PostgreSQL 9.5.24 Documentation

The PostgreSQL Global Development Group

PostgreSQL 9.5.24 Documentation

by The PostgreSQL Global Development Group

Copyright © 1996-2020 The PostgreSQL Global Development Group

Legal Notice

PostgreSQL is Copyright © 1996-2020 by the PostgreSQL Global Development Group.

Postgres95 is Copyright © 1994-5 by the Regents of the University of California.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFTWARE PROVIDED HEREUNDER IS ON AN “AS-IS” BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Table of Contents

Preface ..... **xlvi**
1. What is PostgreSQL? xlvi
2. A Brief History of PostgreSQL xlvii
2.1. The Berkeley POSTGRES Project xlvii
2.2. Postgres95 xlviii
2.3. PostgreSQL xlviii
3. Conventions xlix
4. Further Information xlix
5. Bug Reporting Guidelines xlix
5.1. Identifying Bugs l
5.2. What to Report l
5.3. Where to Report Bugs lii
I. Tutorial **1**
1. Getting Started 1
1.1. Installation 1
1.2. Architectural Fundamentals 1
1.3. Creating a Database 2
1.4. Accessing a Database 3
2. The SQL Language 6
2.1. Introduction 6
2.2. Concepts 6
2.3. Creating a New Table 6
2.4. Populating a Table With Rows 7
2.5. Querying a Table 8
2.6. Joins Between Tables 10
2.7. Aggregate Functions 12
2.8. Updates 14
2.9. Deletions 14
3. Advanced Features 16
3.1. Introduction 16
3.2. Views 16
3.3. Foreign Keys 16
3.4. Transactions 17
3.5. Window Functions 19
3.6. Inheritance 22
3.7. Conclusion 24
II. The SQL Language **25**
4. SQL Syntax 27
4.1. Lexical Structure 27
4.1.1. Identifiers and Key Words 27
4.1.2. Constants 29
4.1.2.1. String Constants 29
4.1.2.2. String Constants with C-style Escapes 29
4.1.2.3. String Constants with Unicode Escapes 31
4.1.2.4. Dollar-quoted String Constants 32

4.1.2.5. Bit-string Constants 33
4.1.2.6. Numeric Constants 33
4.1.2.7. Constants of Other Types 33
4.1.3. Operators 34
4.1.4. Special Characters 35
4.1.5. Comments 35
4.1.6. Operator Precedence 36
4.2. Value Expressions 37
4.2.1. Column References 38
4.2.2. Positional Parameters 38
4.2.3. Subscripts 39
4.2.4. Field Selection 39
4.2.5. Operator Invocations 40
4.2.6. Function Calls 40
4.2.7. Aggregate Expressions 40
4.2.8. Window Function Calls 43
4.2.9. Type Casts 45
4.2.10. Collation Expressions 45
4.2.11. Scalar Subqueries 46
4.2.12. Array Constructors 47
4.2.13. Row Constructors 48
4.2.14. Expression Evaluation Rules 50
4.3. Calling Functions 51
4.3.1. Using Positional Notation 52
4.3.2. Using Named Notation 52
4.3.3. Using Mixed Notation 53
5. Data Definition 55
5.1. Table Basics 55
5.2. Default Values 56
5.3. Constraints 57
5.3.1. Check Constraints 57
5.3.2. Not-Null Constraints 60
5.3.3. Unique Constraints 60
5.3.4. Primary Keys 62
5.3.5. Foreign Keys 62
5.3.6. Exclusion Constraints 65
5.4. System Columns 65
5.5. Modifying Tables 67
5.5.1. Adding a Column 67
5.5.2. Removing a Column 68
5.5.3. Adding a Constraint 68
5.5.4. Removing a Constraint 68
5.5.5. Changing a Column's Default Value 69
5.5.6. Changing a Column's Data Type 69
5.5.7. Renaming a Column 69
5.5.8. Renaming a Table 70
5.6. Privileges 70
5.7. Row Security Policies 71

5.8. Schemas 76
5.8.1. Creating a Schema 77
5.8.2. The Public Schema 78
5.8.3. The Schema Search Path 78
5.8.4. Schemas and Privileges 79
5.8.5. The System Catalog Schema 80
5.8.6. Usage Patterns 80
5.8.7. Portability 81
5.9. Inheritance 81
5.9.1. Caveats 84
5.10. Partitioning 85
5.10.1. Overview 85
5.10.2. Implementing Partitioning 86
5.10.3. Managing Partitions 89
5.10.4. Partitioning and Constraint Exclusion 90
5.10.5. Alternative Partitioning Methods 91
5.10.6. Caveats 92
5.11. Foreign Data 93
5.12. Other Database Objects 93
5.13. Dependency Tracking 94
6. Data Manipulation 96
6.1. Inserting Data 96
6.2. Updating Data 97
6.3. Deleting Data 98
6.4. Returning Data From Modified Rows 98
7. Queries 100
7.1. Overview 100
7.2. Table Expressions 100
7.2.1. The FROM Clause 101
7.2.1.1. Joined Tables 101
7.2.1.2. Table and Column Aliases 105
7.2.1.3. Subqueries 106
7.2.1.4. Table Functions 107
7.2.1.5. LATERAL Subqueries 108
7.2.2. The WHERE Clause 110
7.2.3. The GROUP BY and HAVING Clauses 111
7.2.4. GROUPING SETS, CUBE, and ROLLUP 113
7.2.5. Window Function Processing 115
7.3. Select Lists 116
7.3.1. Select-List Items 116
7.3.2. Column Labels 117
7.3.3. DISTINCT 117
7.4. Combining Queries 118
7.5. Sorting Rows 118
7.6. LIMIT and OFFSET 119
7.7. VALUES Lists 120
7.8. WITH Queries (Common Table Expressions) 121
7.8.1. SELECT in WITH 121

7.8.2. Data-Modifying Statements in WITH 125
8. Data Types 127
8.1. Numeric Types 128
8.1.1. Integer Types 129
8.1.2. Arbitrary Precision Numbers 130
8.1.3. Floating-Point Types 131
8.1.4. Serial Types 132
8.2. Monetary Types 133
8.3. Character Types 134
8.4. Binary Data Types 136
8.4.1. bytea Hex Format 136
8.4.2. bytea Escape Format 137
8.5. Date/Time Types 138
8.5.1. Date/Time Input 140
8.5.1.1. Dates 140
8.5.1.2. Times 141
8.5.1.3. Time Stamps 142
8.5.1.4. Special Values 143
8.5.2. Date/Time Output 144
8.5.3. Time Zones 145
8.5.4. Interval Input 146
8.5.5. Interval Output 148
8.6. Boolean Type 149
8.7. Enumerated Types 150
8.7.1. Declaration of Enumerated Types 150
8.7.2. Ordering 151
8.7.3. Type Safety 151
8.7.4. Implementation Details 152
8.8. Geometric Types 152
8.8.1. Points 153
8.8.2. Lines 153
8.8.3. Line Segments 153
8.8.4. Boxes 154
8.8.5. Paths 154
8.8.6. Polygons 154
8.8.7. Circles 155
8.9. Network Address Types 155
8.9.1. inet 155
8.9.2. cidr 156
8.9.3. inet vs. cidr 156
8.9.4. macaddr 157
8.10. Bit String Types 157
8.11. Text Search Types 158
8.11.1. tsvector 158
8.11.2. tsquery 159
8.12. UUID Type 161
8.13. XML Type 161
8.13.1. Creating XML Values 162

8.13.2. Encoding Handling 163
8.13.3. Accessing XML Values 163
8.14. JSON Types 164
8.14.1. JSON Input and Output Syntax 165
8.14.2. Designing JSON documents effectively 166
8.14.3. `jsonb` Containment and Existence 167
8.14.4. `jsonb` Indexing 168
8.15. Arrays 171
8.15.1. Declaration of Array Types 171
8.15.2. Array Value Input 172
8.15.3. Accessing Arrays 173
8.15.4. Modifying Arrays 175
8.15.5. Searching in Arrays 178
8.15.6. Array Input and Output Syntax 179
8.16. Composite Types 180
8.16.1. Declaration of Composite Types 181
8.16.2. Constructing Composite Values 182
8.16.3. Accessing Composite Types 182
8.16.4. Modifying Composite Types 183
8.16.5. Using Composite Types in Queries 184
8.16.6. Composite Type Input and Output Syntax 186
8.17. Range Types 187
8.17.1. Built-in Range Types 187
8.17.2. Examples 187
8.17.3. Inclusive and Exclusive Bounds 188
8.17.4. Infinite (Unbounded) Ranges 188
8.17.5. Range Input/Output 189
8.17.6. Constructing Ranges 190
8.17.7. Discrete Range Types 190
8.17.8. Defining New Range Types 191
8.17.9. Indexing 192
8.17.10. Constraints on Ranges 192
8.18. Object Identifier Types 193
8.19. pg_lsn Type 195
8.20. Pseudo-Types 195
9. Functions and Operators 197
9.1. Logical Operators 197
9.2. Comparison Operators 197
9.3. Mathematical Functions and Operators 199
9.4. String Functions and Operators 204
9.4.1. `format` 218
9.5. Binary String Functions and Operators 220
9.6. Bit String Functions and Operators 222
9.7. Pattern Matching 223
9.7.1. `LIKE` 224
9.7.2. `SIMILAR TO` Regular Expressions 225
9.7.3. POSIX Regular Expressions 226
9.7.3.1. Regular Expression Details 229

9.7.3.2. Bracket Expressions 232
9.7.3.3. Regular Expression Escapes 233
9.7.3.4. Regular Expression Metasyntax 235
9.7.3.5. Regular Expression Matching Rules 237
9.7.3.6. Limits and Compatibility 238
9.7.3.7. Basic Regular Expressions 239
9.8. Data Type Formatting Functions 239
9.9. Date/Time Functions and Operators 247
9.9.1. EXTRACT, date_part 253
9.9.2. date_trunc 256
9.9.3. AT TIME ZONE 257
9.9.4. Current Date/Time 258
9.9.5. Delaying Execution 260
9.10. Enum Support Functions 260
9.11. Geometric Functions and Operators 261
9.12. Network Address Functions and Operators 266
9.13. Text Search Functions and Operators 268
9.14. XML Functions 272
9.14.1. Producing XML Content 272
9.14.1.1. xmlcomment 272
9.14.1.2. xmlconcat 273
9.14.1.3. xmlelement 273
9.14.1.4. xmlforest 275
9.14.1.5. xmlpi 275
9.14.1.6. xmlroot 276
9.14.1.7. xmlagg 276
9.14.2. XML Predicates 277
9.14.2.1. IS DOCUMENT 277
9.14.2.2. IS NOT DOCUMENT 277
9.14.2.3. XMLEXISTS 277
9.14.2.4. xml_is_well_formed 278
9.14.3. Processing XML 279
9.14.4. Mapping Tables to XML 280
9.15. JSON Functions and Operators 283
9.16. Sequence Manipulation Functions 292
9.17. Conditional Expressions 294
9.17.1. CASE 294
9.17.2. COALESCE 296
9.17.3. NULLIF 297
9.17.4. GREATEST and LEAST 297
9.18. Array Functions and Operators 297
9.19. Range Functions and Operators 301
9.20. Aggregate Functions 303
9.21. Window Functions 311
9.22. Subquery Expressions 313
9.22.1. EXISTS 313
9.22.2. IN 313
9.22.3. NOT IN 314

9.22.4. ANY/SOME 315
9.22.5. ALL 315
9.22.6. Single-row Comparison 316
9.23. Row and Array Comparisons 316
9.23.1. IN 316
9.23.2. NOT IN 316
9.23.3. ANY/SOME (array) 317
9.23.4. ALL (array) 317
9.23.5. Row Constructor Comparison 318
9.23.6. Composite Type Comparison 318
9.24. Set Returning Functions 319
9.25. System Information Functions 323
9.26. System Administration Functions 336
9.26.1. Configuration Settings Functions 336
9.26.2. Server Signaling Functions 337
9.26.3. Backup Control Functions 337
9.26.4. Recovery Control Functions 339
9.26.5. Snapshot Synchronization Functions 341
9.26.6. Replication Functions 342
9.26.7. Database Object Management Functions 346
9.26.8. Index Maintenance Functions 349
9.26.9. Generic File Access Functions 349
9.26.10. Advisory Lock Functions 350
9.27. Trigger Functions 352
9.28. Event Trigger Functions 353
9.28.1. Capturing Changes at Command End 353
9.28.2. Processing Objects Dropped by a DDL Command 354
9.28.3. Handling a Table Rewrite Event 355
10. Type Conversion 357
10.1. Overview 357
10.2. Operators 358
10.3. Functions 362
10.4. Value Storage 366
10.5. UNION, CASE, and Related Constructs 367
11. Indexes 370
11.1. Introduction 370
11.2. Index Types 371
11.3. Multicolumn Indexes 373
11.4. Indexes and ORDER BY 374
11.5. Combining Multiple Indexes 375
11.6. Unique Indexes 376
11.7. Indexes on Expressions 377
11.8. Partial Indexes 377
11.9. Operator Classes and Operator Families 380
11.10. Indexes and Collations 381
11.11. Examining Index Usage 382
12. Full Text Search 384
12.1. Introduction 384

12.1.1. What Is a Document? 385
12.1.2. Basic Text Matching 386
12.1.3. Configurations 387
12.2. Tables and Indexes 387
12.2.1. Searching a Table 387
12.2.2. Creating Indexes 388
12.3. Controlling Text Search 389
12.3.1. Parsing Documents 390
12.3.2. Parsing Queries 391
12.3.3. Ranking Search Results 392
12.3.4. Highlighting Results 394
12.4. Additional Features 396
12.4.1. Manipulating Documents 396
12.4.2. Manipulating Queries 397
12.4.2.1. Query Rewriting 397
12.4.3. Triggers for Automatic Updates 399
12.4.4. Gathering Document Statistics 400
12.5. Parsers 401
12.6. Dictionaries 403
12.6.1. Stop Words 404
12.6.2. Simple Dictionary 405
12.6.3. Synonym Dictionary 406
12.6.4. Thesaurus Dictionary 408
12.6.4.1. Thesaurus Configuration 409
12.6.4.2. Thesaurus Example 409
12.6.5. Ispell Dictionary 410
12.6.6. Snowball Dictionary 411
12.7. Configuration Example 412
12.8. Testing and Debugging Text Search 413
12.8.1. Configuration Testing 414
12.8.2. Parser Testing 416
12.8.3. Dictionary Testing 417
12.9. GIN and GiST Index Types 418
12.10. psql Support 419
12.11. Limitations 421
12.12. Migration from Pre-8.3 Text Search 422
13. Concurrency Control 423
13.1. Introduction 423
13.2. Transaction Isolation 423
13.2.1. Read Committed Isolation Level 424
13.2.2. Repeatable Read Isolation Level 426
13.2.3. Serializable Isolation Level 427
13.3. Explicit Locking 429
13.3.1. Table-level Locks 430
13.3.2. Row-level Locks 432
13.3.3. Page-level Locks 434
13.3.4. Deadlocks 434
13.3.5. Advisory Locks 435

13.4. Data Consistency Checks at the Application Level 436
13.4.1. Enforcing Consistency With Serializable Transactions 436
13.4.2. Enforcing Consistency With Explicit Blocking Locks 437
13.5. Caveats 437
13.6. Locking and Indexes 438
14. Performance Tips 440
14.1. Using EXPLAIN 440
14.1.1. EXPLAIN Basics 440
14.1.2. EXPLAIN ANALYZE 446
14.1.3. Caveats 450
14.2. Statistics Used by the Planner 451
14.3. Controlling the Planner with Explicit JOIN Clauses 452
14.4. Populating a Database 454
14.4.1. Disable Autocommit 455
14.4.2. Use COPY 455
14.4.3. Remove Indexes 455
14.4.4. Remove Foreign Key Constraints 455
14.4.5. Increase maintenance_work_mem 456
14.4.6. Increase max_wal_size 456
14.4.7. Disable WAL Archival and Streaming Replication 456
14.4.8. Run ANALYZE Afterwards 457
14.4.9. Some Notes About pg_dump 457
14.5. Non-Durable Settings 458
III. Server Administration 459
15. Installation from Source Code 461
15.1. Short Version 461
15.2. Requirements 461
15.3. Getting The Source 463
15.4. Installation Procedure 463
15.5. Post-Installation Setup 474
15.5.1. Shared Libraries 474
15.5.2. Environment Variables 475
15.6. Supported Platforms 475
15.7. Platform-specific Notes 476
15.7.1. AIX 476
15.7.1.1. GCC Issues 477
15.7.1.2. Unix-Domain Sockets Broken 477
15.7.1.3. Internet Address Issues 477
15.7.1.4. Memory Management 478
15.7.2. Cygwin 479
15.7.3. HP-UX 480
15.7.4. macOS 481
15.7.5. MinGW/Native Windows 481
15.7.5.1. Collecting Crash Dumps on Windows 481
15.7.6. SCO OpenServer and SCO UnixWare 482
15.7.6.1. Skunkware 482
15.7.6.2. GNU Make 482

15.7.6.3. Readline 482
15.7.6.4. Using the UDK on OpenServer 483
15.7.6.5. Reading the PostgreSQL Man Pages 483
15.7.6.6. C99 Issues with the 7.1.1b Feature Supplement 483
15.7.6.7. Threading on UnixWare 483
15.7.7. Solaris 483
15.7.7.1. Required Tools 483
15.7.7.2. Problems with OpenSSL 484
15.7.7.3. configure Complains About a Failed Test Program 484
15.7.7.4. 64-bit Build Sometimes Crashes 484
15.7.7.5. Compiling for Optimal Performance 485
15.7.7.6. Using DTrace for Tracing PostgreSQL 485
16. Installation from Source Code on Windows 486
16.1. Building with Visual C++ or the Microsoft Windows SDK 486
16.1.1. Requirements 487
16.1.2. Special Considerations for 64-bit Windows 489
16.1.3. Building 489
16.1.4. Cleaning and Installing 489
16.1.5. Running the Regression Tests 490
16.1.6. Building the Documentation 491
16.2. Building libpq with Visual C++ or Borland C++ 491
16.2.1. Generated Files 492
17. Server Setup and Operation 493
17.1. The PostgreSQL User Account 493
17.2. Creating a Database Cluster 493
17.2.1. Use of Secondary File Systems 494
17.2.2. Use of Network File Systems 495
17.3. Starting the Database Server 495
17.3.1. Server Start-up Failures 496
17.3.2. Client Connection Problems 497
17.4. Managing Kernel Resources 498
17.4.1. Shared Memory and Semaphores 498
17.4.2. systemd RemoveIPC 504
17.4.3. Resource Limits 504
17.4.4. Linux Memory Overcommit 505
17.4.5. Linux Huge Pages 507
17.5. Shutting Down the Server 508
17.6. Upgrading a PostgreSQL Cluster 509
17.6.1. Upgrading Data via pg_dumpall 509
17.6.2. Upgrading Data via pg_upgrade 511
17.6.3. Upgrading Data via Replication 511
17.7. Preventing Server Spoofing 511
17.8. Encryption Options 512
17.9. Secure TCP/IP Connections with SSL 513
17.9.1. Using Client Certificates 514
17.9.2. SSL Server File Usage 514
17.9.3. Creating Certificates 515
17.10. Secure TCP/IP Connections with SSH Tunnels 516

17.11. Registering Event Log on Windows 517
18. Server Configuration 519
18.1. Setting Parameters 519
18.1.1. Parameter Names and Values 519
18.1.2. Parameter Interaction via the Configuration File 519
18.1.3. Parameter Interaction via SQL 520
18.1.4. Parameter Interaction via the Shell 521
18.1.5. Managing Configuration File Contents 521
18.2. File Locations 523
18.3. Connections and Authentication 524
18.3.1. Connection Settings 524
18.3.2. Security and Authentication 526
18.4. Resource Consumption 529
18.4.1. Memory 529
18.4.2. Disk 531
18.4.3. Kernel Resource Usage 531
18.4.4. Cost-based Vacuum Delay 532
18.4.5. Background Writer 533
18.4.6. Asynchronous Behavior 534
18.5. Write Ahead Log 534
18.5.1. Settings 534
18.5.2. Checkpoints 538
18.5.3. Archiving 539
18.6. Replication 540
18.6.1. Sending Server(s) 540
18.6.2. Master Server 541
18.6.3. Standby Servers 542
18.7. Query Planning 544
18.7.1. Planner Method Configuration 544
18.7.2. Planner Cost Constants 545
18.7.3. Genetic Query Optimizer 546
18.7.4. Other Planner Options 547
18.8. Error Reporting and Logging 549
18.8.1. Where To Log 549
18.8.2. When To Log 551
18.8.3. What To Log 553
18.8.4. Using CSV-Format Log Output 557
18.8.5. Process Title 558
18.9. Run-time Statistics 558
18.9.1. Query and Index Statistics Collector 559
18.9.2. Statistics Monitoring 559
18.10. Automatic Vacuuming 560
18.11. Client Connection Defaults 562
18.11.1. Statement Behavior 562
18.11.2. Locale and Formatting 566
18.11.3. Shared Library Preloading 568
18.11.4. Other Defaults 569
18.12. Lock Management 570

18.13. Version and Platform Compatibility 571
18.13.1. Previous PostgreSQL Versions 571
18.13.2. Platform and Client Compatibility 573
18.14. Error Handling 573
18.15. Preset Options 574
18.16. Customized Options 575
18.17. Developer Options 576
18.18. Short Options 578
19. Client Authentication 580
19.1. The `pg_hba.conf` File 580
19.2. User Name Maps 586
19.3. Authentication Methods 588
19.3.1. Trust Authentication 588
19.3.2. Password Authentication 588
19.3.3. GSSAPI Authentication 589
19.3.4. SSPI Authentication 590
19.3.5. Ident Authentication 591
19.3.6. Peer Authentication 592
19.3.7. LDAP Authentication 592
19.3.8. RADIUS Authentication 594
19.3.9. Certificate Authentication 595
19.3.10. PAM Authentication 595
19.4. Authentication Problems 596
20. Database Roles 597
20.1. Database Roles 597
20.2. Role Attributes 598
20.3. Role Membership 599
20.4. Dropping Roles 601
20.5. Function Security 602
21. Managing Databases 603
21.1. Overview 603
21.2. Creating a Database 603
21.3. Template Databases 604
21.4. Database Configuration 606
21.5. Destroying a Database 606
21.6. Tablespaces 606
22. Localization 609
22.1. Locale Support 609
22.1.1. Overview 609
22.1.2. Behavior 610
22.1.3. Problems 611
22.2. Collation Support 611
22.2.1. Concepts 612
22.2.2. Managing Collations 613
22.3. Character Set Support 614
22.3.1. Supported Character Sets 615
22.3.2. Setting the Character Set 617
22.3.3. Automatic Character Set Conversion Between Server and Client 618

22.3.4. Further Reading 621
23. Routine Database Maintenance Tasks 622
23.1. Routine Vacuuming 622
23.1.1. Vacuuming Basics 622
23.1.2. Recovering Disk Space 623
23.1.3. Updating Planner Statistics 624
23.1.4. Updating The Visibility Map 625
23.1.5. Preventing Transaction ID Wraparound Failures 625
23.1.5.1. Multixacts and Wraparound 628
23.1.6. The Autovacuum Daemon 629
23.2. Routine Reindexing 630
23.3. Log File Maintenance 631
24. Backup and Restore 633
24.1. SQL Dump 633
24.1.1. Restoring the Dump 634
24.1.2. Using pg_dumpall 635
24.1.3. Handling Large Databases 635
24.2. File System Level Backup 636
24.3. Continuous Archiving and Point-in-Time Recovery (PITR) 637
24.3.1. Setting Up WAL Archiving 638
24.3.2. Making a Base Backup 640
24.3.3. Making a Base Backup Using the Low Level API 641
24.3.4. Recovering Using a Continuous Archive Backup 643
24.3.5. Timelines 645
24.3.6. Tips and Examples 646
24.3.6.1. Standalone Hot Backups 646
24.3.6.2. Compressed Archive Logs 646
24.3.6.3. `archive_command` Scripts 647
24.3.7. Caveats 647
25. High Availability, Load Balancing, and Replication 649
25.1. Comparison of Different Solutions 649
25.2. Log-Shipping Standby Servers 653
25.2.1. Planning 653
25.2.2. Standby Server Operation 654
25.2.3. Preparing the Master for Standby Servers 654
25.2.4. Setting Up a Standby Server 654
25.2.5. Streaming Replication 655
25.2.5.1. Authentication 656
25.2.5.2. Monitoring 656
25.2.6. Replication Slots 657
25.2.6.1. Querying and manipulating replication slots 657
25.2.6.2. Configuration Example 657
25.2.7. Cascading Replication 658
25.2.8. Synchronous Replication 658
25.2.8.1. Basic Configuration 659
25.2.8.2. Planning for Performance 660
25.2.8.3. Planning for High Availability 660
25.2.9. Continuous archiving in standby 661

25.3. Failover 661
25.4. Alternative Method for Log Shipping 662
25.4.1. Implementation 663
25.4.2. Record-based Log Shipping 664
25.5. Hot Standby 664
25.5.1. User's Overview 664
25.5.2. Handling Query Conflicts 666
25.5.3. Administrator's Overview 668
25.5.4. Hot Standby Parameter Reference 671
25.5.5. Caveats 671
26. Recovery Configuration 673
26.1. Archive Recovery Settings 673
26.2. Recovery Target Settings 674
26.3. Standby Server Settings 675
27. Monitoring Database Activity 677
27.1. Standard Unix Tools 677
27.2. The Statistics Collector 678
27.2.1. Statistics Collection Configuration 678
27.2.2. Viewing Statistics 679
27.2.3. Statistics Functions 694
27.3. Viewing Locks 696
27.4. Dynamic Tracing 696
27.4.1. Compiling for Dynamic Tracing 697
27.4.2. Built-in Probes 697
27.4.3. Using Probes 706
27.4.4. Defining New Probes 707
28. Monitoring Disk Usage 709
28.1. Determining Disk Usage 709
28.2. Disk Full Failure 710
29. Reliability and the Write-Ahead Log 711
29.1. Reliability 711
29.2. Write-Ahead Logging (WAL) 713
29.3. Asynchronous Commit 713
29.4. WAL Configuration 715
29.5. WAL Internals 718
30. Regression Tests 720
30.1. Running the Tests 720
30.1.1. Running the Tests Against a Temporary Installation 720
30.1.2. Running the Tests Against an Existing Installation 721
30.1.3. Additional Test Suites 721
30.1.4. Locale and Encoding 722
30.1.5. Extra Tests 722
30.1.6. Testing Hot Standby 722
30.2. Test Evaluation 723
30.2.1. Error Message Differences 723
30.2.2. Locale Differences 724
30.2.3. Date and Time Differences 724
30.2.4. Floating-Point Differences 724

30.2.5. Row Ordering Differences 725
30.2.6. Insufficient Stack Depth 725
30.2.7. The “random” Test 725
30.2.8. Configuration Parameters 726
30.3. Variant Comparison Files 726
30.4. TAP Tests 727
30.5. Test Coverage Examination 727
IV. Client Interfaces 728
31. libpq - C Library 730
31.1. Database Connection Control Functions 730
31.1.1. Connection Strings 736
31.1.1.1. Keyword/Value Connection Strings 737
31.1.1.2. Connection URIs 737
31.1.2. Parameter Key Words 738
31.2. Connection Status Functions 742
31.3. Command Execution Functions 747
31.3.1. Main Functions 747
31.3.2. Retrieving Query Result Information 754
31.3.3. Retrieving Other Result Information 758
31.3.4. Escaping Strings for Inclusion in SQL Commands 759
31.4. Asynchronous Command Processing 762
31.5. Retrieving Query Results Row-By-Row 766
31.6. Canceling Queries in Progress 767
31.7. The Fast-Path Interface 768
31.8. Asynchronous Notification 769
31.9. Functions Associated with the COPY Command 770
31.9.1. Functions for Sending COPY Data 771
31.9.2. Functions for Receiving COPY Data 771
31.9.3. Obsolete Functions for COPY 772
31.10. Control Functions 774
31.11. Miscellaneous Functions 775
31.12. Notice Processing 778
31.13. Event System 779
31.13.1. Event Types 779
31.13.2. Event Callback Procedure 781
31.13.3. Event Support Functions 782
31.13.4. Event Example 783
31.14. Environment Variables 786
31.15. The Password File 787
31.16. The Connection Service File 787
31.17. LDAP Lookup of Connection Parameters 788
31.18. SSL Support 789
31.18.1. Client Verification of Server Certificates 789
31.18.2. Client Certificates 790
31.18.3. Protection Provided in Different Modes 790
31.18.4. SSL Client File Usage 792
31.18.5. SSL Library Initialization 793

31.19. Behavior in Threaded Programs 793
31.20. Building libpq Programs 794
31.21. Example Programs 795
32. Large Objects 806
32.1. Introduction 806
32.2. Implementation Features 806
32.3. Client Interfaces 806
32.3.1. Creating a Large Object 807
32.3.2. Importing a Large Object 807
32.3.3. Exporting a Large Object 808
32.3.4. Opening an Existing Large Object 808
32.3.5. Writing Data to a Large Object 809
32.3.6. Reading Data from a Large Object 809
32.3.7. Seeking in a Large Object 809
32.3.8. Obtaining the Seek Position of a Large Object 810
32.3.9. Truncating a Large Object 810
32.3.10. Closing a Large Object Descriptor 810
32.3.11. Removing a Large Object 811
32.4. Server-side Functions 811
32.5. Example Program 812
33. ECPG - Embedded SQL in C 818
33.1. The Concept 818
33.2. Managing Database Connections 818
33.2.1. Connecting to the Database Server 818
33.2.2. Choosing a Connection 820
33.2.3. Closing a Connection 821
33.3. Running SQL Commands 821
33.3.1. Executing SQL Statements 822
33.3.2. Using Cursors 823
33.3.3. Managing Transactions 823
33.3.4. Prepared Statements 823
33.4. Using Host Variables 824
33.4.1. Overview 825
33.4.2. Declare Sections 825
33.4.3. Retrieving Query Results 826
33.4.4. Type Mapping 827
33.4.4.1. Handling Character Strings 827
33.4.4.2. Accessing Special Data Types 828
33.4.4.2.1. timestamp, date 828
33.4.4.2.2. interval 829
33.4.4.2.3. numeric, decimal 830
33.4.4.3. Host Variables with Nonprimitive Types 831
33.4.4.3.1. Arrays 831
33.4.4.3.2. Structures 832
33.4.4.3.3. Typedefs 833
33.4.4.3.4. Pointers 834
33.4.5. Handling Nonprimitive SQL Data Types 834
33.4.5.1. Arrays 834

33.4.5.2. Composite Types 836
33.4.5.3. User-defined Base Types 838
33.4.6. Indicators 839
33.5. Dynamic SQL 840
33.5.1. Executing Statements without a Result Set 840
33.5.2. Executing a Statement with Input Parameters 840
33.5.3. Executing a Statement with a Result Set 841
33.6. pgtypes Library 842
33.6.1. Character Strings 842
33.6.2. The numeric Type 842
33.6.3. The date Type 845
33.6.4. The timestamp Type 849
33.6.5. The interval Type 852
33.6.6. The decimal Type 853
33.6.7. errno Values of pgtypeslib 854
33.6.8. Special Constants of pgtypeslib 855
33.7. Using Descriptor Areas 855
33.7.1. Named SQL Descriptor Areas 855
33.7.2. SQLDA Descriptor Areas 858
33.7.2.1. SQLDA Data Structure 858
33.7.2.1.1. sqlda_t Structure 858
33.7.2.1.2. sqlvar_t Structure 859
33.7.2.1.3. struct sqlname Structure 860
33.7.2.2. Retrieving a Result Set Using an SQLDA 861
33.7.2.3. Passing Query Parameters Using an SQLDA 862
33.7.2.4. A Sample Application Using SQLDA 863
33.8. Error Handling 869
33.8.1. Setting Callbacks 869
33.8.2. sqlca 871
33.8.3. SQLSTATE vs. SQLCODE 873
33.9. Preprocessor Directives 876
33.9.1. Including Files 877
33.9.2. The define and undef Directives 877
33.9.3. ifdef, ifndef, else, elif, and endif Directives 878
33.10. Processing Embedded SQL Programs 879
33.11. Library Functions 880
33.12. Large Objects 880
33.13. C++ Applications 882
33.13.1. Scope for Host Variables 883
33.13.2. C++ Application Development with External C Module 884
33.14. Embedded SQL Commands 886
ALLOCATE DESCRIPTOR 886
CONNECT 888
DEALLOCATE DESCRIPTOR 891
DECLARE 892
DESCRIBE 894
DISCONNECT 896
EXECUTE IMMEDIATE 898

GET DESCRIPTOR 899
OPEN 902
PREPARE 904
SET AUTOCOMMIT 906
SET CONNECTION 907
SET DESCRIPTOR 908
TYPE 910
VAR 913
WHENEVER 914
33.15. Informix Compatibility Mode 916
33.15.1. Additional Types 916
33.15.2. Additional/Missing Embedded SQL Statements 916
33.15.3. Informix-compatible SQLDA Descriptor Areas 917
33.15.4. Additional Functions 920
33.15.5. Additional Constants 929
33.16. Internals 930
34. The Information Schema 933
34.1. The Schema 933
34.2. Data Types 933
34.3. information_schema_catalog_name 934
34.4. administrable_role_authorizations 934
34.5. applicable_roles 935
34.6. attributes 935
34.7. character_sets 939
34.8. check_constraint_routine_usage 940
34.9. check_constraints 941
34.10. collations 941
34.11. collation_character_set_applicability 942
34.12. column_domain_usage 942
34.13. column_options 943
34.14. column_privileges 943
34.15. column_udt_usage 944
34.16. columns 945
34.17. constraint_column_usage 950
34.18. constraint_table_usage 951
34.19. data_type_privileges 951
34.20. domain_constraints 952
34.21. domain_udt_usage 953
34.22. domains 953
34.23. element_types 957
34.24. enabled_roles 960
34.25. foreign_data_wrapper_options 960
34.26. foreign_data_wrappers 961
34.27. foreign_server_options 961
34.28. foreign_servers 962
34.29. foreign_table_options 962
34.30. foreign_tables 963
34.31. key_column_usage 963

34.32. parameters 964
34.33. referential_constraints 967
34.34. role_column_grants 968
34.35. role_routine_grants 969
34.36. role_table_grants 969
34.37. role_udt_grants 970
34.38. role_usage_grants 971
34.39. routine_privileges 972
34.40. routines 972
34.41. schemata 978
34.42. sequences 979
34.43. sql_features 980
34.44. sql_implementation_info 981
34.45. sql_languages 982
34.46. sql_packages 982
34.47. sql_parts 983
34.48. sql_sizing 983
34.49. sql_sizing_profiles 984
34.50. table_constraints 984
34.51. table_privileges 985
34.52. tables 986
34.53. transforms 987
34.54. triggered_update_columns 988
34.55. triggers 989
34.56. udt_privileges 990
34.57. usage_privileges 991
34.58. user_defined_types 992
34.59. user_mapping_options 993
34.60. user_mappings 994
34.61. view_column_usage 995
34.62. view_routine_usage 995
34.63. view_table_usage 996
34.64. views 997
V. Server Programming 999
35. Extending SQL 1001
35.1. How Extensibility Works 1001
35.2. The PostgreSQL Type System 1001
35.2.1. Base Types 1001
35.2.2. Composite Types 1002
35.2.3. Domains 1002
35.2.4. Pseudo-Types 1002
35.2.5. Polymorphic Types 1002
35.3. User-defined Functions 1003
35.4. Query Language (SQL) Functions 1004
35.4.1. Arguments for SQL Functions 1005
35.4.2. SQL Functions on Base Types 1005
35.4.3. SQL Functions on Composite Types 1007

35.4.4. SQL Functions with Output Parameters 1010
35.4.5. SQL Functions with Variable Numbers of Arguments 1011
35.4.6. SQL Functions with Default Values for Arguments 1012
35.4.7. SQL Functions as Table Sources 1013
35.4.8. SQL Functions Returning Sets 1014
35.4.9. SQL Functions Returning TABLE 1016
35.4.10. Polymorphic SQL Functions 1016
35.4.11. SQL Functions with Collations 1018
35.5. Function Overloading 1019
35.6. Function Volatility Categories 1020
35.7. Procedural Language Functions 1021
35.8. Internal Functions 1021
35.9. C-Language Functions 1022
35.9.1. Dynamic Loading 1022
35.9.2. Base Types in C-Language Functions 1023
35.9.3. Version 0 Calling Conventions 1026
35.9.4. Version 1 Calling Conventions 1029
35.9.5. Writing Code 1031
35.9.6. Compiling and Linking Dynamically-loaded Functions 1032
35.9.7. Composite-type Arguments 1034
35.9.8. Returning Rows (Composite Types) 1036
35.9.9. Returning Sets 1037
35.9.10. Polymorphic Arguments and Return Types 1043
35.9.11. Transform Functions 1045
35.9.12. Shared Memory and LWLocks 1045
35.9.13. Using C++ for Extensibility 1046
35.10. User-defined Aggregates 1046
35.10.1. Moving-Aggregate Mode 1048
35.10.2. Polymorphic and Variadic Aggregates 1049
35.10.3. Ordered-Set Aggregates 1051
35.10.4. Support Functions for Aggregates 1052
35.11. User-defined Types 1053
35.11.1. TOAST Considerations 1056
35.12. User-defined Operators 1058
35.13. Operator Optimization Information 1058
35.13.1. COMMUTATOR 1059
35.13.2. NEGATOR 1060
35.13.3. RESTRICT 1060
35.13.4. JOIN 1061
35.13.5. HASHES 1061
35.13.6. MERGES 1062
35.14. Interfacing Extensions To Indexes 1063
35.14.1. Index Methods and Operator Classes 1063
35.14.2. Index Method Strategies 1064
35.14.3. Index Method Support Routines 1066
35.14.4. An Example 1069
35.14.5. Operator Classes and Operator Families 1071
35.14.6. System Dependencies on Operator Classes 1074

35.14.7. Ordering Operators 1075
35.14.8. Special Features of Operator Classes 1076
35.15. Packaging Related Objects into an Extension 1077
35.15.1. Extension Files 1077
35.15.2. Extension Relocatability 1079
35.15.3. Extension Configuration Tables 1080
35.15.4. Extension Updates 1081
35.15.5. Security Considerations for Extensions 1082
35.15.5.1. Security Considerations for Extension Functions 1082
35.15.5.2. Security Considerations for Extension Scripts 1083
35.15.6. Extension Example 1083
35.16. Extension Building Infrastructure 1085
36. Triggers 1088
36.1. Overview of Trigger Behavior 1088
36.2. Visibility of Data Changes 1090
36.3. Writing Trigger Functions in C 1091
36.4. A Complete Trigger Example 1094
37. Event Triggers 1098
37.1. Overview of Event Trigger Behavior 1098
37.2. Event Trigger Firing Matrix 1099
37.3. Writing Event Trigger Functions in C 1104
37.4. A Complete Event Trigger Example 1105
37.5. A Table Rewrite Event Trigger Example 1106
38. The Rule System 1108
38.1. The Query Tree 1108
38.2. Views and the Rule System 1110
38.2.1. How SELECT Rules Work 1110
38.2.2. View Rules in Non-SELECT Statements 1115
38.2.3. The Power of Views in PostgreSQL 1116
38.2.4. Updating a View 1116
38.3. Materialized Views 1117
38.4. Rules on INSERT, UPDATE, and DELETE 1120
38.4.1. How Update Rules Work 1121
38.4.1.1. A First Rule Step by Step 1122
38.4.2. Cooperation with Views 1125
38.5. Rules and Privileges 1131
38.6. Rules and Command Status 1133
38.7. Rules Versus Triggers 1134
39. Procedural Languages 1137
39.1. Installing Procedural Languages 1137
40. PL/pgSQL - SQL Procedural Language 1140
40.1. Overview 1140
40.1.1. Advantages of Using PL/pgSQL 1140
40.1.2. Supported Argument and Result Data Types 1141
40.2. Structure of PL/pgSQL 1141
40.3. Declarations 1143
40.3.1. Declaring Function Parameters 1144
40.3.2. ALIAS 1146

40.3.3. Copying Types 1147
40.3.4. Row Types 1147
40.3.5. Record Types 1148
40.3.6. Collation of PL/pgSQL Variables 1148
40.4. Expressions 1149
40.5. Basic Statements 1150
40.5.1. Assignment 1150
40.5.2. Executing a Command With No Result 1150
40.5.3. Executing a Query with a Single-row Result 1151
40.5.4. Executing Dynamic Commands 1153
40.5.5. Obtaining the Result Status 1156
40.5.6. Doing Nothing At All 1157
40.6. Control Structures 1158
40.6.1. Returning From a Function 1158
40.6.1.1. RETURN 1158
40.6.1.2. RETURN NEXT and RETURN QUERY 1159
40.6.2. Conditionals 1160
40.6.2.1. IF-THEN 1161
40.6.2.2. IF-THEN-ELSE 1161
40.6.2.3. IF-THEN-ELSIF 1162
40.6.2.4. Simple CASE 1163
40.6.2.5. Searched CASE 1163
40.6.3. Simple Loops 1164
40.6.3.1. LOOP 1164
40.6.3.2. EXIT 1164
40.6.3.3. CONTINUE 1165
40.6.3.4. WHILE 1166
40.6.3.5. FOR (Integer Variant) 1166
40.6.4. Looping Through Query Results 1167
40.6.5. Looping Through Arrays 1168
40.6.6. Trapping Errors 1169
40.6.6.1. Obtaining Information About an Error 1171
40.6.7. Obtaining Execution Location Information 1173
40.7. Cursors 1173
40.7.1. Declaring Cursor Variables 1174
40.7.2. Opening Cursors 1174
40.7.2.1. OPEN FOR *query* 1174
40.7.2.2. OPEN FOR EXECUTE 1175
40.7.2.3. Opening a Bound Cursor 1175
40.7.3. Using Cursors 1176
40.7.3.1. FETCH 1176
40.7.3.2. MOVE 1177
40.7.3.3. UPDATE/DELETE WHERE CURRENT OF 1177
40.7.3.4. CLOSE 1177
40.7.3.5. Returning Cursors 1178
40.7.4. Looping Through a Cursor's Result 1179
40.8. Errors and Messages 1180
40.8.1. Reporting Errors and Messages 1180

40.8.2. Checking Assertions 1182
40.9. Trigger Procedures 1182
40.9.1. Triggers on Data Changes 1183
40.9.2. Triggers on Events 1190
40.10. PL/pgSQL Under the Hood 1190
40.10.1. Variable Substitution 1190
40.10.2. Plan Caching 1193
40.11. Tips for Developing in PL/pgSQL 1194
40.11.1. Handling of Quotation Marks 1195
40.11.2. Additional Compile-time Checks 1196
40.12. Porting from Oracle PL/SQL 1197
40.12.1. Porting Examples 1198
40.12.2. Other Things to Watch For 1203
40.12.2.1. Implicit Rollback after Exceptions 1203
40.12.2.2. `EXECUTE` 1204
40.12.2.3. Optimizing PL/pgSQL Functions 1204
40.12.3. Appendix 1204
41. PL/Tcl - Tcl Procedural Language 1208
41.1. Overview 1208
41.2. PL/Tcl Functions and Arguments 1208
41.3. Data Values in PL/Tcl 1210
41.4. Global Data in PL/Tcl 1210
41.5. Database Access from PL/Tcl 1210
41.6. Trigger Procedures in PL/Tcl 1213
41.7. Event Trigger Procedures in PL/Tcl 1215
41.8. Modules and the `unknown` Command 1215
41.9. Tcl Procedure Names 1216
42. PL/Perl - Perl Procedural Language 1217
42.1. PL/Perl Functions and Arguments 1217
42.2. Data Values in PL/Perl 1221
42.3. Built-in Functions 1221
42.3.1. Database Access from PL/Perl 1221
42.3.2. Utility Functions in PL/Perl 1224
42.4. Global Values in PL/Perl 1226
42.5. Trusted and Untrusted PL/Perl 1227
42.6. PL/Perl Triggers 1228
42.7. PL/Perl Event Triggers 1229
42.8. PL/Perl Under the Hood 1230
42.8.1. Configuration 1230
42.8.2. Limitations and Missing Features 1231
43. PL/Python - Python Procedural Language 1233
43.1. Python 2 vs. Python 3 1233
43.2. PL/Python Functions 1234
43.3. Data Values 1236
43.3.1. Data Type Mapping 1236
43.3.2. Null, None 1237
43.3.3. Arrays, Lists 1237
43.3.4. Composite Types 1238

43.3.5. Set-returning Functions 1240
43.4. Sharing Data 1241
43.5. Anonymous Code Blocks 1241
43.6. Trigger Functions 1242
43.7. Database Access 1242
43.7.1. Database Access Functions 1243
43.7.2. Trapping Errors 1245
43.8. Explicit Subtransactions 1246
43.8.1. Subtransaction Context Managers 1246
43.8.2. Older Python Versions 1247
43.9. Utility Functions 1248
43.10. Environment Variables 1249
44. Server Programming Interface 1250
44.1. Interface Functions 1250
SPI_connect 1250
SPI_finish 1252
SPI_push 1253
SPI_pop 1254
SPI_execute 1255
SPI_exec 1259
SPI_execute_with_args 1260
SPI_prepare 1262
SPI_prepare_cursor 1264
SPI_prepare_params 1265
SPI_getargcount 1266
SPI_getargtypeid 1267
SPI_is_cursor_plan 1268
SPI_execute_plan 1269
SPI_execute_plan_with_paramlist 1271
SPI_execp 1272
SPI_cursor_open 1273
SPI_cursor_open_with_args 1275
SPI_cursor_open_with_paramlist 1277
SPI_cursor_find 1278
SPI_cursor_fetch 1279
SPI_cursor_move 1280
SPI_scroll_cursor_fetch 1281
SPI_scroll_cursor_move 1282
SPI_cursor_close 1283
SPI_keepplan 1284
SPI_saveplan 1285
44.2. Interface Support Functions 1286
SPI_fname 1286
SPI_fnumber 1287
SPI_getvalue 1288
SPI_getbinval 1289
SPI_gettype 1290
SPI_gettypeid 1291

SPI_getrelname 1292
SPI_getnspname 1293
44.3. Memory Management 1294
SPI_palloc 1294
SPI_repalloc 1296
SPI_pfree 1297
SPI_copytuple 1298
SPI_returntuple 1299
SPI_modifytuple 1300
SPI_freetuple 1302
SPI_freetuptable 1303
SPI_freeplan 1304
44.4. Visibility of Data Changes 1305
44.5. Examples 1305
45. Background Worker Processes 1309
46. Logical Decoding 1312
46.1. Logical Decoding Examples 1312
46.2. Logical Decoding Concepts 1314
46.2.1. Logical Decoding 1314
46.2.2. Replication Slots 1314
46.2.3. Output Plugins 1315
46.2.4. Exported Snapshots 1315
46.3. Streaming Replication Protocol Interface 1315
46.4. Logical Decoding SQL Interface 1316
46.5. System Catalogs Related to Logical Decoding 1316
46.6. Logical Decoding Output Plugins 1316
46.6.1. Initialization Function 1316
46.6.2. Capabilities 1317
46.6.3. Output Modes 1317
46.6.4. Output Plugin Callbacks 1317
46.6.4.1. Startup Callback 1317
46.6.4.2. Shutdown Callback 1318
46.6.4.3. Transaction Begin Callback 1318
46.6.4.4. Transaction End Callback 1318
46.6.4.5. Change Callback 1318
46.6.4.6. Origin Filter Callback 1319
46.6.5. Functions for Producing Output 1319
46.7. Logical Decoding Output Writers 1320
46.8. Synchronous Replication Support for Logical Decoding 1320
47. Replication Progress Tracking 1321
VI. Reference 1322
I. SQL Commands 1324
ABORT 1325
ALTER AGGREGATE 1327
ALTER COLLATION 1330
ALTER CONVERSION 1332
ALTER DATABASE 1334

ALTER DEFAULT PRIVILEGES 1337
ALTER DOMAIN 1340
ALTER EVENT TRIGGER 1344
ALTER EXTENSION 1345
ALTER FOREIGN DATA WRAPPER 1349
ALTER FOREIGN TABLE 1351
ALTER FUNCTION 1357
ALTER GROUP 1361
ALTER INDEX 1363
ALTER LANGUAGE 1366
ALTER LARGE OBJECT 1367
ALTER MATERIALIZED VIEW 1368
ALTER OPERATOR 1370
ALTER OPERATOR CLASS 1372
ALTER OPERATOR FAMILY 1374
ALTER POLICY 1378
ALTER ROLE 1380
ALTER RULE 1385
ALTER SCHEMA 1387
ALTER SEQUENCE 1388
ALTER SERVER 1391
ALTER SYSTEM 1393
ALTER TABLE 1395
ALTER TABLESPACE 1408
ALTER TEXT SEARCH CONFIGURATION 1410
ALTER TEXT SEARCH DICTIONARY 1412
ALTER TEXT SEARCH PARSER 1414
ALTER TEXT SEARCH TEMPLATE 1415
ALTER TRIGGER 1416
ALTER TYPE 1418
ALTER USER 1422
ALTER USER MAPPING 1424
ALTER VIEW 1426
ANALYZE 1429
BEGIN 1432
CHECKPOINT 1434
CLOSE 1435
CLUSTER 1437
COMMENT 1440
COMMIT 1444
COMMIT PREPARED 1446
COPY 1448
CREATE AGGREGATE 1459
CREATE CAST 1466
CREATE COLLATION 1471
CREATE CONVERSION 1473
CREATE DATABASE 1475
CREATE DOMAIN 1479

CREATE EVENT TRIGGER....1482
CREATE EXTENSION....1484
CREATE FOREIGN DATA WRAPPER....1487
CREATE FOREIGN TABLE....1489
CREATE FUNCTION....1493
CREATE GROUP....1501
CREATE INDEX....1502
CREATE LANGUAGE....1509
CREATE MATERIALIZED VIEW....1513
CREATE OPERATOR....1515
CREATE OPERATOR CLASS....1519
CREATE OPERATOR FAMILY....1523
CREATE POLICY....1525
CREATE ROLE....1531
CREATE RULE....1536
CREATE SCHEMA....1539
CREATE SEQUENCE....1542
CREATE SERVER....1546
CREATE TABLE....1548
CREATE TABLE AS....1564
CREATE TABLESPACE....1567
CREATE TEXT SEARCH CONFIGURATION....1569
CREATE TEXT SEARCH DICTIONARY....1571
CREATE TEXT SEARCH PARSER....1573
CREATE TEXT SEARCH TEMPLATE....1575
CREATE TRANSFORM....1577
CREATE TRIGGER....1580
CREATE TYPE....1586
CREATE USER....1596
CREATE USER MAPPING....1597
CREATE VIEW....1599
DEALLOCATE....1604
DECLARE....1605
DELETE....1609
DISCARD....1612
DO....1614
DROP AGGREGATE....1616
DROP CAST....1618
DROP COLLATION....1620
DROP CONVERSION....1622
DROP DATABASE....1624
DROP DOMAIN....1625
DROP EVENT TRIGGER....1627
DROP EXTENSION....1629
DROP FOREIGN DATA WRAPPER....1631
DROP FOREIGN TABLE....1633
DROP FUNCTION....1635
DROP GROUP....1637

DROP INDEX 1638
DROP LANGUAGE 1640
DROP MATERIALIZED VIEW 1642
DROP OPERATOR 1644
DROP OPERATOR CLASS 1646
DROP OPERATOR FAMILY 1648
DROP OWNED 1650
DROP POLICY 1652
DROP ROLE 1654
DROP RULE 1656
DROP SCHEMA 1658
DROP SEQUENCE 1660
DROP SERVER 1662
DROP TABLE 1664
DROP TABLESPACE 1666
DROP TEXT SEARCH CONFIGURATION 1668
DROP TEXT SEARCH DICTIONARY 1670
DROP TEXT SEARCH PARSER 1672
DROP TEXT SEARCH TEMPLATE 1674
DROP TRANSFORM 1676
DROP TRIGGER 1678
DROP TYPE 1680
DROP USER 1682
DROP USER MAPPING 1683
DROP VIEW 1685
END 1687
EXECUTE 1689
EXPLAIN 1691
FETCH 1697
GRANT 1701
IMPORT FOREIGN SCHEMA 1709
INSERT 1711
LISTEN 1718
LOAD 1720
LOCK 1721
MOVE 1724
NOTIFY 1726
PREPARE 1729
PREPARE TRANSACTION 1732
REASSIGN OWNED 1734
REFRESH MATERIALIZED VIEW 1736
REINDEX 1738
RELEASE SAVEPOINT 1741
RESET 1743
REVOKE 1745
ROLLBACK 1749
ROLLBACK PREPARED 1751
ROLLBACK TO SAVEPOINT 1753

SAVEPOINT 1755
SECURITY LABEL 1757
SELECT 1760
SELECT INTO 1782
SET 1784
SET CONSTRAINTS 1788
SET ROLE 1790
SET SESSION AUTHORIZATION 1792
SET TRANSACTION 1794
SHOW 1797
START TRANSACTION 1800
TRUNCATE 1801
UNLISTEN 1804
UPDATE 1806
VACUUM 1811
VALUES 1814
II. PostgreSQL Client Applications 1817
clusterdb 1818
createdb 1821
createlang 1825
createuser 1828
dropdb 1833
droplang 1836
dropuser 1839
ecpg 1842
pg_basebackup 1845
pgbench 1852
pg_config 1864
pg_dump 1867
pg_dumpall 1880
pg_isready 1886
pg_receivexlog 1889
pg_recvlogical 1893
pg_restore 1897
psql 1906
reindexdb 1940
vacuumdb 1944
III. PostgreSQL Server Applications 1948
initdb 1949
pg_archivecleanup 1953
pg_controldata 1956
pg_ctl 1957
pg_resetxlog 1963
pg_rewind 1966
pg_test_fsync 1969
pg_test_timing 1971
pg_upgrade 1975
pg_xlogdump 1983

postgres 1986
postmaster 1994
VII. Internals 1995
48. Overview of PostgreSQL Internals 1997
48.1. The Path of a Query 1997
48.2. How Connections are Established 1997
48.3. The Parser Stage 1998
48.3.1. Parser 1998
48.3.2. Transformation Process 1999
48.4. The PostgreSQL Rule System 1999
48.5. Planner/Optimizer 2000
48.5.1. Generating Possible Plans 2000
48.6. Executor 2001
49. System Catalogs 2003
49.1. Overview 2003
49.2. pg_aggregate 2004
49.3. pg_am 2007
49.4. pg_amop 2009
49.5. pg_amproc 2011
49.6. pg_attrdef 2011
49.7. pg_attribute 2012
49.8. pg_authid 2015
49.9. pg_auth_members 2017
49.10. pg_cast 2017
49.11. pg_class 2019
49.12. pg_collation 2023
49.13. pg_constraint 2024
49.14. pg_conversion 2027
49.15. pg_database 2028
49.16. pg_db_role_setting 2030
49.17. pg_default_acl 2031
49.18. pg_depend 2032
49.19. pg_description 2033
49.20. pg_enum 2034
49.21. pg_event_trigger 2035
49.22. pg_extension 2035
49.23. pg_foreign_data_wrapper 2036
49.24. pg_foreign_server 2037
49.25. pg_foreign_table 2038
49.26. pg_index 2038
49.27. pg_inherits 2041
49.28. pg_language 2042
49.29. pg_largeobject 2043
49.30. pg_largeobject_metadata 2044
49.31. pg_namespace 2044
49.32. pg_opclass 2045
49.33. pg_operator 2046

49.34. pg_opfamily 2047
49.35. pg_pltemplate 2048
49.36. pg_policy 2048
49.37. pg_proc 2049
49.38. pg_range 2054
49.39. pg_replication_origin 2055
49.40. pg_rewrite 2055
49.41. pg_seclabel 2056
49.42. pg_shdepend 2057
49.43. pg_shdescription 2058
49.44. pg_shseclabel 2059
49.45. pg_statistic 2059
49.46. pg_tablespace 2062
49.47. pg_transform 2062
49.48. pg_trigger 2063
49.49. pg_ts_config 2065
49.50. pg_ts_config_map 2066
49.51. pg_ts_dict 2066
49.52. pg_ts_parser 2067
49.53. pg_ts_template 2068
49.54. pg_type 2068
49.55. pg_user_mapping 2077
49.56. System Views 2077
49.57. pg_available_extensions 2078
49.58. pg_available_extension_versions 2079
49.59. pg_cursors 2079
49.60. pg_file_settings 2080
49.61. pg_group 2081
49.62. pg_indexes 2082
49.63. pg_locks 2082
49.64. pg_matviews 2086
49.65. pg_policies 2086
49.66. pg_prepared_statements 2087
49.67. pg_prepared_xacts 2088
49.68. pg_replication_origin_status 2089
49.69. pg_replication_slots 2089
49.70. pg_roles 2091
49.71. pg_rules 2092
49.72. pg_seclabels 2093
49.73. pg_settings 2094
49.74. pg_shadow 2096
49.75. pg_stats 2097
49.76. pg_tables 2100
49.77. pg_timezone_abbrevs 2101
49.78. pg_timezone_names 2101
49.79. pg_user 2102
49.80. pg_user_mappings 2102
49.81. pg_views 2103

50. Frontend/Backend Protocol 2104
50.1. Overview 2104
50.1.1. Messaging Overview 2104
50.1.2. Extended Query Overview 2105
50.1.3. Formats and Format Codes 2105
50.2. Message Flow 2106
50.2.1. Start-up 2106
50.2.2. Simple Query 2108
50.2.3. Extended Query 2110
50.2.4. Function Call 2112
50.2.5. COPY Operations 2113
50.2.6. Asynchronous Operations 2114
50.2.7. Canceling Requests in Progress 2115
50.2.8. Termination 2116
50.2.9. SSL Session Encryption 2116
50.3. Streaming Replication Protocol 2117
50.4. Message Data Types 2123
50.5. Message Formats 2124
50.6. Error and Notice Message Fields 2140
50.7. Summary of Changes since Protocol 2.0 2142
51. PostgreSQL Coding Conventions 2144
51.1. Formatting 2144
51.2. Reporting Errors Within the Server 2145
51.3. Error Message Style Guide 2148
51.3.1. What Goes Where 2148
51.3.2. Formatting 2148
51.3.3. Quotation Marks 2148
51.3.4. Use of Quotes 2149
51.3.5. Grammar and Punctuation 2149
51.3.6. Upper Case vs. Lower Case 2149
51.3.7. Avoid Passive Voice 2150
51.3.8. Present vs. Past Tense 2150
51.3.9. Type of the Object 2150
51.3.10. Brackets 2150
51.3.11. Assembling Error Messages 2150
51.3.12. Reasons for Errors 2151
51.3.13. Function Names 2151
51.3.14. Tricky Words to Avoid 2151
51.3.15. Proper Spelling 2152
51.3.16. Localization 2152
52. Native Language Support 2153
52.1. For the Translator 2153
52.1.1. Requirements 2153
52.1.2. Concepts 2153
52.1.3. Creating and Maintaining Message Catalogs 2154
52.1.4. Editing the PO Files 2155
52.2. For the Programmer 2156
52.2.1. Mechanics 2156

52.2.2. Message-writing Guidelines 2157
53. Writing A Procedural Language Handler 2159
54. Writing A Foreign Data Wrapper 2162
54.1. Foreign Data Wrapper Functions 2162
54.2. Foreign Data Wrapper Callback Routines 2162
54.2.1. FDW Routines For Scanning Foreign Tables 2163
54.2.2. FDW Routines For Scanning Foreign Joins 2165
54.2.3. FDW Routines For Updating Foreign Tables 2165
54.2.4. FDW Routines For Row Locking 2169
54.2.5. FDW Routines for `EXPLAIN` 2170
54.2.6. FDW Routines for `ANALYZE` 2171
54.2.7. FDW Routines For `IMPORT FOREIGN SCHEMA` 2171
54.3. Foreign Data Wrapper Helper Functions 2172
54.4. Foreign Data Wrapper Query Planning 2173
54.5. Row Locking in Foreign Data Wrappers 2175
55. Writing A Table Sampling Method 2177
55.1. Sampling Method Support Functions 2177
56. Writing A Custom Scan Provider 2181
56.1. Creating Custom Scan Paths 2181
56.1.1. Custom Scan Path Callbacks 2182
56.2. Creating Custom Scan Plans 2182
56.2.1. Custom Scan Plan Callbacks 2183
56.3. Executing Custom Scans 2184
56.3.1. Custom Scan Execution Callbacks 2184
57. Genetic Query Optimizer 2186
57.1. Query Handling as a Complex Optimization Problem 2186
57.2. Genetic Algorithms 2186
57.3. Genetic Query Optimization (GEQO) in PostgreSQL 2187
57.3.1. Generating Possible Plans with GEQO 2188
57.3.2. Future Implementation Tasks for PostgreSQL GEQO 2188
57.4. Further Reading 2189
58. Index Access Method Interface Definition 2190
58.1. Catalog Entries for Indexes 2190
58.2. Index Access Method Functions 2191
58.3. Index Scanning 2195
58.4. Index Locking Considerations 2197
58.5. Index Uniqueness Checks 2198
58.6. Index Cost Estimation Functions 2199
59. GiST Indexes 2203
59.1. Introduction 2203
59.2. Built-in Operator Classes 2203
59.3. Extensibility 2204
59.4. Implementation 2212
59.4.1. GiST buffering build 2212
59.5. Examples 2213
60. SP-GiST Indexes 2214
60.1. Introduction 2214
60.2. Built-in Operator Classes 2214

60.3. Extensibility 2215
60.4. Implementation 2221
60.4.1. SP-GiST Limits 2221
60.4.2. SP-GiST Without Node Labels 2222
60.4.3. “All-the-same” Inner Tuples 2222
60.5. Examples 2222
61. GIN Indexes 2224
61.1. Introduction 2224
61.2. Built-in Operator Classes 2224
61.3. Extensibility 2225
61.4. Implementation 2228
61.4.1. GIN Fast Update Technique 2228
61.4.2. Partial Match Algorithm 2229
61.5. GIN Tips and Tricks 2229
61.6. Limitations 2230
61.7. Examples 2230
62. BRIN Indexes 2231
62.1. Introduction 2231
62.1.1. Index Maintenance 2231
62.2. Built-in Operator Classes 2231
62.3. Extensibility 2233
63. Database Physical Storage 2237
63.1. Database File Layout 2237
63.2. TOAST 2239
63.2.1. Out-of-line, on-disk TOAST storage 2240
63.2.2. Out-of-line, in-memory TOAST storage 2242
63.3. Free Space Map 2242
63.4. Visibility Map 2243
63.5. The Initialization Fork 2243
63.6. Database Page Layout 2243
64. BKI Backend Interface 2247
64.1. BKI File Format 2247
64.2. BKI Commands 2247
64.3. Structure of the Bootstrap BKI File 2248
64.4. Example 2249
65. How the Planner Uses Statistics 2250
65.1. Row Estimation Examples 2250
65.2. Planner Statistics and Security 2255
VIII. Appendixes 2257
A. PostgreSQL Error Codes 2258
B. Date/Time Support 2267
B.1. Date/Time Input Interpretation 2267
B.2. Handling of Invalid or Ambiguous Timestamps 2268
B.3. Date/Time Key Words 2269
B.4. Date/Time Configuration Files 2270
B.5. POSIX Time Zone Specifications 2272
B.6. History of Units 2274

C. SQL Key Words 2276
D. SQL Conformance 2301
D.1. Supported Features 2302
D.2. Unsupported Features 2318
E. Release Notes 2334
E.1. Release 9.5.24 2334
E.1.1. Migration to Version 9.5.24 2334
E.1.2. Changes 2334
E.2. Release 9.5.23 2337
E.2.1. Migration to Version 9.5.23 2337
E.2.2. Changes 2337
E.3. Release 9.5.22 2340
E.3.1. Migration to Version 9.5.22 2340
E.3.2. Changes 2340
E.4. Release 9.5.21 2342
E.4.1. Migration to Version 9.5.21 2343
E.4.2. Changes 2343
E.5. Release 9.5.20 2345
E.5.1. Migration to Version 9.5.20 2345
E.5.2. Changes 2345
E.6. Release 9.5.19 2349
E.6.1. Migration to Version 9.5.19 2349
E.6.2. Changes 2349
E.7. Release 9.5.18 2351
E.7.1. Migration to Version 9.5.18 2351
E.7.2. Changes 2351
E.8. Release 9.5.17 2352
E.8.1. Migration to Version 9.5.17 2353
E.8.2. Changes 2353
E.9. Release 9.5.16 2355
E.9.1. Migration to Version 9.5.16 2356
E.9.2. Changes 2356
E.10. Release 9.5.15 2359
E.10.1. Migration to Version 9.5.15 2359
E.10.2. Changes 2359
E.11. Release 9.5.14 2363
E.11.1. Migration to Version 9.5.14 2363
E.11.2. Changes 2363
E.12. Release 9.5.13 2366
E.12.1. Migration to Version 9.5.13 2366
E.12.2. Changes 2366
E.13. Release 9.5.12 2369
E.13.1. Migration to Version 9.5.12 2369
E.13.2. Changes 2370
E.14. Release 9.5.11 2371
E.14.1. Migration to Version 9.5.11 2371
E.14.2. Changes 2371
E.15. Release 9.5.10 2374

E.15.1. Migration to Version 9.5.10 2374
E.15.2. Changes 2374
E.16. Release 9.5.9 2376
E.16.1. Migration to Version 9.5.9 2376
E.16.2. Changes 2377
E.17. Release 9.5.8 2378
E.17.1. Migration to Version 9.5.8 2378
E.17.2. Changes 2378
E.18. Release 9.5.7 2383
E.18.1. Migration to Version 9.5.7 2383
E.18.2. Changes 2384
E.19. Release 9.5.6 2387
E.19.1. Migration to Version 9.5.6 2388
E.19.2. Changes 2388
E.20. Release 9.5.5 2392
E.20.1. Migration to Version 9.5.5 2392
E.20.2. Changes 2392
E.21. Release 9.5.4 2396
E.21.1. Migration to Version 9.5.4 2396
E.21.2. Changes 2396
E.22. Release 9.5.3 2400
E.22.1. Migration to Version 9.5.3 2400
E.22.2. Changes 2401
E.23. Release 9.5.2 2402
E.23.1. Migration to Version 9.5.2 2402
E.23.2. Changes 2403
E.24. Release 9.5.1 2405
E.24.1. Migration to Version 9.5.1 2405
E.24.2. Changes 2405
E.25. Release 9.5 2407
E.25.1. Overview 2407
E.25.2. Migration to Version 9.5 2407
E.25.3. Changes 2409
E.25.3.1. Server 2409
E.25.3.1.1. Indexes 2409
E.25.3.1.2. General Performance 2409
E.25.3.1.3. Monitoring 2410
E.25.3.1.4. SSL 2410
E.25.3.1.5. Server Settings 2411
E.25.3.2. Replication and Recovery 2412
E.25.3.3. Queries 2412
E.25.3.4. Utility Commands 2413
E.25.3.4.1. REINDEX 2413
E.25.3.5. Object Manipulation 2413
E.25.3.5.1. Foreign Tables 2414
E.25.3.5.2. Event Triggers 2414
E.25.3.6. Data Types 2415
E.25.3.6.1. JSON 2415

E.25.3.7. Functions 2415
E.25.3.7.1. System Information Functions and Views 2416
E.25.3.7.2. Aggregates 2416
E.25.3.8. Server-Side Languages 2416
E.25.3.8.1. PL/pgSQL Server-Side Language 2417
E.25.3.9. Client Applications 2417
E.25.3.9.1. psql 2417
E.25.3.9.1.1. Backslash Commands 2418
E.25.3.9.2. pg_dump 2418
E.25.3.9.3. pg_ctl 2419
E.25.3.9.4. pg_upgrade 2419
E.25.3.9.5. pgbench 2419
E.25.3.10. Source Code 2420
E.25.3.10.1. MS Windows 2420
E.25.3.11. Additional Modules 2421
E.26. Prior Releases 2422
F. Additional Supplied Modules 2423
F.1. adminpack 2424
F.2. auth_delay 2425
F.2.1. Configuration Parameters 2425
F.2.2. Author 2425
F.3. auto_explain 2425
F.3.1. Configuration Parameters 2426
F.3.2. Example 2427
F.3.3. Author 2428
F.4. btree_gin 2428
F.4.1. Example Usage 2428
F.4.2. Authors 2428
F.5. btree_gist 2428
F.5.1. Example Usage 2429
F.5.2. Authors 2429
F.6. chkpass 2430
F.6.1. Author 2431
F.7. citext 2431
F.7.1. Rationale 2431
F.7.2. How to Use It 2431
F.7.3. String Comparison Behavior 2432
F.7.4. Limitations 2432
F.7.5. Author 2433
F.8. cube 2433
F.8.1. Syntax 2433
F.8.2. Precision 2434
F.8.3. Usage 2434
F.8.4. Defaults 2436
F.8.5. Notes 2437
F.8.6. Credits 2437
F.9. dblink 2437
dblink_connect 2438

dblink_connect_u 2441
dblink_disconnect 2442
dblink 2443
dblink_exec 2447
dblink_open 2449
dblink_fetch 2451
dblink_close 2453
dblink_get_connections 2455
dblink_error_message 2456
dblink_send_query 2457
dblink_is_busy 2458
dblink_get_notify 2459
dblink_get_result 2461
dblink_cancel_query 2464
dblink_get_pkey 2465
dblink_build_sql_insert 2467
dblink_build_sql_delete 2469
dblink_build_sql_update 2471
F.10. dict_int 2473
F.10.1. Configuration 2473
F.10.2. Usage 2473
F.11. dict_xsyn 2473
F.11.1. Configuration 2474
F.11.2. Usage 2474
F.12. earthdistance 2475
F.12.1. Cube-based Earth Distances 2476
F.12.2. Point-based Earth Distances 2477
F.13. file_fdw 2477
F.14. fuzzystrmatch 2479
F.14.1. Soundex 2480
F.14.2. Levenshtein 2480
F.14.3. Metaphone 2481
F.14.4. Double Metaphone 2482
F.15. hstore 2482
F.15.1. `hstore` External Representation 2482
F.15.2. `hstore` Operators and Functions 2483
F.15.3. Indexes 2487
F.15.4. Examples 2487
F.15.5. Statistics 2488
F.15.6. Compatibility 2489
F.15.7. Transforms 2489
F.15.8. Authors 2490
F.16. intagg 2490
F.16.1. Functions 2490
F.16.2. Sample Uses 2490
F.17. intarray 2491
F.17.1. `intarray` Functions and Operators 2491
F.17.2. Index Support 2493

F.17.3. Example 2493
F.17.4. Benchmark 2494
F.17.5. Authors 2494
F.18. isn 2494
F.18.1. Data Types 2494
F.18.2. Casts 2495
F.18.3. Functions and Operators 2496
F.18.4. Examples 2497
F.18.5. Bibliography 2498
F.18.6. Author 2498
F.19. lo 2498
F.19.1. Rationale 2498
F.19.2. How to Use It 2499
F.19.3. Limitations 2499
F.19.4. Author 2499
F.20. ltree 2500
F.20.1. Definitions 2500
F.20.2. Operators and Functions 2501
F.20.3. Indexes 2504
F.20.4. Example 2504
F.20.5. Transforms 2506
F.20.6. Authors 2507
F.21. pageinspect 2507
F.21.1. Functions 2507
F.22. passwordcheck 2511
F.23. pg_buffercache 2511
F.23.1. The `pg_buffercache` View 2511
F.23.2. Sample Output 2512
F.23.3. Authors 2513
F.24. pgcrypto 2513
F.24.1. General Hashing Functions 2513
F.24.1.1. `digest()` 2513
F.24.1.2. `hmac()` 2513
F.24.2. Password Hashing Functions 2514
F.24.2.1. `crypt()` 2514
F.24.2.2. `gen_salt()` 2515
F.24.3. PGP Encryption Functions 2516
F.24.3.1. `pgp_sym_encrypt()` 2517
F.24.3.2. `pgp_sym_decrypt()` 2517
F.24.3.3. `pgp_pub_encrypt()` 2517
F.24.3.4. `pgp_pub_decrypt()` 2517
F.24.3.5. `pgp_key_id()` 2518
F.24.3.6. `armor()`, `dearmor()` 2518
F.24.3.7. `pgp_armor_headers` 2518
F.24.3.8. Options for PGP Functions 2519
F.24.3.8.1. cipher-algo 2519
F.24.3.8.2. compress-algo 2519
F.24.3.8.3. compress-level 2519

F.24.3.8.4. convert-crlf 2519
F.24.3.8.5. disable-mdc 2520
F.24.3.8.6. sess-key 2520
F.24.3.8.7. s2k-mode 2520
F.24.3.8.8. s2k-digest-algo 2520
F.24.3.8.9. s2k-cipher-algo 2520
F.24.3.8.10. unicode-mode 2521
F.24.3.9. Generating PGP Keys with GnuPG 2521
F.24.3.10. Limitations of PGP Code 2522
F.24.4. Raw Encryption Functions 2522
F.24.5. Random-Data Functions 2523
F.24.6. Notes 2523
F.24.6.1. Configuration 2523
F.24.6.2. NULL Handling 2524
F.24.6.3. Security Limitations 2524
F.24.6.4. Useful Reading 2524
F.24.6.5. Technical References 2525
F.24.7. Author 2525
F.25. pg_freespacemap 2526
F.25.1. Functions 2526
F.25.2. Sample Output 2526
F.25.3. Author 2527
F.26. pg_prewarm 2527
F.26.1. Functions 2527
F.26.2. Author 2528
F.27. pgrowlocks 2528
F.27.1. Overview 2528
F.27.2. Sample Output 2529
F.27.3. Author 2529
F.28. pg_stat_statements 2529
F.28.1. The `pg_stat_statements` View 2529
F.28.2. Functions 2532
F.28.3. Configuration Parameters 2533
F.28.4. Sample Output 2534
F.28.5. Authors 2534
F.29. pgstattuple 2534
F.29.1. Functions 2535
F.29.2. Authors 2538
F.30. pg_trgm 2538
F.30.1. Trigram (or Trigraph) Concepts 2539
F.30.2. Functions and Operators 2539
F.30.3. Index Support 2540
F.30.4. Text Search Integration 2541
F.30.5. References 2542
F.30.6. Authors 2542
F.31. postgres_fdw 2542
F.31.1. FDW Options of postgres_fdw 2543
F.31.1.1. Connection Options 2543

F.31.1.2. Object Name Options 2543
F.31.1.3. Cost Estimation Options 2544
F.31.1.4. Updatability Options 2544
F.31.1.5. Importing Options 2545
F.31.2. Connection Management 2546
F.31.3. Transaction Management 2546
F.31.4. Remote Query Optimization 2546
F.31.5. Remote Query Execution Environment 2546
F.31.6. Cross-Version Compatibility 2547
F.31.7. Examples 2547
F.31.8. Author 2548
F.32. seg 2548
F.32.1. Rationale 2548
F.32.2. Syntax 2549
F.32.3. Precision 2550
F.32.4. Usage 2550
F.32.5. Notes 2551
F.32.6. Credits 2551
F.33. sepgsql 2551
F.33.1. Overview 2552
F.33.2. Installation 2552
F.33.3. Regression Tests 2553
F.33.4. GUC Parameters 2554
F.33.5. Features 2554
F.33.5.1. Controlled Object Classes 2555
F.33.5.2. DML Permissions 2555
F.33.5.3. DDL Permissions 2556
F.33.5.4. Trusted Procedures 2556
F.33.5.5. Dynamic Domain Transitions 2557
F.33.5.6. Miscellaneous 2558
F.33.6. Sepgsql Functions 2558
F.33.7. Limitations 2559
F.33.8. External Resources 2559
F.33.9. Author 2560
F.34. spi 2560
F.34.1. refint — Functions for Implementing Referential Integrity 2560
F.34.2. timetravel — Functions for Implementing Time Travel 2560
F.34.3. autoinc — Functions for Autoincrementing Fields 2561
F.34.4. insert_username — Functions for Tracking Who Changed a Table 2562
F.34.5. moddatetime — Functions for Tracking Last Modification Time 2562
F.35. sslinfo 2562
F.35.1. Functions Provided 2562
F.35.2. Author 2564
F.36. tablefunc 2564
F.36.1. Functions Provided 2564
F.36.1.1. `normal_rand` 2565
F.36.1.2. `crosstab(text)` 2566
F.36.1.3. `crosstab`*N*`(text)` 2568

F.36.1.4. crosstab(text, text) 2569
F.36.1.5. connectby 2572
F.36.2. Author 2574
F.37. tcn 2574
F.38. test_decoding 2575
F.39. tsearch2 2576
F.39.1. Portability Issues 2576
F.39.2. Converting a pre-8.3 Installation 2577
F.39.3. References 2577
F.40. tsm_system_rows 2577
F.40.1. Examples 2578
F.41. tsm_system_time 2578
F.41.1. Examples 2578
F.42. unaccent 2579
F.42.1. Configuration 2579
F.42.2. Usage 2580
F.42.3. Functions 2581
F.43. uuid-ossp 2581
F.43.1. uuid-ossp Functions 2581
F.43.2. Building uuid-ossp 2582
F.43.3. Author 2583
F.44. xml2 2583
F.44.1. Deprecation Notice 2583
F.44.2. Description of Functions 2583
F.44.3. xpath_table 2585
F.44.3.1. Multivalued Results 2586
F.44.4. XSLT Functions 2587
F.44.4.1. xslt_process 2587
F.44.5. Author 2588
G. Additional Supplied Programs 2589
G.1. Client Applications 2589
oid2name 2589
vacuumlo 2594
G.2. Server Applications 2596
pg_standby 2596
H. External Projects 2600
H.1. Client Interfaces 2600
H.2. Administration Tools 2600
H.3. Procedural Languages 2601
H.4. Extensions 2601
I. The Source Code Repository 2602
I.1. Getting The Source via Git 2602
J. Documentation 2603
J.1. DocBook 2603
J.2. Tool Sets 2603
J.2.1. Installation on Fedora, RHEL, and Derivatives 2604
J.2.2. Installation on FreeBSD 2605
J.2.3. Debian Packages 2605

J.2.4. OS X 2605
J.2.5. Manual Installation from Source 2605
J.2.5.1. Installing OpenJade 2606
J.2.5.2. Installing the DocBook DTD Kit 2606
J.2.5.3. Installing the DocBook DSSSL Style Sheets 2607
J.2.5.4. Installing JadeTeX 2607
J.2.6. Detection by `configure` 2608
J.3. Building The Documentation 2608
J.3.1. HTML 2608
J.3.2. Manpages 2609
J.3.3. Print Output via JadeTeX 2609
J.3.4. Overflow Text 2610
J.3.5. Print Output via RTF 2610
J.3.6. Plain Text Files 2611
J.3.7. Syntax Check 2612
J.4. Documentation Authoring 2612
J.4.1. Emacs/PSGML 2612
J.4.2. Other Emacs Modes 2613
J.5. Style Guide 2613
J.5.1. Reference Pages 2613
K. Acronyms 2616
Bibliography 2622
Index 2625

Preface

This book is the official documentation of PostgreSQL. It has been written by the PostgreSQL developers and other volunteers in parallel to the development of the PostgreSQL software. It describes all the functionality that the current version of PostgreSQL officially supports.

To make the large amount of information about PostgreSQL manageable, this book has been organized in several parts. Each part is targeted at a different class of users, or at users in different stages of their PostgreSQL experience:

- Part I is an informal introduction for new users.
- Part II documents the SQL query language environment, including data types and functions, as well as user-level performance tuning. Every PostgreSQL user should read this.
- Part III describes the installation and administration of the server. Everyone who runs a PostgreSQL server, be it for private use or for others, should read this part.
- Part IV describes the programming interfaces for PostgreSQL client programs.
- Part V contains information for advanced users about the extensibility capabilities of the server. Topics include user-defined data types and functions.
- Part VI contains reference information about SQL commands, client and server programs. This part supports the other parts with structured information sorted by command or program.
- Part VII contains assorted information that might be of use to PostgreSQL developers.

1. What is PostgreSQL?

PostgreSQL is an object-relational database management system (ORDBMS) based on POSTGRES, Version 4.2[1], developed at the University of California at Berkeley Computer Science Department. POSTGRES pioneered many concepts that only became available in some commercial database systems much later.

PostgreSQL is an open-source descendant of this original Berkeley code. It supports a large part of the SQL standard and offers many modern features:

- complex queries
- foreign keys
- triggers
- updatable views
- transactional integrity
- multiversion concurrency control

Also, PostgreSQL can be extended by the user in many ways, for example by adding new

- data types

1. https://dsf.berkeley.edu/postgres.html

- functions
- operators
- aggregate functions
- index methods
- procedural languages

And because of the liberal license, PostgreSQL can be used, modified, and distributed by anyone free of charge for any purpose, be it private, commercial, or academic.

2. A Brief History of PostgreSQL

The object-relational database management system now known as PostgreSQL is derived from the POSTGRES package written at the University of California at Berkeley. With over two decades of development behind it, PostgreSQL is now the most advanced open-source database available anywhere.

2.1. The Berkeley POSTGRES Project

The POSTGRES project, led by Professor Michael Stonebraker, was sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc. The implementation of POSTGRES began in 1986. The initial concepts for the system were presented in *The design of POSTGRES* , and the definition of the initial data model appeared in *The POSTGRES data model* . The design of the rule system at that time was described in *The design of the POSTGRES rules system*. The rationale and architecture of the storage manager were detailed in *The design of the POSTGRES storage system* .

POSTGRES has undergone several major releases since then. The first "demoware" system became operational in 1987 and was shown at the 1988 ACM-SIGMOD Conference. Version 1, described in *The implementation of POSTGRES* , was released to a few external users in June 1989. In response to a critique of the first rule system (*A commentary on the POSTGRES rules system*), the rule system was redesigned (*On Rules, Procedures, Caching and Views in Database Systems*), and Version 2 was released in June 1990 with the new rule system. Version 3 appeared in 1991 and added support for multiple storage managers, an improved query executor, and a rewritten rule system. For the most part, subsequent releases until Postgres95 (see below) focused on portability and reliability.

POSTGRES has been used to implement many different research and production applications. These include: a financial data analysis system, a jet engine performance monitoring package, an asteroid tracking database, a medical information database, and several geographic information systems. POSTGRES has also been used as an educational tool at several universities. Finally, Illustra Information Technologies (later merged into Informix[2], which is now owned by IBM[3]) picked up the code and commercialized it. In late 1992, POSTGRES became the primary data manager for the Sequoia 2000 scientific computing project[4].

The size of the external user community nearly doubled during 1993. It became increasingly obvious that maintenance of the prototype code and support was taking up large amounts of time that should have been

2. http://www.informix.com/
3. http://www.ibm.com/
4. http://meteora.ucsd.edu/s2k/s2k_home.html

devoted to database research. In an effort to reduce this support burden, the Berkeley POSTGRES project officially ended with Version 4.2.

2.2. Postgres95

In 1994, Andrew Yu and Jolly Chen added an SQL language interpreter to POSTGRES. Under a new name, Postgres95 was subsequently released to the web to find its own way in the world as an open-source descendant of the original POSTGRES Berkeley code.

Postgres95 code was completely ANSI C and trimmed in size by 25%. Many internal changes improved performance and maintainability. Postgres95 release 1.0.x ran about 30-50% faster on the Wisconsin Benchmark compared to POSTGRES, Version 4.2. Apart from bug fixes, the following were the major enhancements:

- The query language PostQUEL was replaced with SQL (implemented in the server). (Interface library libpq was named after PostQUEL.) Subqueries were not supported until PostgreSQL (see below), but they could be imitated in Postgres95 with user-defined SQL functions. Aggregate functions were re-implemented. Support for the `GROUP BY` query clause was also added.
- A new program (psql) was provided for interactive SQL queries, which used GNU Readline. This largely superseded the old monitor program.
- A new front-end library, `libpgtcl`, supported Tcl-based clients. A sample shell, `pgtclsh`, provided new Tcl commands to interface Tcl programs with the Postgres95 server.
- The large-object interface was overhauled. The inversion large objects were the only mechanism for storing large objects. (The inversion file system was removed.)
- The instance-level rule system was removed. Rules were still available as rewrite rules.
- A short tutorial introducing regular SQL features as well as those of Postgres95 was distributed with the source code
- GNU make (instead of BSD make) was used for the build. Also, Postgres95 could be compiled with an unpatched GCC (data alignment of doubles was fixed).

2.3. PostgreSQL

By 1996, it became clear that the name “Postgres95” would not stand the test of time. We chose a new name, PostgreSQL, to reflect the relationship between the original POSTGRES and the more recent versions with SQL capability. At the same time, we set the version numbering to start at 6.0, putting the numbers back into the sequence originally begun by the Berkeley POSTGRES project.

Many people continue to refer to PostgreSQL as “Postgres” (now rarely in all capital letters) because of tradition or because it is easier to pronounce. This usage is widely accepted as a nickname or alias.

The emphasis during development of Postgres95 was on identifying and understanding existing problems in the server code. With PostgreSQL, the emphasis has shifted to augmenting features and capabilities, although work continues in all areas.

Details about what has happened in PostgreSQL since then can be found in Appendix E.

3. Conventions

The following conventions are used in the synopsis of a command: brackets (`[` and `]`) indicate optional parts. (In the synopsis of a Tcl command, question marks (`?`) are used instead, as is usual in Tcl.) Braces (`{` and `}`) and vertical lines (`|`) indicate that you must choose one alternative. Dots (`...`) mean that the preceding element can be repeated.

Where it enhances the clarity, SQL commands are preceded by the prompt =>, and shell commands are preceded by the prompt `$`. Normally, prompts are not shown, though.

An *administrator* is generally a person who is in charge of installing and running the server. A *user* could be anyone who is using, or wants to use, any part of the PostgreSQL system. These terms should not be interpreted too narrowly; this book does not have fixed presumptions about system administration procedures.

4. Further Information

Besides the documentation, that is, this book, there are other resources about PostgreSQL:

Wiki

The PostgreSQL wiki[5] contains the project's FAQ[6] (Frequently Asked Questions) list, TODO[7] list, and detailed information about many more topics.

Web Site

The PostgreSQL web site[8] carries details on the latest release and other information to make your work or play with PostgreSQL more productive.

Mailing Lists

The mailing lists are a good place to have your questions answered, to share experiences with other users, and to contact the developers. Consult the PostgreSQL web site for details.

Yourself!

PostgreSQL is an open-source project. As such, it depends on the user community for ongoing support. As you begin to use PostgreSQL, you will rely on others for help, either through the documentation or through the mailing lists. Consider contributing your knowledge back. Read the mailing lists and answer questions. If you learn something which is not in the documentation, write it up and contribute it. If you add features to the code, contribute them.

5. https://wiki.postgresql.org
6. https://wiki.postgresql.org/wiki/Frequently_Asked_Questions
7. https://wiki.postgresql.org/wiki/Todo
8. https://www.postgresql.org

5. Bug Reporting Guidelines

When you find a bug in PostgreSQL we want to hear about it. Your bug reports play an important part in making PostgreSQL more reliable because even the utmost care cannot guarantee that every part of PostgreSQL will work on every platform under every circumstance.

The following suggestions are intended to assist you in forming bug reports that can be handled in an effective fashion. No one is required to follow them but doing so tends to be to everyone's advantage.

We cannot promise to fix every bug right away. If the bug is obvious, critical, or affects a lot of users, chances are good that someone will look into it. It could also happen that we tell you to update to a newer version to see if the bug happens there. Or we might decide that the bug cannot be fixed before some major rewrite we might be planning is done. Or perhaps it is simply too hard and there are more important things on the agenda. If you need help immediately, consider obtaining a commercial support contract.

5.1. Identifying Bugs

Before you report a bug, please read and re-read the documentation to verify that you can really do whatever it is you are trying. If it is not clear from the documentation whether you can do something or not, please report that too; it is a bug in the documentation. If it turns out that a program does something different from what the documentation says, that is a bug. That might include, but is not limited to, the following circumstances:

- A program terminates with a fatal signal or an operating system error message that would point to a problem in the program. (A counterexample might be a "disk full" message, since you have to fix that yourself.)
- A program produces the wrong output for any given input.
- A program refuses to accept valid input (as defined in the documentation).
- A program accepts invalid input without a notice or error message. But keep in mind that your idea of invalid input might be our idea of an extension or compatibility with traditional practice.
- PostgreSQL fails to compile, build, or install according to the instructions on supported platforms.

Here "program" refers to any executable, not only the backend process.

Being slow or resource-hogging is not necessarily a bug. Read the documentation or ask on one of the mailing lists for help in tuning your applications. Failing to comply to the SQL standard is not necessarily a bug either, unless compliance for the specific feature is explicitly claimed.

Before you continue, check on the TODO list and in the FAQ to see if your bug is already known. If you cannot decode the information on the TODO list, report your problem. The least we can do is make the TODO list clearer.

5.2. What to Report

The most important thing to remember about bug reporting is to state all the facts and only facts. Do not speculate what you think went wrong, what "it seemed to do", or which part of the program has a fault. If you are not familiar with the implementation you would probably guess wrong and not help us a bit.

And even if you are, educated explanations are a great supplement to but no substitute for facts. If we are going to fix the bug we still have to see it happen for ourselves first. Reporting the bare facts is relatively straightforward (you can probably copy and paste them from the screen) but all too often important details are left out because someone thought it does not matter or the report would be understood anyway.

The following items should be contained in every bug report:

- The exact sequence of steps *from program start-up* necessary to reproduce the problem. This should be self-contained; it is not enough to send in a bare `SELECT` statement without the preceding `CREATE TABLE` and `INSERT` statements, if the output should depend on the data in the tables. We do not have the time to reverse-engineer your database schema, and if we are supposed to make up our own data we would probably miss the problem.

  The best format for a test case for SQL-related problems is a file that can be run through the psql frontend that shows the problem. (Be sure to not have anything in your `~/.psqlrc` start-up file.) An easy way to create this file is to use pg_dump to dump out the table declarations and data needed to set the scene, then add the problem query. You are encouraged to minimize the size of your example, but this is not absolutely necessary. If the bug is reproducible, we will find it either way.

  If your application uses some other client interface, such as PHP, then please try to isolate the offending queries. We will probably not set up a web server to reproduce your problem. In any case remember to provide the exact input files; do not guess that the problem happens for "large files" or "midsize databases", etc. since this information is too inexact to be of use.

- The output you got. Please do not say that it "didn't work" or "crashed". If there is an error message, show it, even if you do not understand it. If the program terminates with an operating system error, say which. If nothing at all happens, say so. Even if the result of your test case is a program crash or otherwise obvious it might not happen on our platform. The easiest thing is to copy the output from the terminal, if possible.

  > **Note:** If you are reporting an error message, please obtain the most verbose form of the message. In psql, say `\set VERBOSITY verbose` beforehand. If you are extracting the message from the server log, set the run-time parameter log_error_verbosity to `verbose` so that all details are logged.

  > **Note:** In case of fatal errors, the error message reported by the client might not contain all the information available. Please also look at the log output of the database server. If you do not keep your server's log output, this would be a good time to start doing so.

- The output you expected is very important to state. If you just write "This command gives me that output." or "This is not what I expected.", we might run it ourselves, scan the output, and think it looks OK and is exactly what we expected. We should not have to spend the time to decode the exact semantics behind your commands. Especially refrain from merely saying that "This is not what SQL says/Oracle does." Digging out the correct behavior from SQL is not a fun undertaking, nor do we all know how all the other relational databases out there behave. (If your problem is a program crash, you can obviously omit this item.)

- Any command line options and other start-up options, including any relevant environment variables or configuration files that you changed from the default. Again, please provide exact information. If you are using a prepackaged distribution that starts the database server at boot time, you should try to find out how that is done.
- Anything you did at all differently from the installation instructions.
- The PostgreSQL version. You can run the command `SELECT version();` to find out the version of the server you are connected to. Most executable programs also support a `--version` option; at least `postgres --version` and `psql --version` should work. If the function or the options do not exist then your version is more than old enough to warrant an upgrade. If you run a prepackaged version, such as RPMs, say so, including any subversion the package might have. If you are talking about a Git snapshot, mention that, including the commit hash.

  If your version is older than 9.5.24 we will almost certainly tell you to upgrade. There are many bug fixes and improvements in each new release, so it is quite possible that a bug you have encountered in an older release of PostgreSQL has already been fixed. We can only provide limited support for sites using older releases of PostgreSQL; if you require more than we can provide, consider acquiring a commercial support contract.

- Platform information. This includes the kernel name and version, C library, processor, memory information, and so on. In most cases it is sufficient to report the vendor and version, but do not assume everyone knows what exactly "Debian" contains or that everyone runs on x86_64. If you have installation problems then information about the toolchain on your machine (compiler, make, and so on) is also necessary.

Do not be afraid if your bug report becomes rather lengthy. That is a fact of life. It is better to report everything the first time than us having to squeeze the facts out of you. On the other hand, if your input files are huge, it is fair to ask first whether somebody is interested in looking into it. Here is an article[9] that outlines some more tips on reporting bugs.

Do not spend all your time to figure out which changes in the input make the problem go away. This will probably not help solving it. If it turns out that the bug cannot be fixed right away, you will still have time to find and share your work-around. Also, once again, do not waste your time guessing why the bug exists. We will find that out soon enough.

When writing a bug report, please avoid confusing terminology. The software package in total is called "PostgreSQL", sometimes "Postgres" for short. If you are specifically talking about the backend process, mention that, do not just say "PostgreSQL crashes". A crash of a single backend process is quite different from crash of the parent "postgres" process; please don't say "the server crashed" when you mean a single backend process went down, nor vice versa. Also, client programs such as the interactive frontend "psql" are completely separate from the backend. Please try to be specific about whether the problem is on the client or server side.

5.3. Where to Report Bugs

In general, send bug reports to the bug report mailing list at <pgsql-bugs@lists.postgresql.org>. You are requested to use a descriptive subject for your email message, perhaps parts of the error message.

9. http://www.chiark.greenend.org.uk/~sgtatham/bugs.html

Another method is to fill in the bug report web-form available at the project's web site[10]. Entering a bug report this way causes it to be mailed to the <pgsql-bugs@lists.postgresql.org> mailing list.

If your bug report has security implications and you'd prefer that it not become immediately visible in public archives, don't send it to pgsql-bugs. Security issues can be reported privately to <security@postgresql.org>.

Do not send bug reports to any of the user mailing lists, such as <pgsql-sql@lists.postgresql.org> or <pgsql-general@lists.postgresql.org>. These mailing lists are for answering user questions, and their subscribers normally do not wish to receive bug reports. More importantly, they are unlikely to fix them.

Also, please do *not* send reports to the developers' mailing list <pgsql-hackers@lists.postgresql.org>. This list is for discussing the development of PostgreSQL, and it would be nice if we could keep the bug reports separate. We might choose to take up a discussion about your bug report on pgsql-hackers, if the problem needs more review.

If you have a problem with the documentation, the best place to report it is the documentation mailing list <pgsql-docs@lists.postgresql.org>. Please be specific about what part of the documentation you are unhappy with.

If your bug is a portability problem on a non-supported platform, send mail to <pgsql-hackers@lists.postgresql.org>, so we (and you) can work on porting PostgreSQL to your platform.

> **Note:** Due to the unfortunate amount of spam going around, all of the above lists will be moderated unless you are subscribed. That means there will be some delay before the email is delivered. If you wish to subscribe to the lists, please visit https://lists.postgresql.org/ for instructions.

10. https://www.postgresql.org/

I. Tutorial

Welcome to the PostgreSQL Tutorial. The following few chapters are intended to give a simple introduction to PostgreSQL, relational database concepts, and the SQL language to those who are new to any one of these aspects. We only assume some general knowledge about how to use computers. No particular Unix or programming experience is required. This part is mainly intended to give you some hands-on experience with important aspects of the PostgreSQL system. It makes no attempt to be a complete or thorough treatment of the topics it covers.

After you have worked through this tutorial you might want to move on to reading Part II to gain a more formal knowledge of the SQL language, or Part IV for information about developing applications for PostgreSQL. Those who set up and manage their own server should also read Part III.

Chapter 1. Getting Started

1.1. Installation

Before you can use PostgreSQL you need to install it, of course. It is possible that PostgreSQL is already installed at your site, either because it was included in your operating system distribution or because the system administrator already installed it. If that is the case, you should obtain information from the operating system documentation or your system administrator about how to access PostgreSQL.

If you are not sure whether PostgreSQL is already available or whether you can use it for your experimentation then you can install it yourself. Doing so is not hard and it can be a good exercise. PostgreSQL can be installed by any unprivileged user; no superuser (root) access is required.

If you are installing PostgreSQL yourself, then refer to Chapter 15 for instructions on installation, and return to this guide when the installation is complete. Be sure to follow closely the section about setting up the appropriate environment variables.

If your site administrator has not set things up in the default way, you might have some more work to do. For example, if the database server machine is a remote machine, you will need to set the `PGHOST` environment variable to the name of the database server machine. The environment variable `PGPORT` might also have to be set. The bottom line is this: if you try to start an application program and it complains that it cannot connect to the database, you should consult your site administrator or, if that is you, the documentation to make sure that your environment is properly set up. If you did not understand the preceding paragraph then read the next section.

1.2. Architectural Fundamentals

Before we proceed, you should understand the basic PostgreSQL system architecture. Understanding how the parts of PostgreSQL interact will make this chapter somewhat clearer.

In database jargon, PostgreSQL uses a client/server model. A PostgreSQL session consists of the following cooperating processes (programs):

- A server process, which manages the database files, accepts connections to the database from client applications, and performs database actions on behalf of the clients. The database server program is called `postgres`.
- The user's client (frontend) application that wants to perform database operations. Client applications can be very diverse in nature: a client could be a text-oriented tool, a graphical application, a web server that accesses the database to display web pages, or a specialized database maintenance tool. Some client applications are supplied with the PostgreSQL distribution; most are developed by users.

As is typical of client/server applications, the client and the server can be on different hosts. In that case they communicate over a TCP/IP network connection. You should keep this in mind, because the files that can be accessed on a client machine might not be accessible (or might only be accessible using a different file name) on the database server machine.

The PostgreSQL server can handle multiple concurrent connections from clients. To achieve this it starts ("forks") a new process for each connection. From that point on, the client and the new server process communicate without intervention by the original `postgres` process. Thus, the master server process is always running, waiting for client connections, whereas client and associated server processes come and go. (All of this is of course invisible to the user. We only mention it here for completeness.)

1.3. Creating a Database

The first test to see whether you can access the database server is to try to create a database. A running PostgreSQL server can manage many databases. Typically, a separate database is used for each project or for each user.

Possibly, your site administrator has already created a database for your use. He should have told you what the name of your database is. In that case you can omit this step and skip ahead to the next section.

To create a new database, in this example named `mydb`, you use the following command:

```
$ createdb mydb
```

If this produces no response then this step was successful and you can skip over the remainder of this section.

If you see a message similar to:

```
createdb: command not found
```

then PostgreSQL was not installed properly. Either it was not installed at all or your shell's search path was not set to include it. Try calling the command with an absolute path instead:

```
$ /usr/local/pgsql/bin/createdb mydb
```

The path at your site might be different. Contact your site administrator or check the installation instructions to correct the situation.

Another response could be this:

```
createdb: could not connect to database postgres: could not connect to server: No such file
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

This means that the server was not started, or it was not started where `createdb` expected it. Again, check the installation instructions or consult the administrator.

Another response could be this:

```
createdb: could not connect to database postgres: FATAL:  role "joe" does not exist
```

where your own login name is mentioned. This will happen if the administrator has not created a PostgreSQL user account for you. (PostgreSQL user accounts are distinct from operating system user accounts.) If you are the administrator, see Chapter 20 for help creating accounts. You will need to become the operating system user under which PostgreSQL was installed (usually `postgres`) to create the first user account. It could also be that you were assigned a PostgreSQL user name that is different from your

operating system user name; in that case you need to use the `-U` switch or set the `PGUSER` environment variable to specify your PostgreSQL user name.

If you have a user account but it does not have the privileges required to create a database, you will see the following:

```
createdb: database creation failed: ERROR:  permission denied to create database
```

Not every user has authorization to create new databases. If PostgreSQL refuses to create databases for you then the site administrator needs to grant you permission to create databases. Consult your site administrator if this occurs. If you installed PostgreSQL yourself then you should log in for the purposes of this tutorial under the user account that you started the server as. [1]

You can also create databases with other names. PostgreSQL allows you to create any number of databases at a given site. Database names must have an alphabetic first character and are limited to 63 bytes in length. A convenient choice is to create a database with the same name as your current user name. Many tools assume that database name as the default, so it can save you some typing. To create that database, simply type:

```
$ createdb
```

If you do not want to use your database anymore you can remove it. For example, if you are the owner (creator) of the database `mydb`, you can destroy it using the following command:

```
$ dropdb mydb
```

(For this command, the database name does not default to the user account name. You always need to specify it.) This action physically removes all files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

More about `createdb` and `dropdb` can be found in createdb and dropdb respectively.

1.4. Accessing a Database

Once you have created a database, you can access it by:

- Running the PostgreSQL interactive terminal program, called *psql*, which allows you to interactively enter, edit, and execute SQL commands.
- Using an existing graphical frontend tool like pgAdmin or an office suite with ODBC or JDBC support to create and manipulate a database. These possibilities are not covered in this tutorial.
- Writing a custom application, using one of the several available language bindings. These possibilities are discussed further in Part IV.

1. As an explanation for why this works: PostgreSQL user names are separate from operating system user accounts. When you connect to a database, you can choose what PostgreSQL user name to connect as; if you don't, it will default to the same name as your current operating system account. As it happens, there will always be a PostgreSQL user account that has the same name as the operating system user that started the server, and it also happens that that user always has permission to create databases. Instead of logging in as that user you can also specify the `-U` option everywhere to select a PostgreSQL user name to connect as.

You probably want to start up `psql` to try the examples in this tutorial. It can be activated for the `mydb` database by typing the command:

```
$ psql mydb
```

If you do not supply the database name then it will default to your user account name. You already discovered this scheme in the previous section using `createdb`.

In `psql`, you will be greeted with the following message:

```
psql (9.5.24)
Type "help" for help.

mydb=>
```

The last line could also be:

```
mydb=#
```

That would mean you are a database superuser, which is most likely the case if you installed the PostgreSQL instance yourself. Being a superuser means that you are not subject to access controls. For the purposes of this tutorial that is not important.

If you encounter problems starting `psql` then go back to the previous section. The diagnostics of `createdb` and `psql` are similar, and if the former worked the latter should work as well.

The last line printed out by `psql` is the prompt, and it indicates that `psql` is listening to you and that you can type SQL queries into a work space maintained by `psql`. Try out these commands:

```
mydb=> SELECT version();
                                version
 ---------------------------------------------------------------------
 PostgreSQL 9.5.24 on i586-pc-linux-gnu, compiled by GCC 2.96, 32-bit
(1 row)

mydb=> SELECT current_date;
    date
------------
 2002-08-31
(1 row)

mydb=> SELECT 2 + 2;
 ?column?
----------
        4
(1 row)
```

The `psql` program has a number of internal commands that are not SQL commands. They begin with the backslash character, "\". For example, you can get help on the syntax of various PostgreSQL SQL commands by typing:

```
mydb=> \h
```

To get out of `psql`, type:

```
mydb=> \q
```

and `psql` will quit and return you to your command shell. (For more internal commands, type `\?` at the `psql` prompt.) The full capabilities of `psql` are documented in psql. In this tutorial we will not use these features explicitly, but you can use them yourself when it is helpful.

Chapter 2. The SQL Language

2.1. Introduction

This chapter provides an overview of how to use SQL to perform simple operations. This tutorial is only intended to give you an introduction and is in no way a complete tutorial on SQL. Numerous books have been written on SQL, including *Understanding the New SQL* and *A Guide to the SQL Standard*. You should be aware that some PostgreSQL language features are extensions to the standard.

In the examples that follow, we assume that you have created a database named `mydb`, as described in the previous chapter, and have been able to start psql.

Examples in this manual can also be found in the PostgreSQL source distribution in the directory `src/tutorial/`. (Binary distributions of PostgreSQL might not compile these files.) To use those files, first change to that directory and run make:

```
$ cd ..../src/tutorial
$ make
```

This creates the scripts and compiles the C files containing user-defined functions and types. Then, to start the tutorial, do the following:

```
$ cd ..../tutorial
$ psql -s mydb
...

mydb=> \i basics.sql
```

The `\i` command reads in commands from the specified file. `psql`'s `-s` option puts you in single step mode which pauses before sending each statement to the server. The commands used in this section are in the file `basics.sql`.

2.2. Concepts

PostgreSQL is a *relational database management system* (RDBMS). That means it is a system for managing data stored in *relations*. Relation is essentially a mathematical term for *table*. The notion of storing data in tables is so commonplace today that it might seem inherently obvious, but there are a number of other ways of organizing databases. Files and directories on Unix-like operating systems form an example of a hierarchical database. A more modern development is the object-oriented database.

Each table is a named collection of *rows*. Each row of a given table has the same set of named *columns*, and each column is of a specific data type. Whereas columns have a fixed order in each row, it is important to remember that SQL does not guarantee the order of the rows within the table in any way (although they can be explicitly sorted for display).

Tables are grouped into databases, and a collection of databases managed by a single PostgreSQL server instance constitutes a database *cluster*.

2.3. Creating a New Table

You can create a new table by specifying the table name, along with all column names and their types:

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- low temperature
    temp_hi         int,           -- high temperature
    prcp            real,          -- precipitation
    date            date
);
```

You can enter this into `psql` with the line breaks. `psql` will recognize that the command is not terminated until the semicolon.

White space (i.e., spaces, tabs, and newlines) can be used freely in SQL commands. That means you can type the command aligned differently than above, or even all on one line. Two dashes ("`--`") introduce comments. Whatever follows them is ignored up to the end of the line. SQL is case insensitive about key words and identifiers, except when identifiers are double-quoted to preserve the case (not done above).

`varchar(80)` specifies a data type that can store arbitrary character strings up to 80 characters in length. `int` is the normal integer type. `real` is a type for storing single precision floating-point numbers. `date` should be self-explanatory. (Yes, the column of type `date` is also named `date`. This might be convenient or confusing — you choose.)

PostgreSQL supports the standard SQL types `int`, `smallint`, `real`, `double precision`, `char(`*N*`)`, `varchar(`*N*`)`, `date`, `time`, `timestamp`, and `interval`, as well as other types of general utility and a rich set of geometric types. PostgreSQL can be customized with an arbitrary number of user-defined data types. Consequently, type names are not key words in the syntax, except where required to support special cases in the SQL standard.

The second example will store cities and their associated geographical location:

```
CREATE TABLE cities (
    name            varchar(80),
    location        point
);
```

The `point` type is an example of a PostgreSQL-specific data type.

Finally, it should be mentioned that if you don't need a table any longer or want to recreate it differently you can remove it using the following command:

```
DROP TABLE tablename;
```

2.4. Populating a Table With Rows

The `INSERT` statement is used to populate a table with rows:

```
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

Note that all data types use rather obvious input formats. Constants that are not simple numeric values usually must be surrounded by single quotes (’), as in the example. The date type is actually quite flexible in what it accepts, but for this tutorial we will stick to the unambiguous format shown here.

The point type requires a coordinate pair as input, as shown here:

```
INSERT INTO cities VALUES (’San Francisco’, ’(-194.0, 53.0)’);
```

The syntax used so far requires you to remember the order of the columns. An alternative syntax allows you to list the columns explicitly:

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
    VALUES (’San Francisco’, 43, 57, 0.0, ’1994-11-29’);
```

You can list the columns in a different order if you wish or even omit some columns, e.g., if the precipitation is unknown:

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
    VALUES (’1994-11-29’, ’Hayward’, 54, 37);
```

Many developers consider explicitly listing the columns better style than relying on the order implicitly.

Please enter all the commands shown above so you have some data to work with in the following sections.

You could also have used COPY to load large amounts of data from flat-text files. This is usually faster because the COPY command is optimized for this application while allowing less flexibility than INSERT. An example would be:

```
COPY weather FROM ’/home/user/weather.txt’;
```

where the file name for the source file must be available on the machine running the backend process, not the client, since the backend process reads the file directly. You can read more about the COPY command in COPY.

2.5. Querying a Table

To retrieve data from a table, the table is *queried*. An SQL SELECT statement is used to do this. The statement is divided into a select list (the part that lists the columns to be returned), a table list (the part that lists the tables from which to retrieve the data), and an optional qualification (the part that specifies any restrictions). For example, to retrieve all the rows of table weather, type:

```
SELECT * FROM weather;
```

Here * is a shorthand for “all columns”. [1] So the same result would be had with:

```
SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
```

The output should be:

1. While SELECT * is useful for off-the-cuff queries, it is widely considered bad style in production code, since adding a column to the table would change the results.

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      43 |      57 |    0 | 1994-11-29
 Hayward       |      37 |      54 |      | 1994-11-29
(3 rows)
```

You can write expressions, not just simple column references, in the select list. For example, you can do:

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

This should give:

```
     city      | temp_avg |    date
---------------+----------+------------
 San Francisco |       48 | 1994-11-27
 San Francisco |       50 | 1994-11-29
 Hayward       |       45 | 1994-11-29
(3 rows)
```

Notice how the `AS` clause is used to relabel the output column. (The `AS` clause is optional.)

A query can be "qualified" by adding a `WHERE` clause that specifies which rows are wanted. The `WHERE` clause contains a Boolean (truth value) expression, and only rows for which the Boolean expression is true are returned. The usual Boolean operators (`AND`, `OR`, and `NOT`) are allowed in the qualification. For example, the following retrieves the weather of San Francisco on rainy days:

```
SELECT * FROM weather
    WHERE city = 'San Francisco' AND prcp > 0.0;
```

Result:

```
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
(1 row)
```

You can request that the results of a query be returned in sorted order:

```
SELECT * FROM weather
    ORDER BY city;

     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 Hayward       |      37 |      54 |      | 1994-11-29
 San Francisco |      43 |      57 |    0 | 1994-11-29
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
```

In this example, the sort order isn't fully specified, and so you might get the San Francisco rows in either order. But you'd always get the results shown above if you do:

```
SELECT * FROM weather
    ORDER BY city, temp_lo;
```

You can request that duplicate rows be removed from the result of a query:

```
SELECT DISTINCT city
    FROM weather;

     city
---------------
 Hayward
 San Francisco
(2 rows)
```

Here again, the result row ordering might vary. You can ensure consistent results by using `DISTINCT` and `ORDER BY` together: [2]

```
SELECT DISTINCT city
    FROM weather
    ORDER BY city;
```

2.6. Joins Between Tables

Thus far, our queries have only accessed one table at a time. Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being processed at the same time. A query that accesses multiple rows of the same or different tables at one time is called a *join* query. As an example, say you wish to list all the weather records together with the location of the associated city. To do that, we need to compare the `city` column of each row of the `weather` table with the `name` column of all rows in the `cities` table, and select the pairs of rows where these values match.

> **Note:** This is only a conceptual model. The join is usually performed in a more efficient manner than actually comparing each possible pair of rows, but this is invisible to the user.

This would be accomplished by the following query:

```
SELECT *
    FROM weather, cities
    WHERE city = name;

     city      | temp_lo | temp_hi | prcp |    date    |     name      | location
---------------+---------+---------+------+------------+---------------+-----------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(2 rows)
```

2. In some database systems, including older versions of PostgreSQL, the implementation of `DISTINCT` automatically orders the rows and so `ORDER BY` is unnecessary. But this is not required by the SQL standard, and current PostgreSQL does not guarantee that `DISTINCT` causes the rows to be ordered.

Observe two things about the result set:

- There is no result row for the city of Hayward. This is because there is no matching entry in the `cities` table for Hayward, so the join ignores the unmatched rows in the `weather` table. We will see shortly how this can be fixed.
- There are two columns containing the city name. This is correct because the lists of columns from the `weather` and `cities` tables are concatenated. In practice this is undesirable, though, so you will probably want to list the output columns explicitly rather than using `*`:

```
SELECT city, temp_lo, temp_hi, prcp, date, location
    FROM weather, cities
    WHERE city = name;
```

Exercise: Attempt to determine the semantics of this query when the `WHERE` clause is omitted.

Since the columns all had different names, the parser automatically found which table they belong to. If there were duplicate column names in the two tables you'd need to *qualify* the column names to show which one you meant, as in:

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.location
    FROM weather, cities
    WHERE cities.name = weather.city;
```

It is widely considered good style to qualify all column names in a join query, so that the query won't fail if a duplicate column name is later added to one of the tables.

Join queries of the kind seen thus far can also be written in this alternative form:

```
SELECT *
    FROM weather INNER JOIN cities ON (weather.city = cities.name);
```

This syntax is not as commonly used as the one above, but we show it here to help you understand the following topics.

Now we will figure out how we can get the Hayward records back in. What we want the query to do is to scan the `weather` table and for each row to find the matching `cities` row(s). If no matching row is found we want some "empty values" to be substituted for the `cities` table's columns. This kind of query is called an *outer join*. (The joins we have seen so far are inner joins.) The command looks like this:

```
SELECT *
    FROM weather LEFT OUTER JOIN cities ON (weather.city = cities.name);

     city      | temp_lo | temp_hi | prcp |    date    |     name      | location
---------------+---------+---------+------+------------+---------------+-----------
 Hayward       |      37 |      54 |      | 1994-11-29 |               |
 San Francisco |      46 |      50 | 0.25 | 1994-11-27 | San Francisco | (-194,53)
 San Francisco |      43 |      57 |    0 | 1994-11-29 | San Francisco | (-194,53)
(3 rows)
```

This query is called a *left outer join* because the table mentioned on the left of the join operator will have each of its rows in the output at least once, whereas the table on the right will only have those rows output that match some row of the left table. When outputting a left-table row for which there is no right-table match, empty (null) values are substituted for the right-table columns.

Exercise: There are also right outer joins and full outer joins. Try to find out what those do.

We can also join a table against itself. This is called a *self join*. As an example, suppose we wish to find all the weather records that are in the temperature range of other weather records. So we need to compare the `temp_lo` and `temp_hi` columns of each `weather` row to the `temp_lo` and `temp_hi` columns of all other `weather` rows. We can do this with the following query:

```
SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
    W2.city, W2.temp_lo AS low, W2.temp_hi AS high
    FROM weather W1, weather W2
    WHERE W1.temp_lo < W2.temp_lo
    AND W1.temp_hi > W2.temp_hi;

     city      | low | high |     city      | low | high
---------------+-----+------+---------------+-----+------
 San Francisco |  43 |   57 | San Francisco |  46 |   50
 Hayward       |  37 |   54 | San Francisco |  46 |   50
(2 rows)
```

Here we have relabeled the weather table as `W1` and `W2` to be able to distinguish the left and right side of the join. You can also use these kinds of aliases in other queries to save some typing, e.g.:

```
SELECT *
    FROM weather w, cities c
    WHERE w.city = c.name;
```

You will encounter this style of abbreviating quite frequently.

2.7. Aggregate Functions

Like most other relational database products, PostgreSQL supports *aggregate functions*. An aggregate function computes a single result from multiple input rows. For example, there are aggregates to compute the `count`, `sum`, `avg` (average), `max` (maximum) and `min` (minimum) over a set of rows.

As an example, we can find the highest low-temperature reading anywhere with:

```
SELECT max(temp_lo) FROM weather;

 max
-----
  46
(1 row)
```

If we wanted to know what city (or cities) that reading occurred in, we might try:

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);     WRONG
```

but this will not work since the aggregate `max` cannot be used in the `WHERE` clause. (This restriction exists because the `WHERE` clause determines which rows will be included in the aggregate calculation; so obviously it has to be evaluated before aggregate functions are computed.) However, as is often the case the query can be restated to accomplish the desired result, here by using a *subquery*:

```
SELECT city FROM weather
    WHERE temp_lo = (SELECT max(temp_lo) FROM weather);

     city
---------------
 San Francisco
(1 row)
```

This is OK because the subquery is an independent computation that computes its own aggregate separately from what is happening in the outer query.

Aggregates are also very useful in combination with `GROUP BY` clauses. For example, we can get the maximum low temperature observed in each city with:

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city;

     city      | max
---------------+-----
 Hayward       |  37
 San Francisco |  46
(2 rows)
```

which gives us one output row per city. Each aggregate result is computed over the table rows matching that city. We can filter these grouped rows using `HAVING`:

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city
    HAVING max(temp_lo) < 40;

  city   | max
---------+-----
 Hayward |  37
(1 row)
```

which gives us the same results for only the cities that have all `temp_lo` values below 40. Finally, if we only care about cities whose names begin with "`S`", we might do:

```
SELECT city, max(temp_lo)
    FROM weather
    WHERE city LIKE 'S%'❶
    GROUP BY city
    HAVING max(temp_lo) < 40;
```

❶ The LIKE operator does pattern matching and is explained in Section 9.7.

It is important to understand the interaction between aggregates and SQL's WHERE and HAVING clauses. The fundamental difference between WHERE and HAVING is this: WHERE selects input rows before groups and aggregates are computed (thus, it controls which rows go into the aggregate computation), whereas HAVING selects group rows after groups and aggregates are computed. Thus, the WHERE clause must not contain aggregate functions; it makes no sense to try to use an aggregate to determine which rows will be inputs to the aggregates. On the other hand, the HAVING clause always contains aggregate functions. (Strictly speaking, you are allowed to write a HAVING clause that doesn't use aggregates, but it's seldom useful. The same condition could be used more efficiently at the WHERE stage.)

In the previous example, we can apply the city name restriction in WHERE, since it needs no aggregate. This is more efficient than adding the restriction to HAVING, because we avoid doing the grouping and aggregate calculations for all rows that fail the WHERE check.

2.8. Updates

You can update existing rows using the UPDATE command. Suppose you discover the temperature readings are all off by 2 degrees after November 28. You can correct the data as follows:

```
UPDATE weather
    SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
    WHERE date > '1994-11-28';
```

Look at the new state of the data:

```
SELECT * FROM weather;

     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
 Hayward       |      35 |      52 |      | 1994-11-29
(3 rows)
```

2.9. Deletions

Rows can be removed from a table using the DELETE command. Suppose you are no longer interested in the weather of Hayward. Then you can do the following to delete those rows from the table:

```
DELETE FROM weather WHERE city = 'Hayward';
```

All weather records belonging to Hayward are removed.

```
SELECT * FROM weather;

     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      41 |      55 |    0 | 1994-11-29
(2 rows)
```

One should be wary of statements of the form

```
DELETE FROM tablename;
```

Without a qualification, `DELETE` will remove *all* rows from the given table, leaving it empty. The system will not request confirmation before doing this!

Chapter 3. Advanced Features

3.1. Introduction

In the previous chapter we have covered the basics of using SQL to store and access your data in PostgreSQL. We will now discuss some more advanced features of SQL that simplify management and prevent loss or corruption of your data. Finally, we will look at some PostgreSQL extensions.

This chapter will on occasion refer to examples found in Chapter 2 to change or improve them, so it will be useful to have read that chapter. Some examples from this chapter can also be found in `advanced.sql` in the tutorial directory. This file also contains some sample data to load, which is not repeated here. (Refer to Section 2.1 for how to use the file.)

3.2. Views

Refer back to the queries in Section 2.6. Suppose the combined listing of weather records and city location is of particular interest to your application, but you do not want to type the query each time you need it. You can create a *view* over the query, which gives a name to the query that you can refer to like an ordinary table:

```
CREATE VIEW myview AS
    SELECT city, temp_lo, temp_hi, prcp, date, location
        FROM weather, cities
        WHERE city = name;

SELECT * FROM myview;
```

Making liberal use of views is a key aspect of good SQL database design. Views allow you to encapsulate the details of the structure of your tables, which might change as your application evolves, behind consistent interfaces.

Views can be used in almost any place a real table can be used. Building views upon other views is not uncommon.

3.3. Foreign Keys

Recall the `weather` and `cities` tables from Chapter 2. Consider the following problem: You want to make sure that no one can insert rows in the `weather` table that do not have a matching entry in the `cities` table. This is called maintaining the *referential integrity* of your data. In simplistic database systems this would be implemented (if at all) by first looking at the `cities` table to check if a matching record exists, and then inserting or rejecting the new `weather` records. This approach has a number of problems and is very inconvenient, so PostgreSQL can do this for you.

The new declaration of the tables would look like this:

```
CREATE TABLE cities (
        city     varchar(80) primary key,
        location point
);

CREATE TABLE weather (
        city      varchar(80) references cities(city),
        temp_lo   int,
        temp_hi   int,
        prcp      real,
        date      date
);
```

Now try inserting an invalid record:

```
INSERT INTO weather VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');

ERROR:  insert or update on table "weather" violates foreign key constraint "weather_city_f
DETAIL:  Key (city)=(Berkeley) is not present in table "cities".
```

The behavior of foreign keys can be finely tuned to your application. We will not go beyond this simple example in this tutorial, but just refer you to Chapter 5 for more information. Making correct use of foreign keys will definitely improve the quality of your database applications, so you are strongly encouraged to learn about them.

3.4. Transactions

Transactions are a fundamental concept of all database systems. The essential point of a transaction is that it bundles multiple steps into a single, all-or-nothing operation. The intermediate states between the steps are not visible to other concurrent transactions, and if some failure occurs that prevents the transaction from completing, then none of the steps affect the database at all.

For example, consider a bank database that contains balances for various customer accounts, as well as total deposit balances for branches. Suppose that we want to record a payment of $100.00 from Alice's account to Bob's account. Simplifying outrageously, the SQL commands for this might look like:

```
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
UPDATE branches SET balance = balance - 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice');
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
UPDATE branches SET balance = balance + 100.00
    WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
```

The details of these commands are not important here; the important point is that there are several separate updates involved to accomplish this rather simple operation. Our bank's officers will want to be assured that either all these updates happen, or none of them happen. It would certainly not do for a system failure

to result in Bob receiving $100.00 that was not debited from Alice. Nor would Alice long remain a happy customer if she was debited without Bob being credited. We need a guarantee that if something goes wrong partway through the operation, none of the steps executed so far will take effect. Grouping the updates into a *transaction* gives us this guarantee. A transaction is said to be *atomic*: from the point of view of other transactions, it either happens completely or not at all.

We also want a guarantee that once a transaction is completed and acknowledged by the database system, it has indeed been permanently recorded and won't be lost even if a crash ensues shortly thereafter. For example, if we are recording a cash withdrawal by Bob, we do not want any chance that the debit to his account will disappear in a crash just after he walks out the bank door. A transactional database guarantees that all the updates made by a transaction are logged in permanent storage (i.e., on disk) before the transaction is reported complete.

Another important property of transactional databases is closely related to the notion of atomic updates: when multiple transactions are running concurrently, each one should not be able to see the incomplete changes made by others. For example, if one transaction is busy totalling all the branch balances, it would not do for it to include the debit from Alice's branch but not the credit to Bob's branch, nor vice versa. So transactions must be all-or-nothing not only in terms of their permanent effect on the database, but also in terms of their visibility as they happen. The updates made so far by an open transaction are invisible to other transactions until the transaction completes, whereupon all the updates become visible simultaneously.

In PostgreSQL, a transaction is set up by surrounding the SQL commands of the transaction with `BEGIN` and `COMMIT` commands. So our banking transaction would actually look like:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
-- etc etc
COMMIT;
```

If, partway through the transaction, we decide we do not want to commit (perhaps we just noticed that Alice's balance went negative), we can issue the command `ROLLBACK` instead of `COMMIT`, and all our updates so far will be canceled.

PostgreSQL actually treats every SQL statement as being executed within a transaction. If you do not issue a `BEGIN` command, then each individual statement has an implicit `BEGIN` and (if successful) `COMMIT` wrapped around it. A group of statements surrounded by `BEGIN` and `COMMIT` is sometimes called a *transaction block*.

> **Note:** Some client libraries issue `BEGIN` and `COMMIT` commands automatically, so that you might get the effect of transaction blocks without asking. Check the documentation for the interface you are using.

It's possible to control the statements in a transaction in a more granular fashion through the use of *savepoints*. Savepoints allow you to selectively discard parts of the transaction, while committing the rest. After defining a savepoint with `SAVEPOINT`, you can if needed roll back to the savepoint with `ROLLBACK TO`. All the transaction's database changes between defining the savepoint and rolling back to it are discarded, but changes earlier than the savepoint are kept.

After rolling back to a savepoint, it continues to be defined, so you can roll back to it several times. Conversely, if you are sure you won't need to roll back to a particular savepoint again, it can be released, so the system can free some resources. Keep in mind that either releasing or rolling back to a savepoint will automatically release all savepoints that were defined after it.

All this is happening within the transaction block, so none of it is visible to other database sessions. When and if you commit the transaction block, the committed actions become visible as a unit to other sessions, while the rolled-back actions never become visible at all.

Remembering the bank database, suppose we debit $100.00 from Alice's account, and credit Bob's account, only to find later that we should have credited Wally's account. We could do it using savepoints like this:

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';
SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob';
-- oops ... forget that and use Wally's account
ROLLBACK TO my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

This example is, of course, oversimplified, but there's a lot of control possible in a transaction block through the use of savepoints. Moreover, `ROLLBACK TO` is the only way to regain control of a transaction block that was put in aborted state by the system due to an error, short of rolling it back completely and starting again.

3.5. Window Functions

A *window function* performs a calculation across a set of table rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row — the rows retain their separate identities. Behind the scenes, the window function is able to access more than just the current row of the query result.

Here is an example that shows how to compare each employee's salary with the average salary in his or her department:

```
SELECT depname, empno, salary, avg(salary) OVER (PARTITION BY depname) FROM empsalary;

  depname  | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     9 |   4500 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
```

```
 develop   |    10 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.6666666666666667
(10 rows)
```

The first three output columns come directly from the table `empsalary`, and there is one output row for each row in the table. The fourth column represents an average taken across all the table rows that have the same `depname` value as the current row. (This actually is the same function as the regular `avg` aggregate function, but the `OVER` clause causes it to be treated as a window function and computed across an appropriate set of rows.)

A window function call always contains an `OVER` clause directly following the window function's name and argument(s). This is what syntactically distinguishes it from a regular function or aggregate function. The `OVER` clause determines exactly how the rows of the query are split up for processing by the window function. The `PARTITION BY` list within `OVER` specifies dividing the rows into groups, or partitions, that share the same values of the `PARTITION BY` expression(s). For each row, the window function is computed across the rows that fall into the same partition as the current row.

You can also control the order in which rows are processed by window functions using `ORDER BY` within `OVER`. (The window `ORDER BY` does not even have to match the order in which the rows are output.) Here is an example:

```
SELECT depname, empno, salary,
       rank() OVER (PARTITION BY depname ORDER BY salary DESC)
FROM empsalary;

  depname  | empno | salary | rank
-----------+-------+--------+------
 develop   |     8 |   6000 |    1
 develop   |    10 |   5200 |    2
 develop   |    11 |   5200 |    2
 develop   |     9 |   4500 |    4
 develop   |     7 |   4200 |    5
 personnel |     2 |   3900 |    1
 personnel |     5 |   3500 |    2
 sales     |     1 |   5000 |    1
 sales     |     4 |   4800 |    2
 sales     |     3 |   4800 |    2
(10 rows)
```

As shown here, the `rank` function produces a numerical rank within the current row's partition for each distinct `ORDER BY` value, in the order defined by the `ORDER BY` clause. `rank` needs no explicit parameter, because its behavior is entirely determined by the `OVER` clause.

The rows considered by a window function are those of the "virtual table" produced by the query's `FROM` clause as filtered by its `WHERE`, `GROUP BY`, and `HAVING` clauses if any. For example, a row removed because it does not meet the `WHERE` condition is not seen by any window function. A query can contain multiple window functions that slice up the data in different ways by means of different `OVER` clauses, but they all act on the same collection of rows defined by this virtual table.

We already saw that ORDER BY can be omitted if the ordering of rows is not important. It is also possible to omit PARTITION BY, in which case there is just one partition containing all the rows.

There is another important concept associated with window functions: for each row, there is a set of rows within its partition called its *window frame*. Many (but not all) window functions act only on the rows of the window frame, rather than of the whole partition. By default, if ORDER BY is supplied then the frame consists of all rows from the start of the partition up through the current row, plus any following rows that are equal to the current row according to the ORDER BY clause. When ORDER BY is omitted the default frame consists of all rows in the partition. [1] Here is an example using sum:

```
SELECT salary, sum(salary) OVER () FROM empsalary;

 salary |  sum
--------+-------
   5200 | 47100
   5000 | 47100
   3500 | 47100
   4800 | 47100
   3900 | 47100
   4200 | 47100
   4500 | 47100
   4800 | 47100
   6000 | 47100
   5200 | 47100
(10 rows)
```

Above, since there is no ORDER BY in the OVER clause, the window frame is the same as the partition, which for lack of PARTITION BY is the whole table; in other words each sum is taken over the whole table and so we get the same result for each output row. But if we add an ORDER BY clause, we get very different results:

```
SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary;

 salary |  sum
--------+-------
   3500 |  3500
   3900 |  7400
   4200 | 11600
   4500 | 16100
   4800 | 25700
   4800 | 25700
   5000 | 30700
   5200 | 41100
   5200 | 41100
   6000 | 47100
(10 rows)
```

Here the sum is taken from the first (lowest) salary up through the current one, including any duplicates of the current one (notice the results for the duplicated salaries).

Window functions are permitted only in the SELECT list and the ORDER BY clause of the query. They are forbidden elsewhere, such as in GROUP BY, HAVING and WHERE clauses. This is because they logically

1. There are options to define the window frame in other ways, but this tutorial does not cover them. See Section 4.2.8 for details.

execute after the processing of those clauses. Also, window functions execute after regular aggregate functions. This means it is valid to include an aggregate function call in the arguments of a window function, but not vice versa.

If there is a need to filter or group rows after the window calculations are performed, you can use a sub-select. For example:

```
SELECT depname, empno, salary, enroll_date
FROM
  (SELECT depname, empno, salary, enroll_date,
          rank() OVER (PARTITION BY depname ORDER BY salary DESC, empno) AS pos
     FROM empsalary
  ) AS ss
WHERE pos < 3;
```

The above query only shows the rows from the inner query having `rank` less than 3.

When a query involves multiple window functions, it is possible to write out each one with a separate `OVER` clause, but this is duplicative and error-prone if the same windowing behavior is wanted for several functions. Instead, each windowing behavior can be named in a `WINDOW` clause and then referenced in `OVER`. For example:

```
SELECT sum(salary) OVER w, avg(salary) OVER w
  FROM empsalary
  WINDOW w AS (PARTITION BY depname ORDER BY salary DESC);
```

More details about window functions can be found in Section 4.2.8, Section 9.21, Section 7.2.5, and the SELECT reference page.

3.6. Inheritance

Inheritance is a concept from object-oriented databases. It opens up interesting new possibilities of database design.

Let's create two tables: A table `cities` and a table `capitals`. Naturally, capitals are also cities, so you want some way to show the capitals implicitly when you list all cities. If you're really clever you might invent some scheme like this:

```
CREATE TABLE capitals (
  name       text,
  population real,
  elevation  int,    -- (in ft)
  state      char(2)
);

CREATE TABLE non_capitals (
  name       text,
  population real,
  elevation  int     -- (in ft)
);
```

```
CREATE VIEW cities AS
  SELECT name, population, elevation FROM capitals
    UNION
  SELECT name, population, elevation FROM non_capitals;
```

This works OK as far as querying goes, but it gets ugly when you need to update several rows, for one thing.

A better solution is this:

```
CREATE TABLE cities (
  name       text,
  population real,
  elevation  int     -- (in ft)
);

CREATE TABLE capitals (
  state      char(2) UNIQUE NOT NULL
) INHERITS (cities);
```

In this case, a row of `capitals` *inherits* all columns (`name`, `population`, and `elevation`) from its *parent*, `cities`. The type of the column `name` is `text`, a native PostgreSQL type for variable length character strings. The `capitals` table has an additional column, `state`, which shows its state abbreviation. In PostgreSQL, a table can inherit from zero or more other tables.

For example, the following query finds the names of all cities, including state capitals, that are located at an elevation over 500 feet:

```
SELECT name, elevation
  FROM cities
  WHERE elevation > 500;
```

which returns:

```
   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
 Madison   |       845
(3 rows)
```

On the other hand, the following query finds all the cities that are not state capitals and are situated at an elevation over 500 feet:

```
SELECT name, elevation
    FROM ONLY cities
    WHERE elevation > 500;

   name    | elevation
-----------+-----------
```

```
 Las Vegas |      2174
 Mariposa  |      1953
(2 rows)
```

Here the `ONLY` before `cities` indicates that the query should be run over only the `cities` table, and not tables below `cities` in the inheritance hierarchy. Many of the commands that we have already discussed — `SELECT`, `UPDATE`, and `DELETE` — support this `ONLY` notation.

> **Note:** Although inheritance is frequently useful, it has not been integrated with unique constraints or foreign keys, which limits its usefulness. See Section 5.9 for more detail.

3.7. Conclusion

PostgreSQL has many features not touched upon in this tutorial introduction, which has been oriented toward newer users of SQL. These features are discussed in more detail in the remainder of this book.

If you feel you need more introductory material, please visit the PostgreSQL web site[2] for links to more resources.

2. https://www.postgresql.org

II. The SQL Language

This part describes the use of the SQL language in PostgreSQL. We start with describing the general syntax of SQL, then explain how to create the structures to hold data, how to populate the database, and how to query it. The middle part lists the available data types and functions for use in SQL commands. The rest treats several aspects that are important for tuning a database for optimal performance.

The information in this part is arranged so that a novice user can follow it start to end to gain a full understanding of the topics without having to refer forward too many times. The chapters are intended to be self-contained, so that advanced users can read the chapters individually as they choose. The information in this part is presented in a narrative fashion in topical units. Readers looking for a complete description of a particular command should see Part VI.

Readers of this part should know how to connect to a PostgreSQL database and issue SQL commands. Readers that are unfamiliar with these issues are encouraged to read Part I first. SQL commands are typically entered using the PostgreSQL interactive terminal psql, but other programs that have similar functionality can be used as well.

Chapter 4. SQL Syntax

This chapter describes the syntax of SQL. It forms the foundation for understanding the following chapters which will go into detail about how SQL commands are applied to define and modify data.

We also advise users who are already familiar with SQL to read this chapter carefully because it contains several rules and concepts that are implemented inconsistently among SQL databases or that are specific to PostgreSQL.

4.1. Lexical Structure

SQL input consists of a sequence of *commands*. A command is composed of a sequence of *tokens*, terminated by a semicolon (";"). The end of the input stream also terminates a command. Which tokens are valid depends on the syntax of the particular command.

A token can be a *key word*, an *identifier*, a *quoted identifier*, a *literal* (or constant), or a special character symbol. Tokens are normally separated by whitespace (space, tab, newline), but need not be if there is no ambiguity (which is generally only the case if a special character is adjacent to some other token type).

For example, the following is (syntactically) valid SQL input:

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

This is a sequence of three commands, one per line (although this is not required; more than one command can be on a line, and commands can usefully be split across lines).

Additionally, *comments* can occur in SQL input. They are not tokens, they are effectively equivalent to whitespace.

The SQL syntax is not very consistent regarding what tokens identify commands and which are operands or parameters. The first few tokens are generally the command name, so in the above example we would usually speak of a "SELECT", an "UPDATE", and an "INSERT" command. But for instance the `UPDATE` command always requires a `SET` token to appear in a certain position, and this particular variation of `INSERT` also requires a `VALUES` in order to be complete. The precise syntax rules for each command are described in Part VI.

4.1.1. Identifiers and Key Words

Tokens such as `SELECT`, `UPDATE`, or `VALUES` in the example above are examples of *key words*, that is, words that have a fixed meaning in the SQL language. The tokens `MY_TABLE` and `A` are examples of *identifiers*. They identify names of tables, columns, or other database objects, depending on the command they are used in. Therefore they are sometimes simply called "names". Key words and identifiers have the same lexical structure, meaning that one cannot know whether a token is an identifier or a key word without knowing the language. A complete list of key words can be found in Appendix C.

SQL identifiers and key words must begin with a letter (`a-z`, but also letters with diacritical marks and non-Latin letters) or an underscore (_). Subsequent characters in an identifier or key word can be letters, underscores, digits (`0-9`), or dollar signs (`$`). Note that dollar signs are not allowed in identifiers according

to the letter of the SQL standard, so their use might render applications less portable. The SQL standard will not define a key word that contains digits or starts or ends with an underscore, so identifiers of this form are safe against possible conflict with future extensions of the standard.

The system uses no more than NAMEDATALEN-1 bytes of an identifier; longer names can be written in commands, but they will be truncated. By default, NAMEDATALEN is 64 so the maximum identifier length is 63 bytes. If this limit is problematic, it can be raised by changing the NAMEDATALEN constant in src/include/pg_config_manual.h.

Key words and unquoted identifiers are case insensitive. Therefore:

```
UPDATE MY_TABLE SET A = 5;
```

can equivalently be written as:

```
uPDaTE my_TabLE SeT a = 5;
```

A convention often used is to write key words in upper case and names in lower case, e.g.:

```
UPDATE my_table SET a = 5;
```

There is a second kind of identifier: the *delimited identifier* or *quoted identifier*. It is formed by enclosing an arbitrary sequence of characters in double-quotes ("). A delimited identifier is always an identifier, never a key word. So "select" could be used to refer to a column or table named "select", whereas an unquoted select would be taken as a key word and would therefore provoke a parse error when used where a table or column name is expected. The example can be written with quoted identifiers like this:

```
UPDATE "my_table" SET "a" = 5;
```

Quoted identifiers can contain any character, except the character with code zero. (To include a double quote, write two double quotes.) This allows constructing table or column names that would otherwise not be possible, such as ones containing spaces or ampersands. The length limitation still applies.

A variant of quoted identifiers allows including escaped Unicode characters identified by their code points. This variant starts with U& (upper or lower case U followed by ampersand) immediately before the opening double quote, without any spaces in between, for example U&"foo". (Note that this creates an ambiguity with the operator &. Use spaces around the operator to avoid this problem.) Inside the quotes, Unicode characters can be specified in escaped form by writing a backslash followed by the four-digit hexadecimal code point number or alternatively a backslash followed by a plus sign followed by a six-digit hexadecimal code point number. For example, the identifier "data" could be written as

```
U&"d\0061t\+000061"
```

The following less trivial example writes the Russian word "slon" (elephant) in Cyrillic letters:

```
U&"\0441\043B\043E\043D"
```

If a different escape character than backslash is desired, it can be specified using the UESCAPE clause after the string, for example:

```
U&"d!0061t!+000061" UESCAPE '!'
```

The escape character can be any single character other than a hexadecimal digit, the plus sign, a single quote, a double quote, or a whitespace character. Note that the escape character is written in single quotes, not double quotes.

To include the escape character in the identifier literally, write it twice.

The Unicode escape syntax works only when the server encoding is `UTF8`. When other server encodings are used, only code points in the ASCII range (up to `\007F`) can be specified. Both the 4-digit and the 6-digit form can be used to specify UTF-16 surrogate pairs to compose characters with code points larger than U+FFFF, although the availability of the 6-digit form technically makes this unnecessary. (Surrogate pairs are not stored directly, but combined into a single code point that is then encoded in UTF-8.)

Quoting an identifier also makes it case-sensitive, whereas unquoted names are always folded to lower case. For example, the identifiers `FOO`, `foo`, and `"foo"` are considered the same by PostgreSQL, but `"Foo"` and `"FOO"` are different from these three and each other. (The folding of unquoted names to lower case in PostgreSQL is incompatible with the SQL standard, which says that unquoted names should be folded to upper case. Thus, `foo` should be equivalent to `"FOO"` not `"foo"` according to the standard. If you want to write portable applications you are advised to always quote a particular name or never quote it.)

4.1.2. Constants

There are three kinds of *implicitly-typed constants* in PostgreSQL: strings, bit strings, and numbers. Constants can also be specified with explicit types, which can enable more accurate representation and more efficient handling by the system. These alternatives are discussed in the following subsections.

4.1.2.1. String Constants

A string constant in SQL is an arbitrary sequence of characters bounded by single quotes (`'`), for example `'This is a string'`. To include a single-quote character within a string constant, write two adjacent single quotes, e.g., `'Dianne''s horse'`. Note that this is *not* the same as a double-quote character (`"`).

Two string constants that are only separated by whitespace *with at least one newline* are concatenated and effectively treated as if the string had been written as one constant. For example:

```
SELECT 'foo'
'bar';
```

is equivalent to:

```
SELECT 'foobar';
```

but:

```
SELECT 'foo'      'bar';
```

is not valid syntax. (This slightly bizarre behavior is specified by SQL; PostgreSQL is following the standard.)

4.1.2.2. String Constants with C-style Escapes

PostgreSQL also accepts "escape" string constants, which are an extension to the SQL standard. An escape string constant is specified by writing the letter `E` (upper or lower case) just before the opening single quote, e.g., `E'foo'`. (When continuing an escape string constant across lines, write `E` only before the first opening quote.) Within an escape string, a backslash character (`\`) begins a C-like *backslash escape* sequence, in which the combination of backslash and following character(s) represent a special byte value, as shown in Table 4-1.

Table 4-1. Backslash Escape Sequences

Backslash Escape Sequence	Interpretation
`\b`	backspace
`\f`	form feed
`\n`	newline
`\r`	carriage return
`\t`	tab
`\`*o*, `\`*oo*, `\`*ooo* (*o* = 0 - 7)	octal byte value
`\x`*h*, `\x`*hh* (*h* = 0 - 9, A - F)	hexadecimal byte value
`\u`*xxxx*, `\U`*xxxxxxxx* (*x* = 0 - 9, A - F)	16 or 32-bit hexadecimal Unicode character value

Any other character following a backslash is taken literally. Thus, to include a backslash character, write two backslashes (`\\`). Also, a single quote can be included in an escape string by writing `\'`, in addition to the normal way of `''`.

It is your responsibility that the byte sequences you create, especially when using the octal or hexadecimal escapes, compose valid characters in the server character set encoding. When the server encoding is UTF-8, then the Unicode escapes or the alternative Unicode escape syntax, explained in Section 4.1.2.3, should be used instead. (The alternative would be doing the UTF-8 encoding by hand and writing out the bytes, which would be very cumbersome.)

The Unicode escape syntax works fully only when the server encoding is `UTF8`. When other server encodings are used, only code points in the ASCII range (up to `\u007F`) can be specified. Both the 4-digit and the 8-digit form can be used to specify UTF-16 surrogate pairs to compose characters with code points larger than U+FFFF, although the availability of the 8-digit form technically makes this unnecessary. (When surrogate pairs are used when the server encoding is `UTF8`, they are first combined into a single code point that is then encoded in UTF-8.)

Caution

If the configuration parameter standard_conforming_strings is `off`, then PostgreSQL recognizes backslash escapes in both regular and escape string constants. However, as of PostgreSQL 9.1, the default is `on`, meaning that backslash escapes are recognized only in escape string constants. This behavior is more standards-compliant, but might break applications which rely on the historical behavior, where backslash escapes were always recognized. As a workaround, you can set this parameter to `off`, but it is better to migrate away from using backslash escapes. If you need to use a backslash escape to represent a special character, write the string constant with an `E`.

In addition to `standard_conforming_strings`, the configuration parameters escape_string_warning and backslash_quote govern treatment of backslashes in string constants.

The character with the code zero cannot be in a string constant.

4.1.2.3. String Constants with Unicode Escapes

PostgreSQL also supports another type of escape syntax for strings that allows specifying arbitrary Unicode characters by code point. A Unicode escape string constant starts with `U&` (upper or lower case letter U followed by ampersand) immediately before the opening quote, without any spaces in between, for example `U&'foo'`. (Note that this creates an ambiguity with the operator `&`. Use spaces around the operator to avoid this problem.) Inside the quotes, Unicode characters can be specified in escaped form by writing a backslash followed by the four-digit hexadecimal code point number or alternatively a backslash followed by a plus sign followed by a six-digit hexadecimal code point number. For example, the string `'data'` could be written as

```
U&'d\0061t\+000061'
```

The following less trivial example writes the Russian word “slon” (elephant) in Cyrillic letters:

```
U&'\0441\043B\043E\043D'
```

If a different escape character than backslash is desired, it can be specified using the `UESCAPE` clause after the string, for example:

```
U&'d!0061t!+000061' UESCAPE '!'
```

The escape character can be any single character other than a hexadecimal digit, the plus sign, a single quote, a double quote, or a whitespace character.

The Unicode escape syntax works only when the server encoding is `UTF8`. When other server encodings are used, only code points in the ASCII range (up to `\007F`) can be specified. Both the 4-digit and the 6-digit form can be used to specify UTF-16 surrogate pairs to compose characters with code points larger than U+FFFF, although the availability of the 6-digit form technically makes this unnecessary. (When surrogate pairs are used when the server encoding is `UTF8`, they are first combined into a single code point that is then encoded in UTF-8.)

Also, the Unicode escape syntax for string constants only works when the configuration parameter standard_conforming_strings is turned on. This is because otherwise this syntax could confuse clients that parse the SQL statements to the point that it could lead to SQL injections and similar security issues. If the parameter is set to off, this syntax will be rejected with an error message.

To include the escape character in the string literally, write it twice.

4.1.2.4. Dollar-quoted String Constants

While the standard syntax for specifying string constants is usually convenient, it can be difficult to understand when the desired string contains many single quotes or backslashes, since each of those must be doubled. To allow more readable queries in such situations, PostgreSQL provides another way, called "dollar quoting", to write string constants. A dollar-quoted string constant consists of a dollar sign (`$`), an optional "tag" of zero or more characters, another dollar sign, an arbitrary sequence of characters that makes up the string content, a dollar sign, the same tag that began this dollar quote, and a dollar sign. For example, here are two different ways to specify the string "Dianne's horse" using dollar quoting:

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

Notice that inside the dollar-quoted string, single quotes can be used without needing to be escaped. Indeed, no characters inside a dollar-quoted string are ever escaped: the string content is always written literally. Backslashes are not special, and neither are dollar signs, unless they are part of a sequence matching the opening tag.

It is possible to nest dollar-quoted string constants by choosing different tags at each nesting level. This is most commonly used in writing function definitions. For example:

```
$function$
BEGIN
    RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

Here, the sequence `$q$[\t\r\n\v\\]$q$` represents a dollar-quoted literal string `[\t\r\n\v\\]`, which will be recognized when the function body is executed by PostgreSQL. But since the sequence does not match the outer dollar quoting delimiter `$function$`, it is just some more characters within the constant so far as the outer string is concerned.

The tag, if any, of a dollar-quoted string follows the same rules as an unquoted identifier, except that it cannot contain a dollar sign. Tags are case sensitive, so `$tag$String content$tag$` is correct, but `$TAG$String content$tag$` is not.

A dollar-quoted string that follows a keyword or identifier must be separated from it by whitespace; otherwise the dollar quoting delimiter would be taken as part of the preceding identifier.

Dollar quoting is not part of the SQL standard, but it is often a more convenient way to write complicated string literals than the standard-compliant single quote syntax. It is particularly useful when representing string constants inside other constants, as is often needed in procedural function definitions. With single-quote syntax, each backslash in the above example would have to be written as four backslashes, which would be reduced to two backslashes in parsing the original string constant, and then to one when the inner string constant is re-parsed during function execution.

4.1.2.5. Bit-string Constants

Bit-string constants look like regular string constants with a `B` (upper or lower case) immediately before the opening quote (no intervening whitespace), e.g., `B'1001'`. The only characters allowed within bit-string constants are `0` and `1`.

Alternatively, bit-string constants can be specified in hexadecimal notation, using a leading `X` (upper or lower case), e.g., `X'1FF'`. This notation is equivalent to a bit-string constant with four binary digits for each hexadecimal digit.

Both forms of bit-string constant can be continued across lines in the same way as regular string constants. Dollar quoting cannot be used in a bit-string constant.

4.1.2.6. Numeric Constants

Numeric constants are accepted in these general forms:

```
digits
digits.[digits][e[+-]digits]
[digits].digits[e[+-]digits]
digitse[+-]digits
```

where *digits* is one or more decimal digits (0 through 9). At least one digit must be before or after the decimal point, if one is used. At least one digit must follow the exponent marker (`e`), if one is present. There cannot be any spaces or other characters embedded in the constant. Note that any leading plus or minus sign is not actually considered part of the constant; it is an operator applied to the constant.

These are some examples of valid numeric constants:

42
3.5
4.
.001
5e2
1.925e-3

A numeric constant that contains neither a decimal point nor an exponent is initially presumed to be type `integer` if its value fits in type `integer` (32 bits); otherwise it is presumed to be type `bigint` if its value fits in type `bigint` (64 bits); otherwise it is taken to be type `numeric`. Constants that contain decimal points and/or exponents are always initially presumed to be type `numeric`.

The initially assigned data type of a numeric constant is just a starting point for the type resolution algorithms. In most cases the constant will be automatically coerced to the most appropriate type depending on context. When necessary, you can force a numeric value to be interpreted as a specific data type by casting it. For example, you can force a numeric value to be treated as type `real` (`float4`) by writing:

```
REAL '1.23'  -- string style
1.23::REAL   -- PostgreSQL (historical) style
```

These are actually just special cases of the general casting notations discussed next.

4.1.2.7. Constants of Other Types

A constant of an *arbitrary* type can be entered using any one of the following notations:

```
type 'string'
'string'::type
CAST ( 'string' AS type )
```

The string constant's text is passed to the input conversion routine for the type called *type*. The result is a constant of the indicated type. The explicit type cast can be omitted if there is no ambiguity as to the type the constant must be (for example, when it is assigned directly to a table column), in which case it is automatically coerced.

The string constant can be written using either regular SQL notation or dollar-quoting.

It is also possible to specify a type coercion using a function-like syntax:

```
typename ( 'string' )
```

but not all type names can be used in this way; see Section 4.2.9 for details.

The `::`, `CAST()`, and function-call syntaxes can also be used to specify run-time type conversions of arbitrary expressions, as discussed in Section 4.2.9. To avoid syntactic ambiguity, the *type* `'`*string*`'` syntax can only be used to specify the type of a simple literal constant. Another restriction on the *type* `'`*string*`'` syntax is that it does not work for array types; use `::` or `CAST()` to specify the type of an array constant.

The `CAST()` syntax conforms to SQL. The *type* `'`*string*`'` syntax is a generalization of the standard: SQL specifies this syntax only for a few data types, but PostgreSQL allows it for all types. The syntax with `::` is historical PostgreSQL usage, as is the function-call syntax.

4.1.3. Operators

An operator name is a sequence of up to `NAMEDATALEN`-1 (63 by default) characters from the following list:

+ - * / < > = ~ ! @ # % ^ & | ` ?

There are a few restrictions on operator names, however:

- `--` and `/*` cannot appear anywhere in an operator name, since they will be taken as the start of a comment.
- A multiple-character operator name cannot end in + or -, unless the name also contains at least one of these characters:

  ~ ! @ # % ^ & | ` ?

  For example, `@-` is an allowed operator name, but `*-` is not. This restriction allows PostgreSQL to parse SQL-compliant queries without requiring spaces between tokens.

When working with non-SQL-standard operator names, you will usually need to separate adjacent operators with spaces to avoid ambiguity. For example, if you have defined a left unary operator named `@`, you cannot write `X*@Y`; you must write `X* @Y` to ensure that PostgreSQL reads it as two operator names not one.

4.1.4. Special Characters

Some characters that are not alphanumeric have a special meaning that is different from being an operator. Details on the usage can be found at the location where the respective syntax element is described. This section only exists to advise the existence and summarize the purposes of these characters.

- A dollar sign (`$`) followed by digits is used to represent a positional parameter in the body of a function definition or a prepared statement. In other contexts the dollar sign can be part of an identifier or a dollar-quoted string constant.
- Parentheses (`()`) have their usual meaning to group expressions and enforce precedence. In some cases parentheses are required as part of the fixed syntax of a particular SQL command.
- Brackets (`[]`) are used to select the elements of an array. See Section 8.15 for more information on arrays.
- Commas (`,`) are used in some syntactical constructs to separate the elements of a list.
- The semicolon (`;`) terminates an SQL command. It cannot appear anywhere within a command, except within a string constant or quoted identifier.
- The colon (`:`) is used to select "slices" from arrays. (See Section 8.15.) In certain SQL dialects (such as Embedded SQL), the colon is used to prefix variable names.
- The asterisk (`*`) is used in some contexts to denote all the fields of a table row or composite value. It also has a special meaning when used as the argument of an aggregate function, namely that the aggregate does not require any explicit parameter.
- The period (`.`) is used in numeric constants, and to separate schema, table, and column names.

4.1.5. Comments

A comment is a sequence of characters beginning with double dashes and extending to the end of the line, e.g.:

```
-- This is a standard SQL comment
```

Alternatively, C-style block comments can be used:

```
/* multiline comment
 * with nesting: /* nested block comment */
 */
```

where the comment begins with /* and extends to the matching occurrence of */. These block comments nest, as specified in the SQL standard but unlike C, so that one can comment out larger blocks of code that might contain existing block comments.

A comment is removed from the input stream before further syntax analysis and is effectively replaced by whitespace.

4.1.6. Operator Precedence

Table 4-2 shows the precedence and associativity of the operators in PostgreSQL. Most operators have the same precedence and are left-associative. The precedence and associativity of the operators is hard-wired into the parser.

You will sometimes need to add parentheses when using combinations of binary and unary operators. For instance:

```
SELECT 5 ! - 6;
```

will be parsed as:

```
SELECT 5 ! (- 6);
```

because the parser has no idea — until it is too late — that ! is defined as a postfix operator, not an infix one. To get the desired behavior in this case, you must write:

```
SELECT (5 !) - 6;
```

This is the price one pays for extensibility.

Table 4-2. Operator Precedence (highest to lowest)

Operator/Element	Associativity	Description
.	left	table/column name separator
::	left	PostgreSQL-style typecast
[ ]	left	array element selection
+ -	right	unary plus, unary minus
^	left	exponentiation
* / %	left	multiplication, division, modulo
+ -	left	addition, subtraction
(any other operator)	left	all other native and user-defined operators
BETWEEN IN LIKE ILIKE SIMILAR		range containment, set membership, string matching
< > = <= >= <>		comparison operators
IS ISNULL NOTNULL		IS TRUE, IS FALSE, IS NULL, IS DISTINCT FROM, etc
NOT	right	logical negation

Operator/Element	Associativity	Description
AND	left	logical conjunction
OR	left	logical disjunction

Note that the operator precedence rules also apply to user-defined operators that have the same names as the built-in operators mentioned above. For example, if you define a "+" operator for some custom data type it will have the same precedence as the built-in "+" operator, no matter what yours does.

When a schema-qualified operator name is used in the OPERATOR syntax, as for example in:

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

the OPERATOR construct is taken to have the default precedence shown in Table 4-2 for "any other operator". This is true no matter which specific operator appears inside OPERATOR().

> **Note:** PostgreSQL versions before 9.5 used slightly different operator precedence rules. In particular, <= >= and <> used to be treated as generic operators; IS tests used to have higher priority; and NOT BETWEEN and related constructs acted inconsistently, being taken in some cases as having the precedence of NOT rather than BETWEEN. These rules were changed for better compliance with the SQL standard and to reduce confusion from inconsistent treatment of logically equivalent constructs. In most cases, these changes will result in no behavioral change, or perhaps in "no such operator" failures which can be resolved by adding parentheses. However there are corner cases in which a query might change behavior without any parsing error being reported. If you are concerned about whether these changes have silently broken something, you can test your application with the configuration parameter operator_precedence_warning turned on to see if any warnings are logged.

4.2. Value Expressions

Value expressions are used in a variety of contexts, such as in the target list of the SELECT command, as new column values in INSERT or UPDATE, or in search conditions in a number of commands. The result of a value expression is sometimes called a *scalar*, to distinguish it from the result of a table expression (which is a table). Value expressions are therefore also called *scalar expressions* (or even simply *expressions*). The expression syntax allows the calculation of values from primitive parts using arithmetic, logical, set, and other operations.

A value expression is one of the following:

- A constant or literal value
- A column reference
- A positional parameter reference, in the body of a function definition or prepared statement
- A subscripted expression
- A field selection expression
- An operator invocation

- A function call
- An aggregate expression
- A window function call
- A type cast
- A collation expression
- A scalar subquery
- An array constructor
- A row constructor
- Another value expression in parentheses (used to group subexpressions and override precedence)

In addition to this list, there are a number of constructs that can be classified as an expression but do not follow any general syntax rules. These generally have the semantics of a function or operator and are explained in the appropriate location in Chapter 9. An example is the `IS NULL` clause.

We have already discussed constants in Section 4.1.2. The following sections discuss the remaining options.

4.2.1. Column References

A column can be referenced in the form:

correlation.*columnname*

correlation is the name of a table (possibly qualified with a schema name), or an alias for a table defined by means of a `FROM` clause. The correlation name and separating dot can be omitted if the column name is unique across all the tables being used in the current query. (See also Chapter 7.)

4.2.2. Positional Parameters

A positional parameter reference is used to indicate a value that is supplied externally to an SQL statement. Parameters are used in SQL function definitions and in prepared queries. Some client libraries also support specifying data values separately from the SQL command string, in which case parameters are used to refer to the out-of-line data values. The form of a parameter reference is:

$*number*

For example, consider the definition of a function, `dept`, as:

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$
    LANGUAGE SQL;
```

Here the `$1` references the value of the first function argument whenever the function is invoked.

4.2.3. Subscripts

If an expression yields a value of an array type, then a specific element of the array value can be extracted by writing

```
expression[subscript]
```

or multiple adjacent elements (an "array slice") can be extracted by writing

```
expression[lower_subscript:upper_subscript]
```

(Here, the brackets `[ ]` are meant to appear literally.) Each *subscript* is itself an expression, which will be rounded to the nearest integer value.

In general the array *expression* must be parenthesized, but the parentheses can be omitted when the expression to be subscripted is just a column reference or positional parameter. Also, multiple subscripts can be concatenated when the original array is multidimensional. For example:

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]
(arrayfunction(a,b))[42]
```

The parentheses in the last example are required. See Section 8.15 for more about arrays.

4.2.4. Field Selection

If an expression yields a value of a composite type (row type), then a specific field of the row can be extracted by writing

```
expression.fieldname
```

In general the row *expression* must be parenthesized, but the parentheses can be omitted when the expression to be selected from is just a table reference or positional parameter. For example:

```
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3
```

(Thus, a qualified column reference is actually just a special case of the field selection syntax.) An important special case is extracting a field from a table column that is of a composite type:

```
(compositecol).somefield
(mytable.compositecol).somefield
```

The parentheses are required here to show that `compositecol` is a column name not a table name, or that `mytable` is a table name not a schema name in the second case.

You can ask for all fields of a composite value by writing `.*`:

```
(compositecol).*
```

This notation behaves differently depending on context; see Section 8.16.5 for details.

4.2.5. Operator Invocations

There are three possible syntaxes for an operator invocation:

expression *operator* *expression* (binary infix operator)
operator *expression* (unary prefix operator)
expression *operator* (unary postfix operator)

where the *operator* token follows the syntax rules of Section 4.1.3, or is one of the key words `AND`, `OR`, and `NOT`, or is a qualified operator name in the form:

```
OPERATOR(schema.operatorname)
```

Which particular operators exist and whether they are unary or binary depends on what operators have been defined by the system or the user. Chapter 9 describes the built-in operators.

4.2.6. Function Calls

The syntax for a function call is the name of a function (possibly qualified with a schema name), followed by its argument list enclosed in parentheses:

```
function_name ([expression [, expression ... ]] )
```

For example, the following computes the square root of 2:

```
sqrt(2)
```

The list of built-in functions is in Chapter 9. Other functions can be added by the user.

When issuing queries in a database where some users mistrust other users, observe security precautions from Section 10.3 when writing function calls.

The arguments can optionally have names attached. See Section 4.3 for details.

> **Note:** A function that takes a single argument of composite type can optionally be called using field-selection syntax, and conversely field selection can be written in functional style. That is, the notations `col(table)` and `table.col` are interchangeable. This behavior is not SQL-standard but is provided in PostgreSQL because it allows use of functions to emulate “computed fields”. For more information see Section 8.16.5.

4.2.7. Aggregate Expressions

An *aggregate expression* represents the application of an aggregate function across the rows selected by a query. An aggregate function reduces multiple inputs to a single output value, such as the sum or average of the inputs. The syntax of an aggregate expression is one of the following:

```
aggregate_name (expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE filter_clause ) ]
aggregate_name (ALL expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE filter_clause ) ]
aggregate_name (DISTINCT expression [ , ... ] [ order_by_clause ] ) [ FILTER ( WHERE filter_clau
aggregate_name ( * ) [ FILTER ( WHERE filter_clause ) ]
aggregate_name ( [ expression [ , ... ] ] ) WITHIN GROUP ( order_by_clause ) [ FILTER ( WHERE f
```

where *aggregate_name* is a previously defined aggregate (possibly qualified with a schema name) and *expression* is any value expression that does not itself contain an aggregate expression or a window function call. The optional *order_by_clause* and *filter_clause* are described below.

The first form of aggregate expression invokes the aggregate once for each input row. The second form is the same as the first, since `ALL` is the default. The third form invokes the aggregate once for each distinct value of the expression (or distinct set of values, for multiple expressions) found in the input rows. The fourth form invokes the aggregate once for each input row; since no particular input value is specified, it is generally only useful for the `count(*)` aggregate function. The last form is used with *ordered-set* aggregate functions, which are described below.

Most aggregate functions ignore null inputs, so that rows in which one or more of the expression(s) yield null are discarded. This can be assumed to be true, unless otherwise specified, for all built-in aggregates.

For example, `count(*)` yields the total number of input rows; `count(f1)` yields the number of input rows in which `f1` is non-null, since `count` ignores nulls; and `count(distinct f1)` yields the number of distinct non-null values of `f1`.

Ordinarily, the input rows are fed to the aggregate function in an unspecified order. In many cases this does not matter; for example, `min` produces the same result no matter what order it receives the inputs in. However, some aggregate functions (such as `array_agg` and `string_agg`) produce results that depend on the ordering of the input rows. When using such an aggregate, the optional *order_by_clause* can be used to specify the desired ordering. The *order_by_clause* has the same syntax as for a query-level `ORDER BY` clause, as described in Section 7.5, except that its expressions are always just expressions and cannot be output-column names or numbers. For example:

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

When dealing with multiple-argument aggregate functions, note that the `ORDER BY` clause goes after all the aggregate arguments. For example, write this:

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

not this:

```
SELECT string_agg(a ORDER BY a, ',') FROM table;  -- incorrect
```

The latter is syntactically valid, but it represents a call of a single-argument aggregate function with two `ORDER BY` keys (the second one being rather useless since it's a constant).

If `DISTINCT` is specified in addition to an *`order_by_clause`*, then all the `ORDER BY` expressions must match regular arguments of the aggregate; that is, you cannot sort on an expression that is not included in the `DISTINCT` list.

> **Note:** The ability to specify both `DISTINCT` and `ORDER BY` in an aggregate function is a PostgreSQL extension.

Placing `ORDER BY` within the aggregate's regular argument list, as described so far, is used when ordering the input rows for a "normal" aggregate for which ordering is optional. There is a subclass of aggregate functions called *ordered-set aggregates* for which an *`order_by_clause`* is *required*, usually because the aggregate's computation is only sensible in terms of a specific ordering of its input rows. Typical examples of ordered-set aggregates include rank and percentile calculations. For an ordered-set aggregate, the *`order_by_clause`* is written inside `WITHIN GROUP (...)`, as shown in the final syntax alternative above. The expressions in the *`order_by_clause`* are evaluated once per input row just like normal aggregate arguments, sorted as per the *`order_by_clause`*'s requirements, and fed to the aggregate function as input arguments. (This is unlike the case for a non-`WITHIN GROUP` *`order_by_clause`*, which is not treated as argument(s) to the aggregate function.) The argument expressions preceding `WITHIN GROUP`, if any, are called *direct arguments* to distinguish them from the *aggregated arguments* listed in the *`order_by_clause`*. Unlike normal aggregate arguments, direct arguments are evaluated only once per aggregate call, not once per input row. This means that they can contain variables only if those variables are grouped by `GROUP BY`; this restriction is the same as if the direct arguments were not inside an aggregate expression at all. Direct arguments are typically used for things like percentile fractions, which only make sense as a single value per aggregation calculation. The direct argument list can be empty; in this case, write just `()` not `(*)`. (PostgreSQL will actually accept either spelling, but only the first way conforms to the SQL standard.)

An example of an ordered-set aggregate call is:

```
SELECT percentile_cont(0.5) WITHIN GROUP (ORDER BY income) FROM households;
 percentile_cont
-----------------
           50489
```

which obtains the 50th percentile, or median, value of the `income` column from table `households`. Here, `0.5` is a direct argument; it would make no sense for the percentile fraction to be a value varying across rows.

If `FILTER` is specified, then only the input rows for which the *`filter_clause`* evaluates to true are fed to the aggregate function; other rows are discarded. For example:

```
SELECT
    count(*) AS unfiltered,
    count(*) FILTER (WHERE i < 5) AS filtered
FROM generate_series(1,10) AS s(i);
 unfiltered | filtered
------------+----------
         10 |        4
(1 row)
```

The predefined aggregate functions are described in Section 9.20. Other aggregate functions can be added by the user.

An aggregate expression can only appear in the result list or HAVING clause of a SELECT command. It is forbidden in other clauses, such as WHERE, because those clauses are logically evaluated before the results of aggregates are formed.

When an aggregate expression appears in a subquery (see Section 4.2.11 and Section 9.22), the aggregate is normally evaluated over the rows of the subquery. But an exception occurs if the aggregate's arguments (and *filter_clause* if any) contain only outer-level variables: the aggregate then belongs to the nearest such outer level, and is evaluated over the rows of that query. The aggregate expression as a whole is then an outer reference for the subquery it appears in, and acts as a constant over any one evaluation of that subquery. The restriction about appearing only in the result list or HAVING clause applies with respect to the query level that the aggregate belongs to.

4.2.8. Window Function Calls

A *window function call* represents the application of an aggregate-like function over some portion of the rows selected by a query. Unlike regular aggregate function calls, this is not tied to grouping of the selected rows into a single output row — each row remains separate in the query output. However the window function is able to scan all the rows that would be part of the current row's group according to the grouping specification (PARTITION BY list) of the window function call. The syntax of a window function call is one of the following:

```
function_name ([expression [, expression ... ]]) [ FILTER ( WHERE filter_clause ) ] OVER window_n
function_name ([expression [, expression ... ]]) [ FILTER ( WHERE filter_clause ) ] OVER ( windo
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER window_name
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER ( window_definition )
```

where *window_definition* has the syntax

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

and the optional *frame_clause* can be one of

```
{ RANGE | ROWS } frame_start
{ RANGE | ROWS } BETWEEN frame_start AND frame_end
```

where *frame_start* and *frame_end* can be one of

```
UNBOUNDED PRECEDING
value PRECEDING
CURRENT ROW
value FOLLOWING
UNBOUNDED FOLLOWING
```

Here, *expression* represents any value expression that does not itself contain window function calls.

window_name is a reference to a named window specification defined in the query's WINDOW clause. Alternatively, a full *window_definition* can be given within parentheses, using the same syntax as for defining a named window in the WINDOW clause; see the SELECT reference page for details. It's worth pointing out that OVER wname is not exactly equivalent to OVER (wname); the latter implies copying and modifying the window definition, and will be rejected if the referenced window specification includes a frame clause.

The PARTITION BY option groups the rows of the query into *partitions*, which are processed separately by the window function. PARTITION BY works similarly to a query-level GROUP BY clause, except that its expressions are always just expressions and cannot be output-column names or numbers. Without PARTITION BY, all rows produced by the query are treated as a single partition. The ORDER BY option determines the order in which the rows of a partition are processed by the window function. It works similarly to a query-level ORDER BY clause, but likewise cannot use output-column names or numbers. Without ORDER BY, rows are processed in an unspecified order.

The *frame_clause* specifies the set of rows constituting the *window frame*, which is a subset of the current partition, for those window functions that act on the frame instead of the whole partition. The frame can be specified in either RANGE or ROWS mode; in either case, it runs from the *frame_start* to the *frame_end*. If *frame_end* is omitted, it defaults to CURRENT ROW.

A *frame_start* of UNBOUNDED PRECEDING means that the frame starts with the first row of the partition, and similarly a *frame_end* of UNBOUNDED FOLLOWING means that the frame ends with the last row of the partition.

In RANGE mode, a *frame_start* of CURRENT ROW means the frame starts with the current row's first *peer* row (a row that ORDER BY considers equivalent to the current row), while a *frame_end* of CURRENT ROW means the frame ends with the last equivalent ORDER BY peer. In ROWS mode, CURRENT ROW simply means the current row.

The *value* PRECEDING and *value* FOLLOWING cases are currently only allowed in ROWS mode. They indicate that the frame starts or ends the specified number of rows before or after the current row. *value* must be an integer expression not containing any variables, aggregate functions, or window functions. The value must not be null or negative; but it can be zero, which just selects the current row.

The default framing option is RANGE UNBOUNDED PRECEDING, which is the same as RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. With ORDER BY, this sets the frame to be all rows from the partition start up through the current row's last ORDER BY peer. Without ORDER BY, all rows of the partition are included in the window frame, since all rows become peers of the current row.

Restrictions are that *frame_start* cannot be UNBOUNDED FOLLOWING, *frame_end* cannot be UNBOUNDED PRECEDING, and the *frame_end* choice cannot appear earlier in the above list than the *frame_start* choice — for example RANGE BETWEEN CURRENT ROW AND *value* PRECEDING is not allowed.

If FILTER is specified, then only the input rows for which the *filter_clause* evaluates to true are fed to the window function; other rows are discarded. Only window functions that are aggregates accept a FILTER clause.

The built-in window functions are described in Table 9-54. Other window functions can be added by the user. Also, any built-in or user-defined normal aggregate function can be used as a window function. Ordered-set aggregates presently cannot be used as window functions, however.

The syntaxes using * are used for calling parameter-less aggregate functions as window functions, for example count(*) OVER (PARTITION BY x ORDER BY y). The asterisk (*) is customarily not used

for non-aggregate window functions. Aggregate window functions, unlike normal aggregate functions, do not allow DISTINCT or ORDER BY to be used within the function argument list.

Window function calls are permitted only in the SELECT list and the ORDER BY clause of the query.

More information about window functions can be found in Section 3.5, Section 9.21, and Section 7.2.5.

4.2.9. Type Casts

A type cast specifies a conversion from one data type to another. PostgreSQL accepts two equivalent syntaxes for type casts:

```
CAST ( expression AS type )
expression::type
```

The CAST syntax conforms to SQL; the syntax with :: is historical PostgreSQL usage.

When a cast is applied to a value expression of a known type, it represents a run-time type conversion. The cast will succeed only if a suitable type conversion operation has been defined. Notice that this is subtly different from the use of casts with constants, as shown in Section 4.1.2.7. A cast applied to an unadorned string literal represents the initial assignment of a type to a literal constant value, and so it will succeed for any type (if the contents of the string literal are acceptable input syntax for the data type).

An explicit type cast can usually be omitted if there is no ambiguity as to the type that a value expression must produce (for example, when it is assigned to a table column); the system will automatically apply a type cast in such cases. However, automatic casting is only done for casts that are marked "OK to apply implicitly" in the system catalogs. Other casts must be invoked with explicit casting syntax. This restriction is intended to prevent surprising conversions from being applied silently.

It is also possible to specify a type cast using a function-like syntax:

```
typename ( expression )
```

However, this only works for types whose names are also valid as function names. For example, double precision cannot be used this way, but the equivalent float8 can. Also, the names interval, time, and timestamp can only be used in this fashion if they are double-quoted, because of syntactic conflicts. Therefore, the use of the function-like cast syntax leads to inconsistencies and should probably be avoided.

> **Note:** The function-like syntax is in fact just a function call. When one of the two standard cast syntaxes is used to do a run-time conversion, it will internally invoke a registered function to perform the conversion. By convention, these conversion functions have the same name as their output type, and thus the "function-like syntax" is nothing more than a direct invocation of the underlying conversion function. Obviously, this is not something that a portable application should rely on. For further details see CREATE CAST.

4.2.10. Collation Expressions

The COLLATE clause overrides the collation of an expression. It is appended to the expression it applies to:

```
expr COLLATE collation
```

where *collation* is a possibly schema-qualified identifier. The COLLATE clause binds tighter than operators; parentheses can be used when necessary.

If no collation is explicitly specified, the database system either derives a collation from the columns involved in the expression, or it defaults to the default collation of the database if no column is involved in the expression.

The two common uses of the COLLATE clause are overriding the sort order in an ORDER BY clause, for example:

```
SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";
```

and overriding the collation of a function or operator call that has locale-sensitive results, for example:

```
SELECT * FROM tbl WHERE a > 'foo' COLLATE "C";
```

Note that in the latter case the COLLATE clause is attached to an input argument of the operator we wish to affect. It doesn't matter which argument of the operator or function call the COLLATE clause is attached to, because the collation that is applied by the operator or function is derived by considering all arguments, and an explicit COLLATE clause will override the collations of all other arguments. (Attaching non-matching COLLATE clauses to more than one argument, however, is an error. For more details see Section 22.2.) Thus, this gives the same result as the previous example:

```
SELECT * FROM tbl WHERE a COLLATE "C" > 'foo';
```

But this is an error:

```
SELECT * FROM tbl WHERE (a > 'foo') COLLATE "C";
```

because it attempts to apply a collation to the result of the > operator, which is of the non-collatable data type boolean.

4.2.11. Scalar Subqueries

A scalar subquery is an ordinary SELECT query in parentheses that returns exactly one row with one column. (See Chapter 7 for information about writing queries.) The SELECT query is executed and the single returned value is used in the surrounding value expression. It is an error to use a query that returns more than one row or more than one column as a scalar subquery. (But if, during a particular execution, the subquery returns no rows, there is no error; the scalar result is taken to be null.) The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery. See also Section 9.22 for other expressions involving subqueries.

For example, the following finds the largest city population in each state:

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
```

```
    FROM states;
```

4.2.12. Array Constructors

An array constructor is an expression that builds an array value using values for its member elements. A simple array constructor consists of the key word ARRAY, a left square bracket [, a list of expressions (separated by commas) for the array element values, and finally a right square bracket]. For example:

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
(1 row)
```

By default, the array element type is the common type of the member expressions, determined using the same rules as for UNION or CASE constructs (see Section 10.5). You can override this by explicitly casting the array constructor to the desired type, for example:

```
SELECT ARRAY[1,2,22.7]::integer[];
  array
----------
 {1,2,23}
(1 row)
```

This has the same effect as casting each expression to the array element type individually. For more on casting, see Section 4.2.9.

Multidimensional array values can be built by nesting array constructors. In the inner constructors, the key word ARRAY can be omitted. For example, these produce the same result:

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)
```

Since multidimensional arrays must be rectangular, inner constructors at the same level must produce sub-arrays of identical dimensions. Any cast applied to the outer ARRAY constructor propagates automatically to all the inner constructors.

Multidimensional array constructor elements can be anything yielding an array of the proper kind, not only a sub-ARRAY construct. For example:

```
CREATE TABLE arr(f1 int[], f2 int[]);
```

```
INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                     array
-----------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
(1 row)
```

You can construct an empty array, but since it's impossible to have an array with no type, you must explicitly cast your empty array to the desired type. For example:

```
SELECT ARRAY[]::integer[];
 array
-------
 {}
(1 row)
```

It is also possible to construct an array from the results of a subquery. In this form, the array constructor is written with the key word `ARRAY` followed by a parenthesized (not bracketed) subquery. For example:

```
SELECT ARRAY(SELECT oid FROM pg_proc WHERE proname LIKE 'bytea%');
                                  array
------------------------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31,2412,2413}
(1 row)

SELECT ARRAY(SELECT ARRAY[i, i*2] FROM generate_series(1,5) AS a(i));
              array
----------------------------------
 {{1,2},{2,4},{3,6},{4,8},{5,10}}
(1 row)
```

The subquery must return a single column. If the subquery's output column is of a non-array type, the resulting one-dimensional array will have an element for each row in the subquery result, with an element type matching that of the subquery's output column. If the subquery's output column is of an array type, the result will be an array of the same type but one higher dimension; in this case all the subquery rows must yield arrays of identical dimensionality, else the result would not be rectangular.

The subscripts of an array value built with `ARRAY` always begin with one. For more information about arrays, see Section 8.15.

4.2.13. Row Constructors

A row constructor is an expression that builds a row value (also called a composite value) using values for its member fields. A row constructor consists of the key word ROW, a left parenthesis, zero or more expressions (separated by commas) for the row field values, and finally a right parenthesis. For example:

```
SELECT ROW(1,2.5,'this is a test');
```

The key word ROW is optional when there is more than one expression in the list.

A row constructor can include the syntax *rowvalue*.*, which will be expanded to a list of the elements of the row value, just as occurs when the .* syntax is used at the top level of a SELECT list (see Section 8.16.5). For example, if table t has columns f1 and f2, these are the same:

```
SELECT ROW(t.*, 42) FROM t;
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

> **Note:** Before PostgreSQL 8.2, the .* syntax was not expanded in row constructors, so that writing ROW(t.*, 42) created a two-field row whose first field was another row value. The new behavior is usually more useful. If you need the old behavior of nested row values, write the inner row value without .*, for instance ROW(t, 42).

By default, the value created by a ROW expression is of an anonymous record type. If necessary, it can be cast to a named composite type — either the row type of a table, or a composite type created with CREATE TYPE AS. An explicit cast might be needed to avoid ambiguity. For example:

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);

CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- No cast needed since only one getf1() exists
SELECT getf1(ROW(1,2.5,'this is a test'));
 getf1
-------
     1
(1 row)

CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- Now we need a cast to indicate which function to call:
SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR:  function getf1(record) is not unique

SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
 getf1
-------
     1
(1 row)
```

```
SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS myrowtype));
 getf1
-------
    11
(1 row)
```

Row constructors can be used to build composite values to be stored in a composite-type table column, or to be passed to a function that accepts a composite parameter. Also, it is possible to compare two row values or test a row with `IS NULL` or `IS NOT NULL`, for example:

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');

SELECT ROW(table.*) IS NULL FROM table;  -- detect all-null rows
```

For more detail see Section 9.23. Row constructors can also be used in connection with subqueries, as discussed in Section 9.22.

4.2.14. Expression Evaluation Rules

The order of evaluation of subexpressions is not defined. In particular, the inputs of an operator or function are not necessarily evaluated left-to-right or in any other fixed order.

Furthermore, if the result of an expression can be determined by evaluating only some parts of it, then other subexpressions might not be evaluated at all. For instance, if one wrote:

```
SELECT true OR somefunc();
```

then `somefunc()` would (probably) not be called at all. The same would be the case if one wrote:

```
SELECT somefunc() OR true;
```

Note that this is not the same as the left-to-right "short-circuiting" of Boolean operators that is found in some programming languages.

As a consequence, it is unwise to use functions with side effects as part of complex expressions. It is particularly dangerous to rely on side effects or evaluation order in `WHERE` and `HAVING` clauses, since those clauses are extensively reprocessed as part of developing an execution plan. Boolean expressions (`AND`/`OR`/`NOT` combinations) in those clauses can be reorganized in any manner allowed by the laws of Boolean algebra.

When it is essential to force evaluation order, a `CASE` construct (see Section 9.17) can be used. For example, this is an untrustworthy way of trying to avoid division by zero in a `WHERE` clause:

```
SELECT ... WHERE x > 0 AND y/x > 1.5;
```

But this is safe:

```
SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;
```

A CASE construct used in this fashion will defeat optimization attempts, so it should only be done when necessary. (In this particular example, it would be better to sidestep the problem by writing y > 1.5*x instead.)

CASE is not a cure-all for such issues, however. One limitation of the technique illustrated above is that it does not prevent early evaluation of constant subexpressions. As described in Section 35.6, functions and operators marked IMMUTABLE can be evaluated when the query is planned rather than when it is executed. Thus for example

```
SELECT CASE WHEN x > 0 THEN x ELSE 1/0 END FROM tab;
```

is likely to result in a division-by-zero failure due to the planner trying to simplify the constant subexpression, even if every row in the table has x > 0 so that the ELSE arm would never be entered at run time.

While that particular example might seem silly, related cases that don't obviously involve constants can occur in queries executed within functions, since the values of function arguments and local variables can be inserted into queries as constants for planning purposes. Within PL/pgSQL functions, for example, using an IF-THEN-ELSE statement to protect a risky computation is much safer than just nesting it in a CASE expression.

Another limitation of the same kind is that a CASE cannot prevent evaluation of an aggregate expression contained within it, because aggregate expressions are computed before other expressions in a SELECT list or HAVING clause are considered. For example, the following query can cause a division-by-zero error despite seemingly having protected against it:

```
SELECT CASE WHEN min(employees) > 0
            THEN avg(expenses / employees)
       END
    FROM departments;
```

The min() and avg() aggregates are computed concurrently over all the input rows, so if any row has employees equal to zero, the division-by-zero error will occur before there is any opportunity to test the result of min(). Instead, use a WHERE or FILTER clause to prevent problematic input rows from reaching an aggregate function in the first place.

4.3. Calling Functions

PostgreSQL allows functions that have named parameters to be called using either *positional* or *named* notation. Named notation is especially useful for functions that have a large number of parameters, since it makes the associations between parameters and actual arguments more explicit and reliable. In positional notation, a function call is written with its argument values in the same order as they are defined in the function declaration. In named notation, the arguments are matched to the function parameters by name and can be written in any order. For each notation, also consider the effect of function argument types, documented in Section 10.3.

In either notation, parameters that have default values given in the function declaration need not be written in the call at all. But this is particularly useful in named notation, since any combination of parameters can be omitted; while in positional notation parameters can only be omitted from right to left.

PostgreSQL also supports *mixed* notation, which combines positional and named notation. In this case, positional parameters are written first and named parameters appear after them.

The following examples will illustrate the usage of all three notations, using the following function definition:

```
CREATE FUNCTION concat_lower_or_upper(a text, b text, uppercase boolean DEFAULT false)
RETURNS text
AS
$$
 SELECT CASE
        WHEN $3 THEN UPPER($1 || ' ' || $2)
        ELSE LOWER($1 || ' ' || $2)
        END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

Function `concat_lower_or_upper` has two mandatory parameters, `a` and `b`. Additionally there is one optional parameter `uppercase` which defaults to `false`. The `a` and `b` inputs will be concatenated, and forced to either upper or lower case depending on the `uppercase` parameter. The remaining details of this function definition are not important here (see Chapter 35 for more information).

4.3.1. Using Positional Notation

Positional notation is the traditional mechanism for passing arguments to functions in PostgreSQL. An example is:

```
SELECT concat_lower_or_upper('Hello', 'World', true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

All arguments are specified in order. The result is upper case since `uppercase` is specified as `true`. Another example is:

```
SELECT concat_lower_or_upper('Hello', 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

Here, the `uppercase` parameter is omitted, so it receives its default value of `false`, resulting in lower case output. In positional notation, arguments can be omitted from right to left so long as they have defaults.

4.3.2. Using Named Notation

In named notation, each argument's name is specified using => to separate it from the argument expression. For example:

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

Again, the argument `uppercase` was omitted so it is set to `false` implicitly. One advantage of using named notation is that the arguments may be specified in any order, for example:

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)

SELECT concat_lower_or_upper(a => 'Hello', uppercase => true, b => 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

An older syntax based on ":=" is supported for backward compatibility:

```
SELECT concat_lower_or_upper(a := 'Hello', uppercase := true, b := 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

4.3.3. Using Mixed Notation

The mixed notation combines positional and named notation. However, as already mentioned, named arguments cannot precede positional arguments. For example:

```
SELECT concat_lower_or_upper('Hello', 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

In the above query, the arguments `a` and `b` are specified positionally, while `uppercase` is specified by name. In this example, that adds little except documentation. With a more complex function having nu-

merous parameters that have default values, named or mixed notation can save a great deal of writing and reduce chances for error.

> **Note:** Named and mixed call notations currently cannot be used when calling an aggregate function (but they do work when an aggregate function is used as a window function).

Chapter 5. Data Definition

This chapter covers how one creates the database structures that will hold one's data. In a relational database, the raw data is stored in tables, so the majority of this chapter is devoted to explaining how tables are created and modified and what features are available to control what data is stored in the tables. Subsequently, we discuss how tables can be organized into schemas, and how privileges can be assigned to tables. Finally, we will briefly look at other features that affect the data storage, such as inheritance, views, functions, and triggers.

5.1. Table Basics

A table in a relational database is much like a table on paper: It consists of rows and columns. The number and order of the columns is fixed, and each column has a name. The number of rows is variable — it reflects how much data is stored at a given moment. SQL does not make any guarantees about the order of the rows in a table. When a table is read, the rows will appear in an unspecified order, unless sorting is explicitly requested. This is covered in Chapter 7. Furthermore, SQL does not assign unique identifiers to rows, so it is possible to have several completely identical rows in a table. This is a consequence of the mathematical model that underlies SQL but is usually not desirable. Later in this chapter we will see how to deal with this issue.

Each column has a data type. The data type constrains the set of possible values that can be assigned to a column and assigns semantics to the data stored in the column so that it can be used for computations. For instance, a column declared to be of a numerical type will not accept arbitrary text strings, and the data stored in such a column can be used for mathematical computations. By contrast, a column declared to be of a character string type will accept almost any kind of data but it does not lend itself to mathematical calculations, although other operations such as string concatenation are available.

PostgreSQL includes a sizable set of built-in data types that fit many applications. Users can also define their own data types. Most built-in data types have obvious names and semantics, so we defer a detailed explanation to Chapter 8. Some of the frequently used data types are `integer` for whole numbers, `numeric` for possibly fractional numbers, `text` for character strings, `date` for dates, `time` for time-of-day values, and `timestamp` for values containing both date and time.

To create a table, you use the aptly named CREATE TABLE command. In this command you specify at least a name for the new table, the names of the columns and the data type of each column. For example:

```
CREATE TABLE my_first_table (
    first_column text,
    second_column integer
);
```

This creates a table named `my_first_table` with two columns. The first column is named `first_column` and has a data type of `text`; the second column has the name `second_column` and the type `integer`. The table and column names follow the identifier syntax explained in Section 4.1.1. The type names are usually also identifiers, but there are some exceptions. Note that the column list is comma-separated and surrounded by parentheses.

Of course, the previous example was heavily contrived. Normally, you would give names to your tables and columns that convey what kind of data they store. So let's look at a more realistic example:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

(The `numeric` type can store fractional components, as would be typical of monetary amounts.)

> **Tip:** When you create many interrelated tables it is wise to choose a consistent naming pattern for the tables and columns. For instance, there is a choice of using singular or plural nouns for table names, both of which are favored by some theorist or other.

There is a limit on how many columns a table can contain. Depending on the column types, it is between 250 and 1600. However, defining a table with anywhere near this many columns is highly unusual and often a questionable design.

If you no longer need a table, you can remove it using the DROP TABLE command. For example:

```
DROP TABLE my_first_table;
DROP TABLE products;
```

Attempting to drop a table that does not exist is an error. Nevertheless, it is common in SQL script files to unconditionally try to drop each table before creating it, ignoring any error messages, so that the script works whether or not the table exists. (If you like, you can use the `DROP TABLE IF EXISTS` variant to avoid the error messages, but this is not standard SQL.)

If you need to modify a table that already exists, see Section 5.5 later in this chapter.

With the tools discussed so far you can create fully functional tables. The remainder of this chapter is concerned with adding features to the table definition to ensure data integrity, security, or convenience. If you are eager to fill your tables with data now you can skip ahead to Chapter 6 and read the rest of this chapter later.

5.2. Default Values

A column can be assigned a default value. When a new row is created and no values are specified for some of the columns, those columns will be filled with their respective default values. A data manipulation command can also request explicitly that a column be set to its default value, without having to know what that value is. (Details about data manipulation commands are in Chapter 6.)

If no default value is declared explicitly, the default value is the null value. This usually makes sense because a null value can be considered to represent unknown data.

In a table definition, default values are listed after the column data type. For example:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric DEFAULT 9.99
);
```

The default value can be an expression, which will be evaluated whenever the default value is inserted (*not* when the table is created). A common example is for a `timestamp` column to have a default of `CURRENT_TIMESTAMP`, so that it gets set to the time of row insertion. Another common example is generating a "serial number" for each row. In PostgreSQL this is typically done by something like:

```
CREATE TABLE products (
    product_no integer DEFAULT nextval('products_product_no_seq'),
    ...
);
```

where the `nextval()` function supplies successive values from a *sequence object* (see Section 9.16). This arrangement is sufficiently common that there's a special shorthand for it:

```
CREATE TABLE products (
    product_no SERIAL,
    ...
);
```

The `SERIAL` shorthand is discussed further in Section 8.1.4.

5.3. Constraints

Data types are a way to limit the kind of data that can be stored in a table. For many applications, however, the constraint they provide is too coarse. For example, a column containing a product price should probably only accept positive values. But there is no standard data type that accepts only positive numbers. Another issue is that you might want to constrain column data with respect to other columns or rows. For example, in a table containing product information, there should be only one row for each product number.

To that end, SQL allows you to define constraints on columns and tables. Constraints give you as much control over the data in your tables as you wish. If a user attempts to store data in a column that would violate a constraint, an error is raised. This applies even if the value came from the default value definition.

5.3.1. Check Constraints

A check constraint is the most generic constraint type. It allows you to specify that the value in a certain column must satisfy a Boolean (truth-value) expression. For instance, to require positive product prices, you could use:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0)
);
```

As you see, the constraint definition comes after the data type, just like default value definitions. Default values and constraints can be listed in any order. A check constraint consists of the key word CHECK followed by an expression in parentheses. The check constraint expression should involve the column thus constrained, otherwise the constraint would not make too much sense.

You can also give the constraint a separate name. This clarifies error messages and allows you to refer to the constraint when you need to change it. The syntax is:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CONSTRAINT positive_price CHECK (price > 0)
);
```

So, to specify a named constraint, use the key word CONSTRAINT followed by an identifier followed by the constraint definition. (If you don't specify a constraint name in this way, the system chooses a name for you.)

A check constraint can also refer to several columns. Say you store a regular price and a discounted price, and you want to ensure that the discounted price is lower than the regular price:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

The first two constraints should look familiar. The third one uses a new syntax. It is not attached to a particular column, instead it appears as a separate item in the comma-separated column list. Column definitions and these constraint definitions can be listed in mixed order.

We say that the first two constraints are column constraints, whereas the third one is a table constraint because it is written separately from any one column definition. Column constraints can also be written as table constraints, while the reverse is not necessarily possible, since a column constraint is supposed to refer to only the column it is attached to. (PostgreSQL doesn't enforce that rule, but you should follow it if you want your table definitions to work with other database systems.) The above example could also be written as:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

or even:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0 AND price > discounted_price)
);
```

It's a matter of taste.

Names can be assigned to table constraints in the same way as column constraints:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CONSTRAINT valid_discount CHECK (price > discounted_price)
);
```

It should be noted that a check constraint is satisfied if the check expression evaluates to true or the null value. Since most expressions will evaluate to the null value if any operand is null, they will not prevent null values in the constrained columns. To ensure that a column does not contain null values, the not-null constraint described in the next section can be used.

> **Note:** PostgreSQL does not support `CHECK` constraints that reference table data other than the new or updated row being checked. While a `CHECK` constraint that violates this rule may appear to work in simple tests, it cannot guarantee that the database will not reach a state in which the constraint condition is false (due to subsequent changes of the other row(s) involved). This would cause a database dump and reload to fail. The reload could fail even when the complete database state is consistent with the constraint, due to rows not being loaded in an order that will satisfy the constraint. If possible, use `UNIQUE`, `EXCLUDE`, or `FOREIGN KEY` constraints to express cross-row and cross-table restrictions.
>
> If what you desire is a one-time check against other rows at row insertion, rather than a continuously-maintained consistency guarantee, a custom trigger can be used to implement that. (This approach avoids the dump/reload problem because pg_dump does not reinstall triggers until after reloading data, so that the check will not be enforced during a dump/reload.)

> **Note:** PostgreSQL assumes that `CHECK` constraints' conditions are immutable, that is, they will always give the same result for the same input row. This assumption is what justifies examining `CHECK` constraints only when rows are inserted or updated, and not at other times. (The warning above about not referencing other table data is really a special case of this restriction.)
>
> An example of a common way to break this assumption is to reference a user-defined function in a `CHECK` expression, and then change the behavior of that function. PostgreSQL does not disallow that, but it will not notice if there are rows in the table that now violate the `CHECK` constraint. That would cause a subsequent database dump and reload to fail. The recommended way to handle such

a change is to drop the constraint (using `ALTER TABLE`), adjust the function definition, and re-add the constraint, thereby rechecking it against all table rows.

5.3.2. Not-Null Constraints

A not-null constraint simply specifies that a column must not assume the null value. A syntax example:

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric
);
```

A not-null constraint is always written as a column constraint. A not-null constraint is functionally equivalent to creating a check constraint `CHECK (column_name IS NOT NULL)`, but in PostgreSQL creating an explicit not-null constraint is more efficient. The drawback is that you cannot give explicit names to not-null constraints created this way.

Of course, a column can have more than one constraint. Just write the constraints one after another:

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric NOT NULL CHECK (price > 0)
);
```

The order doesn't matter. It does not necessarily determine in which order the constraints are checked.

The `NOT NULL` constraint has an inverse: the `NULL` constraint. This does not mean that the column must be null, which would surely be useless. Instead, this simply selects the default behavior that the column might be null. The `NULL` constraint is not present in the SQL standard and should not be used in portable applications. (It was only added to PostgreSQL to be compatible with some other database systems.) Some users, however, like it because it makes it easy to toggle the constraint in a script file. For example, you could start with:

```
CREATE TABLE products (
    product_no integer NULL,
    name text NULL,
    price numeric NULL
);
```

and then insert the `NOT` key word where desired.

Tip: In most database designs the majority of columns should be marked not null.

5.3.3. Unique Constraints

Unique constraints ensure that the data contained in a column, or a group of columns, is unique among all the rows in the table. The syntax is:

```
CREATE TABLE products (
    product_no integer UNIQUE,
    name text,
    price numeric
);
```

when written as a column constraint, and:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no)
);
```

when written as a table constraint.

To define a unique constraint for a group of columns, write it as a table constraint with the column names separated by commas:

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c)
);
```

This specifies that the combination of values in the indicated columns is unique across the whole table, though any one of the columns need not be (and ordinarily isn't) unique.

You can assign your own name for a unique constraint, in the usual way:

```
CREATE TABLE products (
    product_no integer CONSTRAINT must_be_different UNIQUE,
    name text,
    price numeric
);
```

Adding a unique constraint will automatically create a unique B-tree index on the column or group of columns listed in the constraint. A uniqueness restriction covering only some rows cannot be written as a unique constraint, but it is possible to enforce such a restriction by creating a unique partial index.

In general, a unique constraint is violated if there is more than one row in the table where the values of all of the columns included in the constraint are equal. However, two null values are never considered equal in this comparison. That means even in the presence of a unique constraint it is possible to store duplicate rows that contain a null value in at least one of the constrained columns. This behavior conforms to the SQL standard, but we have heard that other SQL databases might not follow this rule. So be careful when developing applications that are intended to be portable.

5.3.4. Primary Keys

A primary key constraint indicates that a column, or group of columns, can be used as a unique identifier for rows in the table. This requires that the values be both unique and not null. So, the following two table definitions accept the same data:

```
CREATE TABLE products (
    product_no integer UNIQUE NOT NULL,
    name text,
    price numeric
);

CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

Primary keys can span more than one column; the syntax is similar to unique constraints:

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c)
);
```

Adding a primary key will automatically create a unique B-tree index on the column or group of columns listed in the primary key, and will force the column(s) to be marked `NOT NULL`.

A table can have at most one primary key. (There can be any number of unique and not-null constraints, which are functionally almost the same thing, but only one can be identified as the primary key.) Relational database theory dictates that every table must have a primary key. This rule is not enforced by PostgreSQL, but it is usually best to follow it.

Primary keys are useful both for documentation purposes and for client applications. For example, a GUI application that allows modifying row values probably needs to know the primary key of a table to be able to identify rows uniquely. There are also various ways in which the database system makes use of a primary key if one has been declared; for example, the primary key defines the default target column(s) for foreign keys referencing its table.

5.3.5. Foreign Keys

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table. We say this maintains the *referential integrity* between two related tables.

Say you have the product table that we have used several times already:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

Let's also assume you have a table storing orders of those products. We want to ensure that the orders table only contains orders of products that actually exist. So we define a foreign key constraint in the orders table that references the products table:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products (product_no),
    quantity integer
);
```

Now it is impossible to create orders with non-NULL `product_no` entries that do not appear in the products table.

We say that in this situation the orders table is the *referencing* table and the products table is the *referenced* table. Similarly, there are referencing and referenced columns.

You can also shorten the above command to:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products,
    quantity integer
);
```

because in absence of a column list the primary key of the referenced table is used as the referenced column(s).

A foreign key can also constrain and reference a group of columns. As usual, it then needs to be written in table constraint form. Here is a contrived syntax example:

```
CREATE TABLE t1 (
  a integer PRIMARY KEY,
  b integer,
  c integer,
  FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
);
```

Of course, the number and type of the constrained columns need to match the number and type of the referenced columns.

You can assign your own name for a foreign key constraint, in the usual way.

A table can have more than one foreign key constraint. This is used to implement many-to-many relationships between tables. Say you have tables about products and orders, but now you want to allow one order to contain possibly many products (which the structure above did not allow). You could use this table structure:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
```

```
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products,
    order_id integer REFERENCES orders,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

Notice that the primary key overlaps with the foreign keys in the last table.

We know that the foreign keys disallow creation of orders that do not relate to any products. But what if a product is removed after an order is created that references it? SQL allows you to handle that as well. Intuitively, we have a few options:

- Disallow deleting a referenced product
- Delete the orders as well
- Something else?

To illustrate this, let's implement the following policy on the many-to-many relationship example above: when someone wants to remove a product that is still referenced by an order (via `order_items`), we disallow it. If someone removes an order, the order items are removed as well:

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products ON DELETE RESTRICT,
    order_id integer REFERENCES orders ON DELETE CASCADE,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

Restricting and cascading deletes are the two most common options. RESTRICT prevents deletion of a referenced row. NO ACTION means that if any referencing rows still exist when the constraint is checked, an error is raised; this is the default behavior if you do not specify anything. (The essential difference between these two choices is that NO ACTION allows the check to be deferred until later in the transaction, whereas RESTRICT does not.) CASCADE specifies that when a referenced row is deleted, row(s) referencing it should be automatically deleted as well. There are two other options: SET NULL and SET DEFAULT. These cause the referencing column(s) in the referencing row(s) to be set to nulls or their default values, respectively, when the referenced row is deleted. Note that these do not excuse you from observing any constraints. For example, if an action specifies SET DEFAULT but the default value would not satisfy the foreign key constraint, the operation will fail.

Analogous to ON DELETE there is also ON UPDATE which is invoked when a referenced column is changed (updated). The possible actions are the same. In this case, CASCADE means that the updated values of the referenced column(s) should be copied into the referencing row(s).

Normally, a referencing row need not satisfy the foreign key constraint if any of its referencing columns are null. If MATCH FULL is added to the foreign key declaration, a referencing row escapes satisfying the constraint only if all its referencing columns are null (so a mix of null and non-null values is guaranteed to fail a MATCH FULL constraint). If you don't want referencing rows to be able to avoid satisfying the foreign key constraint, declare the referencing column(s) as NOT NULL.

A foreign key must reference columns that either are a primary key or form a unique constraint. This means that the referenced columns always have an index (the one underlying the primary key or unique constraint); so checks on whether a referencing row has a match will be efficient. Since a DELETE of a row from the referenced table or an UPDATE of a referenced column will require a scan of the referencing table for rows matching the old value, it is often a good idea to index the referencing columns too. Because this is not always needed, and there are many choices available on how to index, declaration of a foreign key constraint does not automatically create an index on the referencing columns.

More information about updating and deleting data is in Chapter 6. Also see the description of foreign key constraint syntax in the reference documentation for CREATE TABLE.

5.3.6. Exclusion Constraints

Exclusion constraints ensure that if any two rows are compared on the specified columns or expressions using the specified operators, at least one of these operator comparisons will return false or null. The syntax is:

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

See also CREATE TABLE ... CONSTRAINT ... EXCLUDE for details.

Adding an exclusion constraint will automatically create an index of the type specified in the constraint declaration.

5.4. System Columns

Every table has several *system columns* that are implicitly defined by the system. Therefore, these names cannot be used as names of user-defined columns. (Note that these restrictions are separate from whether the name is a key word or not; quoting a name will not allow you to escape these restrictions.) You do not really need to be concerned about these columns; just know they exist.

`oid`

The object identifier (object ID) of a row. This column is only present if the table was created using `WITH OIDS`, or if the default_with_oids configuration variable was set at the time. This column is of type `oid` (same name as the column); see Section 8.18 for more information about the type.

`tableoid`

The OID of the table containing this row. This column is particularly handy for queries that select from inheritance hierarchies (see Section 5.9), since without it, it's difficult to tell which individual table a row came from. The `tableoid` can be joined against the `oid` column of `pg_class` to obtain the table name.

`xmin`

The identity (transaction ID) of the inserting transaction for this row version. (A row version is an individual state of a row; each update of a row creates a new row version for the same logical row.)

`cmin`

The command identifier (starting at zero) within the inserting transaction.

`xmax`

The identity (transaction ID) of the deleting transaction, or zero for an undeleted row version. It is possible for this column to be nonzero in a visible row version. That usually indicates that the deleting transaction hasn't committed yet, or that an attempted deletion was rolled back.

`cmax`

The command identifier within the deleting transaction, or zero.

`ctid`

The physical location of the row version within its table. Note that although the `ctid` can be used to locate the row version very quickly, a row's `ctid` will change if it is updated or moved by `VACUUM FULL`. Therefore `ctid` is useless as a long-term row identifier. The OID, or even better a user-defined serial number, should be used to identify logical rows.

OIDs are 32-bit quantities and are assigned from a single cluster-wide counter. In a large or long-lived database, it is possible for the counter to wrap around. Hence, it is bad practice to assume that OIDs are unique, unless you take steps to ensure that this is the case. If you need to identify the rows in a table, using a sequence generator is strongly recommended. However, OIDs can be used as well, provided that a few additional precautions are taken:

- A unique constraint should be created on the OID column of each table for which the OID will be used to identify rows. When such a unique constraint (or unique index) exists, the system takes care not to generate an OID matching an already-existing row. (Of course, this is only possible if the table contains fewer than 2^{32} (4 billion) rows, and in practice the table size had better be much less than that, or performance might suffer.)

- OIDs should never be assumed to be unique across tables; use the combination of `tableoid` and row OID if you need a database-wide identifier.
- Of course, the tables in question must be created `WITH OIDS`. As of PostgreSQL 8.1, `WITHOUT OIDS` is the default.

Transaction identifiers are also 32-bit quantities. In a long-lived database it is possible for transaction IDs to wrap around. This is not a fatal problem given appropriate maintenance procedures; see Chapter 23 for details. It is unwise, however, to depend on the uniqueness of transaction IDs over the long term (more than one billion transactions).

Command identifiers are also 32-bit quantities. This creates a hard limit of 2^{32} (4 billion) SQL commands within a single transaction. In practice this limit is not a problem — note that the limit is on the number of SQL commands, not the number of rows processed. Also, only commands that actually modify the database contents will consume a command identifier.

5.5. Modifying Tables

When you create a table and you realize that you made a mistake, or the requirements of the application change, you can drop the table and create it again. But this is not a convenient option if the table is already filled with data, or if the table is referenced by other database objects (for instance a foreign key constraint). Therefore PostgreSQL provides a family of commands to make modifications to existing tables. Note that this is conceptually distinct from altering the data contained in the table: here we are interested in altering the definition, or structure, of the table.

You can:

- Add columns
- Remove columns
- Add constraints
- Remove constraints
- Change default values
- Change column data types
- Rename columns
- Rename tables

All these actions are performed using the ALTER TABLE command, whose reference page contains details beyond those given here.

5.5.1. Adding a Column

To add a column, use a command like:

```
ALTER TABLE products ADD COLUMN description text;
```

The new column is initially filled with whatever default value is given (null if you don't specify a `DEFAULT` clause).

You can also define constraints on the column at the same time, using the usual syntax:

```
ALTER TABLE products ADD COLUMN description text CHECK (description <> '');
```

In fact all the options that can be applied to a column description in `CREATE TABLE` can be used here. Keep in mind however that the default value must satisfy the given constraints, or the `ADD` will fail. Alternatively, you can add constraints later (see below) after you've filled in the new column correctly.

> **Tip:** Adding a column with a default requires updating each row of the table (to store the new column value). However, if no default is specified, PostgreSQL is able to avoid the physical update. So if you intend to fill the column with mostly nondefault values, it's best to add the column with no default, insert the correct values using `UPDATE`, and then add any desired default as described below.

5.5.2. Removing a Column

To remove a column, use a command like:

```
ALTER TABLE products DROP COLUMN description;
```

Whatever data was in the column disappears. Table constraints involving the column are dropped, too. However, if the column is referenced by a foreign key constraint of another table, PostgreSQL will not silently drop that constraint. You can authorize dropping everything that depends on the column by adding `CASCADE`:

```
ALTER TABLE products DROP COLUMN description CASCADE;
```

See Section 5.13 for a description of the general mechanism behind this.

5.5.3. Adding a Constraint

To add a constraint, the table constraint syntax is used. For example:

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES product_groups;
```

To add a not-null constraint, which cannot be written as a table constraint, use this syntax:

```
ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;
```

The constraint will be checked immediately, so the table data must satisfy the constraint before it can be added.

5.5.4. Removing a Constraint

To remove a constraint you need to know its name. If you gave it a name then that's easy. Otherwise the system assigned a generated name, which you need to find out. The psql command `\d tablename` can be helpful here; other interfaces might also provide a way to inspect table details. Then the command is:

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

(If you are dealing with a generated constraint name like `$2`, don't forget that you'll need to double-quote it to make it a valid identifier.)

As with dropping a column, you need to add `CASCADE` if you want to drop a constraint that something else depends on. An example is that a foreign key constraint depends on a unique or primary key constraint on the referenced column(s).

This works the same for all constraint types except not-null constraints. To drop a not null constraint use:

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

(Recall that not-null constraints do not have names.)

5.5.5. Changing a Column's Default Value

To set a new default for a column, use a command like:

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

Note that this doesn't affect any existing rows in the table, it just changes the default for future `INSERT` commands.

To remove any default value, use:

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

This is effectively the same as setting the default to null. As a consequence, it is not an error to drop a default where one hadn't been defined, because the default is implicitly the null value.

5.5.6. Changing a Column's Data Type

To convert a column to a different data type, use a command like:

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

This will succeed only if each existing entry in the column can be converted to the new type by an implicit cast. If a more complex conversion is needed, you can add a `USING` clause that specifies how to compute the new values from the old.

PostgreSQL will attempt to convert the column's default value (if any) to the new type, as well as any constraints that involve the column. But these conversions might fail, or might produce surprising results. It's often best to drop any constraints on the column before altering its type, and then add back suitably modified constraints afterwards.

5.5.7. Renaming a Column

To rename a column:

```
ALTER TABLE products RENAME COLUMN product_no TO product_number;
```

5.5.8. Renaming a Table

To rename a table:

```
ALTER TABLE products RENAME TO items;
```

5.6. Privileges

When an object is created, it is assigned an owner. The owner is normally the role that executed the creation statement. For most kinds of objects, the initial state is that only the owner (or a superuser) can do anything with the object. To allow other roles to use it, *privileges* must be granted.

There are different kinds of privileges: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, `REFERENCES`, `TRIGGER`, `CREATE`, `CONNECT`, `TEMPORARY`, `EXECUTE`, and `USAGE`. The privileges applicable to a particular object vary depending on the object's type (table, function, etc). For complete information on the different types of privileges supported by PostgreSQL, refer to the GRANT reference page. The following sections and chapters will also show you how those privileges are used.

The right to modify or destroy an object is always the privilege of the owner only.

An object can be assigned to a new owner with an `ALTER` command of the appropriate kind for the object, e.g., ALTER TABLE. Superusers can always do this; ordinary roles can only do it if they are both the current owner of the object (or a member of the owning role) and a member of the new owning role.

To assign privileges, the `GRANT` command is used. For example, if `joe` is an existing user, and `accounts` is an existing table, the privilege to update the table can be granted with:

```
GRANT UPDATE ON accounts TO joe;
```

Writing `ALL` in place of a specific privilege grants all privileges that are relevant for the object type.

The special "user" name `PUBLIC` can be used to grant a privilege to every user on the system. Also, "group" roles can be set up to help manage privileges when there are many users of a database — for details see Chapter 20.

To revoke a privilege, use the fittingly named `REVOKE` command:

```
REVOKE ALL ON accounts FROM PUBLIC;
```

The special privileges of the object owner (i.e., the right to do DROP, GRANT, REVOKE, etc.) are always implicit in being the owner, and cannot be granted or revoked. But the object owner can choose to revoke his own ordinary privileges, for example to make a table read-only for himself as well as others.

Ordinarily, only the object's owner (or a superuser) can grant or revoke privileges on an object. However, it is possible to grant a privilege "with grant option", which gives the recipient the right to grant it in turn to others. If the grant option is subsequently revoked then all who received the privilege from that recipient (directly or through a chain of grants) will lose the privilege. For details see the GRANT and REVOKE reference pages.

5.7. Row Security Policies

In addition to the SQL-standard privilege system available through GRANT, tables can have *row security policies* that restrict, on a per-user basis, which rows can be returned by normal queries or inserted, updated, or deleted by data modification commands. This feature is also known as *Row-Level Security*. By default, tables do not have any policies, so that if a user has access privileges to a table according to the SQL privilege system, all rows within it are equally available for querying or updating.

When row security is enabled on a table (with ALTER TABLE ... ENABLE ROW LEVEL SECURITY), all normal access to the table for selecting rows or modifying rows must be allowed by a row security policy. (However, the table's owner is typically not subject to row security policies.) If no policy exists for the table, a default-deny policy is used, meaning that no rows are visible or can be modified. Operations that apply to the whole table, such as TRUNCATE and REFERENCES, are not subject to row security.

Row security policies can be specific to commands, or to roles, or to both. A policy can be specified to apply to ALL commands, or to SELECT, INSERT, UPDATE, or DELETE. Multiple roles can be assigned to a given policy, and normal role membership and inheritance rules apply.

To specify which rows are visible or modifiable according to a policy, an expression is required that returns a Boolean result. This expression will be evaluated for each row prior to any conditions or functions coming from the user's query. (The only exceptions to this rule are leakproof functions, which are guaranteed to not leak information; the optimizer may choose to apply such functions ahead of the row-security check.) Rows for which the expression does not return true will not be processed. Separate expressions may be specified to provide independent control over the rows which are visible and the rows which are allowed to be modified. Policy expressions are run as part of the query and with the privileges of the user running the query, although security-definer functions can be used to access data not available to the calling user.

Superusers and roles with the BYPASSRLS attribute always bypass the row security system when accessing a table. Table owners normally bypass row security as well, though a table owner can choose to be subject to row security with ALTER TABLE ... FORCE ROW LEVEL SECURITY.

Enabling and disabling row security, as well as adding policies to a table, is always the privilege of the table owner only.

Policies are created using the CREATE POLICY command, altered using the ALTER POLICY command, and dropped using the DROP POLICY command. To enable and disable row security for a given table, use the ALTER TABLE command.

Each policy has a name and multiple policies can be defined for a table. As policies are table-specific, each policy for a table must have a unique name. Different tables may have policies with the same name.

When multiple policies apply to a given query, they are combined using `OR`, so that a row is accessible if any policy allows it. This is similar to the rule that a given role has the privileges of all roles that they are a member of.

As a simple example, here is how to create a policy on the `account` relation to allow only members of the `managers` role to access rows, and only rows of their accounts:

```
CREATE TABLE accounts (manager text, company text, contact_email text);

ALTER TABLE accounts ENABLE ROW LEVEL SECURITY;

CREATE POLICY account_managers ON accounts TO managers
    USING (manager = current_user);
```

The policy above implicitly provides a `WITH CHECK` clause identical to its `USING` clause, so that the constraint applies both to rows selected by a command (so a manager cannot `SELECT`, `UPDATE`, or `DELETE` existing rows belonging to a different manager) and to rows modified by a command (so rows belonging to a different manager cannot be created via `INSERT` or `UPDATE`).

If no role is specified, or the special user name `PUBLIC` is used, then the policy applies to all users on the system. To allow all users to access only their own row in a `users` table, a simple policy can be used:

```
CREATE POLICY user_policy ON users
    USING (user_name = current_user);
```

This works similarly to the previous example.

To use a different policy for rows that are being added to the table compared to those rows that are visible, multiple policies can be combined. This pair of policies would allow all users to view all rows in the `users` table, but only modify their own:

```
CREATE POLICY user_sel_policy ON users
    FOR SELECT
    USING (true);
CREATE POLICY user_mod_policy ON users
    USING (user_name = current_user);
```

In a `SELECT` command, these two policies are combined using `OR`, with the net effect being that all rows can be selected. In other command types, only the second policy applies, so that the effects are the same as before.

Row security can also be disabled with the `ALTER TABLE` command. Disabling row security does not remove any policies that are defined on the table; they are simply ignored. Then all rows in the table are visible and modifiable, subject to the standard SQL privileges system.

Below is a larger example of how this feature can be used in production environments. The table `passwd` emulates a Unix password file:

```
-- Simple passwd-file based example
CREATE TABLE passwd (
  user_name             text UNIQUE NOT NULL,
  pwhash                text,
  uid                   int  PRIMARY KEY,
  gid                   int  NOT NULL,
```

```
  real_name             text NOT NULL,
  home_phone            text,
  extra_info            text,
  home_dir              text NOT NULL,
  shell                 text NOT NULL
);

CREATE ROLE admin;  -- Administrator
CREATE ROLE bob;    -- Normal user
CREATE ROLE alice;  -- Normal user

-- Populate the table
INSERT INTO passwd VALUES
  ('admin','xxx',0,0,'Admin','111-222-3333',null,'/root','/bin/dash');
INSERT INTO passwd VALUES
  ('bob','xxx',1,1,'Bob','123-456-7890',null,'/home/bob','/bin/zsh');
INSERT INTO passwd VALUES
  ('alice','xxx',2,1,'Alice','098-765-4321',null,'/home/alice','/bin/zsh');

-- Be sure to enable row level security on the table
ALTER TABLE passwd ENABLE ROW LEVEL SECURITY;

-- Create policies
-- Administrator can see all rows and add any rows
CREATE POLICY admin_all ON passwd TO admin USING (true) WITH CHECK (true);
-- Normal users can view all rows
CREATE POLICY all_view ON passwd FOR SELECT USING (true);
-- Normal users can update their own records, but
-- limit which shells a normal user is allowed to set
CREATE POLICY user_mod ON passwd FOR UPDATE
  USING (current_user = user_name)
  WITH CHECK (
    current_user = user_name AND
    shell IN ('/bin/bash','/bin/sh','/bin/dash','/bin/zsh','/bin/tcsh')
  );

-- Allow admin all normal rights
GRANT SELECT, INSERT, UPDATE, DELETE ON passwd TO admin;
-- Users only get select access on public columns
GRANT SELECT
  (user_name, uid, gid, real_name, home_phone, extra_info, home_dir, shell)
  ON passwd TO public;
-- Allow users to update certain columns
GRANT UPDATE
  (pwhash, real_name, home_phone, extra_info, shell)
  ON passwd TO public;
```

As with any security settings, it's important to test and ensure that the system is behaving as expected. Using the example above, this demonstrates that the permission system is working properly.

```
-- admin can view all rows and fields
postgres=> set role admin;
SET
```

```
postgres=> table passwd;
 user_name | pwhash | uid | gid | real_name |  home_phone  | extra_info | home_dir    |   s
-----------+--------+-----+-----+-----------+--------------+------------+-------------+----
 admin     | xxx    |   0 |   0 | Admin     | 111-222-3333 |            | /root       | /bi
 bob       | xxx    |   1 |   1 | Bob       | 123-456-7890 |            | /home/bob   | /bi
 alice     | xxx    |   2 |   1 | Alice     | 098-765-4321 |            | /home/alice | /bi
(3 rows)

-- Test what Alice is able to do
postgres=> set role alice;
SET
postgres=> table passwd;
ERROR:  permission denied for relation passwd
postgres=> select user_name,real_name,home_phone,extra_info,home_dir,shell from passwd;
 user_name | real_name |  home_phone  | extra_info | home_dir    |   shell
-----------+-----------+--------------+------------+-------------+-----------
 admin     | Admin     | 111-222-3333 |            | /root       | /bin/dash
 bob       | Bob       | 123-456-7890 |            | /home/bob   | /bin/zsh
 alice     | Alice     | 098-765-4321 |            | /home/alice | /bin/zsh
(3 rows)

postgres=> update passwd set user_name = 'joe';
ERROR:  permission denied for relation passwd
-- Alice is allowed to change her own real_name, but no others
postgres=> update passwd set real_name = 'Alice Doe';
UPDATE 1
postgres=> update passwd set real_name = 'John Doe' where user_name = 'admin';
UPDATE 0
postgres=> update passwd set shell = '/bin/xx';
ERROR:  new row violates WITH CHECK OPTION for "passwd"
postgres=> delete from passwd;
ERROR:  permission denied for relation passwd
postgres=> insert into passwd (user_name) values ('xxx');
ERROR:  permission denied for relation passwd
-- Alice can change her own password; RLS silently prevents updating other rows
postgres=> update passwd set pwhash = 'abc';
UPDATE 1
```

Referential integrity checks, such as unique or primary key constraints and foreign key references, always bypass row security to ensure that data integrity is maintained. Care must be taken when developing schemas and row level policies to avoid "covert channel" leaks of information through such referential integrity checks.

In some contexts it is important to be sure that row security is not being applied. For example, when taking a backup, it could be disastrous if row security silently caused some rows to be omitted from the backup. In such a situation, you can set the row_security configuration parameter to `off`. This does not in itself bypass row security; what it does is throw an error if any query's results would get filtered by a policy. The reason for the error can then be investigated and fixed.

In the examples above, the policy expressions consider only the current values in the row to be accessed or updated. This is the simplest and best-performing case; when possible, it's best to design row security applications to work this way. If it is necessary to consult other rows or other tables to make a policy decision, that can be accomplished using sub-`SELECT`s, or functions that contain `SELECT`s, in the policy

expressions. Be aware however that such accesses can create race conditions that could allow information leakage if care is not taken. As an example, consider the following table design:

```
-- definition of privilege groups
CREATE TABLE groups (group_id int PRIMARY KEY,
                     group_name text NOT NULL);

INSERT INTO groups VALUES
  (1, 'low'),
  (2, 'medium'),
  (5, 'high');

GRANT ALL ON groups TO alice;  -- alice is the administrator
GRANT SELECT ON groups TO public;

-- definition of users' privilege levels
CREATE TABLE users (user_name text PRIMARY KEY,
                    group_id int NOT NULL REFERENCES groups);

INSERT INTO users VALUES
  ('alice', 5),
  ('bob', 2),
  ('mallory', 2);

GRANT ALL ON users TO alice;
GRANT SELECT ON users TO public;

-- table holding the information to be protected
CREATE TABLE information (info text,
                          group_id int NOT NULL REFERENCES groups);

INSERT INTO information VALUES
  ('barely secret', 1),
  ('slightly secret', 2),
  ('very secret', 5);

ALTER TABLE information ENABLE ROW LEVEL SECURITY;

-- a row should be visible to/updatable by users whose security group_id is
-- greater than or equal to the row's group_id
CREATE POLICY fp_s ON information FOR SELECT
  USING (group_id <= (SELECT group_id FROM users WHERE user_name = current_user));
CREATE POLICY fp_u ON information FOR UPDATE
  USING (group_id <= (SELECT group_id FROM users WHERE user_name = current_user));

-- we rely only on RLS to protect the information table
GRANT ALL ON information TO public;
```

Now suppose that `alice` wishes to change the "slightly secret" information, but decides that `mallory` should not be trusted with the new content of that row, so she does:

```
BEGIN;
UPDATE users SET group_id = 1 WHERE user_name = 'mallory';
```

```
UPDATE information SET info = 'secret from mallory' WHERE group_id = 2;
COMMIT;
```

That looks safe; there is no window wherein `mallory` should be able to see the "secret from mallory" string. However, there is a race condition here. If `mallory` is concurrently doing, say,

```
SELECT * FROM information WHERE group_id = 2 FOR UPDATE;
```

and her transaction is in `READ COMMITTED` mode, it is possible for her to see "secret from mallory". That happens if her transaction reaches the `information` row just after `alice`'s does. It blocks waiting for `alice`'s transaction to commit, then fetches the updated row contents thanks to the `FOR UPDATE` clause. However, it does *not* fetch an updated row for the implicit `SELECT` from `users`, because that sub-`SELECT` did not have `FOR UPDATE`; instead the `users` row is read with the snapshot taken at the start of the query. Therefore, the policy expression tests the old value of `mallory`'s privilege level and allows her to see the updated row.

There are several ways around this problem. One simple answer is to use `SELECT ... FOR SHARE` in sub-`SELECT`s in row security policies. However, that requires granting `UPDATE` privilege on the referenced table (here `users`) to the affected users, which might be undesirable. (But another row security policy could be applied to prevent them from actually exercising that privilege; or the sub-`SELECT` could be embedded into a security definer function.) Also, heavy concurrent use of row share locks on the referenced table could pose a performance problem, especially if updates of it are frequent. Another solution, practical if updates of the referenced table are infrequent, is to take an exclusive lock on the referenced table when updating it, so that no concurrent transactions could be examining old row values. Or one could just wait for all concurrent transactions to end after committing an update of the referenced table and before making changes that rely on the new security situation.

For additional details see CREATE POLICY and ALTER TABLE.

5.8. Schemas

A PostgreSQL database cluster contains one or more named databases. Roles and a few other object types are shared across the entire cluster. A client connection to the server can only access data in a single database, the one specified in the connection request.

> **Note:** Users of a cluster do not necessarily have the privilege to access every database in the cluster. Sharing of role names means that there cannot be different roles named, say, `joe` in two databases in the same cluster; but the system can be configured to allow `joe` access to only some of the databases.

A database contains one or more named *schemas*, which in turn contain tables. Schemas also contain other kinds of named objects, including data types, functions, and operators. The same object name can be used in different schemas without conflict; for example, both `schema1` and `myschema` can contain tables named `mytable`. Unlike databases, schemas are not rigidly separated: a user can access objects in any of the schemas in the database he is connected to, if he has privileges to do so.

There are several reasons why one might want to use schemas:

- To allow many users to use one database without interfering with each other.

- To organize database objects into logical groups to make them more manageable.
- Third-party applications can be put into separate schemas so they do not collide with the names of other objects.

Schemas are analogous to directories at the operating system level, except that schemas cannot be nested.

5.8.1. Creating a Schema

To create a schema, use the CREATE SCHEMA command. Give the schema a name of your choice. For example:

```
CREATE SCHEMA myschema;
```

To create or access objects in a schema, write a *qualified name* consisting of the schema name and table name separated by a dot:

```
schema.table
```

This works anywhere a table name is expected, including the table modification commands and the data access commands discussed in the following chapters. (For brevity we will speak of tables only, but the same ideas apply to other kinds of named objects, such as types and functions.)

Actually, the even more general syntax

```
database.schema.table
```

can be used too, but at present this is just for *pro forma* compliance with the SQL standard. If you write a database name, it must be the same as the database you are connected to.

So to create a table in the new schema, use:

```
CREATE TABLE myschema.mytable (
 ...
);
```

To drop a schema if it's empty (all objects in it have been dropped), use:

```
DROP SCHEMA myschema;
```

To drop a schema including all contained objects, use:

```
DROP SCHEMA myschema CASCADE;
```

See Section 5.13 for a description of the general mechanism behind this.

Often you will want to create a schema owned by someone else (since this is one of the ways to restrict the activities of your users to well-defined namespaces). The syntax for that is:

```
CREATE SCHEMA schema_name AUTHORIZATION user_name;
```

You can even omit the schema name, in which case the schema name will be the same as the user name. See Section 5.8.6 for how this can be useful.

Schema names beginning with `pg_` are reserved for system purposes and cannot be created by users.

5.8.2. The Public Schema

In the previous sections we created tables without specifying any schema names. By default such tables (and other objects) are automatically put into a schema named "public". Every new database contains such a schema. Thus, the following are equivalent:

```
CREATE TABLE products ( ... );
```

and:

```
CREATE TABLE public.products ( ... );
```

5.8.3. The Schema Search Path

Qualified names are tedious to write, and it's often best not to wire a particular schema name into applications anyway. Therefore tables are often referred to by *unqualified names*, which consist of just the table name. The system determines which table is meant by following a *search path*, which is a list of schemas to look in. The first matching table in the search path is taken to be the one wanted. If there is no match in the search path, an error is reported, even if matching table names exist in other schemas in the database.

The ability to create like-named objects in different schemas complicates writing a query that references precisely the same objects every time. It also opens up the potential for users to change the behavior of other users' queries, maliciously or accidentally. Due to the prevalence of unqualified names in queries and their use in PostgreSQL internals, adding a schema to `search_path` effectively trusts all users having `CREATE` privilege on that schema. When you run an ordinary query, a malicious user able to create objects in a schema of your search path can take control and execute arbitrary SQL functions as though you executed them.

The first schema named in the search path is called the current schema. Aside from being the first schema searched, it is also the schema in which new tables will be created if the `CREATE TABLE` command does not specify a schema name.

To show the current search path, use the following command:

```
SHOW search_path;
```

In the default setup this returns:

```
 search_path
--------------
 "$user", public
```

The first element specifies that a schema with the same name as the current user is to be searched. If no such schema exists, the entry is ignored. The second element refers to the public schema that we have seen already.

The first schema in the search path that exists is the default location for creating new objects. That is the reason that by default objects are created in the public schema. When objects are referenced in any other context without schema qualification (table modification, data modification, or query commands) the search path is traversed until a matching object is found. Therefore, in the default configuration, any unqualified access again can only refer to the public schema.

To put our new schema in the path, we use:

```
SET search_path TO myschema,public;
```

(We omit the `$user` here because we have no immediate need for it.) And then we can access the table without schema qualification:

```
DROP TABLE mytable;
```

Also, since `myschema` is the first element in the path, new objects would by default be created in it.

We could also have written:

```
SET search_path TO myschema;
```

Then we no longer have access to the public schema without explicit qualification. There is nothing special about the public schema except that it exists by default. It can be dropped, too.

See also Section 9.25 for other ways to manipulate the schema search path.

The search path works in the same way for data type names, function names, and operator names as it does for table names. Data type and function names can be qualified in exactly the same way as table names. If you need to write a qualified operator name in an expression, there is a special provision: you must write

```
OPERATOR(schema.operator)
```

This is needed to avoid syntactic ambiguity. An example is:

```
SELECT 3 OPERATOR(pg_catalog.+) 4;
```

In practice one usually relies on the search path for operators, so as not to have to write anything so ugly as that.

5.8.4. Schemas and Privileges

By default, users cannot access any objects in schemas they do not own. To allow that, the owner of the schema must grant the `USAGE` privilege on the schema. To allow users to make use of the objects in the schema, additional privileges might need to be granted, as appropriate for the object.

A user can also be allowed to create objects in someone else's schema. To allow that, the `CREATE` privilege on the schema needs to be granted. Note that by default, everyone has `CREATE` and `USAGE` privileges on

the schema public. This allows all users that are able to connect to a given database to create objects in its public schema. Some usage patterns call for revoking that privilege:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

(The first "public" is the schema, the second "public" means "every user". In the first sense it is an identifier, in the second sense it is a key word, hence the different capitalization; recall the guidelines from Section 4.1.1.)

5.8.5. The System Catalog Schema

In addition to public and user-created schemas, each database contains a pg_catalog schema, which contains the system tables and all the built-in data types, functions, and operators. pg_catalog is always effectively part of the search path. If it is not named explicitly in the path then it is implicitly searched *before* searching the path's schemas. This ensures that built-in names will always be findable. However, you can explicitly place pg_catalog at the end of your search path if you prefer to have user-defined names override built-in names.

Since system table names begin with pg_, it is best to avoid such names to ensure that you won't suffer a conflict if some future version defines a system table named the same as your table. (With the default search path, an unqualified reference to your table name would then be resolved as the system table instead.) System tables will continue to follow the convention of having names beginning with pg_, so that they will not conflict with unqualified user-table names so long as users avoid the pg_ prefix.

5.8.6. Usage Patterns

Schemas can be used to organize your data in many ways. A *secure schema usage pattern* prevents untrusted users from changing the behavior of other users' queries. When a database does not use a secure schema usage pattern, users wishing to securely query that database would take protective action at the beginning of each session. Specifically, they would begin each session by setting search_path to the empty string or otherwise removing non-superuser-writable schemas from search_path. There are a few usage patterns easily supported by the default configuration:

- Constrain ordinary users to user-private schemas. To implement this, issue REVOKE CREATE ON SCHEMA public FROM PUBLIC, and create a schema for each user with the same name as that user. Recall that the default search path starts with $user, which resolves to the user name. Therefore, if each user has a separate schema, they access their own schemas by default. After adopting this pattern in a database where untrusted users had already logged in, consider auditing the public schema for objects named like objects in schema pg_catalog. This pattern is a secure schema usage pattern unless an untrusted user is the database owner or holds the CREATEROLE privilege, in which case no secure schema usage pattern exists.

- Remove the public schema from the default search path, by modifying postgresql.conf or by issuing ALTER ROLE ALL SET search_path = "$user". Everyone retains the ability to create objects in the public schema, but only qualified names will choose those objects. While qualified table references are fine, calls to functions in the public schema will be unsafe or unreliable. If you create functions

or extensions in the public schema, use the first pattern instead. Otherwise, like the first pattern, this is secure unless an untrusted user is the database owner or holds the `CREATEROLE` privilege.

- Keep the default. All users access the public schema implicitly. This simulates the situation where schemas are not available at all, giving a smooth transition from the non-schema-aware world. However, this is never a secure pattern. It is acceptable only when the database has a single user or a few mutually-trusting users.

For any pattern, to install shared applications (tables to be used by everyone, additional functions provided by third parties, etc.), put them into separate schemas. Remember to grant appropriate privileges to allow the other users to access them. Users can then refer to these additional objects by qualifying the names with a schema name, or they can put the additional schemas into their search path, as they choose.

5.8.7. Portability

In the SQL standard, the notion of objects in the same schema being owned by different users does not exist. Moreover, some implementations do not allow you to create schemas that have a different name than their owner. In fact, the concepts of schema and user are nearly equivalent in a database system that implements only the basic schema support specified in the standard. Therefore, many users consider qualified names to really consist of *`user_name`*`.`*`table_name`*. This is how PostgreSQL will effectively behave if you create a per-user schema for every user.

Also, there is no concept of a `public` schema in the SQL standard. For maximum conformance to the standard, you should not use the `public` schema.

Of course, some SQL database systems might not implement schemas at all, or provide namespace support by allowing (possibly limited) cross-database access. If you need to work with those systems, then maximum portability would be achieved by not using schemas at all.

5.9. Inheritance

PostgreSQL implements table inheritance, which can be a useful tool for database designers. (SQL:1999 and later define a type inheritance feature, which differs in many respects from the features described here.)

Let's start with an example: suppose we are trying to build a data model for cities. Each state has many cities, but only one capital. We want to be able to quickly retrieve the capital city for any particular state. This can be done by creating two tables, one for state capitals and one for cities that are not capitals. However, what happens when we want to ask for data about a city, regardless of whether it is a capital or not? The inheritance feature can help to resolve this problem. We define the `capitals` table so that it inherits from `cities`:

```
CREATE TABLE cities (
    name            text,
    population      float,
    elevation       int     -- in feet
);
```

```
CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

In this case, the `capitals` table *inherits* all the columns of its parent table, `cities`. State capitals also have an extra column, `state`, that shows their state.

In PostgreSQL, a table can inherit from zero or more other tables, and a query can reference either all rows of a table or all rows of a table plus all of its descendant tables. The latter behavior is the default. For example, the following query finds the names of all cities, including state capitals, that are located at an elevation over 500 feet:

```
SELECT name, elevation
    FROM cities
    WHERE elevation > 500;
```

Given the sample data from the PostgreSQL tutorial (see Section 2.1), this returns:

```
   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
 Madison   |       845
```

On the other hand, the following query finds all the cities that are not state capitals and are situated at an elevation over 500 feet:

```
SELECT name, elevation
    FROM ONLY cities
    WHERE elevation > 500;

   name    | elevation
-----------+-----------
 Las Vegas |      2174
 Mariposa  |      1953
```

Here the `ONLY` keyword indicates that the query should apply only to `cities`, and not any tables below `cities` in the inheritance hierarchy. Many of the commands that we have already discussed — `SELECT`, `UPDATE` and `DELETE` — support the `ONLY` keyword.

You can also write the table name with a trailing `*` to explicitly specify that descendant tables are included:

```
SELECT name, elevation
    FROM cities*
    WHERE elevation > 500;
```

Writing `*` is not necessary, since this behavior is the default (unless you have changed the setting of the sql_inheritance configuration option). However writing `*` might be useful to emphasize that additional tables will be searched.

In some cases you might wish to know which table a particular row originated from. There is a system column called `tableoid` in each table which can tell you the originating table:

```
SELECT c.tableoid, c.name, c.elevation
FROM cities c
WHERE c.elevation > 500;
```

which returns:

```
 tableoid |   name    | elevation
----------+-----------+-----------
   139793 | Las Vegas |      2174
   139793 | Mariposa  |      1953
   139798 | Madison   |       845
```

(If you try to reproduce this example, you will probably get different numeric OIDs.) By doing a join with `pg_class` you can see the actual table names:

```
SELECT p.relname, c.name, c.elevation
FROM cities c, pg_class p
WHERE c.elevation > 500 AND c.tableoid = p.oid;
```

which returns:

```
 relname  |   name    | elevation
----------+-----------+-----------
 cities   | Las Vegas |      2174
 cities   | Mariposa  |      1953
 capitals | Madison   |       845
```

Another way to get the same effect is to use the `regclass` pseudo-type, which will print the table OID symbolically:

```
SELECT c.tableoid::regclass, c.name, c.elevation
FROM cities c
WHERE c.elevation > 500;
```

Inheritance does not automatically propagate data from `INSERT` or `COPY` commands to other tables in the inheritance hierarchy. In our example, the following `INSERT` statement will fail:

```
INSERT INTO cities (name, population, elevation, state)
VALUES ('Albany', NULL, NULL, 'NY');
```

We might hope that the data would somehow be routed to the `capitals` table, but this does not happen: `INSERT` always inserts into exactly the table specified. In some cases it is possible to redirect the insertion using a rule (see Chapter 38). However that does not help for the above case because the `cities` table does not contain the column `state`, and so the command will be rejected before the rule can be applied.

All check constraints and not-null constraints on a parent table are automatically inherited by its children, unless explicitly specified otherwise with `NO INHERIT` clauses. Other types of constraints (unique, primary key, and foreign key constraints) are not inherited.

A table can inherit from more than one parent table, in which case it has the union of the columns defined by the parent tables. Any columns declared in the child table's definition are added to these. If the same column name appears in multiple parent tables, or in both a parent table and the child's definition, then these columns are "merged" so that there is only one such column in the child table. To be merged, columns must have the same data types, else an error is raised. Inheritable check constraints and not-null constraints are merged in a similar fashion. Thus, for example, a merged column will be marked not-null if any one of the column definitions it came from is marked not-null. Check constraints are merged if they have the same name, and the merge will fail if their conditions are different.

Table inheritance is typically established when the child table is created, using the `INHERITS` clause of the CREATE TABLE statement. Alternatively, a table which is already defined in a compatible way can have a new parent relationship added, using the `INHERIT` variant of ALTER TABLE. To do this the new child table must already include columns with the same names and types as the columns of the parent. It must also include check constraints with the same names and check expressions as those of the parent. Similarly an inheritance link can be removed from a child using the `NO INHERIT` variant of `ALTER TABLE`. Dynamically adding and removing inheritance links like this can be useful when the inheritance relationship is being used for table partitioning (see Section 5.10).

One convenient way to create a compatible table that will later be made a new child is to use the `LIKE` clause in `CREATE TABLE`. This creates a new table with the same columns as the source table. If there are any `CHECK` constraints defined on the source table, the `INCLUDING CONSTRAINTS` option to `LIKE` should be specified, as the new child must have constraints matching the parent to be considered compatible.

A parent table cannot be dropped while any of its children remain. Neither can columns or check constraints of child tables be dropped or altered if they are inherited from any parent tables. If you wish to remove a table and all of its descendants, one easy way is to drop the parent table with the `CASCADE` option.

ALTER TABLE will propagate any changes in column data definitions and check constraints down the inheritance hierarchy. Again, dropping columns that are depended on by other tables is only possible when using the `CASCADE` option. `ALTER TABLE` follows the same rules for duplicate column merging and rejection that apply during `CREATE TABLE`.

Inherited queries perform access permission checks on the parent table only. Thus, for example, granting `UPDATE` permission on the `cities` table implies permission to update rows in the `capitals` table as well, when they are accessed through `cities`. This preserves the appearance that the data is (also) in the parent table. But the `capitals` table could not be updated directly without an additional grant. Two exceptions to this rule are `TRUNCATE` and `LOCK TABLE`, where permissions on the child tables are always checked, whether they are processed directly or recursively via those commands performed on the parent table.

In a similar way, the parent table's row security policies (see Section 5.7) are applied to rows coming from child tables during an inherited query. A child table's policies, if any, are applied only when it is the table explicitly named in the query; and in that case, any policies attached to its parent(s) are ignored.

Foreign tables (see Section 5.11) can also be part of inheritance hierarchies, either as parent or child tables, just as regular tables can be. If a foreign table is part of an inheritance hierarchy then any operations not supported by the foreign table are not supported on the whole hierarchy either.

5.9.1. Caveats

Note that not all SQL commands are able to work on inheritance hierarchies. Commands that are used for data querying, data modification, or schema modification (e.g., `SELECT`, `UPDATE`, `DELETE`, most variants of `ALTER TABLE`, but not `INSERT` or `ALTER TABLE ... RENAME`) typically default to including child tables and support the `ONLY` notation to exclude them. Commands that do database maintenance and tuning (e.g., `REINDEX`, `VACUUM`) typically only work on individual, physical tables and do not support recursing over inheritance hierarchies. The respective behavior of each individual command is documented in its reference page (Reference I, *SQL Commands*).

A serious limitation of the inheritance feature is that indexes (including unique constraints) and foreign key constraints only apply to single tables, not to their inheritance children. This is true on both the referencing and referenced sides of a foreign key constraint. Thus, in the terms of the above example:

- If we declared `cities.name` to be `UNIQUE` or a `PRIMARY KEY`, this would not stop the `capitals` table from having rows with names duplicating rows in `cities`. And those duplicate rows would by default show up in queries from `cities`. In fact, by default `capitals` would have no unique constraint at all, and so could contain multiple rows with the same name. You could add a unique constraint to `capitals`, but this would not prevent duplication compared to `cities`.
- Similarly, if we were to specify that `cities.name` `REFERENCES` some other table, this constraint would not automatically propagate to `capitals`. In this case you could work around it by manually adding the same `REFERENCES` constraint to `capitals`.
- Specifying that another table's column `REFERENCES cities(name)` would allow the other table to contain city names, but not capital names. There is no good workaround for this case.

These deficiencies will probably be fixed in some future release, but in the meantime considerable care is needed in deciding whether inheritance is useful for your application.

5.10. Partitioning

PostgreSQL supports basic table partitioning. This section describes why and how to implement partitioning as part of your database design.

5.10.1. Overview

Partitioning refers to splitting what is logically one large table into smaller physical pieces. Partitioning can provide several benefits:

- Query performance can be improved dramatically in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. The partitioning substitutes for leading columns of indexes, reducing index size and making it more likely that the heavily-used parts of the indexes fit in memory.

- When queries or updates access a large percentage of a single partition, performance can be improved by taking advantage of sequential scan of that partition instead of using an index and random access reads scattered across the whole table.
- Bulk loads and deletes can be accomplished by adding or removing partitions, if that requirement is planned into the partitioning design. `ALTER TABLE NO INHERIT` and `DROP TABLE` are both far faster than a bulk operation. These commands also entirely avoid the `VACUUM` overhead caused by a bulk `DELETE`.
- Seldom-used data can be migrated to cheaper and slower storage media.

The benefits will normally be worthwhile only when a table would otherwise be very large. The exact point at which a table will benefit from partitioning depends on the application, although a rule of thumb is that the size of the table should exceed the physical memory of the database server.

Currently, PostgreSQL supports partitioning via table inheritance. Each partition must be created as a child table of a single parent table. The parent table itself is normally empty; it exists just to represent the entire data set. You should be familiar with inheritance (see Section 5.9) before attempting to set up partitioning.

The following forms of partitioning can be implemented in PostgreSQL:

Range Partitioning

: The table is partitioned into "ranges" defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions. For example one might partition by date ranges, or by ranges of identifiers for particular business objects.

List Partitioning

: The table is partitioned by explicitly listing which key values appear in each partition.

5.10.2. Implementing Partitioning

To set up a partitioned table, do the following:

1. Create the "master" table, from which all of the partitions will inherit.

   This table will contain no data. Do not define any check constraints on this table, unless you intend them to be applied equally to all partitions. There is no point in defining any indexes or unique constraints on it, either.
2. Create several "child" tables that each inherit from the master table. Normally, these tables will not add any columns to the set inherited from the master.

   We will refer to the child tables as partitions, though they are in every way normal PostgreSQL tables (or, possibly, foreign tables).
3. Add table constraints to the partition tables to define the allowed key values in each partition.

   Typical examples would be:

   ```
   CHECK ( x = 1 )
   CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire', 'Warwickshire' ))
   CHECK ( outletID >= 100 AND outletID < 200 )
   ```

Ensure that the constraints guarantee that there is no overlap between the key values permitted in different partitions. A common mistake is to set up range constraints like:

```
CHECK ( outletID BETWEEN 100 AND 200 )
CHECK ( outletID BETWEEN 200 AND 300 )
```

This is wrong since it is not clear which partition the key value 200 belongs in.

Note that there is no difference in syntax between range and list partitioning; those terms are descriptive only.

4. For each partition, create an index on the key column(s), as well as any other indexes you might want. (The key index is not strictly necessary, but in most scenarios it is helpful. If you intend the key values to be unique then you should always create a unique or primary-key constraint for each partition.)
5. Optionally, define a trigger or rule to redirect data inserted into the master table to the appropriate partition.
6. Ensure that the constraint_exclusion configuration parameter is not disabled in `postgresql.conf`. If it is, queries will not be optimized as desired.

For example, suppose we are constructing a database for a large ice cream company. The company measures peak temperatures every day as well as ice cream sales in each region. Conceptually, we want a table like:

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
);
```

We know that most queries will access just the last week's, month's or quarter's data, since the main use of this table will be to prepare online reports for management. To reduce the amount of old data that needs to be stored, we decide to only keep the most recent 3 years worth of data. At the beginning of each month we will remove the oldest month's data.

In this situation we can use partitioning to help us meet all of our different requirements for the measurements table. Following the steps outlined above, partitioning can be set up as follows:

1. The master table is the `measurement` table, declared exactly as above.
2. Next we create one partition for each active month:

```
CREATE TABLE measurement_y2006m02 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2006m03 ( ) INHERITS (measurement);
...
CREATE TABLE measurement_y2007m11 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2007m12 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2008m01 ( ) INHERITS (measurement);
```

Each of the partitions are complete tables in their own right, but they inherit their definitions from the `measurement` table.

This solves one of our problems: deleting old data. Each month, all we will need to do is perform a `DROP TABLE` on the oldest child table and create a new child table for the new month's data.

3. We must provide non-overlapping table constraints. Rather than just creating the partition tables as above, the table creation script should really be:

```
CREATE TABLE measurement_y2006m02 (
    CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
) INHERITS (measurement);
CREATE TABLE measurement_y2006m03 (
    CHECK ( logdate >= DATE '2006-03-01' AND logdate < DATE '2006-04-01' )
) INHERITS (measurement);
...
CREATE TABLE measurement_y2007m11 (
    CHECK ( logdate >= DATE '2007-11-01' AND logdate < DATE '2007-12-01' )
) INHERITS (measurement);
CREATE TABLE measurement_y2007m12 (
    CHECK ( logdate >= DATE '2007-12-01' AND logdate < DATE '2008-01-01' )
) INHERITS (measurement);
CREATE TABLE measurement_y2008m01 (
    CHECK ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
) INHERITS (measurement);
```

4. We probably need indexes on the key columns too:

```
CREATE INDEX measurement_y2006m02_logdate ON measurement_y2006m02 (logdate);
CREATE INDEX measurement_y2006m03_logdate ON measurement_y2006m03 (logdate);
...
CREATE INDEX measurement_y2007m11_logdate ON measurement_y2007m11 (logdate);
CREATE INDEX measurement_y2007m12_logdate ON measurement_y2007m12 (logdate);
CREATE INDEX measurement_y2008m01_logdate ON measurement_y2008m01 (logdate);
```

   We choose not to add further indexes at this time.

5. We want our application to be able to say `INSERT INTO measurement ...` and have the data be redirected into the appropriate partition table. We can arrange that by attaching a suitable trigger function to the master table. If data will be added only to the latest partition, we can use a very simple trigger function:

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

   After creating the function, we create a trigger which calls the trigger function:

```
CREATE TRIGGER insert_measurement_trigger
    BEFORE INSERT ON measurement
    FOR EACH ROW EXECUTE PROCEDURE measurement_insert_trigger();
```

   We must redefine the trigger function each month so that it always points to the current partition. The trigger definition does not need to be updated, however.

   We might want to insert data and have the server automatically locate the partition into which the row should be added. We could do this with a more complex trigger function, for example:

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
```

```
    IF ( NEW.logdate >= DATE '2006-02-01' AND
         NEW.logdate < DATE '2006-03-01' ) THEN
        INSERT INTO measurement_y2006m02 VALUES (NEW.*);
    ELSIF ( NEW.logdate >= DATE '2006-03-01' AND
            NEW.logdate < DATE '2006-04-01' ) THEN
        INSERT INTO measurement_y2006m03 VALUES (NEW.*);
    ...
    ELSIF ( NEW.logdate >= DATE '2008-01-01' AND
            NEW.logdate < DATE '2008-02-01' ) THEN
        INSERT INTO measurement_y2008m01 VALUES (NEW.*);
    ELSE
        RAISE EXCEPTION 'Date out of range.  Fix the measurement_insert_trigger() functi
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE plpgsql;
```

The trigger definition is the same as before. Note that each `IF` test must exactly match the `CHECK` constraint for its partition.

While this function is more complex than the single-month case, it doesn't need to be updated as often, since branches can be added in advance of being needed.

> **Note:** In practice it might be best to check the newest partition first, if most inserts go into that partition. For simplicity we have shown the trigger's tests in the same order as in other parts of this example.

As we can see, a complex partitioning scheme could require a substantial amount of DDL. In the above example we would be creating a new partition each month, so it might be wise to write a script that generates the required DDL automatically.

5.10.3. Managing Partitions

Normally the set of partitions established when initially defining the table are not intended to remain static. It is common to want to remove old partitions of data and periodically add new partitions for new data. One of the most important advantages of partitioning is precisely that it allows this otherwise painful task to be executed nearly instantaneously by manipulating the partition structure, rather than physically moving large amounts of data around.

The simplest option for removing old data is simply to drop the partition that is no longer necessary:

```
DROP TABLE measurement_y2006m02;
```

This can very quickly delete millions of records because it doesn't have to individually delete every record.

Another option that is often preferable is to remove the partition from the partitioned table but retain access to it as a table in its own right:

```
ALTER TABLE measurement_y2006m02 NO INHERIT measurement;
```

This allows further operations to be performed on the data before it is dropped. For example, this is often a useful time to back up the data using COPY, pg_dump, or similar tools. It might also be a useful time to aggregate data into smaller formats, perform other data manipulations, or run reports.

Similarly we can add a new partition to handle new data. We can create an empty partition in the partitioned table just as the original partitions were created above:

```
CREATE TABLE measurement_y2008m02 (
    CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' )
) INHERITS (measurement);
```

As an alternative, it is sometimes more convenient to create the new table outside the partition structure, and make it a proper partition later. This allows the data to be loaded, checked, and transformed prior to it appearing in the partitioned table:

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );
\copy measurement_y2008m02 from 'measurement_y2008m02'
-- possibly some other data preparation work
ALTER TABLE measurement_y2008m02 INHERIT measurement;
```

5.10.4. Partitioning and Constraint Exclusion

Constraint exclusion is a query optimization technique that improves performance for partitioned tables defined in the fashion described above. As an example:

```
SET constraint_exclusion = on;
SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
```

Without constraint exclusion, the above query would scan each of the partitions of the measurement table. With constraint exclusion enabled, the planner will examine the constraints of each partition and try to prove that the partition need not be scanned because it could not contain any rows meeting the query's WHERE clause. When the planner can prove this, it excludes the partition from the query plan.

You can use the EXPLAIN command to show the difference between a plan with constraint_exclusion on and a plan with it off. A typical unoptimized plan for this type of table setup is:

```
SET constraint_exclusion = off;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';

                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate  (cost=158.66..158.68 rows=1 width=0)
   ->  Append  (cost=0.00..151.88 rows=2715 width=0)
         ->  Seq Scan on measurement  (cost=0.00..30.38 rows=543 width=0)
```

```
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2006m02 measurement  (cost=0.00..30.38 rows=543 width
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2006m03 measurement  (cost=0.00..30.38 rows=543 width
               Filter: (logdate >= '2008-01-01'::date)
...
         ->  Seq Scan on measurement_y2007m12 measurement  (cost=0.00..30.38 rows=543 width
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2008m01 measurement  (cost=0.00..30.38 rows=543 width
               Filter: (logdate >= '2008-01-01'::date)
```

Some or all of the partitions might use index scans instead of full-table sequential scans, but the point here is that there is no need to scan the older partitions at all to answer this query. When we enable constraint exclusion, we get a significantly cheaper plan that will deliver the same answer:

```
SET constraint_exclusion = on;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate  (cost=63.47..63.48 rows=1 width=0)
   ->  Append  (cost=0.00..60.75 rows=1086 width=0)
         ->  Seq Scan on measurement  (cost=0.00..30.38 rows=543 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2008m01 measurement  (cost=0.00..30.38 rows=543 width
               Filter: (logdate >= '2008-01-01'::date)
```

Note that constraint exclusion is driven only by `CHECK` constraints, not by the presence of indexes. Therefore it isn't necessary to define indexes on the key columns. Whether an index needs to be created for a given partition depends on whether you expect that queries that scan the partition will generally scan a large part of the partition or just a small part. An index will be helpful in the latter case but not the former.

The default (and recommended) setting of constraint_exclusion is actually neither `on` nor `off`, but an intermediate setting called `partition`, which causes the technique to be applied only to queries that are likely to be working on partitioned tables. The `on` setting causes the planner to examine `CHECK` constraints in all queries, even simple ones that are unlikely to benefit.

5.10.5. Alternative Partitioning Methods

A different approach to redirecting inserts into the appropriate partition table is to set up rules, instead of a trigger, on the master table. For example:

```
CREATE RULE measurement_insert_y2006m02 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
DO INSTEAD
    INSERT INTO measurement_y2006m02 VALUES (NEW.*);
...
CREATE RULE measurement_insert_y2008m01 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
```

```
DO INSTEAD
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
```

A rule has significantly more overhead than a trigger, but the overhead is paid once per query rather than once per row, so this method might be advantageous for bulk-insert situations. In most cases, however, the trigger method will offer better performance.

Be aware that COPY ignores rules. If you want to use COPY to insert data, you'll need to copy into the correct partition table rather than into the master. COPY does fire triggers, so you can use it normally if you use the trigger approach.

Another disadvantage of the rule approach is that there is no simple way to force an error if the set of rules doesn't cover the insertion date; the data will silently go into the master table instead.

Partitioning can also be arranged using a UNION ALL view, instead of table inheritance. For example,

```
CREATE VIEW measurement AS
          SELECT * FROM measurement_y2006m02
UNION ALL SELECT * FROM measurement_y2006m03
...
UNION ALL SELECT * FROM measurement_y2007m11
UNION ALL SELECT * FROM measurement_y2007m12
UNION ALL SELECT * FROM measurement_y2008m01;
```

However, the need to recreate the view adds an extra step to adding and dropping individual partitions of the data set. In practice this method has little to recommend it compared to using inheritance.

5.10.6. Caveats

The following caveats apply to partitioned tables:

- There is no automatic way to verify that all of the CHECK constraints are mutually exclusive. It is safer to create code that generates partitions and creates and/or modifies associated objects than to write each by hand.
- The schemes shown here assume that the partition key column(s) of a row never change, or at least do not change enough to require it to move to another partition. An UPDATE that attempts to do that will fail because of the CHECK constraints. If you need to handle such cases, you can put suitable update triggers on the partition tables, but it makes management of the structure much more complicated.
- If you are using manual VACUUM or ANALYZE commands, don't forget that you need to run them on each partition individually. A command like:

  ```
  ANALYZE measurement;
  ```
  will only process the master table.
- INSERT statements with ON CONFLICT clauses are unlikely to work as expected, as the ON CONFLICT action is only taken in case of unique violations on the specified target relation, not its child relations.

The following caveats apply to constraint exclusion:

- Constraint exclusion only works when the query's `WHERE` clause contains constants (or externally supplied parameters). For example, a comparison against a non-immutable function such as `CURRENT_TIMESTAMP` cannot be optimized, since the planner cannot know which partition the function value might fall into at run time.
- Keep the partitioning constraints simple, else the planner may not be able to prove that partitions don't need to be visited. Use simple equality conditions for list partitioning, or simple range tests for range partitioning, as illustrated in the preceding examples. A good rule of thumb is that partitioning constraints should contain only comparisons of the partitioning column(s) to constants using B-tree-indexable operators.
- All constraints on all partitions of the master table are examined during constraint exclusion, so large numbers of partitions are likely to increase query planning time considerably. Partitioning using these techniques will work well with up to perhaps a hundred partitions; don't try to use many thousands of partitions.

5.11. Foreign Data

PostgreSQL implements portions of the SQL/MED specification, allowing you to access data that resides outside PostgreSQL using regular SQL queries. Such data is referred to as *foreign data*. (Note that this usage is not to be confused with foreign keys, which are a type of constraint within the database.)

Foreign data is accessed with help from a *foreign data wrapper*. A foreign data wrapper is a library that can communicate with an external data source, hiding the details of connecting to the data source and obtaining data from it. There are some foreign data wrappers available as `contrib` modules; see Appendix F. Other kinds of foreign data wrappers might be found as third party products. If none of the existing foreign data wrappers suit your needs, you can write your own; see Chapter 54.

To access foreign data, you need to create a *foreign server* object, which defines how to connect to a particular external data source according to the set of options used by its supporting foreign data wrapper. Then you need to create one or more *foreign tables*, which define the structure of the remote data. A foreign table can be used in queries just like a normal table, but a foreign table has no storage in the PostgreSQL server. Whenever it is used, PostgreSQL asks the foreign data wrapper to fetch data from the external source, or transmit data to the external source in the case of update commands.

Accessing remote data may require authenticating to the external data source. This information can be provided by a *user mapping*, which can provide additional data such as user names and passwords based on the current PostgreSQL role.

For additional information, see CREATE FOREIGN DATA WRAPPER, CREATE SERVER, CREATE USER MAPPING, CREATE FOREIGN TABLE, and IMPORT FOREIGN SCHEMA.

5.12. Other Database Objects

Tables are the central objects in a relational database structure, because they hold your data. But they are not the only objects that exist in a database. Many other kinds of objects can be created to make the use

and management of the data more efficient or convenient. They are not discussed in this chapter, but we give you a list here so that you are aware of what is possible:

- Views
- Functions and operators
- Data types and domains
- Triggers and rewrite rules

Detailed information on these topics appears in Part V.

5.13. Dependency Tracking

When you create complex database structures involving many tables with foreign key constraints, views, triggers, functions, etc. you implicitly create a net of dependencies between the objects. For instance, a table with a foreign key constraint depends on the table it references.

To ensure the integrity of the entire database structure, PostgreSQL makes sure that you cannot drop objects that other objects still depend on. For example, attempting to drop the products table we considered in Section 5.3.5, with the orders table depending on it, would result in an error message like this:

```
DROP TABLE products;

ERROR:  cannot drop table products because other objects depend on it
DETAIL:  constraint orders_product_no_fkey on table orders depends on table products
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

The error message contains a useful hint: if you do not want to bother deleting all the dependent objects individually, you can run:

```
DROP TABLE products CASCADE;
```

and all the dependent objects will be removed. In this case, it doesn't remove the orders table, it only removes the foreign key constraint. (If you want to check what `DROP ... CASCADE` will do, run `DROP` without `CASCADE` and read the `DETAIL` output.)

All `DROP` commands in PostgreSQL support specifying `CASCADE`. Of course, the nature of the possible dependencies varies with the type of the object. You can also write `RESTRICT` instead of `CASCADE` to get the default behavior, which is to prevent the dropping of objects that other objects depend on.

> **Note:** According to the SQL standard, specifying either `RESTRICT` or `CASCADE` is required in a `DROP` command. No database system actually enforces that rule, but whether the default behavior is `RESTRICT` or `CASCADE` varies across systems.

For user-defined functions, PostgreSQL tracks dependencies associated with a function's externally-visible properties, such as its argument and result types, but *not* dependencies that could only be known by examining the function body. As an example, consider this situation:

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow',
```

```
                                'green', 'blue', 'purple');

CREATE TABLE my_colors (color rainbow, note text);

CREATE FUNCTION get_color_note (rainbow) RETURNS text AS
  'SELECT note FROM my_colors WHERE color = $1'
  LANGUAGE SQL;
```

(See Section 35.4 for an explanation of SQL-language functions.) PostgreSQL will be aware that the `get_color_note` function depends on the `rainbow` type: dropping the type would force dropping the function, because its argument type would no longer be defined. But PostgreSQL will not consider `get_color_note` to depend on the `my_colors` table, and so will not drop the function if the table is dropped. While there are disadvantages to this approach, there are also benefits. The function is still valid in some sense if the table is missing, though executing it would cause an error; creating a new table of the same name would allow the function to work again.

Chapter 6. Data Manipulation

The previous chapter discussed how to create tables and other structures to hold your data. Now it is time to fill the tables with data. This chapter covers how to insert, update, and delete table data. The chapter after this will finally explain how to extract your long-lost data from the database.

6.1. Inserting Data

When a table is created, it contains no data. The first thing to do before a database can be of much use is to insert data. Data is conceptually inserted one row at a time. Of course you can also insert more than one row, but there is no way to insert less than one row. Even if you know only some column values, a complete row must be created.

To create a new row, use the INSERT command. The command requires the table name and column values. For example, consider the products table from Chapter 5:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

An example command to insert a row would be:

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

The data values are listed in the order in which the columns appear in the table, separated by commas. Usually, the data values will be literals (constants), but scalar expressions are also allowed.

The above syntax has the drawback that you need to know the order of the columns in the table. To avoid this you can also list the columns explicitly. For example, both of the following commands have the same effect as the one above:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

Many users consider it good practice to always list the column names.

If you don't have values for all the columns, you can omit some of them. In that case, the columns will be filled with their default values. For example:

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
INSERT INTO products VALUES (1, 'Cheese');
```

The second form is a PostgreSQL extension. It fills the columns from the left with as many values as are given, and the rest will be defaulted.

For clarity, you can also request default values explicitly, for individual columns or for the entire row:

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', DEFAULT);
INSERT INTO products DEFAULT VALUES;
```

You can insert multiple rows in a single command:

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);
```

It is also possible to insert the result of a query (which might be no rows, one row, or many rows):

```
INSERT INTO products (product_no, name, price)
  SELECT product_no, name, price FROM new_products
    WHERE release_date = 'today';
```

This provides the full power of the SQL query mechanism (Chapter 7) for computing the rows to be inserted.

> **Tip:** When inserting a lot of data at the same time, consider using the COPY command. It is not as flexible as the INSERT command, but is more efficient. Refer to Section 14.4 for more information on improving bulk loading performance.

6.2. Updating Data

The modification of data that is already in the database is referred to as updating. You can update individual rows, all the rows in a table, or a subset of all rows. Each column can be updated separately; the other columns are not affected.

To update existing rows, use the UPDATE command. This requires three pieces of information:

1. The name of the table and column to update
2. The new value of the column
3. Which row(s) to update

Recall from Chapter 5 that SQL does not, in general, provide a unique identifier for rows. Therefore it is not always possible to directly specify which row to update. Instead, you specify which conditions a row must meet in order to be updated. Only if you have a primary key in the table (independent of whether you declared it or not) can you reliably address individual rows by choosing a condition that matches the primary key. Graphical database access tools rely on this fact to allow you to update rows individually.

For example, this command updates all products that have a price of 5 to have a price of 10:

```
UPDATE products SET price = 10 WHERE price = 5;
```

This might cause zero, one, or many rows to be updated. It is not an error to attempt an update that does not match any rows.

Let's look at that command in detail. First is the key word `UPDATE` followed by the table name. As usual, the table name can be schema-qualified, otherwise it is looked up in the path. Next is the key word `SET` followed by the column name, an equal sign, and the new column value. The new column value can be any scalar expression, not just a constant. For example, if you want to raise the price of all products by 10% you could use:

```
UPDATE products SET price = price * 1.10;
```

As you see, the expression for the new value can refer to the existing value(s) in the row. We also left out the `WHERE` clause. If it is omitted, it means that all rows in the table are updated. If it is present, only those rows that match the `WHERE` condition are updated. Note that the equals sign in the `SET` clause is an assignment while the one in the `WHERE` clause is a comparison, but this does not create any ambiguity. Of course, the `WHERE` condition does not have to be an equality test. Many other operators are available (see Chapter 9). But the expression needs to evaluate to a Boolean result.

You can update more than one column in an `UPDATE` command by listing more than one assignment in the `SET` clause. For example:

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

6.3. Deleting Data

So far we have explained how to add data to tables and how to change data. What remains is to discuss how to remove data that is no longer needed. Just as adding data is only possible in whole rows, you can only remove entire rows from a table. In the previous section we explained that SQL does not provide a way to directly address individual rows. Therefore, removing rows can only be done by specifying conditions that the rows to be removed have to match. If you have a primary key in the table then you can specify the exact row. But you can also remove groups of rows matching a condition, or you can remove all rows in the table at once.

You use the DELETE command to remove rows; the syntax is very similar to the `UPDATE` command. For instance, to remove all rows from the products table that have a price of 10, use:

```
DELETE FROM products WHERE price = 10;
```

If you simply write:

```
DELETE FROM products;
```

then all rows in the table will be deleted! Caveat programmer.

6.4. Returning Data From Modified Rows

Sometimes it is useful to obtain data from modified rows while they are being manipulated. The `INSERT`, `UPDATE`, and `DELETE` commands all have an optional `RETURNING` clause that supports this. Use of

RETURNING avoids performing an extra database query to collect the data, and is especially valuable when it would otherwise be difficult to identify the modified rows reliably.

The allowed contents of a RETURNING clause are the same as a SELECT command's output list (see Section 7.3). It can contain column names of the command's target table, or value expressions using those columns. A common shorthand is RETURNING *, which selects all columns of the target table in order.

In an INSERT, the data available to RETURNING is the row as it was inserted. This is not so useful in trivial inserts, since it would just repeat the data provided by the client. But it can be very handy when relying on computed default values. For example, when using a serial column to provide unique identifiers, RETURNING can return the ID assigned to a new row:

```
CREATE TABLE users (firstname text, lastname text, id serial primary key);

INSERT INTO users (firstname, lastname) VALUES ('Joe', 'Cool') RETURNING id;
```

The RETURNING clause is also very useful with INSERT ... SELECT.

In an UPDATE, the data available to RETURNING is the new content of the modified row. For example:

```
UPDATE products SET price = price * 1.10
  WHERE price <= 99.99
  RETURNING name, price AS new_price;
```

In a DELETE, the data available to RETURNING is the content of the deleted row. For example:

```
DELETE FROM products
  WHERE obsoletion_date = 'today'
  RETURNING *;
```

If there are triggers (Chapter 36) on the target table, the data available to RETURNING is the row as modified by the triggers. Thus, inspecting columns computed by triggers is another common use-case for RETURNING.

Chapter 7. Queries

The previous chapters explained how to create tables, how to fill them with data, and how to manipulate that data. Now we finally discuss how to retrieve the data from the database.

7.1. Overview

The process of retrieving or the command to retrieve data from a database is called a *query*. In SQL the SELECT command is used to specify queries. The general syntax of the `SELECT` command is

```
[WITH with_queries] SELECT select_list FROM table_expression [sort_specification]
```

The following sections describe the details of the select list, the table expression, and the sort specification. `WITH` queries are treated last since they are an advanced feature.

A simple kind of query has the form:

```
SELECT * FROM table1;
```

Assuming that there is a table called `table1`, this command would retrieve all rows and all user-defined columns from `table1`. (The method of retrieval depends on the client application. For example, the psql program will display an ASCII-art table on the screen, while client libraries will offer functions to extract individual values from the query result.) The select list specification `*` means all columns that the table expression happens to provide. A select list can also select a subset of the available columns or make calculations using the columns. For example, if `table1` has columns named `a`, `b`, and `c` (and perhaps others) you can make the following query:

```
SELECT a, b + c FROM table1;
```

(assuming that `b` and `c` are of a numerical data type). See Section 7.3 for more details.

`FROM table1` is a simple kind of table expression: it reads just one table. In general, table expressions can be complex constructs of base tables, joins, and subqueries. But you can also omit the table expression entirely and use the `SELECT` command as a calculator:

```
SELECT 3 * 4;
```

This is more useful if the expressions in the select list return varying results. For example, you could call a function this way:

```
SELECT random();
```

7.2. Table Expressions

A *table expression* computes a table. The table expression contains a `FROM` clause that is optionally followed by `WHERE`, `GROUP BY`, and `HAVING` clauses. Trivial table expressions simply refer to a table on

disk, a so-called base table, but more complex expressions can be used to modify or combine base tables in various ways.

The optional WHERE, GROUP BY, and HAVING clauses in the table expression specify a pipeline of successive transformations performed on the table derived in the FROM clause. All these transformations produce a virtual table that provides the rows that are passed to the select list to compute the output rows of the query.

7.2.1. The FROM Clause

The *FROM Clause* derives a table from one or more other tables given in a comma-separated table reference list.

```
FROM table_reference [, table_reference [, ...]]
```

A table reference can be a table name (possibly schema-qualified), or a derived table such as a subquery, a JOIN construct, or complex combinations of these. If more than one table reference is listed in the FROM clause, the tables are cross-joined (that is, the Cartesian product of their rows is formed; see below). The result of the FROM list is an intermediate virtual table that can then be subject to transformations by the WHERE, GROUP BY, and HAVING clauses and is finally the result of the overall table expression.

When a table reference names a table that is the parent of a table inheritance hierarchy, the table reference produces rows of not only that table but all of its descendant tables, unless the key word ONLY precedes the table name. However, the reference produces only the columns that appear in the named table — any columns added in subtables are ignored.

Instead of writing ONLY before the table name, you can write * after the table name to explicitly specify that descendant tables are included. Writing * is not necessary since that behavior is the default (unless you have changed the setting of the sql_inheritance configuration option). However writing * might be useful to emphasize that additional tables will be searched.

7.2.1.1. Joined Tables

A joined table is a table derived from two other (real or derived) tables according to the rules of the particular join type. Inner, outer, and cross-joins are available. The general syntax of a joined table is

```
T1 join_type T2 [ join_condition ]
```

Joins of all types can be chained together, or nested: either or both *T1* and *T2* can be joined tables. Parentheses can be used around JOIN clauses to control the join order. In the absence of parentheses, JOIN clauses nest left-to-right.

Join Types

Cross join

```
T1 CROSS JOIN T2
```

For every possible combination of rows from *T1* and *T2* (i.e., a Cartesian product), the joined table will contain a row consisting of all columns in *T1* followed by all columns in *T2*. If the tables have N and M rows respectively, the joined table will have N * M rows.

FROM *T1* CROSS JOIN *T2* is equivalent to FROM *T1* INNER JOIN *T2* ON TRUE (see below). It is also equivalent to FROM *T1*, *T2*.

> **Note:** This latter equivalence does not hold exactly when more than two tables appear, because JOIN binds more tightly than comma. For example FROM *T1* CROSS JOIN *T2* INNER JOIN *T3* ON *condition* is not the same as FROM *T1*, *T2* INNER JOIN *T3* ON *condition* because the *condition* can reference *T1* in the first case but not the second.

Qualified joins

```
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 ON boolean_expression
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2 USING ( join column list )
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN T2
```

The words INNER and OUTER are optional in all forms. INNER is the default; LEFT, RIGHT, and FULL imply an outer join.

The *join condition* is specified in the ON or USING clause, or implicitly by the word NATURAL. The join condition determines which rows from the two source tables are considered to "match", as explained in detail below.

The possible types of qualified join are:

INNER JOIN

For each row R1 of T1, the joined table has a row for each row in T2 that satisfies the join condition with R1.

LEFT OUTER JOIN

First, an inner join is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is added with null values in columns of T2. Thus, the joined table always has at least one row for each row in T1.

RIGHT OUTER JOIN

First, an inner join is performed. Then, for each row in T2 that does not satisfy the join condition with any row in T1, a joined row is added with null values in columns of T1. This is the converse of a left join: the result table will always have a row for each row in T2.

FULL OUTER JOIN

First, an inner join is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is added with null values in columns of T2. Also, for each row of T2 that does not satisfy the join condition with any row in T1, a joined row with null values in the columns of T1 is added.

The ON clause is the most general kind of join condition: it takes a Boolean value expression of the same kind as is used in a WHERE clause. A pair of rows from *T1* and *T2* match if the ON expression evaluates to true.

The USING clause is a shorthand that allows you to take advantage of the specific situation where both sides of the join use the same name for the joining column(s). It takes a comma-separated list of

the shared column names and forms a join condition that includes an equality comparison for each one. For example, joining *T1* and *T2* with `USING (a, b)` produces the join condition `ON T1.a = T2.a AND T1.b = T2.b`.

Furthermore, the output of `JOIN USING` suppresses redundant columns: there is no need to print both of the matched columns, since they must have equal values. While `JOIN ON` produces all columns from *T1* followed by all columns from *T2*, `JOIN USING` produces one output column for each of the listed column pairs (in the listed order), followed by any remaining columns from *T1*, followed by any remaining columns from *T2*.

Finally, `NATURAL` is a shorthand form of `USING`: it forms a `USING` list consisting of all column names that appear in both input tables. As with `USING`, these columns appear only once in the output table. If there are no common column names, `NATURAL JOIN` behaves like `JOIN ... ON TRUE`, producing a cross-product join.

> **Note:** `USING` is reasonably safe from column changes in the joined relations since only the listed columns are combined. `NATURAL` is considerably more risky since any schema changes to either relation that cause a new matching column name to be present will cause the join to combine that new column as well.

To put this together, assume we have tables `t1`:

```
 num | name
-----+------
   1 | a
   2 | b
   3 | c
```

and `t2`:

```
 num | value
-----+-------
   1 | xxx
   3 | yyy
   5 | zzz
```

then we get the following results for the various joins:

```
=> SELECT * FROM t1 CROSS JOIN t2;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   1 | a    |   3 | yyy
   1 | a    |   5 | zzz
   2 | b    |   1 | xxx
   2 | b    |   3 | yyy
   2 | b    |   5 | zzz
   3 | c    |   1 | xxx
   3 | c    |   3 | yyy
   3 | c    |   5 | zzz
```

```
(9 rows)

=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
(2 rows)

=> SELECT * FROM t1 INNER JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
(3 rows)

=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   2 | b    |
   3 | c    | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
     |      |   5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |   3 | yyy
```

```
     |      |   5 | zzz
(4 rows)
```

The join condition specified with `ON` can also contain conditions that do not relate directly to the join. This can prove useful for some queries but needs to be thought out carefully. For example:

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   2 | b    |     |
   3 | c    |     |
(3 rows)
```

Notice that placing the restriction in the `WHERE` clause produces a different result:

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num WHERE t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
(1 row)
```

This is because a restriction placed in the `ON` clause is processed *before* the join, while a restriction placed in the `WHERE` clause is processed *after* the join. That does not matter with inner joins, but it matters a lot with outer joins.

7.2.1.2. Table and Column Aliases

A temporary name can be given to tables and complex table references to be used for references to the derived table in the rest of the query. This is called a *table alias*.

To create a table alias, write

```
FROM table_reference AS alias
```

or

```
FROM table_reference alias
```

The `AS` key word is optional noise. *`alias`* can be any identifier.

A typical application of table aliases is to assign short identifiers to long table names to keep the join clauses readable. For example:

```
SELECT * FROM some_very_long_table_name s JOIN another_fairly_long_name a ON s.id = a.num;
```

The alias becomes the new name of the table reference so far as the current query is concerned — it is not allowed to refer to the table by the original name elsewhere in the query. Thus, this is not valid:

```
SELECT * FROM my_table AS m WHERE my_table.a > 5;    -- wrong
```

Table aliases are mainly for notational convenience, but it is necessary to use them when joining a table to itself, e.g.:

```
SELECT * FROM people AS mother JOIN people AS child ON mother.id = child.mother_id;
```

Additionally, an alias is required if the table reference is a subquery (see Section 7.2.1.3).

Parentheses are used to resolve ambiguities. In the following example, the first statement assigns the alias `b` to the second instance of `my_table`, but the second statement assigns the alias to the result of the join:

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

Another form of table aliasing gives temporary names to the columns of the table, as well as the table itself:

```
FROM table_reference [AS] alias ( column1 [, column2 [, ...]] )
```

If fewer column aliases are specified than the actual table has columns, the remaining columns are not renamed. This syntax is especially useful for self-joins or subqueries.

When an alias is applied to the output of a `JOIN` clause, the alias hides the original name(s) within the `JOIN`. For example:

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

is valid SQL, but:

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

is not valid; the table alias `a` is not visible outside the alias `c`.

7.2.1.3. Subqueries

Subqueries specifying a derived table must be enclosed in parentheses and *must* be assigned a table alias name (as in Section 7.2.1.2). For example:

```
FROM (SELECT * FROM table1) AS alias_name
```

This example is equivalent to `FROM table1 AS alias_name`. More interesting cases, which cannot be reduced to a plain join, arise when the subquery involves grouping or aggregation.

A subquery can also be a `VALUES` list:

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
     AS names(first, last)
```

Again, a table alias is required. Assigning alias names to the columns of the `VALUES` list is optional, but is good practice. For more information see Section 7.7.

7.2.1.4. Table Functions

Table functions are functions that produce a set of rows, made up of either base data types (scalar types) or composite data types (table rows). They are used like a table, view, or subquery in the FROM clause of a query. Columns returned by table functions can be included in SELECT, JOIN, or WHERE clauses in the same manner as columns of a table, view, or subquery.

Table functions may also be combined using the ROWS FROM syntax, with the results returned in parallel columns; the number of result rows in this case is that of the largest function result, with smaller results padded with null values to match.

```
function_call [WITH ORDINALITY] [[AS] table_alias [(column_alias [, ... ])]]
ROWS FROM( function_call [, ... ] ) [WITH ORDINALITY] [[AS] table_alias [(column_alias [, ... ]
```

If the WITH ORDINALITY clause is specified, an additional column of type bigint will be added to the function result columns. This column numbers the rows of the function result set, starting from 1. (This is a generalization of the SQL-standard syntax for UNNEST ... WITH ORDINALITY.) By default, the ordinal column is called ordinality, but a different column name can be assigned to it using an AS clause.

The special table function UNNEST may be called with any number of array parameters, and it returns a corresponding number of columns, as if UNNEST (Section 9.18) had been called on each parameter separately and combined using the ROWS FROM construct.

```
UNNEST( array_expression [, ... ] ) [WITH ORDINALITY] [[AS] table_alias [(column_alias [, ... ]
```

If no *table_alias* is specified, the function name is used as the table name; in the case of a ROWS FROM() construct, the first function's name is used.

If column aliases are not supplied, then for a function returning a base data type, the column name is also the same as the function name. For a function returning a composite type, the result columns get the names of the individual attributes of the type.

Some examples:

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

SELECT * FROM foo
    WHERE foosubid IN (
                        SELECT foosubid
                        FROM getfoo(foo.fooid) z
                        WHERE z.fooid = foo.fooid
                      );

CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);

SELECT * FROM vw_getfoo;
```

In some cases it is useful to define table functions that can return different column sets depending on how they are invoked. To support this, the table function can be declared as returning the pseudotype `record` with no `OUT` parameters. When such a function is used in a query, the expected row structure must be specified in the query itself, so that the system can know how to parse and plan the query. This syntax looks like:

```
function_call [AS] alias (column_definition [, ... ])
function_call AS [alias] (column_definition [, ... ])
ROWS FROM( ... function_call AS (column_definition [, ... ]) [, ... ] )
```

When not using the `ROWS FROM()` syntax, the *`column_definition`* list replaces the column alias list that could otherwise be attached to the `FROM` item; the names in the column definitions serve as column aliases. When using the `ROWS FROM()` syntax, a *`column_definition`* list can be attached to each member function separately; or if there is only one member function and no `WITH ORDINALITY` clause, a *`column_definition`* list can be written in place of a column alias list following `ROWS FROM()`.

Consider this example:

```
SELECT *
    FROM dblink('dbname=mydb', 'SELECT proname, prosrc FROM pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

The dblink function (part of the dblink module) executes a remote query. It is declared to return `record` since it might be used for any kind of query. The actual column set must be specified in the calling query so that the parser knows, for example, what `*` should expand to.

This example uses `ROWS FROM`:

```
SELECT *
FROM ROWS FROM
    (
        json_to_recordset('[{"a":40,"b":"foo"},{"a":"100","b":"bar"}]')
            AS (a INTEGER, b TEXT),
        generate_series(1, 3)
    ) AS x (p, q, s)
ORDER BY p;

  p  |  q  | s
-----+-----+---
  40 | foo | 1
 100 | bar | 2
     |     | 3
```

It joins two functions into a single `FROM` target. `json_to_recordset()` is instructed to return two columns, the first `integer` and the second `text`. The result of `generate_series()` is used directly. The `ORDER BY` clause sorts the column values as integers.

7.2.1.5. LATERAL Subqueries

Subqueries appearing in FROM can be preceded by the key word LATERAL. This allows them to reference columns provided by preceding FROM items. (Without LATERAL, each subquery is evaluated independently and so cannot cross-reference any other FROM item.)

Table functions appearing in FROM can also be preceded by the key word LATERAL, but for functions the key word is optional; the function's arguments can contain references to columns provided by preceding FROM items in any case.

A LATERAL item can appear at top level in the FROM list, or within a JOIN tree. In the latter case it can also refer to any items that are on the left-hand side of a JOIN that it is on the right-hand side of.

When a FROM item contains LATERAL cross-references, evaluation proceeds as follows: for each row of the FROM item providing the cross-referenced column(s), or set of rows of multiple FROM items providing the columns, the LATERAL item is evaluated using that row or row set's values of the columns. The resulting row(s) are joined as usual with the rows they were computed from. This is repeated for each row or set of rows from the column source table(s).

A trivial example of LATERAL is

```
SELECT * FROM foo, LATERAL (SELECT * FROM bar WHERE bar.id = foo.bar_id) ss;
```

This is not especially useful since it has exactly the same result as the more conventional

```
SELECT * FROM foo, bar WHERE bar.id = foo.bar_id;
```

LATERAL is primarily useful when the cross-referenced column is necessary for computing the row(s) to be joined. A common application is providing an argument value for a set-returning function. For example, supposing that vertices(polygon) returns the set of vertices of a polygon, we could identify close-together vertices of polygons stored in a table with:

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1, polygons p2,
     LATERAL vertices(p1.poly) v1,
     LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

This query could also be written

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1 CROSS JOIN LATERAL vertices(p1.poly) v1,
     polygons p2 CROSS JOIN LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

or in several other equivalent formulations. (As already mentioned, the LATERAL key word is unnecessary in this example, but we use it for clarity.)

It is often particularly handy to LEFT JOIN to a LATERAL subquery, so that source rows will appear in the result even if the LATERAL subquery produces no rows for them. For example, if get_product_names() returns the names of products made by a manufacturer, but some manufacturers in our table currently produce no products, we could find out which ones those are like this:

```
SELECT m.name
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true
```

```
WHERE pname IS NULL;
```

7.2.2. The WHERE Clause

The syntax of the *WHERE Clause* is

```
WHERE search_condition
```

where *search_condition* is any value expression (see Section 4.2) that returns a value of type boolean.

After the processing of the FROM clause is done, each row of the derived virtual table is checked against the search condition. If the result of the condition is true, the row is kept in the output table, otherwise (i.e., if the result is false or null) it is discarded. The search condition typically references at least one column of the table generated in the FROM clause; this is not required, but otherwise the WHERE clause will be fairly useless.

> **Note:** The join condition of an inner join can be written either in the WHERE clause or in the JOIN clause. For example, these table expressions are equivalent:
>
> ```
> FROM a, b WHERE a.id = b.id AND b.val > 5
> ```
>
> and:
>
> ```
> FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
> ```
>
> or perhaps even:
>
> ```
> FROM a NATURAL JOIN b WHERE b.val > 5
> ```
>
> Which one of these you use is mainly a matter of style. The JOIN syntax in the FROM clause is probably not as portable to other SQL database management systems, even though it is in the SQL standard. For outer joins there is no choice: they must be done in the FROM clause. The ON or USING clause of an outer join is *not* equivalent to a WHERE condition, because it results in the addition of rows (for unmatched input rows) as well as the removal of rows in the final result.

Here are some examples of WHERE clauses:

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100
```

```
SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)
```

`fdt` is the table derived in the `FROM` clause. Rows that do not meet the search condition of the `WHERE` clause are eliminated from `fdt`. Notice the use of scalar subqueries as value expressions. Just like any other query, the subqueries can employ complex table expressions. Notice also how `fdt` is referenced in the subqueries. Qualifying `c1` as `fdt.c1` is only necessary if `c1` is also the name of a column in the derived input table of the subquery. But qualifying the column name adds clarity even when it is not needed. This example shows how the column naming scope of an outer query extends into its inner queries.

7.2.3. The `GROUP BY` and `HAVING` Clauses

After passing the `WHERE` filter, the derived input table might be subject to grouping, using the `GROUP BY` clause, and elimination of group rows using the `HAVING` clause.

```
SELECT select_list
    FROM ...
    [WHERE ...]
    GROUP BY grouping_column_reference [, grouping_column_reference]...
```

The *GROUP BY Clause* is used to group together those rows in a table that have the same values in all the columns listed. The order in which the columns are listed does not matter. The effect is to combine each set of rows having common values into one group row that represents all rows in the group. This is done to eliminate redundancy in the output and/or compute aggregates that apply to these groups. For instance:

```
=> SELECT * FROM test1;
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> SELECT x FROM test1 GROUP BY x;
 x
---
 a
 b
 c
(3 rows)
```

In the second query, we could not have written `SELECT * FROM test1 GROUP BY x`, because there is no single value for the column `y` that could be associated with each group. The grouped-by columns can be referenced in the select list since they have a single value in each group.

In general, if a table is grouped, columns that are not listed in `GROUP BY` cannot be referenced except in aggregate expressions. An example with aggregate expressions is:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x;
 x | sum
```

```
---+-----
 a |   4
 b |   5
 c |   2
(3 rows)
```

Here sum is an aggregate function that computes a single value over the entire group. More information about the available aggregate functions can be found in Section 9.20.

> **Tip:** Grouping without aggregate expressions effectively calculates the set of distinct values in a column. This can also be achieved using the DISTINCT clause (see Section 7.3.3).

Here is another example: it calculates the total sales for each product (rather than the total sales of all products):

```
SELECT product_id, p.name, (sum(s.units) * p.price) AS sales
    FROM products p LEFT JOIN sales s USING (product_id)
    GROUP BY product_id, p.name, p.price;
```

In this example, the columns product_id, p.name, and p.price must be in the GROUP BY clause since they are referenced in the query select list (but see below). The column s.units does not have to be in the GROUP BY list since it is only used in an aggregate expression (sum(...)), which represents the sales of a product. For each product, the query returns a summary row about all sales of the product.

If the products table is set up so that, say, product_id is the primary key, then it would be enough to group by product_id in the above example, since name and price would be *functionally dependent* on the product ID, and so there would be no ambiguity about which name and price value to return for each product ID group.

In strict SQL, GROUP BY can only group by columns of the source table but PostgreSQL extends this to also allow GROUP BY to group by columns in the select list. Grouping by value expressions instead of simple column names is also allowed.

If a table has been grouped using GROUP BY, but only certain groups are of interest, the HAVING clause can be used, much like a WHERE clause, to eliminate groups from the result. The syntax is:

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ... HAVING boolean_expression
```

Expressions in the HAVING clause can refer both to grouped expressions and to ungrouped expressions (which necessarily involve an aggregate function).

Example:

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum
---+-----
 a |   4
 b |   5
(2 rows)

=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
 x | sum
```

```
---+-----
 a |   4
 b |   5
(2 rows)
```

Again, a more realistic example:

```
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS profit
    FROM products p LEFT JOIN sales s USING (product_id)
    WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
    GROUP BY product_id, p.name, p.price, p.cost
    HAVING sum(p.price * s.units) > 5000;
```

In the example above, the `WHERE` clause is selecting rows by a column that is not grouped (the expression is only true for sales during the last four weeks), while the `HAVING` clause restricts the output to groups with total gross sales over 5000. Note that the aggregate expressions do not necessarily need to be the same in all parts of the query.

If a query contains aggregate function calls, but no `GROUP BY` clause, grouping still occurs: the result is a single group row (or perhaps no rows at all, if the single row is then eliminated by `HAVING`). The same is true if it contains a `HAVING` clause, even without any aggregate function calls or `GROUP BY` clause.

7.2.4. `GROUPING SETS`, `CUBE`, and `ROLLUP`

More complex grouping operations than those described above are possible using the concept of *grouping sets*. The data selected by the `FROM` and `WHERE` clauses is grouped separately by each specified grouping set, aggregates computed for each group just as for simple `GROUP BY` clauses, and then the results returned. For example:

```
=> SELECT * FROM items_sold;
 brand | size | sales
-------+------+-------
 Foo   | L    |  10
 Foo   | M    |  20
 Bar   | M    |  15
 Bar   | L    |  5
(4 rows)

=> SELECT brand, size, sum(sales) FROM items_sold GROUP BY GROUPING SETS ((brand), (size), ());
 brand | size | sum
-------+------+-----
 Foo   |      |  30
 Bar   |      |  20
       | L    |  15
       | M    |  35
       |      |  50
(5 rows)
```

Each sublist of GROUPING SETS may specify zero or more columns or expressions and is interpreted the same way as though it were directly in the GROUP BY clause. An empty grouping set means that all rows are aggregated down to a single group (which is output even if no input rows were present), as described above for the case of aggregate functions with no GROUP BY clause.

References to the grouping columns or expressions are replaced by null values in result rows for grouping sets in which those columns do not appear. To distinguish which grouping a particular output row resulted from, see Table 9-53.

A shorthand notation is provided for specifying two common types of grouping set. A clause of the form

```
ROLLUP ( e1, e2, e3, ... )
```

represents the given list of expressions and all prefixes of the list including the empty list; thus it is equivalent to

```
GROUPING SETS (
     ( e1, e2, e3, ... ),
     ...
     ( e1, e2 ),
     ( e1 ),
     ( )
)
```

This is commonly used for analysis over hierarchical data; e.g., total salary by department, division, and company-wide total.

A clause of the form

```
CUBE ( e1, e2, ... )
```

represents the given list and all of its possible subsets (i.e., the power set). Thus

```
CUBE ( a, b, c )
```

is equivalent to

```
GROUPING SETS (
     ( a, b, c ),
     ( a, b    ),
     ( a,    c ),
     ( a       ),
     (    b, c ),
     (    b    ),
     (       c ),
     (         )
)
```

The individual elements of a CUBE or ROLLUP clause may be either individual expressions, or sublists of elements in parentheses. In the latter case, the sublists are treated as single units for the purposes of generating the individual grouping sets. For example:

```
CUBE ( (a, b), (c, d) )
```

is equivalent to

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b       ),
    (       c, d ),
    (            )
)
```

and

```
ROLLUP ( a, (b, c), d )
```

is equivalent to

```
GROUPING SETS (
    ( a, b, c, d ),
    ( a, b, c    ),
    ( a          ),
    (            )
)
```

The `CUBE` and `ROLLUP` constructs can be used either directly in the `GROUP BY` clause, or nested inside a `GROUPING SETS` clause. If one `GROUPING SETS` clause is nested inside another, the effect is the same as if all the elements of the inner clause had been written directly in the outer clause.

If multiple grouping items are specified in a single `GROUP BY` clause, then the final list of grouping sets is the cross product of the individual items. For example:

```
GROUP BY a, CUBE (b, c), GROUPING SETS ((d), (e))
```

is equivalent to

```
GROUP BY GROUPING SETS (
    (a, b, c, d), (a, b, c, e),
    (a, b, d),    (a, b, e),
    (a, c, d),    (a, c, e),
    (a, d),       (a, e)
)
```

> **Note:** The construct `(a, b)` is normally recognized in expressions as a row constructor. Within the `GROUP BY` clause, this does not apply at the top levels of expressions, and `(a, b)` is parsed as a list of expressions as described above. If for some reason you *need* a row constructor in a grouping expression, use `ROW(a, b)`.

7.2.5. Window Function Processing

If the query contains any window functions (see Section 3.5, Section 9.21 and Section 4.2.8), these functions are evaluated after any grouping, aggregation, and `HAVING` filtering is performed. That is, if the query uses any aggregates, `GROUP BY`, or `HAVING`, then the rows seen by the window functions are the group rows instead of the original table rows from `FROM`/`WHERE`.

When multiple window functions are used, all the window functions having syntactically equivalent `PARTITION BY` and `ORDER BY` clauses in their window definitions are guaranteed to be evaluated in a single pass over the data. Therefore they will see the same sort ordering, even if the `ORDER BY` does not uniquely determine an ordering. However, no guarantees are made about the evaluation of functions having different `PARTITION BY` or `ORDER BY` specifications. (In such cases a sort step is typically required between the passes of window function evaluations, and the sort is not guaranteed to preserve ordering of rows that its `ORDER BY` sees as equivalent.)

Currently, window functions always require presorted data, and so the query output will be ordered according to one or another of the window functions' `PARTITION BY`/`ORDER BY` clauses. It is not recommended to rely on this, however. Use an explicit top-level `ORDER BY` clause if you want to be sure the results are sorted in a particular way.

7.3. Select Lists

As shown in the previous section, the table expression in the `SELECT` command constructs an intermediate virtual table by possibly combining tables, views, eliminating rows, grouping, etc. This table is finally passed on to processing by the *select list*. The select list determines which *columns* of the intermediate table are actually output.

7.3.1. Select-List Items

The simplest kind of select list is `*` which emits all columns that the table expression produces. Otherwise, a select list is a comma-separated list of value expressions (as defined in Section 4.2). For instance, it could be a list of column names:

```
SELECT a, b, c FROM ...
```

The columns names `a`, `b`, and `c` are either the actual names of the columns of tables referenced in the `FROM` clause, or the aliases given to them as explained in Section 7.2.1.2. The name space available in the select list is the same as in the `WHERE` clause, unless grouping is used, in which case it is the same as in the `HAVING` clause.

If more than one table has a column of the same name, the table name must also be given, as in:

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

When working with multiple tables, it can also be useful to ask for all the columns of a particular table:

```
SELECT tbl1.*, tbl2.a FROM ...
```

See Section 8.16.5 for more about the *`table_name`*`.*` notation.

If an arbitrary value expression is used in the select list, it conceptually adds a new virtual column to the returned table. The value expression is evaluated once for each result row, with the row's values substituted for any column references. But the expressions in the select list do not have to reference any columns in the table expression of the FROM clause; they can be constant arithmetic expressions, for instance.

7.3.2. Column Labels

The entries in the select list can be assigned names for subsequent processing, such as for use in an ORDER BY clause or for display by the client application. For example:

```
SELECT a AS value, b + c AS sum FROM ...
```

If no output column name is specified using AS, the system assigns a default column name. For simple column references, this is the name of the referenced column. For function calls, this is the name of the function. For complex expressions, the system will generate a generic name.

The AS keyword is optional, but only if the new column name does not match any PostgreSQL keyword (see Appendix C). To avoid an accidental match to a keyword, you can double-quote the column name. For example, VALUE is a keyword, so this does not work:

```
SELECT a value, b + c AS sum FROM ...
```

but this does:

```
SELECT a "value", b + c AS sum FROM ...
```

For protection against possible future keyword additions, it is recommended that you always either write AS or double-quote the output column name.

> **Note:** The naming of output columns here is different from that done in the FROM clause (see Section 7.2.1.2). It is possible to rename the same column twice, but the name assigned in the select list is the one that will be passed on.

7.3.3. DISTINCT

After the select list has been processed, the result table can optionally be subject to the elimination of duplicate rows. The DISTINCT key word is written directly after SELECT to specify this:

```
SELECT DISTINCT select_list ...
```

(Instead of DISTINCT the key word ALL can be used to specify the default behavior of retaining all rows.)

Obviously, two rows are considered distinct if they differ in at least one column value. Null values are considered equal in this comparison.

Alternatively, an arbitrary expression can determine what rows are to be considered distinct:

```
SELECT DISTINCT ON (expression [, expression ...]) select_list ...
```

Here *expression* is an arbitrary value expression that is evaluated for all rows. A set of rows for which all the expressions are equal are considered duplicates, and only the first row of the set is kept in the output. Note that the "first row" of a set is unpredictable unless the query is sorted on enough columns to guarantee a unique ordering of the rows arriving at the DISTINCT filter. (DISTINCT ON processing occurs after ORDER BY sorting.)

The DISTINCT ON clause is not part of the SQL standard and is sometimes considered bad style because of the potentially indeterminate nature of its results. With judicious use of GROUP BY and subqueries in FROM, this construct can be avoided, but it is often the most convenient alternative.

7.4. Combining Queries

The results of two queries can be combined using the set operations union, intersection, and difference. The syntax is

```
query1 UNION [ALL] query2
query1 INTERSECT [ALL] query2
query1 EXCEPT [ALL] query2
```

query1 and *query2* are queries that can use any of the features discussed up to this point. Set operations can also be nested and chained, for example

```
query1 UNION query2 UNION query3
```

which is executed as:

```
(query1 UNION query2) UNION query3
```

UNION effectively appends the result of *query2* to the result of *query1* (although there is no guarantee that this is the order in which the rows are actually returned). Furthermore, it eliminates duplicate rows from its result, in the same way as DISTINCT, unless UNION ALL is used.

INTERSECT returns all rows that are both in the result of *query1* and in the result of *query2*. Duplicate rows are eliminated unless INTERSECT ALL is used.

EXCEPT returns all rows that are in the result of *query1* but not in the result of *query2*. (This is sometimes called the *difference* between two queries.) Again, duplicates are eliminated unless EXCEPT ALL is used.

In order to calculate the union, intersection, or difference of two queries, the two queries must be "union compatible", which means that they return the same number of columns and the corresponding columns have compatible data types, as described in Section 10.5.

7.5. Sorting Rows

After a query has produced an output table (after the select list has been processed) it can optionally be sorted. If sorting is not chosen, the rows will be returned in an unspecified order. The actual order in that case will depend on the scan and join plan types and the order on disk, but it must not be relied on. A particular output ordering can only be guaranteed if the sort step is explicitly chosen.

The `ORDER BY` clause specifies the sort order:

```
SELECT select_list
    FROM table_expression
    ORDER BY sort_expression1 [ASC | DESC] [NULLS { FIRST | LAST }]
              [, sort_expression2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```

The sort expression(s) can be any expression that would be valid in the query's select list. An example is:

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

When more than one expression is specified, the later values are used to sort rows that are equal according to the earlier values. Each expression can be followed by an optional `ASC` or `DESC` keyword to set the sort direction to ascending or descending. `ASC` order is the default. Ascending order puts smaller values first, where "smaller" is defined in terms of the < operator. Similarly, descending order is determined with the > operator. [1]

The `NULLS FIRST` and `NULLS LAST` options can be used to determine whether nulls appear before or after non-null values in the sort ordering. By default, null values sort as if larger than any non-null value; that is, `NULLS FIRST` is the default for `DESC` order, and `NULLS LAST` otherwise.

Note that the ordering options are considered independently for each sort column. For example `ORDER BY x, y DESC` means `ORDER BY x ASC, y DESC`, which is not the same as `ORDER BY x DESC, y DESC`.

A *`sort_expression`* can also be the column label or number of an output column, as in:

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

both of which sort by the first output column. Note that an output column name has to stand alone, that is, it cannot be used in an expression — for example, this is *not* correct:

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c;          -- wrong
```

This restriction is made to reduce ambiguity. There is still ambiguity if an `ORDER BY` item is a simple name that could match either an output column name or a column from the table expression. The output column is used in such cases. This would only cause confusion if you use `AS` to rename an output column to match some other table column's name.

`ORDER BY` can be applied to the result of a `UNION`, `INTERSECT`, or `EXCEPT` combination, but in this case it is only permitted to sort by output column names or numbers, not by expressions.

1. Actually, PostgreSQL uses the *default B-tree operator class* for the expression's data type to determine the sort ordering for `ASC` and `DESC`. Conventionally, data types will be set up so that the < and > operators correspond to this sort ordering, but a user-defined data type's designer could choose to do something different.

7.6. LIMIT and OFFSET

LIMIT and OFFSET allow you to retrieve just a portion of the rows that are generated by the rest of the query:

```
SELECT select_list
    FROM table_expression
    [ ORDER BY ... ]
    [ LIMIT { number | ALL } ] [ OFFSET number ]
```

If a limit count is given, no more than that many rows will be returned (but possibly less, if the query itself yields less rows). LIMIT ALL is the same as omitting the LIMIT clause.

OFFSET says to skip that many rows before beginning to return rows. OFFSET 0 is the same as omitting the OFFSET clause, and LIMIT NULL is the same as omitting the LIMIT clause. If both OFFSET and LIMIT appear, then OFFSET rows are skipped before starting to count the LIMIT rows that are returned.

When using LIMIT, it is important to use an ORDER BY clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query's rows. You might be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? The ordering is unknown, unless you specified ORDER BY.

The query optimizer takes LIMIT into account when generating query plans, so you are very likely to get different plans (yielding different row orders) depending on what you give for LIMIT and OFFSET. Thus, using different LIMIT/OFFSET values to select different subsets of a query result *will give inconsistent results* unless you enforce a predictable result ordering with ORDER BY. This is not a bug; it is an inherent consequence of the fact that SQL does not promise to deliver the results of a query in any particular order unless ORDER BY is used to constrain the order.

The rows skipped by an OFFSET clause still have to be computed inside the server; therefore a large OFFSET might be inefficient.

7.7. VALUES Lists

VALUES provides a way to generate a "constant table" that can be used in a query without having to actually create and populate a table on-disk. The syntax is

```
VALUES ( expression [, ...] ) [, ...]
```

Each parenthesized list of expressions generates a row in the table. The lists must all have the same number of elements (i.e., the number of columns in the table), and corresponding entries in each list must have compatible data types. The actual data type assigned to each column of the result is determined using the same rules as for UNION (see Section 10.5).

As an example:

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

will return a table of two columns and three rows. It's effectively equivalent to:

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

By default, PostgreSQL assigns the names `column1`, `column2`, etc. to the columns of a `VALUES` table. The column names are not specified by the SQL standard and different database systems do it differently, so it's usually better to override the default names with a table alias list, like this:

```
=> SELECT * FROM (VALUES (1, 'one'), (2, 'two'), (3, 'three')) AS t (num,letter);
 num | letter
-----+--------
   1 | one
   2 | two
   3 | three
(3 rows)
```

Syntactically, `VALUES` followed by expression lists is treated as equivalent to:

```
SELECT select_list FROM table_expression
```

and can appear anywhere a `SELECT` can. For example, you can use it as part of a `UNION`, or attach a *sort_specification* (`ORDER BY`, `LIMIT`, and/or `OFFSET`) to it. `VALUES` is most commonly used as the data source in an `INSERT` command, and next most commonly as a subquery.

For more information see VALUES.

7.8. `WITH` Queries (Common Table Expressions)

`WITH` provides a way to write auxiliary statements for use in a larger query. These statements, which are often referred to as Common Table Expressions or CTEs, can be thought of as defining temporary tables that exist just for one query. Each auxiliary statement in a `WITH` clause can be a `SELECT`, `INSERT`, `UPDATE`, or `DELETE`; and the `WITH` clause itself is attached to a primary statement that can also be a `SELECT`, `INSERT`, `UPDATE`, or `DELETE`.

7.8.1. `SELECT` in `WITH`

The basic value of `SELECT` in `WITH` is to break down complicated queries into simpler parts. An example is:

```
WITH regional_sales AS (
        SELECT region, SUM(amount) AS total_sales
        FROM orders
        GROUP BY region
     ), top_regions AS (
        SELECT region
        FROM regional_sales
```

```
        WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
    )
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

which displays per-product sales totals in only the top sales regions. The `WITH` clause defines two auxiliary statements named `regional_sales` and `top_regions`, where the output of `regional_sales` is used in `top_regions` and the output of `top_regions` is used in the primary `SELECT` query. This example could have been written without `WITH`, but we'd have needed two levels of nested sub-`SELECT`s. It's a bit easier to follow this way.

The optional `RECURSIVE` modifier changes `WITH` from a mere syntactic convenience into a feature that accomplishes things not otherwise possible in standard SQL. Using `RECURSIVE`, a `WITH` query can refer to its own output. A very simple example is this query to sum the integers from 1 through 100:

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

The general form of a recursive `WITH` query is always a *non-recursive term*, then `UNION` (or `UNION ALL`), then a *recursive term*, where only the recursive term can contain a reference to the query's own output. Such a query is executed as follows:

Recursive Query Evaluation

1. Evaluate the non-recursive term. For `UNION` (but not `UNION ALL`), discard duplicate rows. Include all remaining rows in the result of the recursive query, and also place them in a temporary *working table*.
2. So long as the working table is not empty, repeat these steps:
    a. Evaluate the recursive term, substituting the current contents of the working table for the recursive self-reference. For `UNION` (but not `UNION ALL`), discard duplicate rows and rows that duplicate any previous result row. Include all remaining rows in the result of the recursive query, and also place them in a temporary *intermediate table*.
    b. Replace the contents of the working table with the contents of the intermediate table, then empty the intermediate table.

Note: Strictly speaking, this process is iteration not recursion, but `RECURSIVE` is the terminology chosen by the SQL standards committee.

In the example above, the working table has just a single row in each step, and it takes on the values from 1 through 100 in successive steps. In the 100th step, there is no output because of the `WHERE` clause, and so the query terminates.

Recursive queries are typically used to deal with hierarchical or tree-structured data. A useful example is this query to find all the direct and indirect sub-parts of a product, given only a table that shows immediate inclusions:

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
  )
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

When working with recursive queries it is important to be sure that the recursive part of the query will eventually return no tuples, or else the query will loop indefinitely. Sometimes, using `UNION` instead of `UNION ALL` can accomplish this by discarding rows that duplicate previous output rows. However, often a cycle does not involve output rows that are completely duplicate: it may be necessary to check just one or a few fields to see if the same point has been reached before. The standard method for handling such situations is to compute an array of the already-visited values. For example, consider the following query that searches a table `graph` using a `link` field:

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
        SELECT g.id, g.link, g.data, 1
        FROM graph g
      UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1
        FROM graph g, search_graph sg
        WHERE g.id = sg.link
)
SELECT * FROM search_graph;
```

This query will loop if the `link` relationships contain cycles. Because we require a "depth" output, just changing `UNION ALL` to `UNION` would not eliminate the looping. Instead we need to recognize whether we have reached the same row again while following a particular path of links. We add two columns `path` and `cycle` to the loop-prone query:

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
        SELECT g.id, g.link, g.data, 1,
          ARRAY[g.id],
          false
        FROM graph g
      UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1,
          path || g.id,
          g.id = ANY(path)
```

```
        FROM graph g, search_graph sg
        WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

Aside from preventing cycles, the array value is often useful in its own right as representing the "path" taken to reach any particular row.

In the general case where more than one field needs to be checked to recognize a cycle, use an array of rows. For example, if we needed to compare fields `f1` and `f2`:

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
        SELECT g.id, g.link, g.data, 1,
          ARRAY[ROW(g.f1, g.f2)],
          false
        FROM graph g
      UNION ALL
        SELECT g.id, g.link, g.data, sg.depth + 1,
          path || ROW(g.f1, g.f2),
          ROW(g.f1, g.f2) = ANY(path)
        FROM graph g, search_graph sg
        WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

> **Tip:** Omit the `ROW()` syntax in the common case where only one field needs to be checked to recognize a cycle. This allows a simple array rather than a composite-type array to be used, gaining efficiency.

> **Tip:** The recursive query evaluation algorithm produces its output in breadth-first search order. You can display the results in depth-first search order by making the outer query `ORDER BY` a "path" column constructed in this way.

A helpful trick for testing queries when you are not certain if they might loop is to place a `LIMIT` in the parent query. For example, this query would loop forever without the `LIMIT`:

```
WITH RECURSIVE t(n) AS (
    SELECT 1
  UNION ALL
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

This works because PostgreSQL's implementation evaluates only as many rows of a `WITH` query as are actually fetched by the parent query. Using this trick in production is not recommended, because other systems might work differently. Also, it usually won't work if you make the outer query sort the recursive query's results or join them to some other table, because in such cases the outer query will usually try to fetch all of the `WITH` query's output anyway.

A useful property of WITH queries is that they are evaluated only once per execution of the parent query, even if they are referred to more than once by the parent query or sibling WITH queries. Thus, expensive calculations that are needed in multiple places can be placed within a WITH query to avoid redundant work. Another possible application is to prevent unwanted multiple evaluations of functions with side-effects. However, the other side of this coin is that the optimizer is less able to push restrictions from the parent query down into a WITH query than an ordinary subquery. The WITH query will generally be evaluated as written, without suppression of rows that the parent query might discard afterwards. (But, as mentioned above, evaluation might stop early if the reference(s) to the query demand only a limited number of rows.)

The examples above only show WITH being used with SELECT, but it can be attached in the same way to INSERT, UPDATE, or DELETE. In each case it effectively provides temporary table(s) that can be referred to in the main command.

7.8.2. Data-Modifying Statements in WITH

You can use data-modifying statements (INSERT, UPDATE, or DELETE) in WITH. This allows you to perform several different operations in the same query. An example is:

```
WITH moved_rows AS (
    DELETE FROM products
    WHERE
        "date" >= '2010-10-01' AND
        "date" < '2010-11-01'
    RETURNING *
)
INSERT INTO products_log
SELECT * FROM moved_rows;
```

This query effectively moves rows from products to products_log. The DELETE in WITH deletes the specified rows from products, returning their contents by means of its RETURNING clause; and then the primary query reads that output and inserts it into products_log.

A fine point of the above example is that the WITH clause is attached to the INSERT, not the sub-SELECT within the INSERT. This is necessary because data-modifying statements are only allowed in WITH clauses that are attached to the top-level statement. However, normal WITH visibility rules apply, so it is possible to refer to the WITH statement's output from the sub-SELECT.

Data-modifying statements in WITH usually have RETURNING clauses (see Section 6.4), as shown in the example above. It is the output of the RETURNING clause, *not* the target table of the data-modifying statement, that forms the temporary table that can be referred to by the rest of the query. If a data-modifying statement in WITH lacks a RETURNING clause, then it forms no temporary table and cannot be referred to in the rest of the query. Such a statement will be executed nonetheless. A not-particularly-useful example is:

```
WITH t AS (
    DELETE FROM foo
)
DELETE FROM bar;
```

This example would remove all rows from tables foo and bar. The number of affected rows reported to the client would only include rows removed from bar.

Recursive self-references in data-modifying statements are not allowed. In some cases it is possible to work around this limitation by referring to the output of a recursive WITH, for example:

```
WITH RECURSIVE included_parts(sub_part, part) AS (
    SELECT sub_part, part FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
  )
DELETE FROM parts
  WHERE part IN (SELECT part FROM included_parts);
```

This query would remove all direct and indirect subparts of a product.

Data-modifying statements in WITH are executed exactly once, and always to completion, independently of whether the primary query reads all (or indeed any) of their output. Notice that this is different from the rule for SELECT in WITH: as stated in the previous section, execution of a SELECT is carried only as far as the primary query demands its output.

The sub-statements in WITH are executed concurrently with each other and with the main query. Therefore, when using data-modifying statements in WITH, the order in which the specified updates actually happen is unpredictable. All the statements are executed with the same *snapshot* (see Chapter 13), so they cannot "see" one another's effects on the target tables. This alleviates the effects of the unpredictability of the actual order of row updates, and means that RETURNING data is the only way to communicate changes between different WITH sub-statements and the main query. An example of this is that in

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM products;
```

the outer SELECT would return the original prices before the action of the UPDATE, while in

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM t;
```

the outer SELECT would return the updated data.

Trying to update the same row twice in a single statement is not supported. Only one of the modifications takes place, but it is not easy (and sometimes not possible) to reliably predict which one. This also applies to deleting a row that was already updated in the same statement: only the update is performed. Therefore you should generally avoid trying to modify a single row twice in a single statement. In particular avoid writing WITH sub-statements that could affect the same rows changed by the main statement or a sibling sub-statement. The effects of such a statement will not be predictable.

At present, any table used as the target of a data-modifying statement in WITH must not have a conditional rule, nor an ALSO rule, nor an INSTEAD rule that expands to multiple statements.

Chapter 8. Data Types

PostgreSQL has a rich set of native data types available to users. Users can add new types to PostgreSQL using the CREATE TYPE command.

Table 8-1 shows all the built-in general-purpose data types. Most of the alternative names listed in the "Aliases" column are the names used internally by PostgreSQL for historical reasons. In addition, some internally used or deprecated types are available, but are not listed here.

Table 8-1. Data Types

Name	Aliases	Description
bigint	int8	signed eight-byte integer
bigserial	serial8	autoincrementing eight-byte integer
bit [(*n*)]		fixed-length bit string
bit varying [(*n*)]	varbit [(*n*)]	variable-length bit string
boolean	bool	logical Boolean (true/false)
box		rectangular box on a plane
bytea		binary data ("byte array")
character [(*n*)]	char [(*n*)]	fixed-length character string
character varying [(*n*)]	varchar [(*n*)]	variable-length character string
cidr		IPv4 or IPv6 network address
circle		circle on a plane
date		calendar date (year, month, day)
double precision	float8	double precision floating-point number (8 bytes)
inet		IPv4 or IPv6 host address
integer	int, int4	signed four-byte integer
interval [*fields*] [(*p*)]		time span
json		textual JSON data
jsonb		binary JSON data, decomposed
line		infinite line on a plane
lseg		line segment on a plane
macaddr		MAC (Media Access Control) address
money		currency amount
numeric [(*p*, *s*)]	decimal [(*p*, *s*)]	exact numeric of selectable precision
path		geometric path on a plane

Name	Aliases	Description
pg_lsn		PostgreSQL Log Sequence Number
point		geometric point on a plane
polygon		closed geometric path on a plane
real	float4	single precision floating-point number (4 bytes)
smallint	int2	signed two-byte integer
smallserial	serial2	autoincrementing two-byte integer
serial	serial4	autoincrementing four-byte integer
text		variable-length character string
time [(*p*)] [without time zone]		time of day (no time zone)
time [(*p*)] with time zone	timetz	time of day, including time zone
timestamp [(*p*)] [without time zone]		date and time (no time zone)
timestamp [(*p*)] with time zone	timestamptz	date and time, including time zone
tsquery		text search query
tsvector		text search document
txid_snapshot		user-level transaction ID snapshot
uuid		universally unique identifier
xml		XML data

Compatibility: The following types (or spellings thereof) are specified by SQL: bigint, bit, bit varying, boolean, char, character varying, character, varchar, date, double precision, integer, interval, numeric, decimal, real, smallint, time (with or without time zone), timestamp (with or without time zone), xml.

Each data type has an external representation determined by its input and output functions. Many of the built-in types have obvious external formats. However, several types are either unique to PostgreSQL, such as geometric paths, or have several possible formats, such as the date and time types. Some of the input and output functions are not invertible, i.e., the result of an output function might lose accuracy when compared to the original input.

8.1. Numeric Types

Numeric types consist of two-, four-, and eight-byte integers, four- and eight-byte floating-point numbers,

and selectable-precision decimals. Table 8-2 lists the available types.

Table 8-2. Numeric Types

Name	Storage Size	Description	Range
`smallint`	2 bytes	small-range integer	-32768 to +32767
`integer`	4 bytes	typical choice for integer	-2147483648 to +2147483647
`bigint`	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
`decimal`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`numeric`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`real`	4 bytes	variable-precision, inexact	6 decimal digits precision
`double precision`	8 bytes	variable-precision, inexact	15 decimal digits precision
`smallserial`	2 bytes	small autoincrementing integer	1 to 32767
`serial`	4 bytes	autoincrementing integer	1 to 2147483647
`bigserial`	8 bytes	large autoincrementing integer	1 to 9223372036854775807

The syntax of constants for the numeric types is described in Section 4.1.2. The numeric types have a full set of corresponding arithmetic operators and functions. Refer to Chapter 9 for more information. The following sections describe the types in detail.

8.1.1. Integer Types

The types `smallint`, `integer`, and `bigint` store whole numbers, that is, numbers without fractional components, of various ranges. Attempts to store values outside of the allowed range will result in an error.

The type `integer` is the common choice, as it offers the best balance between range, storage size, and performance. The `smallint` type is generally only used if disk space is at a premium. The `bigint` type is designed to be used when the range of the `integer` type is insufficient.

SQL only specifies the integer types `integer` (or `int`), `smallint`, and `bigint`. The type names `int2`, `int4`, and `int8` are extensions, which are also used by some other SQL database systems.

8.1.2. Arbitrary Precision Numbers

The type `numeric` can store numbers with a very large number of digits. It is especially recommended for storing monetary amounts and other quantities where exactness is required. Calculations with `numeric` values yield exact results where possible, e.g., addition, subtraction, multiplication. However, calculations on `numeric` values are very slow compared to the integer types, or to the floating-point types described in the next section.

We use the following terms below: the *precision* of a `numeric` is the total count of significant digits in the whole number, that is, the number of digits to both sides of the decimal point. The *scale* of a `numeric` is the count of decimal digits in the fractional part, to the right of the decimal point. So the number 23.5141 has a precision of 6 and a scale of 4. Integers can be considered to have a scale of zero.

Both the maximum precision and the maximum scale of a `numeric` column can be configured. To declare a column of type `numeric` use the syntax:

```
NUMERIC(precision, scale)
```

The precision must be positive, the scale zero or positive. Alternatively:

```
NUMERIC(precision)
```

selects a scale of 0. Specifying:

```
NUMERIC
```

without any precision or scale creates a column in which numeric values of any precision and scale can be stored, up to the implementation limit on precision. A column of this kind will not coerce input values to any particular scale, whereas `numeric` columns with a declared scale will coerce input values to that scale. (The SQL standard requires a default scale of 0, i.e., coercion to integer precision. We find this a bit useless. If you're concerned about portability, always specify the precision and scale explicitly.)

> **Note:** The maximum allowed precision when explicitly specified in the type declaration is 1000; `NUMERIC` without a specified precision is subject to the limits described in Table 8-2.

If the scale of a value to be stored is greater than the declared scale of the column, the system will round the value to the specified number of fractional digits. Then, if the number of digits to the left of the decimal point exceeds the declared precision minus the declared scale, an error is raised.

Numeric values are physically stored without any extra leading or trailing zeroes. Thus, the declared precision and scale of a column are maximums, not fixed allocations. (In this sense the `numeric` type is more akin to `varchar(n)` than to `char(n)`.) The actual storage requirement is two bytes for each group of four decimal digits, plus three to eight bytes overhead.

In addition to ordinary numeric values, the `numeric` type allows the special value `NaN`, meaning "not-a-number". Any operation on `NaN` yields another `NaN`. When writing this value as a constant in an SQL command, you must put quotes around it, for example `UPDATE table SET x = 'NaN'`. On input, the string `NaN` is recognized in a case-insensitive manner.

Note: In most implementations of the "not-a-number" concept, `NaN` is not considered equal to any other numeric value (including `NaN`). In order to allow `numeric` values to be sorted and used in tree-based indexes, PostgreSQL treats `NaN` values as equal, and greater than all non-`NaN` values.

The types `decimal` and `numeric` are equivalent. Both types are part of the SQL standard.

8.1.3. Floating-Point Types

The data types `real` and `double precision` are inexact, variable-precision numeric types. In practice, these types are usually implementations of IEEE Standard 754 for Binary Floating-Point Arithmetic (single and double precision, respectively), to the extent that the underlying processor, operating system, and compiler support it.

Inexact means that some values cannot be converted exactly to the internal format and are stored as approximations, so that storing and retrieving a value might show slight discrepancies. Managing these errors and how they propagate through calculations is the subject of an entire branch of mathematics and computer science and will not be discussed here, except for the following points:

- If you require exact storage and calculations (such as for monetary amounts), use the `numeric` type instead.
- If you want to do complicated calculations with these types for anything important, especially if you rely on certain behavior in boundary cases (infinity, underflow), you should evaluate the implementation carefully.
- Comparing two floating-point values for equality might not always work as expected.

On most platforms, the `real` type has a range of at least 1E-37 to 1E+37 with a precision of at least 6 decimal digits. The `double precision` type typically has a range of around 1E-307 to 1E+308 with a precision of at least 15 digits. Values that are too large or too small will cause an error. Rounding might take place if the precision of an input number is too high. Numbers too close to zero that are not representable as distinct from zero will cause an underflow error.

Note: The extra_float_digits setting controls the number of extra significant digits included when a floating point value is converted to text for output. With the default value of `0`, the output is the same on every platform supported by PostgreSQL. Increasing it will produce output that more accurately represents the stored value, but may be unportable.

In addition to ordinary numeric values, the floating-point types have several special values:

```
Infinity
-Infinity
NaN
```

These represent the IEEE 754 special values "infinity", "negative infinity", and "not-a-number", respectively. (On a machine whose floating-point arithmetic does not follow IEEE 754, these values will prob-

ably not work as expected.) When writing these values as constants in an SQL command, you must put quotes around them, for example `UPDATE table SET x = 'Infinity'`. On input, these strings are recognized in a case-insensitive manner.

> **Note:** IEEE754 specifies that `NaN` should not compare equal to any other floating-point value (including `NaN`). In order to allow floating-point values to be sorted and used in tree-based indexes, PostgreSQL treats `NaN` values as equal, and greater than all non-`NaN` values.

PostgreSQL also supports the SQL-standard notations `float` and `float(`*p*`)` for specifying inexact numeric types. Here, *p* specifies the minimum acceptable precision in *binary* digits. PostgreSQL accepts `float(1)` to `float(24)` as selecting the `real` type, while `float(25)` to `float(53)` select `double precision`. Values of *p* outside the allowed range draw an error. `float` with no precision specified is taken to mean `double precision`.

> **Note:** The assumption that `real` and `double precision` have exactly 24 and 53 bits in the mantissa respectively is correct for IEEE-standard floating point implementations. On non-IEEE platforms it might be off a little, but for simplicity the same ranges of *p* are used on all platforms.

8.1.4. Serial Types

The data types `smallserial`, `serial` and `bigserial` are not true types, but merely a notational convenience for creating unique identifier columns (similar to the `AUTO_INCREMENT` property supported by some other databases). In the current implementation, specifying:

```
CREATE TABLE tablename (
    colname SERIAL
);
```

is equivalent to specifying:

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename (
    colname integer NOT NULL DEFAULT nextval('tablename_colname_seq')
);
ALTER SEQUENCE tablename_colname_seq OWNED BY tablename.colname;
```

Thus, we have created an integer column and arranged for its default values to be assigned from a sequence generator. A `NOT NULL` constraint is applied to ensure that a null value cannot be inserted. (In most cases you would also want to attach a `UNIQUE` or `PRIMARY KEY` constraint to prevent duplicate values from being inserted by accident, but this is not automatic.) Lastly, the sequence is marked as "owned by" the column, so that it will be dropped if the column or table is dropped.

> **Note:** Because `smallserial`, `serial` and `bigserial` are implemented using sequences, there may be "holes" or gaps in the sequence of values which appears in the column, even if no rows are ever deleted. A value allocated from the sequence is still "used up" even if a row containing that value

is never successfully inserted into the table column. This may happen, for example, if the inserting transaction rolls back. See `nextval()` in Section 9.16 for details.

To insert the next value of the sequence into the `serial` column, specify that the `serial` column should be assigned its default value. This can be done either by excluding the column from the list of columns in the `INSERT` statement, or through the use of the `DEFAULT` key word.

The type names `serial` and `serial4` are equivalent: both create `integer` columns. The type names `bigserial` and `serial8` work the same way, except that they create a `bigint` column. `bigserial` should be used if you anticipate the use of more than 2^{31} identifiers over the lifetime of the table. The type names `smallserial` and `serial2` also work the same way, except that they create a `smallint` column.

The sequence created for a `serial` column is automatically dropped when the owning column is dropped. You can drop the sequence without dropping the column, but this will force removal of the column default expression.

8.2. Monetary Types

The `money` type stores a currency amount with a fixed fractional precision; see Table 8-3. The fractional precision is determined by the database's lc_monetary setting. The range shown in the table assumes there are two fractional digits. Input is accepted in a variety of formats, including integer and floating-point literals, as well as typical currency formatting, such as `'$1,000.00'`. Output is generally in the latter form but depends on the locale.

Table 8-3. Monetary Types

Name	Storage Size	Description	Range
money	8 bytes	currency amount	-92233720368547758.08 to +92233720368547758.07

Since the output of this data type is locale-sensitive, it might not work to load `money` data into a database that has a different setting of `lc_monetary`. To avoid problems, before restoring a dump into a new database make sure `lc_monetary` has the same or equivalent value as in the database that was dumped.

Values of the `numeric`, `int`, and `bigint` data types can be cast to `money`. Conversion from the `real` and `double precision` data types can be done by casting to `numeric` first, for example:

```
SELECT '12.34'::float8::numeric::money;
```

However, this is not recommended. Floating point numbers should not be used to handle money due to the potential for rounding errors.

A `money` value can be cast to `numeric` without loss of precision. Conversion to other types could potentially lose precision, and must also be done in two stages:

```
SELECT '52093.89'::money::numeric::float8;
```

Division of a `money` value by an integer value is performed with truncation of the fractional part towards zero. To get a rounded result, divide by a floating-point value, or cast the `money` value to `numeric` before dividing and back to `money` afterwards. (The latter is preferable to avoid risking precision loss.) When a `money` value is divided by another `money` value, the result is `double precision` (i.e., a pure number, not money); the currency units cancel each other out in the division.

8.3. Character Types

Table 8-4. Character Types

Name	Description
`character varying(n)`, `varchar(n)`	variable-length with limit
`character(n)`, `char(n)`	fixed-length, blank padded
`text`	variable unlimited length

Table 8-4 shows the general-purpose character types available in PostgreSQL.

SQL defines two primary character types: `character varying(n)` and `character(n)`, where *n* is a positive integer. Both of these types can store strings up to *n* characters (not bytes) in length. An attempt to store a longer string into a column of these types will result in an error, unless the excess characters are all spaces, in which case the string will be truncated to the maximum length. (This somewhat bizarre exception is required by the SQL standard.) If the string to be stored is shorter than the declared length, values of type `character` will be space-padded; values of type `character varying` will simply store the shorter string.

If one explicitly casts a value to `character varying(n)` or `character(n)`, then an over-length value will be truncated to *n* characters without raising an error. (This too is required by the SQL standard.)

The notations `varchar(n)` and `char(n)` are aliases for `character varying(n)` and `character(n)`, respectively. `character` without length specifier is equivalent to `character(1)`. If `character varying` is used without length specifier, the type accepts strings of any size. The latter is a PostgreSQL extension.

In addition, PostgreSQL provides the `text` type, which stores strings of any length. Although the type `text` is not in the SQL standard, several other SQL database management systems have it as well.

Values of type `character` are physically padded with spaces to the specified width *n*, and are stored and displayed that way. However, trailing spaces are treated as semantically insignificant and disregarded when comparing two values of type `character`. In collations where whitespace is significant, this behavior can produce unexpected results; for example `SELECT 'a '::CHAR(2) collate "C" < E'a\n'::CHAR(2)` returns true, even though `C` locale would consider a space to be greater than a newline. Trailing spaces are removed when converting a `character` value to one of the other string types. Note that trailing spaces *are* semantically significant in `character varying` and `text` values, and when using pattern matching, that is `LIKE` and regular expressions.

The storage requirement for a short string (up to 126 bytes) is 1 byte plus the actual string, which includes the space padding in the case of `character`. Longer strings have 4 bytes of overhead instead of 1. Long strings are compressed by the system automatically, so the physical requirement on disk might be less. Very long values are also stored in background tables so that they do not interfere with rapid access to shorter column values. In any case, the longest possible character string that can be stored is about 1 GB. (The maximum value that will be allowed for *n* in the data type declaration is less than that. It wouldn't be useful to change this because with multibyte character encodings the number of characters and bytes can be quite different. If you desire to store long strings with no specific upper limit, use `text` or `character varying` without a length specifier, rather than making up an arbitrary length limit.)

> **Tip:** There is no performance difference among these three types, apart from increased storage space when using the blank-padded type, and a few extra CPU cycles to check the length when storing into a length-constrained column. While `character(n)` has performance advantages in some other database systems, there is no such advantage in PostgreSQL; in fact `character(n)` is usually the slowest of the three because of its additional storage costs and slower sorting. In most situations `text` or `character varying` should be used instead.

Refer to Section 4.1.2.1 for information about the syntax of string literals, and to Chapter 9 for information about available operators and functions. The database character set determines the character set used to store textual values; for more information on character set support, refer to Section 22.3.

Example 8-1. Using the Character Types

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1; -- ❶
  a   | char_length
------+-------------
 ok   |           2

CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good      ');
INSERT INTO test2 VALUES ('too long');
ERROR:  value too long for type character varying(5)
INSERT INTO test2 VALUES ('too long'::varchar(5)); -- explicit truncation
SELECT b, char_length(b) FROM test2;
   b   | char_length
-------+-------------
 ok    |           2
 good  |           5
 too l |           5
```

❶ The `char_length` function is discussed in Section 9.4.

There are two other fixed-length character types in PostgreSQL, shown in Table 8-5. The `name` type exists *only* for the storage of identifiers in the internal system catalogs and is not intended for use by the general user. Its length is currently defined as 64 bytes (63 usable characters plus terminator) but should

be referenced using the constant NAMEDATALEN in C source code. The length is set at compile time (and is therefore adjustable for special uses); the default maximum length might change in a future release. The type "char" (note the quotes) is different from char(1) in that it only uses one byte of storage. It is internally used in the system catalogs as a simplistic enumeration type.

Table 8-5. Special Character Types

Name	Storage Size	Description
"char"	1 byte	single-byte internal type
name	64 bytes	internal type for object names

8.4. Binary Data Types

The bytea data type allows storage of binary strings; see Table 8-6.

Table 8-6. Binary Data Types

Name	Storage Size	Description
bytea	1 or 4 bytes plus the actual binary string	variable-length binary string

A binary string is a sequence of octets (or bytes). Binary strings are distinguished from character strings in two ways. First, binary strings specifically allow storing octets of value zero and other "non-printable" octets (usually, octets outside the decimal range 32 to 126). Character strings disallow zero octets, and also disallow any other octet values and sequences of octet values that are invalid according to the database's selected character set encoding. Second, operations on binary strings process the actual bytes, whereas the processing of character strings depends on locale settings. In short, binary strings are appropriate for storing data that the programmer thinks of as "raw bytes", whereas character strings are appropriate for storing text.

The bytea type supports two formats for input and output: "hex" format and PostgreSQL's historical "escape" format. Both of these are always accepted on input. The output format depends on the configuration parameter bytea_output; the default is hex. (Note that the hex format was introduced in PostgreSQL 9.0; earlier versions and some tools don't understand it.)

The SQL standard defines a different binary string type, called BLOB or BINARY LARGE OBJECT. The input format is different from bytea, but the provided functions and operators are mostly the same.

8.4.1. bytea Hex Format

The "hex" format encodes binary data as 2 hexadecimal digits per byte, most significant nibble first. The entire string is preceded by the sequence \x (to distinguish it from the escape format). In some contexts, the initial backslash may need to be escaped by doubling it (see Section 4.1.2.1). For input, the hexadecimal digits can be either upper or lower case, and whitespace is permitted between digit pairs (but not within a digit pair nor in the starting \x sequence). The hex format is compatible with a wide range of

external applications and protocols, and it tends to be faster to convert than the escape format, so its use is preferred.

Example:

```
SELECT '\xDEADBEEF';
```

8.4.2. `bytea` Escape Format

The "escape" format is the traditional PostgreSQL format for the `bytea` type. It takes the approach of representing a binary string as a sequence of ASCII characters, while converting those bytes that cannot be represented as an ASCII character into special escape sequences. If, from the point of view of the application, representing bytes as characters makes sense, then this representation can be convenient. But in practice it is usually confusing because it fuzzes up the distinction between binary strings and character strings, and also the particular escape mechanism that was chosen is somewhat unwieldy. Therefore, this format should probably be avoided for most new applications.

When entering `bytea` values in escape format, octets of certain values *must* be escaped, while all octet values *can* be escaped. In general, to escape an octet, convert it into its three-digit octal value and precede it by a backslash. Backslash itself (octet decimal value 92) can alternatively be represented by double backslashes. Table 8-7 shows the characters that must be escaped, and gives the alternative escape sequences where applicable.

Table 8-7. `bytea` Literal Escaped Octets

Decimal Octet Value	Description	Escaped Input Representation	Example	Hex Representation
0	zero octet	`'\000'`	`SELECT '\000'::bytea;`	`\x00`
39	single quote	`''` or `'\047'`	`SELECT ''::bytea;`	`\x27`
92	backslash	`'\\'` or `'\134'`	`SELECT '\\'::bytea;`	`\x5c`
0 to 31 and 127 to 255	"non-printable" octets	`'\`*xxx*`'` (octal value)	`SELECT '\001'::bytea;`	`\x01`

The requirement to escape *non-printable* octets varies depending on locale settings. In some instances you can get away with leaving them unescaped.

The reason that single quotes must be doubled, as shown in Table 8-7, is that this is true for any string literal in a SQL command. The generic string-literal parser consumes the outermost single quotes and reduces any pair of single quotes to one data character. What the `bytea` input function sees is just one single quote, which it treats as a plain data character. However, the `bytea` input function treats backslashes as special, and the other behaviors shown in Table 8-7 are implemented by that function.

In some contexts, backslashes must be doubled compared to what is shown above, because the generic string-literal parser will also reduce pairs of backslashes to one data character; see Section 4.1.2.1.

Bytea octets are output in hex format by default. If you change bytea_output to escape, "non-printable" octets are converted to their equivalent three-digit octal value and preceded by one backslash. Most "printable" octets are output by their standard representation in the client character set, e.g.:

```
SET bytea_output = 'escape';

SELECT 'abc \153\154\155 \052\251\124'::bytea;
     bytea
----------------
 abc klm *\251T
```

The octet with decimal value 92 (backslash) is doubled in the output. Details are in Table 8-8.

Table 8-8. `bytea` Output Escaped Octets

Decimal Octet Value	Description	Escaped Output Representation	Example	Output Result
92	backslash	\\	`SELECT '\134'::bytea;`	\\
0 to 31 and 127 to 255	"non-printable" octets	*xxx* (octal value)	`SELECT '\001'::bytea;`	\001
32 to 126	"printable" octets	client character set representation	`SELECT '\176'::bytea;`	~

Depending on the front end to PostgreSQL you use, you might have additional work to do in terms of escaping and unescaping bytea strings. For example, you might also have to escape line feeds and carriage returns if your interface automatically translates these.

8.5. Date/Time Types

PostgreSQL supports the full set of SQL date and time types, shown in Table 8-9. The operations available on these data types are described in Section 9.9. Dates are counted according to the Gregorian calendar, even in years before that calendar was introduced (see Section B.6 for more information).

Table 8-9. Date/Time Types

Name	Storage Size	Description	Low Value	High Value	Resolution
timestamp [(*p*)] [without time zone]	8 bytes	both date and time (no time zone)	4713 BC	294276 AD	1 microsecond / 14 digits
timestamp [(*p*)] with time zone	8 bytes	both date and time, with time zone	4713 BC	294276 AD	1 microsecond / 14 digits

Name	Storage Size	Description	Low Value	High Value	Resolution
date	4 bytes	date (no time of day)	4713 BC	5874897 AD	1 day
time [(p)] [without time zone]	8 bytes	time of day (no date)	00:00:00	24:00:00	1 microsecond / 14 digits
time [(p)] with time zone	12 bytes	times of day only, with time zone	00:00:00+1559	24:00:00-1559	1 microsecond / 14 digits
interval [*fields*] [(p)]	16 bytes	time interval	-178000000 years	178000000 years	1 microsecond / 14 digits

Note: The SQL standard requires that writing just `timestamp` be equivalent to `timestamp without time zone`, and PostgreSQL honors that behavior. `timestamptz` is accepted as an abbreviation for `timestamp with time zone`; this is a PostgreSQL extension.

`time`, `timestamp`, and `interval` accept an optional precision value *p* which specifies the number of fractional digits retained in the seconds field. By default, there is no explicit bound on precision. The allowed range of *p* is from 0 to 6 for the `timestamp` and `interval` types.

Note: When `timestamp` values are stored as eight-byte integers (currently the default), microsecond precision is available over the full range of values. When `timestamp` values are stored as double precision floating-point numbers instead (a deprecated compile-time option), the effective limit of precision might be less than 6. `timestamp` values are stored as seconds before or after midnight 2000-01-01. When `timestamp` values are implemented using floating-point numbers, microsecond precision is achieved for dates within a few years of 2000-01-01, but the precision degrades for dates further away. Note that using floating-point datetimes allows a larger range of `timestamp` values to be represented than shown above: from 4713 BC up to 5874897 AD.

The same compile-time option also determines whether `time` and `interval` values are stored as floating-point numbers or eight-byte integers. In the floating-point case, large `interval` values degrade in precision as the size of the interval increases.

For the `time` types, the allowed range of *p* is from 0 to 6 when eight-byte integer storage is used, or from 0 to 10 when floating-point storage is used.

The `interval` type has an additional option, which is to restrict the set of stored fields by writing one of these phrases:

```
YEAR
MONTH
DAY
HOUR
MINUTE
SECOND
YEAR TO MONTH
```

```
DAY TO HOUR
DAY TO MINUTE
DAY TO SECOND
HOUR TO MINUTE
HOUR TO SECOND
MINUTE TO SECOND
```

Note that if both *fields* and *p* are specified, the *fields* must include SECOND, since the precision applies only to the seconds.

The type time with time zone is defined by the SQL standard, but the definition exhibits properties which lead to questionable usefulness. In most cases, a combination of date, time, timestamp without time zone, and timestamp with time zone should provide a complete range of date/time functionality required by any application.

The types abstime and reltime are lower precision types which are used internally. You are discouraged from using these types in applications; these internal types might disappear in a future release.

8.5.1. Date/Time Input

Date and time input is accepted in almost any reasonable format, including ISO 8601, SQL-compatible, traditional POSTGRES, and others. For some formats, ordering of day, month, and year in date input is ambiguous and there is support for specifying the expected ordering of these fields. Set the DateStyle parameter to MDY to select month-day-year interpretation, DMY to select day-month-year interpretation, or YMD to select year-month-day interpretation.

PostgreSQL is more flexible in handling date/time input than the SQL standard requires. See Appendix B for the exact parsing rules of date/time input and for the recognized text fields including months, days of the week, and time zones.

Remember that any date or time literal input needs to be enclosed in single quotes, like text strings. Refer to Section 4.1.2.7 for more information. SQL requires the following syntax

```
type [ (p) ] 'value'
```

where *p* is an optional precision specification giving the number of fractional digits in the seconds field. Precision can be specified for time, timestamp, and interval types. The allowed values are mentioned above. If no precision is specified in a constant specification, it defaults to the precision of the literal value.

8.5.1.1. Dates

Table 8-10 shows some possible inputs for the date type.

Table 8-10. Date Input

Example	Description
1999-01-08	ISO 8601; January 8 in any mode (recommended format)
January 8, 1999	unambiguous in any datestyle input mode
1/8/1999	January 8 in MDY mode; August 1 in DMY mode

Example	Description
1/18/1999	January 18 in `MDY` mode; rejected in other modes
01/02/03	January 2, 2003 in `MDY` mode; February 1, 2003 in `DMY` mode; February 3, 2001 in `YMD` mode
1999-Jan-08	January 8 in any mode
Jan-08-1999	January 8 in any mode
08-Jan-1999	January 8 in any mode
99-Jan-08	January 8 in `YMD` mode, else error
08-Jan-99	January 8, except error in `YMD` mode
Jan-08-99	January 8, except error in `YMD` mode
19990108	ISO 8601; January 8, 1999 in any mode
990108	ISO 8601; January 8, 1999 in any mode
1999.008	year and day of year
J2451187	Julian date
January 8, 99 BC	year 99 BC

8.5.1.2. Times

The time-of-day types are `time [ (`*p*`) ] without time zone` and `time [ (`*p*`) ] with time zone`. `time` alone is equivalent to `time without time zone`.

Valid input for these types consists of a time of day followed by an optional time zone. (See Table 8-11 and Table 8-12.) If a time zone is specified in the input for `time without time zone`, it is silently ignored. You can also specify a date but it will be ignored, except when you use a time zone name that involves a daylight-savings rule, such as `America/New_York`. In this case specifying the date is required in order to determine whether standard or daylight-savings time applies. The appropriate time zone offset is recorded in the `time with time zone` value.

Table 8-11. Time Input

Example	Description
`04:05:06.789`	ISO 8601
`04:05:06`	ISO 8601
`04:05`	ISO 8601
`040506`	ISO 8601
`04:05 AM`	same as 04:05; AM does not affect value
`04:05 PM`	same as 16:05; input hour must be <= 12
`04:05:06.789-8`	ISO 8601
`04:05:06-08:00`	ISO 8601
`04:05-08:00`	ISO 8601
`040506-08`	ISO 8601
`04:05:06 PST`	time zone specified by abbreviation

Example	Description
2003-04-12 04:05:06 America/New_York	time zone specified by full name

Table 8-12. Time Zone Input

Example	Description
PST	Abbreviation (for Pacific Standard Time)
America/New_York	Full time zone name
PST8PDT	POSIX-style time zone specification
-8:00	ISO-8601 offset for PST
-800	ISO-8601 offset for PST
-8	ISO-8601 offset for PST
zulu	Military abbreviation for UTC
z	Short form of zulu

Refer to Section 8.5.3 for more information on how to specify time zones.

8.5.1.3. Time Stamps

Valid input for the time stamp types consists of the concatenation of a date and a time, followed by an optional time zone, followed by an optional AD or BC. (Alternatively, AD/BC can appear before the time zone, but this is not the preferred ordering.) Thus:

```
1999-01-08 04:05:06
```

and:

```
1999-01-08 04:05:06 -8:00
```

are valid values, which follow the ISO 8601 standard. In addition, the common format:

```
January 8 04:05:06 1999 PST
```

is supported.

The SQL standard differentiates timestamp without time zone and timestamp with time zone literals by the presence of a "+" or "-" symbol and time zone offset after the time. Hence, according to the standard,

```
TIMESTAMP '2004-10-19 10:23:54'
```

is a timestamp without time zone, while

```
TIMESTAMP '2004-10-19 10:23:54+02'
```

is a timestamp with time zone. PostgreSQL never examines the content of a literal string before determining its type, and therefore will treat both of the above as timestamp without time zone. To ensure that a literal is treated as timestamp with time zone, give it the correct explicit type:

```
TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'
```

In a literal that has been determined to be `timestamp without time zone`, PostgreSQL will silently ignore any time zone indication. That is, the resulting value is derived from the date/time fields in the input value, and is not adjusted for time zone.

For `timestamp with time zone`, the internally stored value is always in UTC (Universal Coordinated Time, traditionally known as Greenwich Mean Time, GMT). An input value that has an explicit time zone specified is converted to UTC using the appropriate offset for that time zone. If no time zone is stated in the input string, then it is assumed to be in the time zone indicated by the system's TimeZone parameter, and is converted to UTC using the offset for the `timezone` zone.

When a `timestamp with time zone` value is output, it is always converted from UTC to the current `timezone` zone, and displayed as local time in that zone. To see the time in another time zone, either change `timezone` or use the `AT TIME ZONE` construct (see Section 9.9.3).

Conversions between `timestamp without time zone` and `timestamp with time zone` normally assume that the `timestamp without time zone` value should be taken or given as `timezone` local time. A different time zone can be specified for the conversion using `AT TIME ZONE`.

8.5.1.4. Special Values

PostgreSQL supports several special date/time input values for convenience, as shown in Table 8-13. The values `infinity` and `-infinity` are specially represented inside the system and will be displayed unchanged; but the others are simply notational shorthands that will be converted to ordinary date/time values when read. (In particular, `now` and related strings are converted to a specific time value as soon as they are read.) All of these values need to be enclosed in single quotes when used as constants in SQL commands.

Table 8-13. Special Date/Time Inputs

Input String	Valid Types	Description
`epoch`	`date`, `timestamp`	1970-01-01 00:00:00+00 (Unix system time zero)
`infinity`	`date`, `timestamp`	later than all other time stamps
`-infinity`	`date`, `timestamp`	earlier than all other time stamps
`now`	`date`, `time`, `timestamp`	current transaction's start time
`today`	`date`, `timestamp`	midnight (`00:00`) today
`tomorrow`	`date`, `timestamp`	midnight (`00:00`) tomorrow
`yesterday`	`date`, `timestamp`	midnight (`00:00`) yesterday
`allballs`	`time`	00:00:00.00 UTC

The following SQL-compatible functions can also be used to obtain the current time value for the corresponding data type: `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME`, `LOCALTIMESTAMP`. (See Section 9.9.4.) Note that these are SQL functions and are *not* recognized in data input strings.

Caution

While the input strings `now`, `today`, `tomorrow`, and `yesterday` are fine to use in interactive SQL commands, they can have surprising behavior when the command is saved to be executed later, for example in prepared statements, views, and function definitions. The string can be converted to a specific time value that continues to be used long after it becomes stale. Use one of the SQL functions instead in such contexts. For example, `CURRENT_DATE + 1` is safer than `'tomorrow'::date`.

8.5.2. Date/Time Output

The output format of the date/time types can be set to one of the four styles ISO 8601, SQL (Ingres), traditional POSTGRES (Unix date format), or German. The default is the ISO format. (The SQL standard requires the use of the ISO 8601 format. The name of the "SQL" output format is a historical accident.) Table 8-14 shows examples of each output style. The output of the `date` and `time` types is generally only the date or time part in accordance with the given examples. However, the POSTGRES style outputs date-only values in ISO format.

Table 8-14. Date/Time Output Styles

Style Specification	Description	Example
`ISO`	ISO 8601, SQL standard	`1997-12-17 07:37:16-08`
`SQL`	traditional style	`12/17/1997 07:37:16.00 PST`
`Postgres`	original style	`Wed Dec 17 07:37:16 1997 PST`
`German`	regional style	`17.12.1997 07:37:16.00 PST`

Note: ISO 8601 specifies the use of uppercase letter `T` to separate the date and time. PostgreSQL accepts that format on input, but on output it uses a space rather than `T`, as shown above. This is for readability and for consistency with RFC 3339 as well as some other database systems.

In the SQL and POSTGRES styles, day appears before month if DMY field ordering has been specified, otherwise month appears before day. (See Section 8.5.1 for how this setting also affects interpretation of input values.) Table 8-15 shows examples.

Table 8-15. Date Order Conventions

`datestyle` Setting	Input Ordering	Example Output
`SQL, DMY`	*day*/*month*/*year*	`17/12/1997 15:37:16.00 CET`

`datestyle` Setting	Input Ordering	Example Output
`SQL, MDY`	*month*/*day*/*year*	`12/17/1997 07:37:16.00 PST`
`Postgres, DMY`	*day*/*month*/*year*	`Wed 17 Dec 07:37:16 1997 PST`

The date/time style can be selected by the user using the `SET datestyle` command, the DateStyle parameter in the `postgresql.conf` configuration file, or the `PGDATESTYLE` environment variable on the server or client.

The formatting function `to_char` (see Section 9.8) is also available as a more flexible way to format date/time output.

8.5.3. Time Zones

Time zones, and time-zone conventions, are influenced by political decisions, not just earth geometry. Time zones around the world became somewhat standardized during the 1900s, but continue to be prone to arbitrary changes, particularly with respect to daylight-savings rules. PostgreSQL uses the widely-used IANA (Olson) time zone database for information about historical time zone rules. For times in the future, the assumption is that the latest known rules for a given time zone will continue to be observed indefinitely far into the future.

PostgreSQL endeavors to be compatible with the SQL standard definitions for typical usage. However, the SQL standard has an odd mix of date and time types and capabilities. Two obvious problems are:

- Although the `date` type cannot have an associated time zone, the `time` type can. Time zones in the real world have little meaning unless associated with a date as well as a time, since the offset can vary through the year with daylight-saving time boundaries.
- The default time zone is specified as a constant numeric offset from UTC. It is therefore impossible to adapt to daylight-saving time when doing date/time arithmetic across DST boundaries.

To address these difficulties, we recommend using date/time types that contain both date and time when using time zones. We do *not* recommend using the type `time with time zone` (though it is supported by PostgreSQL for legacy applications and for compliance with the SQL standard). PostgreSQL assumes your local time zone for any type containing only date or time.

All timezone-aware dates and times are stored internally in UTC. They are converted to local time in the zone specified by the TimeZone configuration parameter before being displayed to the client.

PostgreSQL allows you to specify time zones in three different forms:

- A full time zone name, for example `America/New_York`. The recognized time zone names are listed in the `pg_timezone_names` view (see Section 49.78). PostgreSQL uses the widely-used IANA time zone data for this purpose, so the same time zone names are also recognized by other software.
- A time zone abbreviation, for example `PST`. Such a specification merely defines a particular offset from UTC, in contrast to full time zone names which can imply a set of daylight savings transition rules as well. The recognized abbreviations are listed in the `pg_timezone_abbrevs` view (see Section 49.77).

You cannot set the configuration parameters TimeZone or log_timezone to a time zone abbreviation, but you can use abbreviations in date/time input values and with the `AT TIME ZONE` operator.

- In addition to the timezone names and abbreviations, PostgreSQL will accept POSIX-style time zone specifications, as described in Section B.5. This option is not normally preferable to using a named time zone, but it may be necessary if no suitable IANA time zone entry is available.

In short, this is the difference between abbreviations and full names: abbreviations represent a specific offset from UTC, whereas many of the full names imply a local daylight-savings time rule, and so have two possible UTC offsets. As an example, `2014-06-04 12:00 America/New_York` represents noon local time in New York, which for this particular date was Eastern Daylight Time (UTC-4). So `2014-06-04 12:00 EDT` specifies that same time instant. But `2014-06-04 12:00 EST` specifies noon Eastern Standard Time (UTC-5), regardless of whether daylight savings was nominally in effect on that date.

To complicate matters, some jurisdictions have used the same timezone abbreviation to mean different UTC offsets at different times; for example, in Moscow `MSK` has meant UTC+3 in some years and UTC+4 in others. PostgreSQL interprets such abbreviations according to whatever they meant (or had most recently meant) on the specified date; but, as with the `EST` example above, this is not necessarily the same as local civil time on that date.

In all cases, timezone names and abbreviations are recognized case-insensitively. (This is a change from PostgreSQL versions prior to 8.2, which were case-sensitive in some contexts but not others.)

Neither timezone names nor abbreviations are hard-wired into the server; they are obtained from configuration files stored under `.../share/timezone/` and `.../share/timezonesets/` of the installation directory (see Section B.4).

The TimeZone configuration parameter can be set in the file `postgresql.conf`, or in any of the other standard ways described in Chapter 18. There are also some special ways to set it:

- The SQL command `SET TIME ZONE` sets the time zone for the session. This is an alternative spelling of `SET TIMEZONE TO` with a more SQL-spec-compatible syntax.
- The `PGTZ` environment variable is used by libpq clients to send a `SET TIME ZONE` command to the server upon connection.

8.5.4. Interval Input

`interval` values can be written using the following verbose syntax:

```
[@] quantity unit [quantity unit...] [direction]
```

where *`quantity`* is a number (possibly signed); *`unit`* is `microsecond`, `millisecond`, `second`, `minute`, `hour`, `day`, `week`, `month`, `year`, `decade`, `century`, `millennium`, or abbreviations or plurals of these units; *`direction`* can be `ago` or empty. The at sign (`@`) is optional noise. The amounts of the different units are implicitly added with appropriate sign accounting. `ago` negates all the fields. This syntax is also used for interval output, if IntervalStyle is set to `postgres_verbose`.

Quantities of days, hours, minutes, and seconds can be specified without explicit unit markings. For example, `'1 12:59:10'` is read the same as `'1 day 12 hours 59 min 10 sec'`. Also, a combination of years and months can be specified with a dash; for example `'200-10'` is read the same as `'200 years`

10 months'. (These shorter forms are in fact the only ones allowed by the SQL standard, and are used for output when IntervalStyle is set to sql_standard.)

Interval values can also be written as ISO 8601 time intervals, using either the "format with designators" of the standard's section 4.4.3.2 or the "alternative format" of section 4.4.3.3. The format with designators looks like this:

```
P quantity unit [ quantity unit ...] [ T [ quantity unit ...]]
```

The string must start with a P, and may include a T that introduces the time-of-day units. The available unit abbreviations are given in Table 8-16. Units may be omitted, and may be specified in any order, but units smaller than a day must appear after T. In particular, the meaning of M depends on whether it is before or after T.

Table 8-16. ISO 8601 Interval Unit Abbreviations

Abbreviation	Meaning
Y	Years
M	Months (in the date part)
W	Weeks
D	Days
H	Hours
M	Minutes (in the time part)
S	Seconds

In the alternative format:

```
P [ years-months-days ] [ T hours:minutes:seconds ]
```

the string must begin with P, and a T separates the date and time parts of the interval. The values are given as numbers similar to ISO 8601 dates.

When writing an interval constant with a *fields* specification, or when assigning a string to an interval column that was defined with a *fields* specification, the interpretation of unmarked quantities depends on the *fields*. For example INTERVAL '1' YEAR is read as 1 year, whereas INTERVAL '1' means 1 second. Also, field values "to the right" of the least significant field allowed by the *fields* specification are silently discarded. For example, writing INTERVAL '1 day 2:03:04' HOUR TO MINUTE results in dropping the seconds field, but not the day field.

According to the SQL standard all fields of an interval value must have the same sign, so a leading negative sign applies to all fields; for example the negative sign in the interval literal '-1 2:03:04' applies to both the days and hour/minute/second parts. PostgreSQL allows the fields to have different signs, and traditionally treats each field in the textual representation as independently signed, so that the hour/minute/second part is considered positive in this example. If IntervalStyle is set to sql_standard then a leading sign is considered to apply to all fields (but only if no additional signs appear). Otherwise the traditional PostgreSQL interpretation is used. To avoid ambiguity, it's recommended to attach an explicit sign to each field if any field is negative.

In the verbose input format, and in some fields of the more compact input formats, field values can have fractional parts; for example '1.5 week' or '01:02:03.45'. Such input is converted to the appropriate

number of months, days, and seconds for storage. When this would result in a fractional number of months or days, the fraction is added to the lower-order fields using the conversion factors 1 month = 30 days and 1 day = 24 hours. For example, `'1.5 month'` becomes 1 month and 15 days. Only seconds will ever be shown as fractional on output.

Table 8-17 shows some examples of valid `interval` input.

Table 8-17. Interval Input

Example	Description
1-2	SQL standard format: 1 year 2 months
3 4:05:06	SQL standard format: 3 days 4 hours 5 minutes 6 seconds
1 year 2 months 3 days 4 hours 5 minutes 6 seconds	Traditional Postgres format: 1 year 2 months 3 days 4 hours 5 minutes 6 seconds
P1Y2M3DT4H5M6S	ISO 8601 “format with designators”: same meaning as above
P0001-02-03T04:05:06	ISO 8601 “alternative format”: same meaning as above

Internally `interval` values are stored as months, days, and seconds. This is done because the number of days in a month varies, and a day can have 23 or 25 hours if a daylight savings time adjustment is involved. The months and days fields are integers while the seconds field can store fractions. Because intervals are usually created from constant strings or `timestamp` subtraction, this storage method works well in most cases, but can cause unexpected results:

```
SELECT EXTRACT(hours from '80 minutes'::interval);
 date_part
-----------
         1

SELECT EXTRACT(days from '80 hours'::interval);
 date_part
-----------
         0
```

Functions `justify_days` and `justify_hours` are available for adjusting days and hours that overflow their normal ranges.

8.5.5. Interval Output

The output format of the interval type can be set to one of the four styles `sql_standard`, `postgres`, `postgres_verbose`, or `iso_8601`, using the command `SET intervalstyle`. The default is the `postgres` format. Table 8-18 shows examples of each output style.

The `sql_standard` style produces output that conforms to the SQL standard’s specification for interval literal strings, if the interval value meets the standard’s restrictions (either year-month only or day-time only, with no mixing of positive and negative components). Otherwise the output looks like a standard

year-month literal string followed by a day-time literal string, with explicit signs added to disambiguate mixed-sign intervals.

The output of the `postgres` style matches the output of PostgreSQL releases prior to 8.4 when the DateStyle parameter was set to `ISO`.

The output of the `postgres_verbose` style matches the output of PostgreSQL releases prior to 8.4 when the `DateStyle` parameter was set to non-`ISO` output.

The output of the `iso_8601` style matches the "format with designators" described in section 4.4.3.2 of the ISO 8601 standard.

Table 8-18. Interval Output Style Examples

Style Specification	Year-Month Interval	Day-Time Interval	Mixed Interval
`sql_standard`	1-2	3 4:05:06	-1-2 +3 -4:05:06
`postgres`	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days -04:05:06
`postgres_verbose`	@ 1 year 2 mons	@ 3 days 4 hours 5 mins 6 secs	@ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago
`iso_8601`	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

8.6. Boolean Type

PostgreSQL provides the standard SQL type `boolean`; see Table 8-19. The `boolean` type can have several states: "true", "false", and a third state, "unknown", which is represented by the SQL null value.

Table 8-19. Boolean Data Type

Name	Storage Size	Description
`boolean`	1 byte	state of true or false

Boolean constants can be represented in SQL queries by the SQL key words `TRUE`, `FALSE`, and `NULL`.

The datatype input function for type `boolean` accepts these string representations for the "true" state:

```
true
yes
on
1
```

and these representations for the "false" state:

```
false
no
```

```
off
0
```

Unique prefixes of these strings are also accepted, for example `t` or `n`. Leading or trailing whitespace is ignored, and case does not matter.

The datatype output function for type `boolean` always emits either `t` or `f`, as shown in Example 8-2.

Example 8-2. Using the `boolean` Type

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
 a |    b
---+---------
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;
 a |    b
---+---------
 t | sic est
```

The key words `TRUE` and `FALSE` are the preferred (SQL-compliant) method for writing Boolean constants in SQL queries. But you can also use the string representations by following the generic string-literal constant syntax described in Section 4.1.2.7, for example `'yes'::boolean`.

Note that the parser automatically understands that `TRUE` and `FALSE` are of type `boolean`, but this is not so for `NULL` because that can have any type. So in some contexts you might have to cast `NULL` to `boolean` explicitly, for example `NULL::boolean`. Conversely, the cast can be omitted from a string-literal Boolean value in contexts where the parser can deduce that the literal must be of type `boolean`.

8.7. Enumerated Types

Enumerated (enum) types are data types that comprise a static, ordered set of values. They are equivalent to the `enum` types supported in a number of programming languages. An example of an enum type might be the days of the week, or a set of status values for a piece of data.

8.7.1. Declaration of Enumerated Types

Enum types are created using the CREATE TYPE command, for example:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Once created, the enum type can be used in table and function definitions much like any other type:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person (
    name text,
```

```
    current_mood mood
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
 name | current_mood
------+--------------
 Moe  | happy
(1 row)
```

8.7.2. Ordering

The ordering of the values in an enum type is the order in which the values were listed when the type was created. All standard comparison operators and related aggregate functions are supported for enums. For example:

```
INSERT INTO person VALUES ('Larry', 'sad');
INSERT INTO person VALUES ('Curly', 'ok');
SELECT * FROM person WHERE current_mood > 'sad';
 name  | current_mood
-------+--------------
 Moe   | happy
 Curly | ok
(2 rows)

SELECT * FROM person WHERE current_mood > 'sad' ORDER BY current_mood;
 name  | current_mood
-------+--------------
 Curly | ok
 Moe   | happy
(2 rows)

SELECT name
FROM person
WHERE current_mood = (SELECT MIN(current_mood) FROM person);
 name
-------
 Larry
(1 row)
```

8.7.3. Type Safety

Each enumerated data type is separate and cannot be compared with other enumerated types. See this example:

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');
CREATE TABLE holidays (
```

```
    num_weeks integer,
    happiness happiness
);
INSERT INTO holidays(num_weeks,happiness) VALUES (4, 'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6, 'very happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8, 'ecstatic');
INSERT INTO holidays(num_weeks,happiness) VALUES (2, 'sad');
ERROR:  invalid input value for enum happiness: "sad"
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood = holidays.happiness;
ERROR:  operator does not exist: mood = happiness
```

If you really need to do something like that, you can either write a custom operator or add explicit casts to your query:

```
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood::text = holidays.happiness::text;
 name | num_weeks
------+-----------
 Moe  |         4
(1 row)
```

8.7.4. Implementation Details

Enum labels are case sensitive, so `'happy'` is not the same as `'HAPPY'`. White space in the labels is significant too.

Although enum types are primarily intended for static sets of values, there is support for adding new values to an existing enum type, and for renaming values (see ALTER TYPE). Existing values cannot be removed from an enum type, nor can the sort ordering of such values be changed, short of dropping and re-creating the enum type.

An enum value occupies four bytes on disk. The length of an enum value's textual label is limited by the `NAMEDATALEN` setting compiled into PostgreSQL; in standard builds this means at most 63 bytes.

The translations from internal enum values to textual labels are kept in the system catalog `pg_enum`. Querying this catalog directly can be useful.

8.8. Geometric Types

Geometric data types represent two-dimensional spatial objects. Table 8-20 shows the geometric types available in PostgreSQL.

Table 8-20. Geometric Types

Name	Storage Size	Description	Representation
`point`	16 bytes	Point on a plane	(x,y)
`line`	32 bytes	Infinite line	{A,B,C}
`lseg`	32 bytes	Finite line segment	((x1,y1),(x2,y2))
`box`	32 bytes	Rectangular box	((x1,y1),(x2,y2))
`path`	16+16n bytes	Closed path (similar to polygon)	((x1,y1),...)
`path`	16+16n bytes	Open path	[(x1,y1),...]
`polygon`	40+16n bytes	Polygon (similar to closed path)	((x1,y1),...)
`circle`	24 bytes	Circle	<(x,y),r> (center point and radius)

A rich set of functions and operators is available to perform various geometric operations such as scaling, translation, rotation, and determining intersections. They are explained in Section 9.11.

8.8.1. Points

Points are the fundamental two-dimensional building block for geometric types. Values of type `point` are specified using either of the following syntaxes:

```
( x , y )
  x , y
```

where *x* and *y* are the respective coordinates, as floating-point numbers.

Points are output using the first syntax.

8.8.2. Lines

Lines are represented by the linear equation *A*x + *B*y + *C* = 0, where *A* and *B* are not both zero. Values of type `line` are input and output in the following form:

```
{ A, B, C }
```

Alternatively, any of the following forms can be used for input:

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where `(x1,y1)` and `(x2,y2)` are two different points on the line.

8.8.3. Line Segments

Line segments are represented by pairs of points that are the endpoints of the segment. Values of type `lseg` are specified using any of the following syntaxes:

```
[ ( x1 , y1 ) , ( x2 , y2 ) ]
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where `(x1,y1)` and `(x2,y2)` are the end points of the line segment.

Line segments are output using the first syntax.

8.8.4. Boxes

Boxes are represented by pairs of points that are opposite corners of the box. Values of type `box` are specified using any of the following syntaxes:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where `(x1,y1)` and `(x2,y2)` are any two opposite corners of the box.

Boxes are output using the second syntax.

Any two opposite corners can be supplied on input, but the values will be reordered as needed to store the upper right and lower left corners, in that order.

8.8.5. Paths

Paths are represented by lists of connected points. Paths can be *open*, where the first and last points in the list are considered not connected, or *closed*, where the first and last points are considered connected.

Values of type `path` are specified using any of the following syntaxes:

```
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

where the points are the end points of the line segments comprising the path. Square brackets (`[]`) indicate an open path, while parentheses (`()`) indicate a closed path. When the outermost parentheses are omitted, as in the third through fifth syntaxes, a closed path is assumed.

Paths are output using the first or second syntax, as appropriate.

8.8.6. Polygons

Polygons are represented by lists of points (the vertexes of the polygon). Polygons are very similar to closed paths, but are stored differently and have their own set of support routines.

Values of type `polygon` are specified using any of the following syntaxes:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

where the points are the end points of the line segments comprising the boundary of the polygon.

Polygons are output using the first syntax.

8.8.7. Circles

Circles are represented by a center point and radius. Values of type `circle` are specified using any of the following syntaxes:

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

where `(x,y)` is the center point and `r` is the radius of the circle.

Circles are output using the first syntax.

8.9. Network Address Types

PostgreSQL offers data types to store IPv4, IPv6, and MAC addresses, as shown in Table 8-21. It is better to use these types instead of plain text types to store network addresses, because these types offer input error checking and specialized operators and functions (see Section 9.12).

Table 8-21. Network Address Types

Name	Storage Size	Description
`cidr`	7 or 19 bytes	IPv4 and IPv6 networks
`inet`	7 or 19 bytes	IPv4 and IPv6 hosts and networks
`macaddr`	6 bytes	MAC addresses

When sorting `inet` or `cidr` data types, IPv4 addresses will always sort before IPv6 addresses, including IPv4 addresses encapsulated or mapped to IPv6 addresses, such as ::10.2.3.4 or ::ffff:10.4.3.2.

8.9.1. inet

The inet type holds an IPv4 or IPv6 host address, and optionally its subnet, all in one field. The subnet is represented by the number of network address bits present in the host address (the "netmask"). If the netmask is 32 and the address is IPv4, then the value does not indicate a subnet, only a single host. In IPv6, the address length is 128 bits, so 128 bits specify a unique host address. Note that if you want to accept only networks, you should use the cidr type rather than inet.

The input format for this type is *address/y* where *address* is an IPv4 or IPv6 address and *y* is the number of bits in the netmask. If the */y* portion is missing, the netmask is 32 for IPv4 and 128 for IPv6, so the value represents just a single host. On display, the */y* portion is suppressed if the netmask specifies a single host.

8.9.2. cidr

The cidr type holds an IPv4 or IPv6 network specification. Input and output formats follow Classless Internet Domain Routing conventions. The format for specifying networks is *address/y* where *address* is the network represented as an IPv4 or IPv6 address, and *y* is the number of bits in the netmask. If *y* is omitted, it is calculated using assumptions from the older classful network numbering system, except it will be at least large enough to include all of the octets written in the input. It is an error to specify a network address that has bits set to the right of the specified netmask.

Table 8-22 shows some examples.

Table 8-22. cidr Type Input Examples

cidr Input	cidr Output	abbrev(cidr)
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba::/64
2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

8.9.3. inet vs. cidr

The essential difference between inet and cidr data types is that inet accepts values with nonzero bits to the right of the netmask, whereas cidr does not.

> **Tip:** If you do not like the output format for inet or cidr values, try the functions host, text, and abbrev.

8.9.4. macaddr

The macaddr type stores MAC addresses, known for example from Ethernet card hardware addresses (although MAC addresses are used for other purposes as well). Input is accepted in the following formats:

```
'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'0800-2b01-0203'
'08002b010203'
```

These examples would all specify the same address. Upper and lower case is accepted for the digits a through f. Output is always in the first of the forms shown.

IEEE Std 802-2001 specifies the second shown form (with hyphens) as the canonical form for MAC addresses, and specifies the first form (with colons) as the bit-reversed notation, so that 08-00-2b-01-02-03 = 01:00:4D:08:04:0C. This convention is widely ignored nowadays, and it is relevant only for obsolete network protocols (such as Token Ring). PostgreSQL makes no provisions for bit reversal, and all accepted formats use the canonical LSB order.

The remaining five input formats are not part of any standard.

8.10. Bit String Types

Bit strings are strings of 1's and 0's. They can be used to store or visualize bit masks. There are two SQL bit types: bit(*n*) and bit varying(*n*), where *n* is a positive integer.

bit type data must match the length *n* exactly; it is an error to attempt to store shorter or longer bit strings. bit varying data is of variable length up to the maximum length *n*; longer strings will be rejected. Writing bit without a length is equivalent to bit(1), while bit varying without a length specification means unlimited length.

> **Note:** If one explicitly casts a bit-string value to bit(*n*), it will be truncated or zero-padded on the right to be exactly *n* bits, without raising an error. Similarly, if one explicitly casts a bit-string value to bit varying(*n*), it will be truncated on the right if it is more than *n* bits.

Refer to Section 4.1.2.5 for information about the syntax of bit string constants. Bit-logical operators and string manipulation functions are available; see Section 9.6.

Example 8-3. Using the Bit String Types

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');
ERROR:  bit string length 2 does not match type bit(3)
INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;
  a  |  b
-----+-----
 101 | 00
 100 | 101
```

A bit string value requires 1 byte for each group of 8 bits, plus 5 or 8 bytes overhead depending on the length of the string (but long values may be compressed or moved out-of-line, as explained in Section 8.3 for character strings).

8.11. Text Search Types

PostgreSQL provides two data types that are designed to support full text search, which is the activity of searching through a collection of natural-language *documents* to locate those that best match a *query*. The `tsvector` type represents a document in a form optimized for text search; the `tsquery` type similarly represents a text query. Chapter 12 provides a detailed explanation of this facility, and Section 9.13 summarizes the related functions and operators.

8.11.1. `tsvector`

A `tsvector` value is a sorted list of distinct *lexemes*, which are words that have been *normalized* to merge different variants of the same word (see Chapter 12 for details). Sorting and duplicate-elimination are done automatically during input, as shown in this example:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
                      tsvector
----------------------------------------------------
 'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

To represent lexemes containing whitespace or punctuation, surround them with quotes:

```
SELECT $$the lexeme '    ' contains spaces$$::tsvector;
                 tsvector
-------------------------------------------
 '    ' 'contains' 'lexeme' 'spaces' 'the'
```

(We use dollar-quoted string literals in this example and the next one to avoid the confusion of having to double quote marks within the literals.) Embedded quotes and backslashes must be doubled:

```
SELECT $$the lexeme 'Joe"s' contains a quote$$::tsvector;
                    tsvector
-----------------------------------------------
 'Joe"s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

Optionally, integer *positions* can be attached to lexemes:

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11 rat:12'::tsvector;
                                    tsvector
-------------------------------------------------------------------------------
 'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12 'sat':4
```

A position normally indicates the source word's location in the document. Positional information can be used for *proximity ranking*. Position values can range from 1 to 16383; larger numbers are silently set to 16383. Duplicate positions for the same lexeme are discarded.

Lexemes that have positions can further be labeled with a *weight*, which can be `A`, `B`, `C`, or `D`. `D` is the default and hence is not shown on output:

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
          tsvector
----------------------------
 'a':1A 'cat':5 'fat':2B,4C
```

Weights are typically used to reflect document structure, for example by marking title words differently from body words. Text search ranking functions can assign different priorities to the different weight markers.

It is important to understand that the `tsvector` type itself does not perform any normalization; it assumes the words it is given are normalized appropriately for the application. For example,

```
select 'The Fat Rats'::tsvector;
      tsvector
--------------------
 'Fat' 'Rats' 'The'
```

For most English-text-searching applications the above words would be considered non-normalized, but `tsvector` doesn't care. Raw document text should usually be passed through `to_tsvector` to normalize the words appropriately for searching:

```
SELECT to_tsvector('english', 'The Fat Rats');
   to_tsvector
-----------------
 'fat':2 'rat':3
```

Again, see Chapter 12 for more detail.

8.11.2. tsquery

A tsquery value stores lexemes that are to be searched for, and combines them honoring the Boolean operators & (AND), | (OR), and ! (NOT). Parentheses can be used to enforce grouping of the operators:

```
SELECT 'fat & rat'::tsquery;
    tsquery
---------------
 'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;
          tsquery
---------------------------
 'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;
        tsquery
------------------------
 'fat' & 'rat' & !'cat'
```

In the absence of parentheses, ! (NOT) binds most tightly, and & (AND) binds more tightly than | (OR).

Optionally, lexemes in a tsquery can be labeled with one or more weight letters, which restricts them to match only tsvector lexemes with matching weights:

```
SELECT 'fat:ab & cat'::tsquery;
    tsquery
------------------
 'fat':AB & 'cat'
```

Also, lexemes in a tsquery can be labeled with * to specify prefix matching:

```
SELECT 'super:*'::tsquery;
  tsquery
-----------
 'super':*
```

This query will match any word in a tsvector that begins with "super". Note that prefixes are first processed by text search configurations, which means this comparison returns true:

```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'postgres:*' );
 ?column?
----------
 t
(1 row)
```

because postgres gets stemmed to postgr:

```
SELECT to_tsquery('postgres:*');
 to_tsquery
------------
 'postgr':*
(1 row)
```

which then matches `postgraduate`.

Quoting rules for lexemes are the same as described previously for lexemes in `tsvector`; and, as with `tsvector`, any required normalization of words must be done before converting to the `tsquery` type. The `to_tsquery` function is convenient for performing such normalization:

```
SELECT to_tsquery('Fat:ab & Cats');
    to_tsquery
------------------
 'fat':AB & 'cat'
```

8.12. UUID Type

The data type `uuid` stores Universally Unique Identifiers (UUID) as defined by RFC 4122, ISO/IEC 9834-8:2005, and related standards. (Some systems refer to this data type as a globally unique identifier, or GUID, instead.) This identifier is a 128-bit quantity that is generated by an algorithm chosen to make it very unlikely that the same identifier will be generated by anyone else in the known universe using the same algorithm. Therefore, for distributed systems, these identifiers provide a better uniqueness guarantee than sequence generators, which are only unique within a single database.

A UUID is written as a sequence of lower-case hexadecimal digits, in several groups separated by hyphens, specifically a group of 8 digits followed by three groups of 4 digits followed by a group of 12 digits, for a total of 32 digits representing the 128 bits. An example of a UUID in this standard form is:

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

PostgreSQL also accepts the following alternative forms for input: use of upper-case digits, the standard format surrounded by braces, omitting some or all hyphens, adding a hyphen after any group of four digits. Examples are:

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

Output is always in the standard form.

PostgreSQL provides storage and comparison functions for UUIDs, but the core database does not include any function for generating UUIDs, because no single algorithm is well suited for every application. The uuid-ossp module provides functions that implement several standard algorithms. The pgcrypto module also provides a generation function for random UUIDs. Alternatively, UUIDs could be generated by client applications or other libraries invoked through a server-side function.

8.13. XML Type

The xml data type can be used to store XML data. Its advantage over storing XML data in a text field is that it checks the input values for well-formedness, and there are support functions to perform type-safe operations on it; see Section 9.14. Use of this data type requires the installation to have been built with configure --with-libxml.

The xml type can store well-formed "documents", as defined by the XML standard, as well as "content" fragments, which are defined by reference to the more permissive "document node"[1] of the XQuery and XPath data model. Roughly, this means that content fragments can have more than one top-level element or character node. The expression *xmlvalue* IS DOCUMENT can be used to evaluate whether a particular xml value is a full document or only a content fragment.

8.13.1. Creating XML Values

To produce a value of type xml from character data, use the function xmlparse:

```
XMLPARSE ( { DOCUMENT | CONTENT } value)
```

Examples:

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title><chapter>...</chapter><
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

While this is the only way to convert character strings into XML values according to the SQL standard, the PostgreSQL-specific syntaxes:

```
xml '<foo>bar</foo>'
'<foo>bar</foo>'::xml
```

can also be used.

The xml type does not validate input values against a document type declaration (DTD), even when the input value specifies a DTD. There is also currently no built-in support for validating against other XML schema languages such as XML Schema.

The inverse operation, producing a character string value from xml, uses the function xmlserialize:

```
XMLSERIALIZE ( { DOCUMENT | CONTENT } value AS type )
```

type can be character, character varying, or text (or an alias for one of those). Again, according to the SQL standard, this is the only way to convert between type xml and character types, but PostgreSQL also allows you to simply cast the value.

When a character string value is cast to or from type xml without going through XMLPARSE or XMLSERIALIZE, respectively, the choice of DOCUMENT versus CONTENT is determined by the "XML option" session configuration parameter, which can be set using the standard command:

```
SET XML OPTION { DOCUMENT | CONTENT };
```

or the more PostgreSQL-like syntax

1. https://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/#DocumentNode

```
SET xmloption TO { DOCUMENT | CONTENT };
```

The default is `CONTENT`, so all forms of XML data are allowed.

8.13.2. Encoding Handling

Care must be taken when dealing with multiple character encodings on the client, server, and in the XML data passed through them. When using the text mode to pass queries to the server and query results to the client (which is the normal mode), PostgreSQL converts all character data passed between the client and the server and vice versa to the character encoding of the respective end; see Section 22.3. This includes string representations of XML values, such as in the above examples. This would ordinarily mean that encoding declarations contained in XML data can become invalid as the character data is converted to other encodings while traveling between client and server, because the embedded encoding declaration is not changed. To cope with this behavior, encoding declarations contained in character strings presented for input to the `xml` type are *ignored*, and content is assumed to be in the current server encoding. Consequently, for correct processing, character strings of XML data must be sent from the client in the current client encoding. It is the responsibility of the client to either convert documents to the current client encoding before sending them to the server, or to adjust the client encoding appropriately. On output, values of type `xml` will not have an encoding declaration, and clients should assume all data is in the current client encoding.

When using binary mode to pass query parameters to the server and query results back to the client, no character set conversion is performed, so the situation is different. In this case, an encoding declaration in the XML data will be observed, and if it is absent, the data will be assumed to be in UTF-8 (as required by the XML standard; note that PostgreSQL does not support UTF-16). On output, data will have an encoding declaration specifying the client encoding, unless the client encoding is UTF-8, in which case it will be omitted.

Needless to say, processing XML data with PostgreSQL will be less error-prone and more efficient if the XML data encoding, client encoding, and server encoding are the same. Since XML data is internally processed in UTF-8, computations will be most efficient if the server encoding is also UTF-8.

> **Caution**
>
> Some XML-related functions may not work at all on non-ASCII data when the server encoding is not UTF-8. This is known to be an issue for `xpath()` in particular.

8.13.3. Accessing XML Values

The `xml` data type is unusual in that it does not provide any comparison operators. This is because there is no well-defined and universally useful comparison algorithm for XML data. One consequence of this is that you cannot retrieve rows by comparing an `xml` column against a search value. XML values should therefore typically be accompanied by a separate key field such as an ID. An alternative solution for comparing XML values is to convert them to character strings first, but note that character string comparison has little to do with a useful XML comparison method.

Since there are no comparison operators for the `xml` data type, it is not possible to create an index directly on a column of this type. If speedy searches in XML data are desired, possible workarounds include casting the expression to a character string type and indexing that, or indexing an XPath expression. Of course, the actual query would have to be adjusted to search by the indexed expression.

The text-search functionality in PostgreSQL can also be used to speed up full-document searches of XML data. The necessary preprocessing support is, however, not yet available in the PostgreSQL distribution.

8.14. JSON Types

JSON data types are for storing JSON (JavaScript Object Notation) data, as specified in RFC 7159[2]. Such data can also be stored as `text`, but the JSON data types have the advantage of enforcing that each stored value is valid according to the JSON rules. There are also assorted JSON-specific functions and operators available for data stored in these data types; see Section 9.15.

There are two JSON data types: `json` and `jsonb`. They accept *almost* identical sets of values as input. The major practical difference is one of efficiency. The `json` data type stores an exact copy of the input text, which processing functions must reparse on each execution; while `jsonb` data is stored in a decomposed binary format that makes it slightly slower to input due to added conversion overhead, but significantly faster to process, since no reparsing is needed. `jsonb` also supports indexing, which can be a significant advantage.

Because the `json` type stores an exact copy of the input text, it will preserve semantically-insignificant white space between tokens, as well as the order of keys within JSON objects. Also, if a JSON object within the value contains the same key more than once, all the key/value pairs are kept. (The processing functions consider the last value as the operative one.) By contrast, `jsonb` does not preserve white space, does not preserve the order of object keys, and does not keep duplicate object keys. If duplicate keys are specified in the input, only the last value is kept.

In general, most applications should prefer to store JSON data as `jsonb`, unless there are quite specialized needs, such as legacy assumptions about ordering of object keys.

PostgreSQL allows only one character set encoding per database. It is therefore not possible for the JSON types to conform rigidly to the JSON specification unless the database encoding is UTF8. Attempts to directly include characters that cannot be represented in the database encoding will fail; conversely, characters that can be represented in the database encoding but not in UTF8 will be allowed.

RFC 7159 permits JSON strings to contain Unicode escape sequences denoted by `\u`*`XXXX`*. In the input function for the `json` type, Unicode escapes are allowed regardless of the database encoding, and are checked only for syntactic correctness (that is, that four hex digits follow `\u`). However, the input function for `jsonb` is stricter: it disallows Unicode escapes for non-ASCII characters (those above `U+007F`) unless the database encoding is UTF8. The `jsonb` type also rejects `\u0000` (because that cannot be represented in PostgreSQL's `text` type), and it insists that any use of Unicode surrogate pairs to designate characters outside the Unicode Basic Multilingual Plane be correct. Valid Unicode escapes are converted to the equivalent ASCII or UTF8 character for storage; this includes folding surrogate pairs into a single character.

2. https://tools.ietf.org/html/rfc7159

Note: Many of the JSON processing functions described in Section 9.15 will convert Unicode escapes to regular characters, and will therefore throw the same types of errors just described even if their input is of type `json` not `jsonb`. The fact that the `json` input function does not make these checks may be considered a historical artifact, although it does allow for simple storage (without processing) of JSON Unicode escapes in a non-UTF8 database encoding. In general, it is best to avoid mixing Unicode escapes in JSON with a non-UTF8 database encoding, if possible.

When converting textual JSON input into `jsonb`, the primitive types described by RFC 7159 are effectively mapped onto native PostgreSQL types, as shown in Table 8-23. Therefore, there are some minor additional constraints on what constitutes valid `jsonb` data that do not apply to the `json` type, nor to JSON in the abstract, corresponding to limits on what can be represented by the underlying data type. Notably, `jsonb` will reject numbers that are outside the range of the PostgreSQL `numeric` data type, while `json` will not. Such implementation-defined restrictions are permitted by RFC 7159. However, in practice such problems are far more likely to occur in other implementations, as it is common to represent JSON's `number` primitive type as IEEE 754 double precision floating point (which RFC 7159 explicitly anticipates and allows for). When using JSON as an interchange format with such systems, the danger of losing numeric precision compared to data originally stored by PostgreSQL should be considered.

Conversely, as noted in the table there are some minor restrictions on the input format of JSON primitive types that do not apply to the corresponding PostgreSQL types.

Table 8-23. JSON primitive types and corresponding PostgreSQL types

JSON primitive type	PostgreSQL type	Notes
`string`	`text`	`\u0000` is disallowed, as are non-ASCII Unicode escapes if database encoding is not UTF8
`number`	`numeric`	`NaN` and `infinity` values are disallowed
`boolean`	`boolean`	Only lowercase `true` and `false` spellings are accepted
`null`	(none)	SQL `NULL` is a different concept

8.14.1. JSON Input and Output Syntax

The input/output syntax for the JSON data types is as specified in RFC 7159.

The following are all valid `json` (or `jsonb`) expressions:

```
-- Simple scalar/primitive value
-- Primitive values can be numbers, quoted strings, true, false, or null
SELECT '5'::json;

-- Array of zero or more elements (elements need not be of same type)
SELECT '[1, 2, "foo", null]'::json;

-- Object containing pairs of keys and values
-- Note that object keys must always be quoted strings
```

```
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;

-- Arrays and objects can be nested arbitrarily
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

As previously stated, when a JSON value is input and then printed without any additional processing, `json` outputs the same text that was input, while `jsonb` does not preserve semantically-insignificant details such as whitespace. For example, note the differences here:

```
SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::json;
                       json
-------------------------------------------------
 {"bar": "baz", "balance": 7.77, "active":false}
(1 row)

SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::jsonb;
                       jsonb
--------------------------------------------------
 {"bar": "baz", "active": false, "balance": 7.77}
(1 row)
```

One semantically-insignificant detail worth noting is that in `jsonb`, numbers will be printed according to the behavior of the underlying `numeric` type. In practice this means that numbers entered with `E` notation will be printed without it, for example:

```
SELECT '{"reading": 1.230e-5}'::json, '{"reading": 1.230e-5}'::jsonb;
         json          |          jsonb
-----------------------+-----------------------
 {"reading": 1.230e-5} | {"reading": 0.00001230}
(1 row)
```

However, `jsonb` will preserve trailing fractional zeroes, as seen in this example, even though those are semantically insignificant for purposes such as equality checks.

8.14.2. Designing JSON documents effectively

Representing data as JSON can be considerably more flexible than the traditional relational data model, which is compelling in environments where requirements are fluid. It is quite possible for both approaches to co-exist and complement each other within the same application. However, even for applications where maximal flexibility is desired, it is still recommended that JSON documents have a somewhat fixed structure. The structure is typically unenforced (though enforcing some business rules declaratively is possible), but having a predictable structure makes it easier to write queries that usefully summarize a set of "documents" (datums) in a table.

JSON data is subject to the same concurrency-control considerations as any other data type when stored in a table. Although storing large documents is practicable, keep in mind that any update acquires a row-level lock on the whole row. Consider limiting JSON documents to a manageable size in order to decrease lock contention among updating transactions. Ideally, JSON documents should each represent an atomic

datum that business rules dictate cannot reasonably be further subdivided into smaller datums that could be modified independently.

8.14.3. `jsonb` Containment and Existence

Testing *containment* is an important capability of `jsonb`. There is no parallel set of facilities for the `json` type. Containment tests whether one `jsonb` document has contained within it another one. These examples return true except as noted:

```
-- Simple scalar/primitive values contain only the identical value:
SELECT '"foo"'::jsonb @> '"foo"'::jsonb;

-- The array on the right side is contained within the one on the left:
SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;

-- Order of array elements is not significant, so this is also true:
SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;

-- Duplicate array elements don't matter either:
SELECT '[1, 2, 3]'::jsonb @> '[1, 2, 2]'::jsonb;

-- The object with a single pair on the right side is contained
-- within the object on the left side:
SELECT '{"product": "PostgreSQL", "version": 9.4, "jsonb": true}'::jsonb @> '{"version": 9.

-- The array on the right side is not considered contained within the
-- array on the left, even though a similar array is nested within it:
SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb;  -- yields false

-- But with a layer of nesting, it is contained:
SELECT '[1, 2, [1, 3]]'::jsonb @> '[[1, 3]]'::jsonb;

-- Similarly, containment is not reported here:
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb;  -- yields false

-- A top-level key and an empty object is contained:
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"foo": {}}'::jsonb;
```

The general principle is that the contained object must match the containing object as to structure and data contents, possibly after discarding some non-matching array elements or object key/value pairs from the containing object. But remember that the order of array elements is not significant when doing a containment match, and duplicate array elements are effectively considered only once.

As a special exception to the general principle that the structures must match, an array may contain a primitive value:

```
-- This array contains the primitive string value:
SELECT '["foo", "bar"]'::jsonb @> '"bar"'::jsonb;

-- This exception is not reciprocal -- non-containment is reported here:
SELECT '"bar"'::jsonb @> '["bar"]'::jsonb;  -- yields false
```

jsonb also has an *existence* operator, which is a variation on the theme of containment: it tests whether a string (given as a text value) appears as an object key or array element at the top level of the jsonb value. These examples return true except as noted:

```
-- String exists as array element:
SELECT '["foo", "bar", "baz"]'::jsonb ? 'bar';

-- String exists as object key:
SELECT '{"foo": "bar"}'::jsonb ? 'foo';

-- Object values are not considered:
SELECT '{"foo": "bar"}'::jsonb ? 'bar';  -- yields false

-- As with containment, existence must match at the top level:
SELECT '{"foo": {"bar": "baz"}}'::jsonb ? 'bar'; -- yields false

-- A string is considered to exist if it matches a primitive JSON string:
SELECT '"foo"'::jsonb ? 'foo';
```

JSON objects are better suited than arrays for testing containment or existence when there are many keys or elements involved, because unlike arrays they are internally optimized for searching, and do not need to be searched linearly.

> **Tip:** Because JSON containment is nested, an appropriate query can skip explicit selection of sub-objects. As an example, suppose that we have a doc column containing objects at the top level, with most objects containing tags fields that contain arrays of sub-objects. This query finds entries in which sub-objects containing both "term":"paris" and "term":"food" appear, while ignoring any such keys outside the tags array:
>
> ```
> SELECT doc->'site_name' FROM websites
>   WHERE doc @> '{"tags":[{"term":"paris"}, {"term":"food"}]}';
> ```
>
> One could accomplish the same thing with, say,
>
> ```
> SELECT doc->'site_name' FROM websites
>   WHERE doc->'tags' @> '[{"term":"paris"}, {"term":"food"}]';
> ```
>
> but that approach is less flexible, and often less efficient as well.
>
> On the other hand, the JSON existence operator is not nested: it will only look for the specified key or array element at top level of the JSON value.

The various containment and existence operators, along with all other JSON operators and functions are documented in Section 9.15.

8.14.4. jsonb Indexing

GIN indexes can be used to efficiently search for keys or key/value pairs occurring within a large number of jsonb documents (datums). Two GIN "operator classes" are provided, offering different performance and flexibility trade-offs.

The default GIN operator class for `jsonb` supports queries with top-level key-exists operators `?`, `?&` and `?|` operators and path/value-exists operator `@>`. (For details of the semantics that these operators implement, see Table 9-41.) An example of creating an index with this operator class is:

```
CREATE INDEX idxgin ON api USING GIN (jdoc);
```

The non-default GIN operator class `jsonb_path_ops` supports indexing the `@>` operator only. An example of creating an index with this operator class is:

```
CREATE INDEX idxginp ON api USING GIN (jdoc jsonb_path_ops);
```

Consider the example of a table that stores JSON documents retrieved from a third-party web service, with a documented schema definition. A typical document is:

```
{
    "guid": "9c36adc1-7fb5-4d5b-83b4-90356a46061a",
    "name": "Angela Barton",
    "is_active": true,
    "company": "Magnafone",
    "address": "178 Howard Place, Gulf, Washington, 702",
    "registered": "2009-11-07T08:53:22 +08:00",
    "latitude": 19.793713,
    "longitude": 86.513373,
    "tags": [
        "enim",
        "aliquip",
        "qui"
    ]
}
```

We store these documents in a table named `api`, in a `jsonb` column named `jdoc`. If a GIN index is created on this column, queries like the following can make use of the index:

```
-- Find documents in which the key "company" has value "Magnafone"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"company": "Magnafone"}';
```

However, the index could not be used for queries like the following, because though the operator `?` is indexable, it is not applied directly to the indexed column `jdoc`:

```
-- Find documents in which the key "tags" contains key or array element "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc -> 'tags' ? 'qui';
```

Still, with appropriate use of expression indexes, the above query can use an index. If querying for particular items within the `"tags"` key is common, defining an index like this may be worthwhile:

```
CREATE INDEX idxgintags ON api USING GIN ((jdoc -> 'tags'));
```

Now, the `WHERE` clause `jdoc -> 'tags' ? 'qui'` will be recognized as an application of the indexable operator `?` to the indexed expression `jdoc -> 'tags'`. (More information on expression indexes can be found in Section 11.7.)

Another approach to querying is to exploit containment, for example:

```
-- Find documents in which the key "tags" contains array element "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"tags": ["qui"]}';
```

A simple GIN index on the `jdoc` column can support this query. But note that such an index will store copies of every key and value in the `jdoc` column, whereas the expression index of the previous example stores only data found under the `tags` key. While the simple-index approach is far more flexible (since it supports queries about any key), targeted expression indexes are likely to be smaller and faster to search than a simple index.

Although the `jsonb_path_ops` operator class supports only queries with the `@>` operator, it has notable performance advantages over the default operator class `jsonb_ops`. A `jsonb_path_ops` index is usually much smaller than a `jsonb_ops` index over the same data, and the specificity of searches is better, particularly when queries contain keys that appear frequently in the data. Therefore search operations typically perform better than with the default operator class.

The technical difference between a `jsonb_ops` and a `jsonb_path_ops` GIN index is that the former creates independent index items for each key and value in the data, while the latter creates index items only for each value in the data. [3] Basically, each `jsonb_path_ops` index item is a hash of the value and the key(s) leading to it; for example to index `{"foo": {"bar": "baz"}}`, a single index item would be created incorporating all three of `foo`, `bar`, and `baz` into the hash value. Thus a containment query looking for this structure would result in an extremely specific index search; but there is no way at all to find out whether `foo` appears as a key. On the other hand, a `jsonb_ops` index would create three index items representing `foo`, `bar`, and `baz` separately; then to do the containment query, it would look for rows containing all three of these items. While GIN indexes can perform such an AND search fairly efficiently, it will still be less specific and slower than the equivalent `jsonb_path_ops` search, especially if there are a very large number of rows containing any single one of the three index items.

A disadvantage of the `jsonb_path_ops` approach is that it produces no index entries for JSON structures not containing any values, such as `{"a": {}}`. If a search for documents containing such a structure is requested, it will require a full-index scan, which is quite slow. `jsonb_path_ops` is therefore ill-suited for applications that often perform such searches.

`jsonb` also supports `btree` and `hash` indexes. These are usually useful only if it's important to check equality of complete JSON documents. The `btree` ordering for `jsonb` datums is seldom of great interest, but for completeness it is:

```
Object > Array > Boolean > Number > String > Null

Object with n pairs > object with n - 1 pairs

Array with n elements > array with n - 1 elements
```

Objects with equal numbers of pairs are compared in the order:

```
key-1, value-1, key-2 ...
```

Note that object keys are compared in their storage order; in particular, since shorter keys are stored before longer keys, this can lead to results that might be unintuitive, such as:

```
{ "aa": 1, "c": 1} > {"b": 1, "d": 1}
```

3. For this purpose, the term "value" includes array elements, though JSON terminology sometimes considers array elements distinct from values within objects.

Similarly, arrays with equal numbers of elements are compared in the order:

```
element-1, element-2 ...
```

Primitive JSON values are compared using the same comparison rules as for the underlying PostgreSQL data type. Strings are compared using the default database collation.

8.15. Arrays

PostgreSQL allows columns of a table to be defined as variable-length multidimensional arrays. Arrays of any built-in or user-defined base type, enum type, or composite type can be created. Arrays of domains are not yet supported.

8.15.1. Declaration of Array Types

To illustrate the use of array types, we create this table:

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

As shown, an array data type is named by appending square brackets (`[]`) to the data type name of the array elements. The above command will create a table named `sal_emp` with a column of type `text` (`name`), a one-dimensional array of type `integer` (`pay_by_quarter`), which represents the employee's salary by quarter, and a two-dimensional array of `text` (`schedule`), which represents the employee's weekly schedule.

The syntax for `CREATE TABLE` allows the exact size of arrays to be specified, for example:

```
CREATE TABLE tictactoe (
    squares   integer[3][3]
);
```

However, the current implementation ignores any supplied array size limits, i.e., the behavior is the same as for arrays of unspecified length.

The current implementation does not enforce the declared number of dimensions either. Arrays of a particular element type are all considered to be of the same type, regardless of size or number of dimensions. So, declaring the array size or number of dimensions in `CREATE TABLE` is simply documentation; it does not affect run-time behavior.

An alternative syntax, which conforms to the SQL standard by using the keyword `ARRAY`, can be used for one-dimensional arrays. `pay_by_quarter` could have been defined as:

```
    pay_by_quarter  integer ARRAY[4],
```

Or, if no array size is to be specified:

```
    pay_by_quarter  integer ARRAY,
```

As before, however, PostgreSQL does not enforce the size restriction in any case.

8.15.2. Array Value Input

To write an array value as a literal constant, enclose the element values within curly braces and separate them by commas. (If you know C, this is not unlike the C syntax for initializing structures.) You can put double quotes around any element value, and must do so if it contains commas or curly braces. (More details appear below.) Thus, the general format of an array constant is the following:

```
'{ val1 delim val2 delim ... }'
```

where *delim* is the delimiter character for the type, as recorded in its `pg_type` entry. Among the standard data types provided in the PostgreSQL distribution, all use a comma (`,`), except for type `box` which uses a semicolon (`;`). Each *val* is either a constant of the array element type, or a subarray. An example of an array constant is:

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

This constant is a two-dimensional, 3-by-3 array consisting of three subarrays of integers.

To set an element of an array constant to NULL, write `NULL` for the element value. (Any upper- or lower-case variant of `NULL` will do.) If you want an actual string value "NULL", you must put double quotes around it.

(These kinds of array constants are actually only a special case of the generic type constants discussed in Section 4.1.2.7. The constant is initially treated as a string and passed to the array input conversion routine. An explicit type specification might be necessary.)

Now we can show some `INSERT` statements:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
```

The result of the previous two inserts looks like this:

```
SELECT * FROM sal_emp;
 name  |      pay_by_quarter       |                  schedule
-------+---------------------------+-------------------------------------------
 Bill  | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
 Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

Multidimensional arrays must have matching extents for each dimension. A mismatch causes an error, for example:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"meeting"}}');
ERROR:  multidimensional arrays must have array expressions with matching dimensions
```

The `ARRAY` constructor syntax can also be used:

```
INSERT INTO sal_emp
    VALUES ('Bill',
    ARRAY[10000, 10000, 10000, 10000],
    ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);

INSERT INTO sal_emp
    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);
```

Notice that the array elements are ordinary SQL constants or expressions; for instance, string literals are single quoted, instead of double quoted as they would be in an array literal. The `ARRAY` constructor syntax is discussed in more detail in Section 4.2.12.

8.15.3. Accessing Arrays

Now, we can run some queries on the table. First, we show how to access a single element of an array. This query retrieves the names of the employees whose pay changed in the second quarter:

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

 name
-------
 Carol
(1 row)
```

The array subscript numbers are written within square brackets. By default PostgreSQL uses a one-based numbering convention for arrays, that is, an array of *n* elements starts with `array[1]` and ends with `array[`*n*`]`.

This query retrieves the third quarter pay of all employees:

```
SELECT pay_by_quarter[3] FROM sal_emp;

 pay_by_quarter
----------------
          10000
          25000
```

```
(2 rows)
```

We can also access arbitrary rectangular slices of an array, or subarrays. An array slice is denoted by writing *lower-bound*:*upper-bound* for one or more array dimensions. For example, this query retrieves the first item on Bill's schedule for the first two days of the week:

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

        schedule
------------------------
 {{meeting},{training}}
(1 row)
```

If any dimension is written as a slice, i.e., contains a colon, then all dimensions are treated as slices. Any dimension that has only a single number (no colon) is treated as being from 1 to the number specified. For example, `[2]` is treated as `[1:2]`, as in this example:

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';

                 schedule
-----------------------------------------
 {{meeting,lunch},{training,presentation}}
(1 row)
```

To avoid confusion with the non-slice case, it's best to use slice syntax for all dimensions, e.g., `[1:2][1:1]`, not `[2][1:1]`.

An array subscript expression will return null if either the array itself or any of the subscript expressions are null. Also, null is returned if a subscript is outside the array bounds (this case does not raise an error). For example, if `schedule` currently has the dimensions `[1:3][1:2]` then referencing `schedule[3][3]` yields NULL. Similarly, an array reference with the wrong number of subscripts yields a null rather than an error.

An array slice expression likewise yields null if the array itself or any of the subscript expressions are null. However, in other cases such as selecting an array slice that is completely outside the current array bounds, a slice expression yields an empty (zero-dimensional) array instead of null. (This does not match non-slice behavior and is done for historical reasons.) If the requested slice partially overlaps the array bounds, then it is silently reduced to just the overlapping region instead of returning null.

The current dimensions of any array value can be retrieved with the `array_dims` function:

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';

 array_dims
------------
 [1:2][1:2]
(1 row)
```

`array_dims` produces a `text` result, which is convenient for people to read but perhaps inconvenient for programs. Dimensions can also be retrieved with `array_upper` and `array_lower`, which return the upper and lower bound of a specified array dimension, respectively:

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_upper
-------------
           2
(1 row)
```

`array_length` will return the length of a specified array dimension:

```
SELECT array_length(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_length
--------------
            2
(1 row)
```

`cardinality` returns the total number of elements in an array across all dimensions. It is effectively the number of rows a call to `unnest` would yield:

```
SELECT cardinality(schedule) FROM sal_emp WHERE name = 'Carol';

 cardinality
-------------
           4
(1 row)
```

8.15.4. Modifying Arrays

An array value can be replaced completely:

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Carol';
```

or using the `ARRAY` expression syntax:

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
    WHERE name = 'Carol';
```

An array can also be updated at a single element:

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
    WHERE name = 'Bill';
```

or updated in a slice:

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
    WHERE name = 'Carol';
```

A stored array value can be enlarged by assigning to elements not already present. Any positions between those previously present and the newly assigned elements will be filled with nulls. For example, if array `myarray` currently has 4 elements, it will have six elements after an update that assigns to `myarray[6]`; `myarray[5]` will contain null. Currently, enlargement in this fashion is only allowed for one-dimensional arrays, not multidimensional arrays.

Subscripted assignment allows creation of arrays that do not use one-based subscripts. For example one might assign to `myarray[-2:7]` to create an array with subscript values from -2 to 7.

New array values can also be constructed using the concatenation operator, `||`:

```
SELECT ARRAY[1,2] || ARRAY[3,4];
 ?column?
-----------
 {1,2,3,4}
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
      ?column?
---------------------
 {{5,6},{1,2},{3,4}}
(1 row)
```

The concatenation operator allows a single element to be pushed onto the beginning or end of a one-dimensional array. It also accepts two *N*-dimensional arrays, or an *N*-dimensional and an *N+1*-dimensional array.

When a single element is pushed onto either the beginning or end of a one-dimensional array, the result is an array with the same lower bound subscript as the array operand. For example:

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);
 array_dims
------------
 [0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
 array_dims
------------
 [1:3]
(1 row)
```

When two arrays with an equal number of dimensions are concatenated, the result retains the lower bound subscript of the left-hand operand's outer dimension. The result is an array comprising every element of the left-hand operand followed by every element of the right-hand operand. For example:

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
 array_dims
------------
 [1:5]
(1 row)
```

```
SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
 array_dims
------------
 [1:5][1:2]
(1 row)
```

When an *N*-dimensional array is pushed onto the beginning or end of an *N+1*-dimensional array, the result is analogous to the element-array case above. Each *N*-dimensional sub-array is essentially an element of the *N+1*-dimensional array's outer dimension. For example:

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
 array_dims
------------
 [1:3][1:2]
(1 row)
```

An array can also be constructed by using the functions `array_prepend`, `array_append`, or `array_cat`. The first two only support one-dimensional arrays, but `array_cat` supports multidimensional arrays. Some examples:

```
SELECT array_prepend(1, ARRAY[2,3]);
 array_prepend
---------------
 {1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);
 array_append
--------------
 {1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
 array_cat
-----------
 {1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
      array_cat
---------------------
 {{1,2},{3,4},{5,6}}
(1 row)

SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
      array_cat
---------------------
 {{5,6},{1,2},{3,4}}
```

In simple cases, the concatenation operator discussed above is preferred over direct use of these functions. However, because the concatenation operator is overloaded to serve all three cases, there are situations where use of one of the functions is helpful to avoid ambiguity. For example consider:

```
SELECT ARRAY[1, 2] || '{3, 4}';  -- the untyped literal is taken as an array
 ?column?
-----------
 {1,2,3,4}

SELECT ARRAY[1, 2] || '7';                -- so is this one
ERROR:  malformed array literal: "7"

SELECT ARRAY[1, 2] || NULL;               -- so is an undecorated NULL
 ?column?
----------
 {1,2}
(1 row)

SELECT array_append(ARRAY[1, 2], NULL);    -- this might have been meant
 array_append
--------------
 {1,2,NULL}
```

In the examples above, the parser sees an integer array on one side of the concatenation operator, and a constant of undetermined type on the other. The heuristic it uses to resolve the constant's type is to assume it's of the same type as the operator's other input — in this case, integer array. So the concatenation operator is presumed to represent `array_cat`, not `array_append`. When that's the wrong choice, it could be fixed by casting the constant to the array's element type; but explicit use of `array_append` might be a preferable solution.

8.15.5. Searching in Arrays

To search for a value in an array, each value must be checked. This can be done manually, if you know the size of the array. For example:

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
                            pay_by_quarter[4] = 10000;
```

However, this quickly becomes tedious for large arrays, and is not helpful if the size of the array is unknown. An alternative method is described in Section 9.23. The above query could be replaced by:

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

In addition, you can find rows where the array has all values equal to 10000 with:

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

Alternatively, the `generate_subscripts` function can be used. For example:

```
SELECT * FROM
   (SELECT pay_by_quarter,
           generate_subscripts(pay_by_quarter, 1) AS s
      FROM sal_emp) AS foo
 WHERE pay_by_quarter[s] = 10000;
```

This function is described in Table 9-56.

You can also search an array using the `&&` operator, which checks whether the left operand overlaps with the right operand. For instance:

```
SELECT * FROM sal_emp WHERE pay_by_quarter && ARRAY[10000];
```

This and other array operators are further described in Section 9.18. It can be accelerated by an appropriate index, as described in Section 11.2.

You can also search for specific values in an array using the `array_position` and `array_positions` functions. The former returns the subscript of the first occurrence of a value in an array; the latter returns an array with the subscripts of all occurrences of the value in the array. For example:

```
SELECT array_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon');
 array_positions
-----------------
 2

SELECT array_positions(ARRAY[1, 4, 3, 1, 3, 4, 2, 1], 1);
 array_positions
-----------------
 {1,4,8}
```

> **Tip:** Arrays are not sets; searching for specific array elements can be a sign of database misdesign. Consider using a separate table with a row for each item that would be an array element. This will be easier to search, and is likely to scale better for a large number of elements.

8.15.6. Array Input and Output Syntax

The external text representation of an array value consists of items that are interpreted according to the I/O conversion rules for the array's element type, plus decoration that indicates the array structure. The decoration consists of curly braces (`{` and `}`) around the array value plus delimiter characters between adjacent items. The delimiter character is usually a comma (`,`) but can be something else: it is determined by the `typdelim` setting for the array's element type. Among the standard data types provided in the PostgreSQL distribution, all use a comma, except for type `box`, which uses a semicolon (`;`). In a multidimensional array, each dimension (row, plane, cube, etc.) gets its own level of curly braces, and delimiters must be written between adjacent curly-braced entities of the same level.

The array output routine will put double quotes around element values if they are empty strings, contain curly braces, delimiter characters, double quotes, backslashes, or white space, or match the word NULL. Double quotes and backslashes embedded in element values will be backslash-escaped. For numeric data types it is safe to assume that double quotes will never appear, but for textual data types one should be prepared to cope with either the presence or absence of quotes.

By default, the lower bound index value of an array's dimensions is set to one. To represent arrays with other lower bounds, the array subscript ranges can be specified explicitly before writing the array contents. This decoration consists of square brackets ([]) around each array dimension's lower and upper bounds, with a colon (:) delimiter character in between. The array dimension decoration is followed by an equal sign (=). For example:

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
 FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;

 e1 | e2
----+----
  1 |  6
(1 row)
```

The array output routine will include explicit dimensions in its result only when there are one or more lower bounds different from one.

If the value written for an element is NULL (in any case variant), the element is taken to be NULL. The presence of any quotes or backslashes disables this and allows the literal string value "NULL" to be entered. Also, for backward compatibility with pre-8.2 versions of PostgreSQL, the array_nulls configuration parameter can be turned off to suppress recognition of NULL as a NULL.

As shown previously, when writing an array value you can use double quotes around any individual array element. You *must* do so if the element value would otherwise confuse the array-value parser. For example, elements containing curly braces, commas (or the data type's delimiter character), double quotes, backslashes, or leading or trailing whitespace must be double-quoted. Empty strings and strings matching the word NULL must be quoted, too. To put a double quote or backslash in a quoted array element value, precede it with a backslash. Alternatively, you can avoid quotes and use backslash-escaping to protect all data characters that would otherwise be taken as array syntax.

You can add whitespace before a left brace or after a right brace. You can also add whitespace before or after any individual item string. In all of these cases the whitespace will be ignored. However, whitespace within double-quoted elements, or surrounded on both sides by non-whitespace characters of an element, is not ignored.

> **Tip:** The ARRAY constructor syntax (see Section 4.2.12) is often easier to work with than the array-literal syntax when writing array values in SQL commands. In ARRAY, individual element values are written the same way they would be written when not members of an array.

8.16. Composite Types

A *composite type* represents the structure of a row or record; it is essentially just a list of field names and their data types. PostgreSQL allows composite types to be used in many of the same ways that simple types can be used. For example, a column of a table can be declared to be of a composite type.

8.16.1. Declaration of Composite Types

Here are two simple examples of defining composite types:

```
CREATE TYPE complex AS (
    r       double precision,
    i       double precision
);

CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric
);
```

The syntax is comparable to `CREATE TABLE`, except that only field names and types can be specified; no constraints (such as `NOT NULL`) can presently be included. Note that the `AS` keyword is essential; without it, the system will think a different kind of `CREATE TYPE` command is meant, and you will get odd syntax errors.

Having defined the types, we can use them to create tables:

```
CREATE TABLE on_hand (
    item      inventory_item,
    count     integer
);

INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

or functions:

```
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric
AS 'SELECT $1.price * $2' LANGUAGE SQL;

SELECT price_extension(item, 10) FROM on_hand;
```

Whenever you create a table, a composite type is also automatically created, with the same name as the table, to represent the table's row type. For example, had we said:

```
CREATE TABLE inventory_item (
    name            text,
    supplier_id     integer REFERENCES suppliers,
    price           numeric CHECK (price > 0)
);
```

then the same inventory_item composite type shown above would come into being as a byproduct, and could be used just as above. Note however an important restriction of the current implementation: since no constraints are associated with a composite type, the constraints shown in the table definition *do not apply* to values of the composite type outside the table. (A partial workaround is to use domain types as members of composite types.)

8.16.2. Constructing Composite Values

To write a composite value as a literal constant, enclose the field values within parentheses and separate them by commas. You can put double quotes around any field value, and must do so if it contains commas or parentheses. (More details appear below.) Thus, the general format of a composite constant is the following:

```
'( val1 , val2 , ... )'
```

An example is:

```
'("fuzzy dice",42,1.99)'
```

which would be a valid value of the inventory_item type defined above. To make a field be NULL, write no characters at all in its position in the list. For example, this constant specifies a NULL third field:

```
'("fuzzy dice",42,)'
```

If you want an empty string rather than NULL, write double quotes:

```
'("",42,)'
```

Here the first field is a non-NULL empty string, the third is NULL.

(These constants are actually only a special case of the generic type constants discussed in Section 4.1.2.7. The constant is initially treated as a string and passed to the composite-type input conversion routine. An explicit type specification might be necessary to tell which type to convert the constant to.)

The ROW expression syntax can also be used to construct composite values. In most cases this is considerably simpler to use than the string-literal syntax since you don't have to worry about multiple layers of quoting. We already used this method above:

```
ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)
```

The ROW keyword is actually optional as long as you have more than one field in the expression, so these can be simplified to:

```
('fuzzy dice', 42, 1.99)
('', 42, NULL)
```

The ROW expression syntax is discussed in more detail in Section 4.2.13.

8.16.3. Accessing Composite Types

To access a field of a composite column, one writes a dot and the field name, much like selecting a field from a table name. In fact, it's so much like selecting from a table name that you often have to use parentheses to keep from confusing the parser. For example, you might try to select some subfields from our `on_hand` example table with something like:

```
SELECT item.name FROM on_hand WHERE item.price > 9.99;
```

This will not work since the name `item` is taken to be a table name, not a column name of `on_hand`, per SQL syntax rules. You must write it like this:

```
SELECT (item).name FROM on_hand WHERE (item).price > 9.99;
```

or if you need to use the table name as well (for instance in a multitable query), like this:

```
SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;
```

Now the parenthesized object is correctly interpreted as a reference to the `item` column, and then the subfield can be selected from it.

Similar syntactic issues apply whenever you select a field from a composite value. For instance, to select just one field from the result of a function that returns a composite value, you'd need to write something like:

```
SELECT (my_func(...)).field FROM ...
```

Without the extra parentheses, this will generate a syntax error.

The special field name $*$ means "all fields", as further explained in Section 8.16.5.

8.16.4. Modifying Composite Types

Here are some examples of the proper syntax for inserting and updating composite columns. First, inserting or updating a whole column:

```
INSERT INTO mytab (complex_col) VALUES((1.1,2.2));

UPDATE mytab SET complex_col = ROW(1.1,2.2) WHERE ...;
```

The first example omits `ROW`, the second uses it; we could have done it either way.

We can update an individual subfield of a composite column:

```
UPDATE mytab SET complex_col.r = (complex_col).r + 1 WHERE ...;
```

Notice here that we don't need to (and indeed cannot) put parentheses around the column name appearing just after `SET`, but we do need parentheses when referencing the same column in the expression to the right of the equal sign.

And we can specify subfields as targets for `INSERT`, too:

```
INSERT INTO mytab (complex_col.r, complex_col.i) VALUES(1.1, 2.2);
```

Had we not supplied values for all the subfields of the column, the remaining subfields would have been filled with null values.

8.16.5. Using Composite Types in Queries

There are various special syntax rules and behaviors associated with composite types in queries. These rules provide useful shortcuts, but can be confusing if you don't know the logic behind them.

In PostgreSQL, a reference to a table name (or alias) in a query is effectively a reference to the composite value of the table's current row. For example, if we had a table `inventory_item` as shown above, we could write:

```
SELECT c FROM inventory_item c;
```

This query produces a single composite-valued column, so we might get output like:

```
           c
------------------------
 ("fuzzy dice",42,1.99)
(1 row)
```

Note however that simple names are matched to column names before table names, so this example works only because there is no column named `c` in the query's tables.

The ordinary qualified-column-name syntax *`table_name.column_name`* can be understood as applying field selection to the composite value of the table's current row. (For efficiency reasons, it's not actually implemented that way.)

When we write

```
SELECT c.* FROM inventory_item c;
```

then, according to the SQL standard, we should get the contents of the table expanded into separate columns:

```
    name    | supplier_id | price
------------+-------------+-------
 fuzzy dice |          42 |  1.99
(1 row)
```

as if the query were

```
SELECT c.name, c.supplier_id, c.price FROM inventory_item c;
```

PostgreSQL will apply this expansion behavior to any composite-valued expression, although as shown above, you need to write parentheses around the value that `.*` is applied to whenever it's not a simple table name. For example, if `myfunc()` is a function returning a composite type with columns `a`, `b`, and `c`, then these two queries have the same result:

```
SELECT (myfunc(x)).* FROM some_table;
SELECT (myfunc(x)).a, (myfunc(x)).b, (myfunc(x)).c FROM some_table;
```

> **Tip:** PostgreSQL handles column expansion by actually transforming the first form into the second. So, in this example, `myfunc()` would get invoked three times per row with either syntax. If it's an expensive function you may wish to avoid that, which you can do with a query like:
>
> ```
> SELECT (m).* FROM (SELECT myfunc(x) AS m FROM some_table OFFSET 0) ss;
> ```
>
> The `OFFSET 0` clause keeps the optimizer from "flattening" the sub-select to arrive at the form with multiple calls of `myfunc()`.

The *composite_value*`.*` syntax results in column expansion of this kind when it appears at the top level of a `SELECT` output list, a `RETURNING` list in `INSERT`/`UPDATE`/`DELETE`, a `VALUES` clause, or a row constructor. In all other contexts (including when nested inside one of those constructs), attaching `.*` to a composite value does not change the value, since it means "all columns" and so the same composite value is produced again. For example, if `somefunc()` accepts a composite-valued argument, these queries are the same:

```
SELECT somefunc(c.*) FROM inventory_item c;
SELECT somefunc(c) FROM inventory_item c;
```

In both cases, the current row of `inventory_item` is passed to the function as a single composite-valued argument. Even though `.*` does nothing in such cases, using it is good style, since it makes clear that a composite value is intended. In particular, the parser will consider `c` in `c.*` to refer to a table name or alias, not to a column name, so that there is no ambiguity; whereas without `.*`, it is not clear whether `c` means a table name or a column name, and in fact the column-name interpretation will be preferred if there is a column named `c`.

Another example demonstrating these concepts is that all these queries mean the same thing:

```
SELECT * FROM inventory_item c ORDER BY c;
SELECT * FROM inventory_item c ORDER BY c.*;
SELECT * FROM inventory_item c ORDER BY ROW(c.*);
```

All of these `ORDER BY` clauses specify the row's composite value, resulting in sorting the rows according to the rules described in Section 9.23.6. However, if `inventory_item` contained a column named `c`, the first case would be different from the others, as it would mean to sort by that column only. Given the column names previously shown, these queries are also equivalent to those above:

```
SELECT * FROM inventory_item c ORDER BY ROW(c.name, c.supplier_id, c.price);
SELECT * FROM inventory_item c ORDER BY (c.name, c.supplier_id, c.price);
```

(The last case uses a row constructor with the key word `ROW` omitted.)

Another special syntactical behavior associated with composite values is that we can use *functional notation* for extracting a field of a composite value. The simple way to explain this is that the notations *field*`(`*table*`)` and *table*`.`*field* are interchangeable. For example, these queries are equivalent:

```
SELECT c.name FROM inventory_item c WHERE c.price > 1000;
SELECT name(c) FROM inventory_item c WHERE price(c) > 1000;
```

Moreover, if we have a function that accepts a single argument of a composite type, we can call it with either notation. These queries are all equivalent:

```
SELECT somefunc(c) FROM inventory_item c;
SELECT somefunc(c.*) FROM inventory_item c;
SELECT c.somefunc FROM inventory_item c;
```

This equivalence between functional notation and field notation makes it possible to use functions on composite types to implement "computed fields". An application using the last query above wouldn't need to be directly aware that `somefunc` isn't a real column of the table.

> **Tip:** Because of this behavior, it's unwise to give a function that takes a single composite-type argument the same name as any of the fields of that composite type. If there is ambiguity, the field-name interpretation will be preferred, so that such a function could not be called without tricks. One way to force the function interpretation is to schema-qualify the function name, that is, write *schema*`.`*func*`(`*compositevalue*`)`.

8.16.6. Composite Type Input and Output Syntax

The external text representation of a composite value consists of items that are interpreted according to the I/O conversion rules for the individual field types, plus decoration that indicates the composite structure. The decoration consists of parentheses (`(` and `)`) around the whole value, plus commas (`,`) between adjacent items. Whitespace outside the parentheses is ignored, but within the parentheses it is considered part of the field value, and might or might not be significant depending on the input conversion rules for the field data type. For example, in:

```
'(  42)'
```

the whitespace will be ignored if the field type is integer, but not if it is text.

As shown previously, when writing a composite value you can write double quotes around any individual field value. You *must* do so if the field value would otherwise confuse the composite-value parser. In particular, fields containing parentheses, commas, double quotes, or backslashes must be double-quoted. To put a double quote or backslash in a quoted composite field value, precede it with a backslash. (Also, a pair of double quotes within a double-quoted field value is taken to represent a double quote character, analogously to the rules for single quotes in SQL literal strings.) Alternatively, you can avoid quoting and use backslash-escaping to protect all data characters that would otherwise be taken as composite syntax.

A completely empty field value (no characters at all between the commas or parentheses) represents a NULL. To write a value that is an empty string rather than NULL, write `""`.

The composite output routine will put double quotes around field values if they are empty strings or contain parentheses, commas, double quotes, backslashes, or white space. (Doing so for white space is not essential, but aids legibility.) Double quotes and backslashes embedded in field values will be doubled.

> **Note:** Remember that what you write in an SQL command will first be interpreted as a string literal, and then as a composite. This doubles the number of backslashes you need (assuming escape string syntax is used). For example, to insert a `text` field containing a double quote and a backslash in a composite value, you'd need to write:

```
INSERT ... VALUES ('("\"\\")');
```

The string-literal processor removes one level of backslashes, so that what arrives at the composite-value parser looks like `("\"\\")`. In turn, the string fed to the `text` data type's input routine becomes `"\`. (If we were working with a data type whose input routine also treated backslashes specially, `bytea` for example, we might need as many as eight backslashes in the command to get one backslash into the stored composite field.) Dollar quoting (see Section 4.1.2.4) can be used to avoid the need to double backslashes.

Tip: The `ROW` constructor syntax is usually easier to work with than the composite-literal syntax when writing composite values in SQL commands. In `ROW`, individual field values are written the same way they would be written when not members of a composite.

8.17. Range Types

Range types are data types representing a range of values of some element type (called the range's *subtype*). For instance, ranges of `timestamp` might be used to represent the ranges of time that a meeting room is reserved. In this case the data type is `tsrange` (short for "timestamp range"), and `timestamp` is the subtype. The subtype must have a total order so that it is well-defined whether element values are within, before, or after a range of values.

Range types are useful because they represent many element values in a single range value, and because concepts such as overlapping ranges can be expressed clearly. The use of time and date ranges for scheduling purposes is the clearest example; but price ranges, measurement ranges from an instrument, and so forth can also be useful.

8.17.1. Built-in Range Types

PostgreSQL comes with the following built-in range types:

- `int4range` — Range of `integer`
- `int8range` — Range of `bigint`
- `numrange` — Range of `numeric`
- `tsrange` — Range of `timestamp without time zone`
- `tstzrange` — Range of `timestamp with time zone`
- `daterange` — Range of `date`

In addition, you can define your own range types; see CREATE TYPE for more information.

8.17.2. Examples

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
    (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');

-- Containment
SELECT int4range(10, 20) @> 3;

-- Overlaps
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- Extract the upper bound
SELECT upper(int8range(15, 25));

-- Compute the intersection
SELECT int4range(10, 20) * int4range(15, 25);

-- Is the range empty?
SELECT isempty(numrange(1, 5));
```

See Table 9-47 and Table 9-48 for complete lists of operators and functions on range types.

8.17.3. Inclusive and Exclusive Bounds

Every non-empty range has two bounds, the lower bound and the upper bound. All points between these values are included in the range. An inclusive bound means that the boundary point itself is included in the range as well, while an exclusive bound means that the boundary point is not included in the range.

In the text form of a range, an inclusive lower bound is represented by "`[`" while an exclusive lower bound is represented by "`(`". Likewise, an inclusive upper bound is represented by "`]`", while an exclusive upper bound is represented by "`)`". (See Section 8.17.5 for more details.)

The functions `lower_inc` and `upper_inc` test the inclusivity of the lower and upper bounds of a range value, respectively.

8.17.4. Infinite (Unbounded) Ranges

The lower bound of a range can be omitted, meaning that all values less than the upper bound are included in the range, e.g., `(,3]`. Likewise, if the upper bound of the range is omitted, then all values greater than the lower bound are included in the range. If both lower and upper bounds are omitted, all values of the element type are considered to be in the range. Specifying a missing bound as inclusive is automatically converted to exclusive, e.g., `[,]` is converted to `(,)`. You can think of these missing values as +/-infinity, but they are special range type values and are considered to be beyond any range element type's +/-infinity values.

Element types that have the notion of "infinity" can use them as explicit bound values. For example, with timestamp ranges, `[today,infinity)` excludes the special `timestamp` value `infinity`, while `[today,infinity]` include it, as does `[today,)` and `[today,]`.

The functions `lower_inf` and `upper_inf` test for infinite lower and upper bounds of a range, respectively.

8.17.5. Range Input/Output

The input for a range value must follow one of the following patterns:

```
(lower-bound,upper-bound)
(lower-bound,upper-bound]
[lower-bound,upper-bound)
[lower-bound,upper-bound]
empty
```

The parentheses or brackets indicate whether the lower and upper bounds are exclusive or inclusive, as described previously. Notice that the final pattern is `empty`, which represents an empty range (a range that contains no points).

The *`lower-bound`* may be either a string that is valid input for the subtype, or empty to indicate no lower bound. Likewise, *`upper-bound`* may be either a string that is valid input for the subtype, or empty to indicate no upper bound.

Each bound value can be quoted using `"` (double quote) characters. This is necessary if the bound value contains parentheses, brackets, commas, double quotes, or backslashes, since these characters would otherwise be taken as part of the range syntax. To put a double quote or backslash in a quoted bound value, precede it with a backslash. (Also, a pair of double quotes within a double-quoted bound value is taken to represent a double quote character, analogously to the rules for single quotes in SQL literal strings.) Alternatively, you can avoid quoting and use backslash-escaping to protect all data characters that would otherwise be taken as range syntax. Also, to write a bound value that is an empty string, write `""`, since writing nothing means an infinite bound.

Whitespace is allowed before and after the range value, but any whitespace between the parentheses or brackets is taken as part of the lower or upper bound value. (Depending on the element type, it might or might not be significant.)

> **Note:** These rules are very similar to those for writing field values in composite-type literals. See Section 8.16.6 for additional commentary.

Examples:

```
-- includes 3, does not include 7, and does include all points in between
SELECT '[3,7)'::int4range;

-- does not include either 3 or 7, but includes all points in between
SELECT '(3,7)'::int4range;

-- includes only the single point 4
SELECT '[4,4]'::int4range;

-- includes no points (and will be normalized to 'empty')
SELECT '[4,4)'::int4range;
```

8.17.6. Constructing Ranges

Each range type has a constructor function with the same name as the range type. Using the constructor function is frequently more convenient than writing a range literal constant, since it avoids the need for extra quoting of the bound values. The constructor function accepts two or three arguments. The two-argument form constructs a range in standard form (lower bound inclusive, upper bound exclusive), while the three-argument form constructs a range with bounds of the form specified by the third argument. The third argument must be one of the strings "`()`", "`(]`", "`[)`", or "`[]`". For example:

```
-- The full form is: lower bound, upper bound, and text argument indicating
-- inclusivity/exclusivity of bounds.
SELECT numrange(1.0, 14.0, '(]');

-- If the third argument is omitted, '[)' is assumed.
SELECT numrange(1.0, 14.0);

-- Although '(]' is specified here, on display the value will be converted to
-- canonical form, since int8range is a discrete range type (see below).
SELECT int8range(1, 14, '(]');

-- Using NULL for either bound causes the range to be unbounded on that side.
SELECT numrange(NULL, 2.2);
```

8.17.7. Discrete Range Types

A discrete range is one whose element type has a well-defined "step", such as `integer` or `date`. In these types two elements can be said to be adjacent, when there are no valid values between them. This contrasts with continuous ranges, where it's always (or almost always) possible to identify other element values between two given values. For example, a range over the `numeric` type is continuous, as is a range over `timestamp`. (Even though `timestamp` has limited precision, and so could theoretically be treated as discrete, it's better to consider it continuous since the step size is normally not of interest.)

Another way to think about a discrete range type is that there is a clear idea of a "next" or "previous" value for each element value. Knowing that, it is possible to convert between inclusive and exclusive representations of a range's bounds, by choosing the next or previous element value instead of the one originally given. For example, in an integer range type `[4,8]` and `(3,9)` denote the same set of values; but this would not be so for a range over numeric.

A discrete range type should have a *canonicalization* function that is aware of the desired step size for the element type. The canonicalization function is charged with converting equivalent values of the range type to have identical representations, in particular consistently inclusive or exclusive bounds. If a canonicalization function is not specified, then ranges with different formatting will always be treated as unequal, even though they might represent the same set of values in reality.

The built-in range types `int4range`, `int8range`, and `daterange` all use a canonical form that includes the lower bound and excludes the upper bound; that is, `[)`. User-defined range types can use other conventions, however.

8.17.8. Defining New Range Types

Users can define their own range types. The most common reason to do this is to use ranges over subtypes not provided among the built-in range types. For example, to define a new range type of subtype `float8`:

```
CREATE TYPE floatrange AS RANGE (
    subtype = float8,
    subtype_diff = float8mi
);

SELECT '[1.234, 5.678]'::floatrange;
```

Because `float8` has no meaningful "step", we do not define a canonicalization function in this example.

Defining your own range type also allows you to specify a different subtype B-tree operator class or collation to use, so as to change the sort ordering that determines which values fall into a given range.

If the subtype is considered to have discrete rather than continuous values, the `CREATE TYPE` command should specify a `canonical` function. The canonicalization function takes an input range value, and must return an equivalent range value that may have different bounds and formatting. The canonical output for two ranges that represent the same set of values, for example the integer ranges `[1, 7]` and `[1, 8)`, must be identical. It doesn't matter which representation you choose to be the canonical one, so long as two equivalent values with different formattings are always mapped to the same value with the same formatting. In addition to adjusting the inclusive/exclusive bounds format, a canonicalization function might round off boundary values, in case the desired step size is larger than what the subtype is capable of storing. For instance, a range type over `timestamp` could be defined to have a step size of an hour, in which case the canonicalization function would need to round off bounds that weren't a multiple of an hour, or perhaps throw an error instead.

In addition, any range type that is meant to be used with GiST or SP-GiST indexes should define a subtype difference, or `subtype_diff`, function. (The index will still work without `subtype_diff`, but it is likely to be considerably less efficient than if a difference function is provided.) The subtype difference function takes two input values of the subtype, and returns their difference (i.e., *`X`* minus *`Y`*) represented as a `float8` value. In our example above, the function `float8mi` that underlies the regular `float8` minus operator can be used; but for any other subtype, some type conversion would be necessary. Some creative thought about how to represent differences as numbers might be needed, too. To the greatest extent possible, the `subtype_diff` function should agree with the sort ordering implied by the selected operator class and collation; that is, its result should be positive whenever its first argument is greater than its second according to the sort ordering.

A less-oversimplified example of a `subtype_diff` function is:

```
CREATE FUNCTION time_subtype_diff(x time, y time) RETURNS float8 AS
'SELECT EXTRACT(EPOCH FROM (x - y))' LANGUAGE sql STRICT IMMUTABLE;

CREATE TYPE timerange AS RANGE (
    subtype = time,
```

```
    subtype_diff = time_subtype_diff
);

SELECT '[11:10, 23:00]'::timerange;
```

See CREATE TYPE for more information about creating range types.

8.17.9. Indexing

GiST and SP-GiST indexes can be created for table columns of range types. For instance, to create a GiST index:

```
CREATE INDEX reservation_idx ON reservation USING GIST (during);
```

A GiST or SP-GiST index can accelerate queries involving these range operators: `=`, `&&`, `<@`, `@>`, `<<`, `>>`, `-|-`, `&<`, and `&>` (see Table 9-47 for more information).

In addition, B-tree and hash indexes can be created for table columns of range types. For these index types, basically the only useful range operation is equality. There is a B-tree sort ordering defined for range values, with corresponding < and > operators, but the ordering is rather arbitrary and not usually useful in the real world. Range types' B-tree and hash support is primarily meant to allow sorting and hashing internally in queries, rather than creation of actual indexes.

8.17.10. Constraints on Ranges

While `UNIQUE` is a natural constraint for scalar values, it is usually unsuitable for range types. Instead, an exclusion constraint is often more appropriate (see CREATE TABLE ... CONSTRAINT ... EXCLUDE). Exclusion constraints allow the specification of constraints such as "non-overlapping" on a range type. For example:

```
CREATE TABLE reservation (
    during tsrange,
    EXCLUDE USING GIST (during WITH &&)
);
```

That constraint will prevent any overlapping values from existing in the table at the same time:

```
INSERT INTO reservation VALUES
    ('[2010-01-01 11:30, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO reservation VALUES
    ('[2010-01-01 14:45, 2010-01-01 15:45)');
ERROR:  conflicting key value violates exclusion constraint "reservation_during_excl"
DETAIL:  Key (during)=(["2010-01-01 14:45:00","2010-01-01 15:45:00")) conflicts
with existing key (during)=(["2010-01-01 11:30:00","2010-01-01 15:00:00")).
```

You can use the btree_gist extension to define exclusion constraints on plain scalar data types, which can then be combined with range exclusions for maximum flexibility. For example, after btree_gist is installed, the following constraint will reject overlapping ranges only if the meeting room numbers are equal:

```
CREATE EXTENSION btree_gist;
CREATE TABLE room_reservation (
    room text,
    during tsrange,
    EXCLUDE USING GIST (room WITH =, during WITH &&)
);

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:00, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO room_reservation VALUES
    ('123A', '[2010-01-01 14:30, 2010-01-01 15:30)');
ERROR:  conflicting key value violates exclusion constraint "room_reservation_room_during_e
DETAIL:  Key (room, during)=(123A, ["2010-01-01 14:30:00","2010-01-01 15:30:00")) conflicts
with existing key (room, during)=(123A, ["2010-01-01 14:00:00","2010-01-01 15:00:00")).

INSERT INTO room_reservation VALUES
    ('123B', '[2010-01-01 14:30, 2010-01-01 15:30)');
INSERT 0 1
```

8.18. Object Identifier Types

Object identifiers (OIDs) are used internally by PostgreSQL as primary keys for various system tables. OIDs are not added to user-created tables, unless WITH OIDS is specified when the table is created, or the default_with_oids configuration variable is enabled. Type oid represents an object identifier. There are also several alias types for oid: regproc, regprocedure, regoper, regoperator, regclass, regtype, regrole, regnamespace, regconfig, and regdictionary. Table 8-24 shows an overview.

The oid type is currently implemented as an unsigned four-byte integer. Therefore, it is not large enough to provide database-wide uniqueness in large databases, or even in large individual tables. So, using a user-created table's OID column as a primary key is discouraged. OIDs are best used only for references to system tables.

The oid type itself has few operations beyond comparison. It can be cast to integer, however, and then manipulated using the standard integer operators. (Beware of possible signed-versus-unsigned confusion if you do this.)

The OID alias types have no operations of their own except for specialized input and output routines. These routines are able to accept and display symbolic names for system objects, rather than the raw numeric value that type oid would use. The alias types allow simplified lookup of OID values for objects. For example, to examine the pg_attribute rows related to a table mytable, one could write:

```
SELECT * FROM pg_attribute WHERE attrelid = 'mytable'::regclass;
```

rather than:

```
SELECT * FROM pg_attribute
  WHERE attrelid = (SELECT oid FROM pg_class WHERE relname = 'mytable');
```

While that doesn't look all that bad by itself, it's still oversimplified. A far more complicated sub-select would be needed to select the right OID if there are multiple tables named `mytable` in different schemas. The `regclass` input converter handles the table lookup according to the schema path setting, and so it does the "right thing" automatically. Similarly, casting a table's OID to `regclass` is handy for symbolic display of a numeric OID.

Table 8-24. Object Identifier Types

Name	References	Description	Value Example
`oid`	any	numeric object identifier	`564182`
`regproc`	`pg_proc`	function name	`sum`
`regprocedure`	`pg_proc`	function with argument types	`sum(int4)`
`regoper`	`pg_operator`	operator name	`+`
`regoperator`	`pg_operator`	operator with argument types	`*(integer,integer)` or `-(NONE,integer)`
`regclass`	`pg_class`	relation name	`pg_type`
`regtype`	`pg_type`	data type name	`integer`
`regrole`	`pg_authid`	role name	`smithee`
`regnamespace`	`pg_namespace`	namespace name	`pg_catalog`
`regconfig`	`pg_ts_config`	text search configuration	`english`
`regdictionary`	`pg_ts_dict`	text search dictionary	`simple`

All of the OID alias types for objects grouped by namespace accept schema-qualified names, and will display schema-qualified names on output if the object would not be found in the current search path without being qualified. The `regproc` and `regoper` alias types will only accept input names that are unique (not overloaded), so they are of limited use; for most uses `regprocedure` or `regoperator` are more appropriate. For `regoperator`, unary operators are identified by writing `NONE` for the unused operand.

An additional property of most of the OID alias types is the creation of dependencies. If a constant of one of these types appears in a stored expression (such as a column default expression or view), it creates a dependency on the referenced object. For example, if a column has a default expression `nextval('my_seq'::regclass)`, PostgreSQL understands that the default expression depends on the sequence `my_seq`; the system will not let the sequence be dropped without first removing the default expression. `regrole` is the only exception for the property. Constants of this type are not allowed in such expressions.

Note: The OID alias types do not completely follow transaction isolation rules. The planner also treats them as simple constants, which may result in sub-optimal planning.

Another identifier type used by the system is `xid`, or transaction (abbreviated xact) identifier. This is the data type of the system columns `xmin` and `xmax`. Transaction identifiers are 32-bit quantities.

A third identifier type used by the system is `cid`, or command identifier. This is the data type of the system columns `cmin` and `cmax`. Command identifiers are also 32-bit quantities.

A final identifier type used by the system is `tid`, or tuple identifier (row identifier). This is the data type of the system column `ctid`. A tuple ID is a pair (block number, tuple index within block) that identifies the physical location of the row within its table.

(The system columns are further explained in Section 5.4.)

8.19. pg_lsn Type

The `pg_lsn` data type can be used to store LSN (Log Sequence Number) data which is a pointer to a location in the XLOG. This type is a representation of `XLogRecPtr` and an internal system type of PostgreSQL.

Internally, an LSN is a 64-bit integer, representing a byte position in the write-ahead log stream. It is printed as two hexadecimal numbers of up to 8 digits each, separated by a slash; for example, `16/B374D848`. The `pg_lsn` type supports the standard comparison operators, like = and >. Two LSNs can be subtracted using the – operator; the result is the number of bytes separating those write-ahead log positions.

8.20. Pseudo-Types

The PostgreSQL type system contains a number of special-purpose entries that are collectively called *pseudo-types*. A pseudo-type cannot be used as a column data type, but it can be used to declare a function's argument or result type. Each of the available pseudo-types is useful in situations where a function's behavior does not correspond to simply taking or returning a value of a specific SQL data type. Table 8-25 lists the existing pseudo-types.

Table 8-25. Pseudo-Types

Name	Description
`any`	Indicates that a function accepts any input data type.
`anyelement`	Indicates that a function accepts any data type (see Section 35.2.5).
`anyarray`	Indicates that a function accepts any array data type (see Section 35.2.5).

Name	Description
anynonarray	Indicates that a function accepts any non-array data type (see Section 35.2.5).
anyenum	Indicates that a function accepts any enum data type (see Section 35.2.5 and Section 8.7).
anyrange	Indicates that a function accepts any range data type (see Section 35.2.5 and Section 8.17).
cstring	Indicates that a function accepts or returns a null-terminated C string.
internal	Indicates that a function accepts or returns a server-internal data type.
language_handler	A procedural language call handler is declared to return language_handler.
fdw_handler	A foreign-data wrapper handler is declared to return fdw_handler.
tsm_handler	A tablesample method handler is declared to return tsm_handler.
record	Identifies a function taking or returning an unspecified row type.
trigger	A trigger function is declared to return trigger.
event_trigger	An event trigger function is declared to return event_trigger.
pg_ddl_command	Identifies a representation of DDL commands that is available to event triggers.
void	Indicates that a function returns no value.
opaque	An obsolete type name that formerly served all the above purposes.

Functions coded in C (whether built-in or dynamically loaded) can be declared to accept or return any of these pseudo data types. It is up to the function author to ensure that the function will behave safely when a pseudo-type is used as an argument type.

Functions coded in procedural languages can use pseudo-types only as allowed by their implementation languages. At present most procedural languages forbid use of a pseudo-type as an argument type, and allow only void and record as a result type (plus trigger or event_trigger when the function is used as a trigger or event trigger). Some also support polymorphic functions using the types anyelement, anyarray, anynonarray, anyenum, and anyrange.

The internal pseudo-type is used to declare functions that are meant only to be called internally by the database system, and not by direct invocation in an SQL query. If a function has at least one internal-type argument then it cannot be called from SQL. To preserve the type safety of this restriction it is important to follow this coding rule: do not create any function that is declared to return internal unless it has at least one internal argument.

Chapter 9. Functions and Operators

PostgreSQL provides a large number of functions and operators for the built-in data types. Users can also define their own functions and operators, as described in Part V. The psql commands `\df` and `\do` can be used to list all available functions and operators, respectively.

If you are concerned about portability then note that most of the functions and operators described in this chapter, with the exception of the most trivial arithmetic and comparison operators and some explicitly marked functions, are not specified by the SQL standard. Some of this extended functionality is present in other SQL database management systems, and in many cases this functionality is compatible and consistent between the various implementations. This chapter is also not exhaustive; additional functions appear in relevant sections of the manual.

9.1. Logical Operators

The usual logical operators are available:

```
AND
OR
NOT
```

SQL uses a three-valued logic system with true, false, and `null`, which represents "unknown". Observe the following truth tables:

a	*b*	*a* AND *b*	*a* OR *b*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	NOT *a*
TRUE	FALSE
FALSE	TRUE
NULL	NULL

The operators `AND` and `OR` are commutative, that is, you can switch the left and right operand without affecting the result. But see Section 4.2.14 for more information about the order of evaluation of subexpressions.

9.2. Comparison Operators

The usual comparison operators are available, shown in Table 9-1.

Table 9-1. Comparison Operators

Operator	Description
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal
<> or !=	not equal

Note: The != operator is converted to <> in the parser stage. It is not possible to implement != and <> operators that do different things.

Comparison operators are available for all relevant data types. All comparison operators are binary operators that return values of type `boolean`; expressions like `1 < 2 < 3` are not valid (because there is no < operator to compare a Boolean value with `3`).

In addition to the comparison operators, the special `BETWEEN` construct is available:

```
a BETWEEN x AND y
```

is equivalent to

```
a >= x AND a <= y
```

Notice that `BETWEEN` treats the endpoint values as included in the range. `NOT BETWEEN` does the opposite comparison:

```
a NOT BETWEEN x AND y
```

is equivalent to

```
a < x OR a > y
```

`BETWEEN SYMMETRIC` is the same as `BETWEEN` except there is no requirement that the argument to the left of `AND` be less than or equal to the argument on the right. If it is not, those two arguments are automatically swapped, so that a nonempty range is always implied.

Ordinary comparison operators yield null (signifying "unknown"), not true or false, when either input is null. For example, `7 = NULL` yields null, as does `7 <> NULL`. When this behavior is not suitable, use the `IS [ NOT ] DISTINCT FROM` constructs:

```
a IS DISTINCT FROM b
a IS NOT DISTINCT FROM b
```

For non-null inputs, `IS DISTINCT FROM` is the same as the <> operator. However, if both inputs are null it returns false, and if only one input is null it returns true. Similarly, `IS NOT DISTINCT FROM` is identical to = for non-null inputs, but it returns true when both inputs are null, and false when only one input is null. Thus, these constructs effectively act as though null were a normal data value, rather than "unknown".

To check whether a value is or is not null, use the constructs:

```
expression IS NULL
expression IS NOT NULL
```

or the equivalent, but nonstandard, constructs:

```
expression ISNULL
expression NOTNULL
```

Do *not* write `expression = NULL` because `NULL` is not "equal to" `NULL`. (The null value represents an unknown value, and it is not known whether two unknown values are equal.)

> **Tip:** Some applications might expect that `expression = NULL` returns true if `expression` evaluates to the null value. It is highly recommended that these applications be modified to comply with the SQL standard. However, if that cannot be done the transform_null_equals configuration variable is available. If it is enabled, PostgreSQL will convert `x = NULL` clauses to `x IS NULL`.

If the `expression` is row-valued, then `IS NULL` is true when the row expression itself is null or when all the row's fields are null, while `IS NOT NULL` is true when the row expression itself is non-null and all the row's fields are non-null. Because of this behavior, `IS NULL` and `IS NOT NULL` do not always return inverse results for row-valued expressions; in particular, a row-valued expression that contains both null and non-null fields will return false for both tests. In some cases, it may be preferable to write `row IS DISTINCT FROM NULL` or `row IS NOT DISTINCT FROM NULL`, which will simply check whether the overall row value is null without any additional tests on the row fields.

Boolean values can also be tested using the constructs

```
expression IS TRUE
expression IS NOT TRUE
expression IS FALSE
expression IS NOT FALSE
expression IS UNKNOWN
expression IS NOT UNKNOWN
```

These will always return true or false, never a null value, even when the operand is null. A null input is treated as the logical value "unknown". Notice that `IS UNKNOWN` and `IS NOT UNKNOWN` are effectively the same as `IS NULL` and `IS NOT NULL`, respectively, except that the input expression must be of Boolean type.

9.3. Mathematical Functions and Operators

Mathematical operators are provided for many PostgreSQL types. For types without standard mathematical conventions (e.g., date/time types) we describe the actual behavior in subsequent sections.

Table 9-2 shows the available mathematical operators.

Table 9-2. Mathematical Operators

Operator	Description	Example	Result
+	addition	2 + 3	5
-	subtraction	2 - 3	-1
*	multiplication	2 * 3	6
/	division (integer division truncates the result)	4 / 2	2
%	modulo (remainder)	5 % 4	1
^	exponentiation (associates left to right)	2.0 ^ 3.0	8
|/	square root	|/ 25.0	5
||/	cube root	||/ 27.0	3
!	factorial (deprecated, use factorial() instead)	5 !	120
!!	factorial as a prefix operator (deprecated, use factorial() instead)	!! 5	120
@	absolute value	@ -5.0	5
&	bitwise AND	91 & 15	11
|	bitwise OR	32 | 3	35
#	bitwise XOR	17 # 5	20
~	bitwise NOT	~1	-2
<<	bitwise shift left	1 << 4	16
>>	bitwise shift right	8 >> 2	2

The bitwise operators work only on integral data types and are also available for the bit string types bit and bit varying, as shown in Table 9-11.

Table 9-3 shows the available mathematical functions. In the table, dp indicates double precision. Many of these functions are provided in multiple forms with different argument types. Except where noted, any given form of a function returns the same data type as its argument. The functions working with double precision data are mostly implemented on top of the host system's C library; accuracy and behavior in boundary cases can therefore vary depending on the host system.

Table 9-3. Mathematical Functions

Function	Return Type	Description	Example	Result
abs(*x*)	(same as input)	absolute value	abs(-17.4)	17.4
cbrt(dp)	dp	cube root	cbrt(27.0)	3
ceil(dp or numeric)	(same as input)	nearest integer greater than or equal to argument	ceil(-42.8)	-42
ceiling(dp or numeric)	(same as input)	nearest integer greater than or equal to argument (same as ceil)	ceiling(-95.3)	-95
degrees(dp)	dp	radians to degrees	degrees(0.5)	28.6478897565412
div(y numeric, x numeric)	numeric	integer quotient of y/x	div(9,4)	2
exp(dp or numeric)	(same as input)	exponential	exp(1.0)	2.71828182845905
factorial(bigint)	numeric	factorial	factorial(5)	120
floor(dp or numeric)	(same as input)	nearest integer less than or equal to argument	floor(-42.8)	-43
ln(dp or numeric)	(same as input)	natural logarithm	ln(2.0)	0.693147180559945
log(dp or numeric)	(same as input)	base 10 logarithm	log(100.0)	2
log(b numeric, x numeric)	numeric	logarithm to base b	log(2.0, 64.0)	6.0000000000
mod(y, x)	(same as argument types)	remainder of y/x	mod(9,4)	1
pi()	dp	"π" constant	pi()	3.14159265358979
power(a dp, b dp)	dp	a raised to the power of b	power(9.0, 3.0)	729
power(a numeric, b numeric)	numeric	a raised to the power of b	power(9.0, 3.0)	729
radians(dp)	dp	degrees to radians	radians(45.0)	0.785398163397448
round(dp or numeric)	(same as input)	round to nearest integer	round(42.4)	42
round(v numeric, s int)	numeric	round to s decimal places	round(42.4382, 2)	42.44

Function	Return Type	Description	Example	Result
sign(dp or numeric)	(same as input)	sign of the argument (-1, 0, +1)	sign(-8.4)	-1
sqrt(dp or numeric)	(same as input)	square root	sqrt(2.0)	1.4142135623731
trunc(dp or numeric)	(same as input)	truncate toward zero	trunc(42.8)	42
trunc(v numeric, s int)	numeric	truncate to s decimal places	trunc(42.4382, 2)	42.43
width_bucket(operand dp, b1 dp, b2 dp, count int)	int	return the bucket number to which operand would be assigned in a histogram having count equal-width buckets spanning the range b1 to b2; returns 0 or count+1 for an input outside the range	width_bucket(5.35, 0.024, 10.06, 5)	3
width_bucket(operand numeric, b1 numeric, b2 numeric, count int)	int	return the bucket number to which operand would be assigned in a histogram having count equal-width buckets spanning the range b1 to b2; returns 0 or count+1 for an input outside the range	width_bucket(5.35, 0.024, 10.06, 5)	3

Function	Return Type	Description	Example	Result
`width_bucket(operand anyelement, thresholds anyarray)`	`int`	return the bucket number to which `operand` would be assigned given an array listing the lower bounds of the buckets; returns `0` for an input less than the first lower bound; the `thresholds` array *must be sorted*, smallest first, or unexpected results will be obtained	`width_bucket(now(), array['yesterday', 'today', 'tomorrow']::timestamptz[])`	`2`

Table 9-4 shows functions for generating random numbers.

Table 9-4. Random Functions

Function	Return Type	Description
`random()`	`dp`	random value in the range 0.0 <= x < 1.0
`setseed(dp)`	`void`	set seed for subsequent `random()` calls (value between -1.0 and 1.0, inclusive)

The characteristics of the values returned by `random()` depend on the system implementation. It is not suitable for cryptographic applications; see pgcrypto module for an alternative.

Finally, Table 9-5 shows the available trigonometric functions. All trigonometric functions take arguments and return values of type `double precision`. Trigonometric functions arguments are expressed in radians. Inverse functions return values are expressed in radians. See unit transformation functions `radians()` and `degrees()` above.

Table 9-5. Trigonometric Functions

Function	Description
`acos(x)`	inverse cosine
`asin(x)`	inverse sine
`atan(x)`	inverse tangent
`atan2(y, x)`	inverse tangent of `y/x`
`cos(x)`	cosine
`cot(x)`	cotangent
`sin(x)`	sine

Function	Description
tan(x)	tangent

9.4. String Functions and Operators

This section describes functions and operators for examining and manipulating string values. Strings in this context include values of the types character, character varying, and text. Unless otherwise noted, all of the functions listed below work on all of these types, but be wary of potential effects of automatic space-padding when using the character type. Some functions also exist natively for the bit-string types.

SQL defines some string functions that use key words, rather than commas, to separate arguments. Details are in Table 9-6. PostgreSQL also provides versions of these functions that use the regular function invocation syntax (see Table 9-7).

> **Note:** Before PostgreSQL 8.3, these functions would silently accept values of several non-string data types as well, due to the presence of implicit coercions from those data types to text. Those coercions have been removed because they frequently caused surprising behaviors. However, the string concatenation operator (||) still accepts non-string input, so long as at least one input is of a string type, as shown in Table 9-6. For other cases, insert an explicit coercion to text if you need to duplicate the previous behavior.

Table 9-6. SQL String Functions and Operators

Function	Return Type	Description	Example	Result
string || string	text	String concatenation	'Post' || 'greSQL'	PostgreSQL
string || non-string or non-string || string	text	String concatenation with one non-string input	'Value: ' || 42	Value: 42
bit_length(string)	int	Number of bits in string	bit_length('jose')	32
char_length(string) or character_length(string)	int	Number of characters in string	char_length('jose')	4
lower(string)	text	Convert string to lower case	lower('TOM')	tom

Function	Return Type	Description	Example	Result
octet_length(string)	int	Number of bytes in string	octet_length('jose')	4
overlay(string placing string from int [for int])	text	Replace substring	overlay('Txxxxas' placing 'hom' from 2 for 4)	Thomas
position(substring in string)	int	Location of specified substring	position('om' in 'Thomas')	3
substring(string [from int] [for int])	text	Extract substring	substring('Thomas' from 2 for 3)	hom
substring(string from *pattern*)	text	Extract substring matching POSIX regular expression. See Section 9.7 for more information on pattern matching.	substring('Thomas' from '...$')	mas
substring(string from *pattern* for *escape*)	text	Extract substring matching SQL regular expression. See Section 9.7 for more information on pattern matching.	substring('Thomas' from '%#"o_a#"_' for '#')	oma
trim([leading | trailing | both] [characters] from string)	text	Remove the longest string containing only characters from characters (a space by default) from the start, end, or both ends (both is the default) of string	trim(both 'xyz' from 'yxTomxx')	Tom
trim([leading | trailing | both] [from] string [, characters])	text	Non-standard syntax for trim()	trim(both from 'yxTomxx', 'xyz')	Tom
upper(string)	text	Convert string to upper case	upper('tom')	TOM

Additional string manipulation functions are available and are listed in Table 9-7. Some of them are used internally to implement the SQL-standard string functions listed in Table 9-6.

Table 9-7. Other String Functions

Function	Return Type	Description	Example	Result
`ascii(string)`	`int`	ASCII code of the first character of the argument. For UTF8 returns the Unicode code point of the character. For other multibyte encodings, the argument must be an ASCII character.	`ascii('x')`	`120`
`btrim(string text [, characters text])`	`text`	Remove the longest string consisting only of characters in `characters` (a space by default) from the start and end of `string`	`btrim('xyxtrimyyx', 'xyz')`	`trim`
`chr(int)`	`text`	Character with the given code. For UTF8 the argument is treated as a Unicode code point. For other multibyte encodings the argument must designate an ASCII character. The NULL (0) character is not allowed because text data types cannot store such bytes.	`chr(65)`	`A`

Function	Return Type	Description	Example	Result
concat(str "any" [, str "any" [, ...]])	text	Concatenate the text representations of all the arguments. NULL arguments are ignored.	concat('abcde', 2, NULL, 22)	abcde222
concat_ws(sep text, str "any" [, str "any" [, ...]])	text	Concatenate all but the first argument with separators. The first argument is used as the separator string. NULL arguments are ignored.	concat_ws(',', 'abcde', 2, NULL, 22)	abcde,2,22
convert(string bytea, src_encoding name, dest_encoding name)	bytea	Convert string to dest_encoding. The original encoding is specified by src_encoding. The string must be valid in this encoding. Conversions can be defined by CREATE CONVERSION. Also there are some predefined conversions. See Table 9-8 for available conversions.	convert('text_in_utf8', 'UTF8', 'LATIN1')	text_in_utf8 represented in Latin-1 encoding (ISO 8859-1)
convert_from(string bytea, src_encoding name)	text	Convert string to the database encoding. The original encoding is specified by src_encoding. The string must be valid in this encoding.	convert_from('text_in_utf8', 'UTF8')	text_in_utf8 represented in the current database encoding

Function	Return Type	Description	Example	Result
convert_to(string text, dest_encoding name)	bytea	Convert string to dest_encoding.	convert_to('some text', 'UTF8')	some text represented in the UTF8 encoding
decode(string text, format text)	bytea	Decode binary data from textual representation in string. Options for format are same as in encode.	decode('MTIzAAE=', 'base64')	3132330001
encode(data bytea, format text)	text	Encode binary data into a textual representation. Supported formats are: base64, hex, escape. escape converts zero bytes and high-bit-set bytes to octal sequences (\nnn) and doubles backslashes.	encode('123\000\001', 'base64')	MTIzAAE=
format(formatstr text [, formatarg "any" [, ...]])	text	Format arguments according to a format string. This function is similar to the C function sprintf. See Section 9.4.1.	format('Hello %s, %1$s', 'World')	Hello World, World
initcap(string)	text	Convert the first letter of each word to upper case and the rest to lower case. Words are sequences of alphanumeric characters separated by non-alphanumeric characters.	initcap('hi THOMAS')	Hi Thomas

Function	Return Type	Description	Example	Result
left(str text, n int)	text	Return first *n* characters in the string. When *n* is negative, return all but last |*n*| characters.	left('abcde', 2)	ab
length(string)	int	Number of characters in string	length('jose')	4
length(string bytea, encoding name)	int	Number of characters in string in the given encoding. The string must be valid in this encoding.	length('jose', 'UTF8')	4
lpad(string text, length int [, fill text])	text	Fill up the string to length length by prepending the characters fill (a space by default). If the string is already longer than length then it is truncated (on the right).	lpad('hi', 5, 'xy')	xyxhi
ltrim(string text [, characters text])	text	Remove the longest string containing only characters from characters (a space by default) from the start of string	ltrim('zzzytest', 'xyz')	test
md5(string)	text	Calculates the MD5 hash of string, returning the result in hexadecimal	md5('abc')	900150983cd24fb0d6963f7d28e17f72
pg_client_encoding()	name	Current client encoding name	pg_client_encoding()	SQL_ASCII

Function	Return Type	Description	Example	Result
quote_ident(string text)	text	Return the given string suitably quoted to be used as an identifier in an SQL statement string. Quotes are added only if necessary (i.e., if the string contains non-identifier characters or would be case-folded). Embedded quotes are properly doubled. See also Example 40-1.	quote_ident('Foo bar')	"Foo bar"
quote_literal(string text)	text	Return the given string suitably quoted to be used as a string literal in an SQL statement string. Embedded single-quotes and backslashes are properly doubled. Note that quote_literal returns null on null input; if the argument might be null, quote_nullable is often more suitable. See also Example 40-1.	quote_literal(E'O\'Reilly')	E'O\'Reilly'
quote_literal(value anyelement)	text	Coerce the given value to text and then quote it as a literal. Embedded single-quotes and backslashes are properly doubled.	quote_literal(42.5)	'42.5'

Function	Return Type	Description	Example	Result
`quote_nullable(string text)`	`text`	Return the given string suitably quoted to be used as a string literal in an SQL statement string; or, if the argument is null, return `NULL`. Embedded single-quotes and backslashes are properly doubled. See also Example 40-1.	`quote_nullable(NULL)`	`NULL`
`quote_nullable(value anyelement)`	`text`	Coerce the given value to text and then quote it as a literal; or, if the argument is null, return `NULL`. Embedded single-quotes and backslashes are properly doubled.	`quote_nullable(42.5)`	`'42.5'`
`regexp_matches(string text, pattern text [, flags text])`	`setof text[]`	Return all captured substrings resulting from matching a POSIX regular expression against the `string`. See Section 9.7.3 for more information.	`regexp_matches('foobarbequebaz', '(bar)(beque)')`	`{bar,beque}`
`regexp_replace(string text, pattern text, replacement text [, flags text])`	`text`	Replace substring(s) matching a POSIX regular expression. See Section 9.7.3 for more information.	`regexp_replace('Thomas', '.[mN]a.', 'M')`	`ThM`

Function	Return Type	Description	Example	Result
regexp_split_to_array(string text, pattern text [, flags text])	text[]	Split string using a POSIX regular expression as the delimiter. See Section 9.7.3 for more information.	regexp_split_to_array('hello world', '\s+')	{hello,world}
regexp_split_to_table(string text, pattern text [, flags text])	setof text	Split string using a POSIX regular expression as the delimiter. See Section 9.7.3 for more information.	regexp_split_to_table('hello world', '\s+')	hello world (2 rows)
repeat(string text, number int)	text	Repeat string the specified number of times	repeat('Pg', 4)	PgPgPgPg
replace(string text, from text, to text)	text	Replace all occurrences in string of substring from with substring to	replace('abcdefabcdef', 'cd', 'XX')	abXXefabXXef
reverse(str)	text	Return reversed string.	reverse('abcde')	edcba
right(str text, n int)	text	Return last *n* characters in the string. When *n* is negative, return all but first |*n*| characters.	right('abcde', 2)	de
rpad(string text, length int [, fill text])	text	Fill up the string to length length by appending the characters fill (a space by default). If the string is already longer than length then it is truncated.	rpad('hi', 5, 'xy')	hixyx

Function	Return Type	Description	Example	Result
rtrim(string text [, characters text])	text	Remove the longest string containing only characters from characters (a space by default) from the end of string	rtrim('testxxzx', 'xyz')	test
split_part(string text, delimiter text, field int)	text	Split string on delimiter and return the given field (counting from one)	split_part('abc~@~def~@~ghi', '~@~', 2)	def
strpos(string, substring)	int	Location of specified substring (same as position(substring in string), but note the reversed argument order)	strpos('high', 'ig')	2
substr(string, from [, count])	text	Extract substring (same as substring(string from from for count))	substr('alphabet', 3, 2)	ph
to_ascii(string text [, encoding text])	text	Convert string to ASCII from another encoding (only supports conversion from LATIN1, LATIN2, LATIN9, and WIN1250 encodings)	to_ascii('Karel')	Karel
to_hex(number int or bigint)	text	Convert number to its equivalent hexadecimal representation	to_hex(2147483647)	7fffffff

Function	Return Type	Description	Example	Result
translate(string text, from text, to text)	text	Any character in string that matches a character in the from set is replaced by the corresponding character in the to set. If from is longer than to, occurrences of the extra characters in from are removed.	translate('12345', '143', 'ax')	a2x5

The concat, concat_ws and format functions are variadic, so it is possible to pass the values to be concatenated or formatted as an array marked with the VARIADIC keyword (see Section 35.4.5). The array's elements are treated as if they were separate ordinary arguments to the function. If the variadic array argument is NULL, concat and concat_ws return NULL, but format treats a NULL as a zero-element array.

See also the aggregate function string_agg in Section 9.20.

Table 9-8. Built-in Conversions

Conversion Name [a]	Source Encoding	Destination Encoding
ascii_to_mic	SQL_ASCII	MULE_INTERNAL
ascii_to_utf8	SQL_ASCII	UTF8
big5_to_euc_tw	BIG5	EUC_TW
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf8	BIG5	UTF8
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf8	EUC_CN	UTF8
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS
euc_jp_to_utf8	EUC_JP	UTF8
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf8	EUC_KR	UTF8
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf8	EUC_TW	UTF8
gb18030_to_utf8	GB18030	UTF8
gbk_to_utf8	GBK	UTF8

Conversion Name [a]	Source Encoding	Destination Encoding
iso_8859_10_to_utf8	LATIN6	UTF8
iso_8859_13_to_utf8	LATIN7	UTF8
iso_8859_14_to_utf8	LATIN8	UTF8
iso_8859_15_to_utf8	LATIN9	UTF8
iso_8859_16_to_utf8	LATIN10	UTF8
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf8	LATIN1	UTF8
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf8	LATIN2	UTF8
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf8	LATIN3	UTF8
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf8	LATIN4	UTF8
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8R
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf8	ISO_8859_5	UTF8
iso_8859_5_to_windows_1251	ISO_8859_5	WIN1251
iso_8859_5_to_windows_866	ISO_8859_5	WIN866
iso_8859_6_to_utf8	ISO_8859_6	UTF8
iso_8859_7_to_utf8	ISO_8859_7	UTF8
iso_8859_8_to_utf8	ISO_8859_8	UTF8
iso_8859_9_to_utf8	LATIN5	UTF8
johab_to_utf8	JOHAB	UTF8
koi8_r_to_iso_8859_5	KOI8R	ISO_8859_5
koi8_r_to_mic	KOI8R	MULE_INTERNAL
koi8_r_to_utf8	KOI8R	UTF8
koi8_r_to_windows_1251	KOI8R	WIN1251
koi8_r_to_windows_866	KOI8R	WIN866
koi8_u_to_utf8	KOI8U	UTF8
mic_to_ascii	MULE_INTERNAL	SQL_ASCII
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW

Conversion Name [a]	Source Encoding	Destination Encoding
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8R
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN1251
mic_to_windows_866	MULE_INTERNAL	WIN866
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf8	SJIS	UTF8
windows_1258_to_utf8	WIN1258	UTF8
uhc_to_utf8	UHC	UTF8
utf8_to_ascii	UTF8	SQL_ASCII
utf8_to_big5	UTF8	BIG5
utf8_to_euc_cn	UTF8	EUC_CN
utf8_to_euc_jp	UTF8	EUC_JP
utf8_to_euc_kr	UTF8	EUC_KR
utf8_to_euc_tw	UTF8	EUC_TW
utf8_to_gb18030	UTF8	GB18030
utf8_to_gbk	UTF8	GBK
utf8_to_iso_8859_1	UTF8	LATIN1
utf8_to_iso_8859_10	UTF8	LATIN6
utf8_to_iso_8859_13	UTF8	LATIN7
utf8_to_iso_8859_14	UTF8	LATIN8
utf8_to_iso_8859_15	UTF8	LATIN9
utf8_to_iso_8859_16	UTF8	LATIN10
utf8_to_iso_8859_2	UTF8	LATIN2
utf8_to_iso_8859_3	UTF8	LATIN3
utf8_to_iso_8859_4	UTF8	LATIN4
utf8_to_iso_8859_5	UTF8	ISO_8859_5
utf8_to_iso_8859_6	UTF8	ISO_8859_6
utf8_to_iso_8859_7	UTF8	ISO_8859_7
utf8_to_iso_8859_8	UTF8	ISO_8859_8
utf8_to_iso_8859_9	UTF8	LATIN5
utf8_to_johab	UTF8	JOHAB

Conversion Name [a]	Source Encoding	Destination Encoding
utf8_to_koi8_r	UTF8	KOI8R
utf8_to_koi8_u	UTF8	KOI8U
utf8_to_sjis	UTF8	SJIS
utf8_to_windows_1258	UTF8	WIN1258
utf8_to_uhc	UTF8	UHC
utf8_to_windows_1250	UTF8	WIN1250
utf8_to_windows_1251	UTF8	WIN1251
utf8_to_windows_1252	UTF8	WIN1252
utf8_to_windows_1253	UTF8	WIN1253
utf8_to_windows_1254	UTF8	WIN1254
utf8_to_windows_1255	UTF8	WIN1255
utf8_to_windows_1256	UTF8	WIN1256
utf8_to_windows_1257	UTF8	WIN1257
utf8_to_windows_866	UTF8	WIN866
utf8_to_windows_874	UTF8	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf8	WIN1250	UTF8
windows_1251_to_iso_8859_5	WIN1251	ISO_8859_5
windows_1251_to_koi8_r	WIN1251	KOI8R
windows_1251_to_mic	WIN1251	MULE_INTERNAL
windows_1251_to_utf8	WIN1251	UTF8
windows_1251_to_windows_866	WIN1251	WIN866
windows_1252_to_utf8	WIN1252	UTF8
windows_1256_to_utf8	WIN1256	UTF8
windows_866_to_iso_8859_5	WIN866	ISO_8859_5
windows_866_to_koi8_r	WIN866	KOI8R
windows_866_to_mic	WIN866	MULE_INTERNAL
windows_866_to_utf8	WIN866	UTF8
windows_866_to_windows_1251	WIN866	WIN
windows_874_to_utf8	WIN874	UTF8
euc_jis_2004_to_utf8	EUC_JIS_2004	UTF8
utf8_to_euc_jis_2004	UTF8	EUC_JIS_2004
shift_jis_2004_to_utf8	SHIFT_JIS_2004	UTF8

Conversion Name [a]	Source Encoding	Destination Encoding
utf8_to_shift_jis_2004	UTF8	SHIFT_JIS_2004
euc_jis_2004_to_shift_jis_2004	EUC_JIS_2004	SHIFT_JIS_2004
shift_jis_2004_to_euc_jis_2004	SHIFT_JIS_2004	EUC_JIS_2004

Notes:
a. The conversion names follow a standard naming scheme: The official name of the source encoding with all non-alphanumeric characters replaced by underscores, followed by _to_, followed by the similarly processed destination encoding name. Therefore, the names might deviate from the customary encoding names.

9.4.1. format

The function format produces output formatted according to a format string, in a style similar to the C function sprintf.

```
format(formatstr text [, formatarg "any" [, ...] ])
```

formatstr is a format string that specifies how the result should be formatted. Text in the format string is copied directly to the result, except where *format specifiers* are used. Format specifiers act as placeholders in the string, defining how subsequent function arguments should be formatted and inserted into the result. Each *formatarg* argument is converted to text according to the usual output rules for its data type, and then formatted and inserted into the result string according to the format specifier(s).

Format specifiers are introduced by a % character and have the form

```
%[position][flags][width]type
```

where the component fields are:

position (optional)

A string of the form *n*$ where *n* is the index of the argument to print. Index 1 means the first argument after *formatstr*. If the *position* is omitted, the default is to use the next argument in sequence.

flags (optional)

Additional options controlling how the format specifier's output is formatted. Currently the only supported flag is a minus sign (-) which will cause the format specifier's output to be left-justified. This has no effect unless the *width* field is also specified.

width (optional)

Specifies the *minimum* number of characters to use to display the format specifier's output. The output is padded on the left or right (depending on the - flag) with spaces as needed to fill the width. A too-small width does not cause truncation of the output, but is simply ignored. The width may be specified using any of the following: a positive integer; an asterisk (*) to use the next function argument as the width; or a string of the form *n*$ to use the *n*th function argument as the width.

If the width comes from a function argument, that argument is consumed before the argument that is used for the format specifier's value. If the width argument is negative, the result is left aligned (as if the - flag had been specified) within a field of length `abs(`*`width`*`)`.

`type` (required)

The type of format conversion to use to produce the format specifier's output. The following types are supported:

- `s` formats the argument value as a simple string. A null value is treated as an empty string.
- `I` treats the argument value as an SQL identifier, double-quoting it if necessary. It is an error for the value to be null (equivalent to `quote_ident`).
- `L` quotes the argument value as an SQL literal. A null value is displayed as the string `NULL`, without quotes (equivalent to `quote_nullable`).

In addition to the format specifiers described above, the special sequence `%%` may be used to output a literal `%` character.

Here are some examples of the basic format conversions:

```
SELECT format('Hello %s', 'World');
Result: Hello World

SELECT format('Testing %s, %s, %s, %%', 'one', 'two', 'three');
Result: Testing one, two, three, %

SELECT format('INSERT INTO %I VALUES(%L)', 'Foo bar', E'O\'Reilly');
Result: INSERT INTO "Foo bar" VALUES('O''Reilly')

SELECT format('INSERT INTO %I VALUES(%L)', 'locations', 'C:\Program Files');
Result: INSERT INTO locations VALUES('C:\Program Files')
```

Here are examples using *`width`* fields and the - flag:

```
SELECT format('|%10s|', 'foo');
Result: |       foo|

SELECT format('|%-10s|', 'foo');
Result: |foo       |

SELECT format('|%*s|', 10, 'foo');
Result: |       foo|

SELECT format('|%*s|', -10, 'foo');
Result: |foo       |

SELECT format('|%-*s|', 10, 'foo');
Result: |foo       |
```

```
SELECT format('|%-*s|', -10, 'foo');
Result: |foo       |
```

These examples show use of *position* fields:

```
SELECT format('Testing %3$s, %2$s, %1$s', 'one', 'two', 'three');
Result: Testing three, two, one

SELECT format('|%*2$s|', 'foo', 10, 'bar');
Result: |       bar|

SELECT format('|%1$*2$s|', 'foo', 10, 'bar');
Result: |       foo|
```

Unlike the standard C function `sprintf`, PostgreSQL's `format` function allows format specifiers with and without *position* fields to be mixed in the same format string. A format specifier without a *position* field always uses the next argument after the last argument consumed. In addition, the `format` function does not require all function arguments to be used in the format string. For example:

```
SELECT format('Testing %3$s, %2$s, %s', 'one', 'two', 'three');
Result: Testing three, two, three
```

The `%I` and `%L` format specifiers are particularly useful for safely constructing dynamic SQL statements. See Example 40-1.

9.5. Binary String Functions and Operators

This section describes functions and operators for examining and manipulating values of type `bytea`.

SQL defines some string functions that use key words, rather than commas, to separate arguments. Details are in Table 9-9. PostgreSQL also provides versions of these functions that use the regular function invocation syntax (see Table 9-10).

> **Note:** The sample results shown on this page assume that the server parameter `bytea_output` is set to `escape` (the traditional PostgreSQL format).

Table 9-9. SQL Binary String Functions and Operators

Function	Return Type	Description	Example	Result

Function	Return Type	Description	Example	Result
string || string	bytea	String concatenation	'\\Post'::bytea || '\047gres\000'::bytea	\\Post'gres\000
octet_length(string)	int	Number of bytes in binary string	octet_length('jo\000se'::bytea)	5
overlay(string placing string from int [for int])	bytea	Replace substring	overlay('Th\000omas'::bytea placing '\002\003'::bytea from 2 for 3)	Th\000\002\003as
position(substring in string)	int	Location of specified substring	position('\000om'::bytea in 'Th\000omas'::bytea)	3
substring(string [from int] [for int])	bytea	Extract substring	substring('Th\000omas'::bytea from 2 for 3)	h\000o
trim([both] bytes from string)	bytea	Remove the longest string containing only bytes appearing in bytes from the start and end of string	trim('\000\001'::bytea from '\000Tom\001'::bytea)	Tom

Additional binary string manipulation functions are available and are listed in Table 9-10. Some of them are used internally to implement the SQL-standard string functions listed in Table 9-9.

Table 9-10. Other Binary String Functions

Function	Return Type	Description	Example	Result
btrim(string bytea, bytes bytea)	bytea	Remove the longest string containing only bytes appearing in bytes from the start and end of string	btrim('\000trim\001'::bytea, '\000\001'::bytea)	trim

Function	Return Type	Description	Example	Result
decode(string text, format text)	bytea	Decode binary data from textual representation in string. Options for format are same as in encode.	decode('123\000456', 'escape')	123\000456
encode(data bytea, format text)	text	Encode binary data into a textual representation. Supported formats are: base64, hex, escape. escape converts zero bytes and high-bit-set bytes to octal sequences (\nnn) and doubles backslashes.	encode('123\000456'::bytea, 'escape')	123\000456
get_bit(string, offset)	int	Extract bit from string	get_bit('Th\000omas'::bytea, 45)	1
get_byte(string, offset)	int	Extract byte from string	get_byte('Th\000omas'::bytea, 4)	109
length(string)	int	Length of binary string	length('jo\000se'::bytea)	5
md5(string)	text	Calculates the MD5 hash of string, returning the result in hexadecimal	md5('Th\000omas'::bytea)	8ab2d3c9689aaf18b4958c334c82d8b1
set_bit(string, offset, newvalue)	bytea	Set bit in string	set_bit('Th\000omas'::bytea, 45, 0)	Th\000omAs
set_byte(string, offset, newvalue)	bytea	Set byte in string	set_byte('Th\000omas'::bytea, 4, 64)	Th\000@as

get_byte and set_byte number the first byte of a binary string as byte 0. get_bit and set_bit number bits from the right within each byte; for example bit 0 is the least significant bit of the first byte, and bit 15 is the most significant bit of the second byte.

See also the aggregate function string_agg in Section 9.20 and the large object functions in Section 32.4.

9.6. Bit String Functions and Operators

This section describes functions and operators for examining and manipulating bit strings, that is values of the types `bit` and `bit varying`. Aside from the usual comparison operators, the operators shown in Table 9-11 can be used. Bit string operands of `&`, `|`, and `#` must be of equal length. When bit shifting, the original length of the string is preserved, as shown in the examples.

Table 9-11. Bit String Operators

Operator	Description	Example	Result
`||`	concatenation	`B'10001' || B'011'`	`10001011`
`&`	bitwise AND	`B'10001' & B'01101'`	`00001`
`|`	bitwise OR	`B'10001' | B'01101'`	`11101`
`#`	bitwise XOR	`B'10001' # B'01101'`	`11100`
`~`	bitwise NOT	`~ B'10001'`	`01110`
`<<`	bitwise shift left	`B'10001' << 3`	`01000`
`>>`	bitwise shift right	`B'10001' >> 2`	`00100`

The following SQL-standard functions work on bit strings as well as character strings: `length`, `bit_length`, `octet_length`, `position`, `substring`, `overlay`.

The following functions work on bit strings as well as binary strings: `get_bit`, `set_bit`. When working with a bit string, these functions number the first (leftmost) bit of the string as bit 0.

In addition, it is possible to cast integral values to and from type `bit`. Some examples:

```
44::bit(10)                    0000101100
44::bit(3)                     100
cast(-44 as bit(12))           111111010100
'1110'::bit(4)::integer        14
```

Note that casting to just "bit" means casting to `bit(1)`, and so will deliver only the least significant bit of the integer.

> **Note:** Casting an integer to `bit(n)` copies the rightmost `n` bits. Casting an integer to a bit string width wider than the integer itself will sign-extend on the left.

9.7. Pattern Matching

There are three separate approaches to pattern matching provided by PostgreSQL: the traditional SQL `LIKE` operator, the more recent `SIMILAR TO` operator (added in SQL:1999), and POSIX-style regular

expressions. Aside from the basic "does this string match this pattern?" operators, functions are available to extract or replace matching substrings and to split a string at matching locations.

Tip: If you have pattern matching needs that go beyond this, consider writing a user-defined function in Perl or Tcl.

> **Caution**
>
> While most regular-expression searches can be executed very quickly, regular expressions can be contrived that take arbitrary amounts of time and memory to process. Be wary of accepting regular-expression search patterns from hostile sources. If you must do so, it is advisable to impose a statement timeout.
>
> Searches using `SIMILAR TO` patterns have the same security hazards, since `SIMILAR TO` provides many of the same capabilities as POSIX-style regular expressions.
>
> `LIKE` searches, being much simpler than the other two options, are safer to use with possibly-hostile pattern sources.

9.7.1. `LIKE`

```
string LIKE pattern [ESCAPE escape-character]
string NOT LIKE pattern [ESCAPE escape-character]
```

The `LIKE` expression returns true if the *`string`* matches the supplied *`pattern`*. (As expected, the `NOT LIKE` expression returns false if `LIKE` returns true, and vice versa. An equivalent expression is `NOT (`*`string`* `LIKE` *`pattern`*`)`.)

If *`pattern`* does not contain percent signs or underscores, then the pattern only represents the string itself; in that case `LIKE` acts like the equals operator. An underscore (_) in *`pattern`* stands for (matches) any single character; a percent sign (`%`) matches any sequence of zero or more characters.

Some examples:

```
'abc' LIKE 'abc'    true
'abc' LIKE 'a%'     true
'abc' LIKE '_b_'    true
'abc' LIKE 'c'      false
```

`LIKE` pattern matching always covers the entire string. Therefore, if it's desired to match a sequence anywhere within a string, the pattern must start and end with a percent sign.

To match a literal underscore or percent sign without matching other characters, the respective character in *`pattern`* must be preceded by the escape character. The default escape character is the backslash but a different one can be selected by using the `ESCAPE` clause. To match the escape character itself, write two escape characters.

Note: If you have standard_conforming_strings turned off, any backslashes you write in literal string constants will need to be doubled. See Section 4.1.2.1 for more information.

It's also possible to select no escape character by writing `ESCAPE ''`. This effectively disables the escape mechanism, which makes it impossible to turn off the special meaning of underscore and percent signs in the pattern.

The key word `ILIKE` can be used instead of `LIKE` to make the match case-insensitive according to the active locale. This is not in the SQL standard but is a PostgreSQL extension.

The operator `~~` is equivalent to `LIKE`, and `~~*` corresponds to `ILIKE`. There are also `!~~` and `!~~*` operators that represent `NOT LIKE` and `NOT ILIKE`, respectively. All of these operators are PostgreSQL-specific. You may see these operator names in `EXPLAIN` output and similar places, since the parser actually translates `LIKE` et al. to these operators.

The phrases `LIKE`, `ILIKE`, `NOT LIKE`, and `NOT ILIKE` are generally treated as operators in PostgreSQL syntax; for example they can be used in *`expression operator`* ANY (*`subquery`*) constructs, although an `ESCAPE` clause cannot be included there. In some obscure cases it may be necessary to use the underlying operator names instead.

9.7.2. `SIMILAR TO` Regular Expressions

`string` SIMILAR TO *`pattern`* [ESCAPE *`escape-character`*]
`string` NOT SIMILAR TO *`pattern`* [ESCAPE *`escape-character`*]

The `SIMILAR TO` operator returns true or false depending on whether its pattern matches the given string. It is similar to `LIKE`, except that it interprets the pattern using the SQL standard's definition of a regular expression. SQL regular expressions are a curious cross between `LIKE` notation and common regular expression notation.

Like `LIKE`, the `SIMILAR TO` operator succeeds only if its pattern matches the entire string; this is unlike common regular expression behavior where the pattern can match any part of the string. Also like `LIKE`, `SIMILAR TO` uses _ and `%` as wildcard characters denoting any single character and any string, respectively (these are comparable to `.` and `.*` in POSIX regular expressions).

In addition to these facilities borrowed from `LIKE`, `SIMILAR TO` supports these pattern-matching metacharacters borrowed from POSIX regular expressions:

- `|` denotes alternation (either of two alternatives).
- `*` denotes repetition of the previous item zero or more times.
- `+` denotes repetition of the previous item one or more times.
- `?` denotes repetition of the previous item zero or one time.
- `{`*`m`*`}` denotes repetition of the previous item exactly *`m`* times.
- `{`*`m`*`,}` denotes repetition of the previous item *`m`* or more times.
- `{`*`m`*`,`*`n`*`}` denotes repetition of the previous item at least *`m`* and not more than *`n`* times.
- Parentheses `()` can be used to group items into a single logical item.

- A bracket expression `[...]` specifies a character class, just as in POSIX regular expressions.

Notice that the period (`.`) is not a metacharacter for `SIMILAR TO`.

As with `LIKE`, a backslash disables the special meaning of any of these metacharacters; or a different escape character can be specified with `ESCAPE`.

Some examples:

```
'abc' SIMILAR TO 'abc'      true
'abc' SIMILAR TO 'a'        false
'abc' SIMILAR TO '%(b|d)%'  true
'abc' SIMILAR TO '(b|c)%'   false
```

The `substring` function with three parameters, `substring(`*`string`* `from` *`pattern`* `for` *`escape-character`*`)`, provides extraction of a substring that matches an SQL regular expression pattern. As with `SIMILAR TO`, the specified pattern must match the entire data string, or else the function fails and returns null. To indicate the part of the pattern that should be returned on success, the pattern must contain two occurrences of the escape character followed by a double quote (`"`). The text matching the portion of the pattern between these markers is returned.

Some examples, with `#"` delimiting the return string:

```
substring('foobar' from '%#"o_b#"%' for '#')   oob
substring('foobar' from '#"o_b#"%' for '#')    NULL
```

9.7.3. POSIX Regular Expressions

Table 9-12 lists the available operators for pattern matching using POSIX regular expressions.

Table 9-12. Regular Expression Match Operators

Operator	Description	Example
`~`	Matches regular expression, case sensitive	`'thomas' ~ '.*thomas.*'`
`~*`	Matches regular expression, case insensitive	`'thomas' ~* '.*Thomas.*'`
`!~`	Does not match regular expression, case sensitive	`'thomas' !~ '.*Thomas.*'`
`!~*`	Does not match regular expression, case insensitive	`'thomas' !~* '.*vadim.*'`

POSIX regular expressions provide a more powerful means for pattern matching than the `LIKE` and `SIMILAR TO` operators. Many Unix tools such as `egrep`, `sed`, or `awk` use a pattern matching language that is similar to the one described here.

A regular expression is a character sequence that is an abbreviated definition of a set of strings (a *regular*

set). A string is said to match a regular expression if it is a member of the regular set described by the regular expression. As with LIKE, pattern characters match string characters exactly unless they are special characters in the regular expression language — but regular expressions use different special characters than LIKE does. Unlike LIKE patterns, a regular expression is allowed to match anywhere within a string, unless the regular expression is explicitly anchored to the beginning or end of the string.

Some examples:

```
'abc' ~ 'abc'    true
'abc' ~ '^a'     true
'abc' ~ '(b|d)'  true
'abc' ~ '^(b|c)' false
```

The POSIX pattern language is described in much greater detail below.

The substring function with two parameters, substring(*string* from *pattern*), provides extraction of a substring that matches a POSIX regular expression pattern. It returns null if there is no match, otherwise the portion of the text that matched the pattern. But if the pattern contains any parentheses, the portion of the text that matched the first parenthesized subexpression (the one whose left parenthesis comes first) is returned. You can put parentheses around the whole expression if you want to use parentheses within it without triggering this exception. If you need parentheses in the pattern before the subexpression you want to extract, see the non-capturing parentheses described below.

Some examples:

```
substring('foobar' from 'o.b')     oob
substring('foobar' from 'o(.)b')   o
```

The regexp_replace function provides substitution of new text for substrings that match POSIX regular expression patterns. It has the syntax regexp_replace(*source*, *pattern*, *replacement* [, *flags*]). The *source* string is returned unchanged if there is no match to the *pattern*. If there is a match, the *source* string is returned with the *replacement* string substituted for the matching substring. The *replacement* string can contain *n*, where *n* is 1 through 9, to indicate that the source substring matching the *n*'th parenthesized subexpression of the pattern should be inserted, and it can contain \& to indicate that the substring matching the entire pattern should be inserted. Write \\ if you need to put a literal backslash in the replacement text. The *flags* parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Flag i specifies case-insensitive matching, while flag g specifies replacement of each matching substring rather than only the first one. Supported flags (though not g) are described in Table 9-20.

Some examples:

```
regexp_replace('foobarbaz', 'b..', 'X')
                                   fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g')
                                   fooXX
regexp_replace('foobarbaz', 'b(..)', 'X\1Y', 'g')
                                   fooXarYXazY
```

The `regexp_matches` function returns a text array of all of the captured substrings resulting from matching a POSIX regular expression pattern. It has the syntax `regexp_matches(`*`string`*`, `*`pattern`*` [, `*`flags`*` ])`. The function can return no rows, one row, or multiple rows (see the `g` flag below). If the *`pattern`* does not match, the function returns no rows. If the pattern contains no parenthesized subexpressions, then each row returned is a single-element text array containing the substring matching the whole pattern. If the pattern contains parenthesized subexpressions, the function returns a text array whose *`n`*'th element is the substring matching the *`n`*'th parenthesized subexpression of the pattern (not counting "non-capturing" parentheses; see below for details). The *`flags`* parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. Flag `g` causes the function to find each match in the string, not only the first one, and return a row for each such match. Supported flags (though not `g`) are described in Table 9-20.

Some examples:

```
SELECT regexp_matches('foobarbequebaz', '(bar)(beque)');
 regexp_matches
----------------
 {bar,beque}
(1 row)

SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
 regexp_matches
----------------
 {bar,beque}
 {bazil,barf}
(2 rows)

SELECT regexp_matches('foobarbequebaz', 'barbeque');
 regexp_matches
----------------
 {barbeque}
(1 row)
```

It is possible to force `regexp_matches()` to always return one row by using a sub-select; this is particularly useful in a `SELECT` target list when you want all rows returned, even non-matching ones:

```
SELECT col1, (SELECT regexp_matches(col2, '(bar)(beque)')) FROM tab;
```

The `regexp_split_to_table` function splits a string using a POSIX regular expression pattern as a delimiter. It has the syntax `regexp_split_to_table(`*`string`*`, `*`pattern`*` [, `*`flags`*` ])`. If there is no match to the *`pattern`*, the function returns the *`string`*. If there is at least one match, for each match it returns the text from the end of the last match (or the beginning of the string) to the beginning of the match. When there are no more matches, it returns the text from the end of the last match to the end of the string. The *`flags`* parameter is an optional text string containing zero or more single-letter flags that change the function's behavior. `regexp_split_to_table` supports the flags described in Table 9-20.

The `regexp_split_to_array` function behaves the same as `regexp_split_to_table`, except that `regexp_split_to_array` returns its result as an array of `text`. It has the syntax

`regexp_split_to_array(`*`string`*`, `*`pattern`* [, *`flags`*]). The parameters are the same as for `regexp_split_to_table`.

Some examples:

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumps over the lazy dog', '\s+')
  foo
-------
 the
 quick
 brown
 fox
 jumps
 over
 the
 lazy
 dog
(9 rows)

SELECT regexp_split_to_array('the quick brown fox jumps over the lazy dog', '\s+');
              regexp_split_to_array
-----------------------------------------------
 {the,quick,brown,fox,jumps,over,the,lazy,dog}
(1 row)

SELECT foo FROM regexp_split_to_table('the quick brown fox', '\s*') AS foo;
 foo
-----
 t
 h
 e
 q
 u
 i
 c
 k
 b
 r
 o
 w
 n
 f
 o
 x
(16 rows)
```

As the last example demonstrates, the regexp split functions ignore zero-length matches that occur at the start or end of the string or immediately after a previous match. This is contrary to the strict definition of regexp matching that is implemented by `regexp_matches`, but is usually the most convenient behavior in practice. Other software systems such as Perl use similar definitions.

9.7.3.1. Regular Expression Details

PostgreSQL's regular expressions are implemented using a software package written by Henry Spencer. Much of the description of regular expressions below is copied verbatim from his manual.

Regular expressions (REs), as defined in POSIX 1003.2, come in two forms: *extended* REs or EREs (roughly those of `egrep`), and *basic* REs or BREs (roughly those of `ed`). PostgreSQL supports both forms, and also implements some extensions that are not in the POSIX standard, but have become widely used due to their availability in programming languages such as Perl and Tcl. REs using these non-POSIX extensions are called *advanced* REs or AREs in this documentation. AREs are almost an exact superset of EREs, but BREs have several notational incompatibilities (as well as being much more limited). We first describe the ARE and ERE forms, noting features that apply only to AREs, and then describe how BREs differ.

> **Note:** PostgreSQL always initially presumes that a regular expression follows the ARE rules. However, the more limited ERE or BRE rules can be chosen by prepending an *embedded option* to the RE pattern, as described in Section 9.7.3.4. This can be useful for compatibility with applications that expect exactly the POSIX 1003.2 rules.

A regular expression is defined as one or more *branches*, separated by `|`. It matches anything that matches one of the branches.

A branch is zero or more *quantified atoms* or *constraints*, concatenated. It matches a match for the first, followed by a match for the second, etc; an empty branch matches the empty string.

A quantified atom is an *atom* possibly followed by a single *quantifier*. Without a quantifier, it matches a match for the atom. With a quantifier, it can match some number of matches of the atom. An *atom* can be any of the possibilities shown in Table 9-13. The possible quantifiers and their meanings are shown in Table 9-14.

A *constraint* matches an empty string, but matches only when specific conditions are met. A constraint can be used where an atom could be used, except it cannot be followed by a quantifier. The simple constraints are shown in Table 9-15; some more constraints are described later.

Table 9-13. Regular Expression Atoms

Atom	Description
`(`*re*`)`	(where *re* is any regular expression) matches a match for *re*, with the match noted for possible reporting
`(?:`*re*`)`	as above, but the match is not noted for reporting (a "non-capturing" set of parentheses) (AREs only)
.	matches any single character
`[`*chars*`]`	a *bracket expression*, matching any one of the *chars* (see Section 9.7.3.2 for more detail)
`\`*k*	(where *k* is a non-alphanumeric character) matches that character taken as an ordinary character, e.g., `\\` matches a backslash character

Atom	Description
\c	where c is alphanumeric (possibly followed by other characters) is an *escape*, see Section 9.7.3.3 (AREs only; in EREs and BREs, this matches c)
{	when followed by a character other than a digit, matches the left-brace character {; when followed by a digit, it is the beginning of a *bound* (see below)
x	where x is a single character with no other significance, matches that character

An RE cannot end with a backslash (\).

> **Note:** If you have standard_conforming_strings turned off, any backslashes you write in literal string constants will need to be doubled. See Section 4.1.2.1 for more information.

Table 9-14. Regular Expression Quantifiers

Quantifier	Matches
*	a sequence of 0 or more matches of the atom
+	a sequence of 1 or more matches of the atom
?	a sequence of 0 or 1 matches of the atom
{m}	a sequence of exactly m matches of the atom
{m,}	a sequence of m or more matches of the atom
{m,n}	a sequence of m through n (inclusive) matches of the atom; m cannot exceed n
*?	non-greedy version of *
+?	non-greedy version of +
??	non-greedy version of ?
{m}?	non-greedy version of {m}
{m,}?	non-greedy version of {m,}
{m,n}?	non-greedy version of {m,n}

The forms using {...} are known as *bounds*. The numbers m and n within a bound are unsigned decimal integers with permissible values from 0 to 255 inclusive.

Non-greedy quantifiers (available in AREs only) match the same possibilities as their corresponding normal (*greedy*) counterparts, but prefer the smallest number rather than the largest number of matches. See Section 9.7.3.5 for more detail.

> **Note:** A quantifier cannot immediately follow another quantifier, e.g., ** is invalid. A quantifier cannot begin an expression or subexpression or follow ^ or |.

Table 9-15. Regular Expression Constraints

Constraint	Description
`^`	matches at the beginning of the string
`$`	matches at the end of the string
`(?=re)`	*positive lookahead* matches at any point where a substring matching `re` begins (AREs only)
`(?!re)`	*negative lookahead* matches at any point where no substring matching `re` begins (AREs only)

Lookahead constraints cannot contain *back references* (see Section 9.7.3.3), and all parentheses within them are considered non-capturing.

9.7.3.2. Bracket Expressions

A *bracket expression* is a list of characters enclosed in `[]`. It normally matches any single character from the list (but see below). If the list begins with `^`, it matches any single character *not* from the rest of the list. If two characters in the list are separated by `-`, this is shorthand for the full range of characters between those two (inclusive) in the collating sequence, e.g., `[0-9]` in ASCII matches any decimal digit. It is illegal for two ranges to share an endpoint, e.g., `a-c-e`. Ranges are very collating-sequence-dependent, so portable programs should avoid relying on them.

To include a literal `]` in the list, make it the first character (after `^`, if that is used). To include a literal `-`, make it the first or last character, or the second endpoint of a range. To use a literal `-` as the first endpoint of a range, enclose it in `[.` and `.]` to make it a collating element (see below). With the exception of these characters, some combinations using `[` (see next paragraphs), and escapes (AREs only), all other special characters lose their special significance within a bracket expression. In particular, `\` is not special when following ERE or BRE rules, though it is special (as introducing an escape) in AREs.

Within a bracket expression, a collating element (a character, a multiple-character sequence that collates as if it were a single character, or a collating-sequence name for either) enclosed in `[.` and `.]` stands for the sequence of characters of that collating element. The sequence is treated as a single element of the bracket expression's list. This allows a bracket expression containing a multiple-character collating element to match more than one character, e.g., if the collating sequence includes a `ch` collating element, then the RE `[[.ch.]]*c` matches the first five characters of `chchcc`.

> **Note:** PostgreSQL currently does not support multi-character collating elements. This information describes possible future behavior.

Within a bracket expression, a collating element enclosed in `[=` and `=]` is an *equivalence class*, standing for the sequences of characters of all collating elements equivalent to that one, including itself. (If there are no other equivalent collating elements, the treatment is as if the enclosing delimiters were `[.` and `.]`.) For example, if `o` and `^` are the members of an equivalence class, then `[[=o=]]`, `[[=^=]]`, and `[o^]` are all synonymous. An equivalence class cannot be an endpoint of a range.

Within a bracket expression, the name of a character class enclosed in `[:` and `:]` stands for the list of all characters belonging to that class. Standard character class names are: `alnum`, `alpha`, `blank`, `cntrl`,

`digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`. These stand for the character classes defined in ctype. A locale can provide others. A character class cannot be used as an endpoint of a range.

There are two special cases of bracket expressions: the bracket expressions `[[:<:]]` and `[[:>:]]` are constraints, matching empty strings at the beginning and end of a word respectively. A word is defined as a sequence of word characters that is neither preceded nor followed by word characters. A word character is an `alnum` character (as defined by ctype) or an underscore. This is an extension, compatible with but not specified by POSIX 1003.2, and should be used with caution in software intended to be portable to other systems. The constraint escapes described below are usually preferable; they are no more standard, but are easier to type.

9.7.3.3. Regular Expression Escapes

Escapes are special sequences beginning with \ followed by an alphanumeric character. Escapes come in several varieties: character entry, class shorthands, constraint escapes, and back references. A \ followed by an alphanumeric character but not constituting a valid escape is illegal in AREs. In EREs, there are no escapes: outside a bracket expression, a \ followed by an alphanumeric character merely stands for that character as an ordinary character, and inside a bracket expression, \ is an ordinary character. (The latter is the one actual incompatibility between EREs and AREs.)

Character-entry escapes exist to make it easier to specify non-printing and other inconvenient characters in REs. They are shown in Table 9-16.

Class-shorthand escapes provide shorthands for certain commonly-used character classes. They are shown in Table 9-17.

A *constraint escape* is a constraint, matching the empty string if specific conditions are met, written as an escape. They are shown in Table 9-18.

A *back reference* (`\`*n*) matches the same string matched by the previous parenthesized subexpression specified by the number *n* (see Table 9-19). For example, `([bc])\1` matches `bb` or `cc` but not `bc` or `cb`. The subexpression must entirely precede the back reference in the RE. Subexpressions are numbered in the order of their leading parentheses. Non-capturing parentheses do not define subexpressions.

Table 9-16. Regular Expression Character-entry Escapes

Escape	Description
`\a`	alert (bell) character, as in C
`\b`	backspace, as in C
`\B`	synonym for backslash (`\`) to help reduce the need for backslash doubling
`\c`*X*	(where *X* is any character) the character whose low-order 5 bits are the same as those of *X*, and whose other bits are all zero
`\e`	the character whose collating-sequence name is `ESC`, or failing that, the character with octal value `033`
`\f`	form feed, as in C
`\n`	newline, as in C

Escape	Description
\r	carriage return, as in C
\t	horizontal tab, as in C
\u*wxyz*	(where *wxyz* is exactly four hexadecimal digits) the character whose hexadecimal value is 0x*wxyz*
\U*stuvwxyz*	(where *stuvwxyz* is exactly eight hexadecimal digits) the character whose hexadecimal value is 0x*stuvwxyz*
\v	vertical tab, as in C
\x*hhh*	(where *hhh* is any sequence of hexadecimal digits) the character whose hexadecimal value is 0x*hhh* (a single character no matter how many hexadecimal digits are used)
\0	the character whose value is 0 (the null byte)
xy	(where *xy* is exactly two octal digits, and is not a *back reference*) the character whose octal value is 0*xy*
xyz	(where *xyz* is exactly three octal digits, and is not a *back reference*) the character whose octal value is 0*xyz*

Hexadecimal digits are 0-9, a-f, and A-F. Octal digits are 0-7.

Numeric character-entry escapes specifying values outside the ASCII range (0-127) have meanings dependent on the database encoding. When the encoding is UTF-8, escape values are equivalent to Unicode code points, for example \u1234 means the character U+1234. For other multibyte encodings, character-entry escapes usually just specify the concatenation of the byte values for the character. If the escape value does not correspond to any legal character in the database encoding, no error will be raised, but it will never match any data.

The character-entry escapes are always taken as ordinary characters. For example, \135 is] in ASCII, but \135 does not terminate a bracket expression.

Table 9-17. Regular Expression Class-shorthand Escapes

Escape	Description
\d	[[:digit:]]
\s	[[:space:]]
\w	[[:alnum:]_] (note underscore is included)
\D	[^[:digit:]]
\S	[^[:space:]]
\W	[^[:alnum:]_] (note underscore is included)

Within bracket expressions, \d, \s, and \w lose their outer brackets, and \D, \S, and \W are illegal. (So, for example, [a-c\d] is equivalent to [a-c[:digit:]]. Also, [a-c\D], which is equivalent to [a-c^[:digit:]], is illegal.)

Table 9-18. Regular Expression Constraint Escapes

Escape	Description
`\A`	matches only at the beginning of the string (see Section 9.7.3.5 for how this differs from `^`)
`\m`	matches only at the beginning of a word
`\M`	matches only at the end of a word
`\y`	matches only at the beginning or end of a word
`\Y`	matches only at a point that is not the beginning or end of a word
`\Z`	matches only at the end of the string (see Section 9.7.3.5 for how this differs from `$`)

A word is defined as in the specification of `[[:<:]]` and `[[:>:]]` above. Constraint escapes are illegal within bracket expressions.

Table 9-19. Regular Expression Back References

Escape	Description
`\`*`m`*	(where *`m`* is a nonzero digit) a back reference to the *`m`*'th subexpression
`\`*`mnn`*	(where *`m`* is a nonzero digit, and *`nn`* is some more digits, and the decimal value *`mnn`* is not greater than the number of closing capturing parentheses seen so far) a back reference to the *`mnn`*'th subexpression

> **Note:** There is an inherent ambiguity between octal character-entry escapes and back references, which is resolved by the following heuristics, as hinted at above. A leading zero always indicates an octal escape. A single non-zero digit, not followed by another digit, is always taken as a back reference. A multi-digit sequence not starting with a zero is taken as a back reference if it comes after a suitable subexpression (i.e., the number is in the legal range for a back reference), and otherwise is taken as octal.

9.7.3.4. Regular Expression Metasyntax

In addition to the main syntax described above, there are some special forms and miscellaneous syntactic facilities available.

An RE can begin with one of two special *director* prefixes. If an RE begins with `***:`, the rest of the RE is taken as an ARE. (This normally has no effect in PostgreSQL, since REs are assumed to be AREs; but it does have an effect if ERE or BRE mode had been specified by the *`flags`* parameter to a regex function.) If an RE begins with `***=`, the rest of the RE is taken to be a literal string, with all characters considered ordinary characters.

An ARE can begin with *embedded options*: a sequence `(?xyz)` (where *xyz* is one or more alphabetic characters) specifies options affecting the rest of the RE. These options override any previously determined options — in particular, they can override the case-sensitivity behavior implied by a regex operator, or the *flags* parameter to a regex function. The available option letters are shown in Table 9-20. Note that these same option letters are used in the *flags* parameters of regex functions.

Table 9-20. ARE Embedded-option Letters

Option	Description
b	rest of RE is a BRE
c	case-sensitive matching (overrides operator type)
e	rest of RE is an ERE
i	case-insensitive matching (see Section 9.7.3.5) (overrides operator type)
m	historical synonym for n
n	newline-sensitive matching (see Section 9.7.3.5)
p	partial newline-sensitive matching (see Section 9.7.3.5)
q	rest of RE is a literal ("quoted") string, all ordinary characters
s	non-newline-sensitive matching (default)
t	tight syntax (default; see below)
w	inverse partial newline-sensitive ("weird") matching (see Section 9.7.3.5)
x	expanded syntax (see below)

Embedded options take effect at the `)` terminating the sequence. They can appear only at the start of an ARE (after the `***:` director if any).

In addition to the usual (*tight*) RE syntax, in which all characters are significant, there is an *expanded* syntax, available by specifying the embedded `x` option. In the expanded syntax, white-space characters in the RE are ignored, as are all characters between a `#` and the following newline (or the end of the RE). This permits paragraphing and commenting a complex RE. There are three exceptions to that basic rule:

- a white-space character or `#` preceded by `\` is retained
- white space or `#` within a bracket expression is retained
- white space and comments cannot appear within multi-character symbols, such as `(?:`

For this purpose, white-space characters are blank, tab, newline, and any character that belongs to the *space* character class.

Finally, in an ARE, outside bracket expressions, the sequence `(?#ttt)` (where *ttt* is any text not containing a `)`) is a comment, completely ignored. Again, this is not allowed between the characters of multi-character symbols, like `(?:`. Such comments are more a historical artifact than a useful facility, and their use is deprecated; use the expanded syntax instead.

None of these metasyntax extensions is available if an initial `***=` director has specified that the user's input be treated as a literal string rather than as an RE.

9.7.3.5. Regular Expression Matching Rules

In the event that an RE could match more than one substring of a given string, the RE matches the one starting earliest in the string. If the RE could match more than one substring starting at that point, either the longest possible match or the shortest possible match will be taken, depending on whether the RE is *greedy* or *non-greedy*.

Whether an RE is greedy or not is determined by the following rules:

- Most atoms, and all constraints, have no greediness attribute (because they cannot match variable amounts of text anyway).
- Adding parentheses around an RE does not change its greediness.
- A quantified atom with a fixed-repetition quantifier (`{m}` or `{m}?`) has the same greediness (possibly none) as the atom itself.
- A quantified atom with other normal quantifiers (including `{m,n}` with *m* equal to *n*) is greedy (prefers longest match).
- A quantified atom with a non-greedy quantifier (including `{m,n}?` with *m* equal to *n*) is non-greedy (prefers shortest match).
- A branch — that is, an RE that has no top-level `|` operator — has the same greediness as the first quantified atom in it that has a greediness attribute.
- An RE consisting of two or more branches connected by the `|` operator is always greedy.

The above rules associate greediness attributes not only with individual quantified atoms, but with branches and entire REs that contain quantified atoms. What that means is that the matching is done in such a way that the branch, or whole RE, matches the longest or shortest possible substring *as a whole*. Once the length of the entire match is determined, the part of it that matches any particular subexpression is determined on the basis of the greediness attribute of that subexpression, with subexpressions starting earlier in the RE taking priority over ones starting later.

An example of what this means:

```
SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
Result: 123
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
Result: 1
```

In the first case, the RE as a whole is greedy because `Y*` is greedy. It can match beginning at the `Y`, and it matches the longest possible string starting there, i.e., `Y123`. The output is the parenthesized part of that, or `123`. In the second case, the RE as a whole is non-greedy because `Y*?` is non-greedy. It can match beginning at the `Y`, and it matches the shortest possible string starting there, i.e., `Y1`. The subexpression `[0-9]{1,3}` is greedy but it cannot change the decision as to the overall match length; so it is forced to match just `1`.

In short, when an RE contains both greedy and non-greedy subexpressions, the total match length is either as long as possible or as short as possible, according to the attribute assigned to the whole RE. The attributes assigned to the subexpressions only affect how much of that match they are allowed to "eat" relative to each other.

The quantifiers `{1,1}` and `{1,1}?` can be used to force greediness or non-greediness, respectively, on a subexpression or a whole RE. This is useful when you need the whole RE to have a greediness attribute different from what's deduced from its elements. As an example, suppose that we are trying to separate a string containing some digits into the digits and the parts before and after them. We might try to do that like this:

```
SELECT regexp_matches('abc01234xyz', '(.*)(\d+)(.*)');
Result: {abc0123,4,xyz}
```

That didn't work: the first `.*` is greedy so it "eats" as much as it can, leaving the `\d+` to match at the last possible place, the last digit. We might try to fix that by making it non-greedy:

```
SELECT regexp_matches('abc01234xyz', '(.*?)(\d+)(.*)');
Result: {abc,0,""}
```

That didn't work either, because now the RE as a whole is non-greedy and so it ends the overall match as soon as possible. We can get what we want by forcing the RE as a whole to be greedy:

```
SELECT regexp_matches('abc01234xyz', '(?:(.*?)(\d+)(.*)){1,1}');
Result: {abc,01234,xyz}
```

Controlling the RE's overall greediness separately from its components' greediness allows great flexibility in handling variable-length patterns.

When deciding what is a longer or shorter match, match lengths are measured in characters, not collating elements. An empty string is considered longer than no match at all. For example: `bb*` matches the three middle characters of `abbbc`; `(week|wee)(night|knights)` matches all ten characters of `weeknights`; when `(.*).*` is matched against `abc` the parenthesized subexpression matches all three characters; and when `(a*)*` is matched against `bc` both the whole RE and the parenthesized subexpression match an empty string.

If case-independent matching is specified, the effect is much as if all case distinctions had vanished from the alphabet. When an alphabetic that exists in multiple cases appears as an ordinary character outside a bracket expression, it is effectively transformed into a bracket expression containing both cases, e.g., `x` becomes `[xX]`. When it appears inside a bracket expression, all case counterparts of it are added to the bracket expression, e.g., `[x]` becomes `[xX]` and `[^x]` becomes `[^xX]`.

If newline-sensitive matching is specified, `.` and bracket expressions using `^` will never match the newline character (so that matches will never cross newlines unless the RE explicitly arranges it) and `^` and `$` will match the empty string after and before a newline respectively, in addition to matching at beginning and end of string respectively. But the ARE escapes `\A` and `\Z` continue to match beginning or end of string *only*.

If partial newline-sensitive matching is specified, this affects `.` and bracket expressions as with newline-sensitive matching, but not `^` and `$`.

If inverse partial newline-sensitive matching is specified, this affects `^` and `$` as with newline-sensitive matching, but not `.` and bracket expressions. This isn't very useful but is provided for symmetry.

9.7.3.6. Limits and Compatibility

No particular limit is imposed on the length of REs in this implementation. However, programs intended to be highly portable should not employ REs longer than 256 bytes, as a POSIX-compliant implementation can refuse to accept such REs.

The only feature of AREs that is actually incompatible with POSIX EREs is that `\` does not lose its special significance inside bracket expressions. All other ARE features use syntax which is illegal or has undefined or unspecified effects in POSIX EREs; the `***` syntax of directors likewise is outside the POSIX syntax for both BREs and EREs.

Many of the ARE extensions are borrowed from Perl, but some have been changed to clean them up, and a few Perl extensions are not present. Incompatibilities of note include `\b`, `\B`, the lack of special treatment for a trailing newline, the addition of complemented bracket expressions to the things affected by newline-sensitive matching, the restrictions on parentheses and back references in lookahead constraints, and the longest/shortest-match (rather than first-match) matching semantics.

Two significant incompatibilities exist between AREs and the ERE syntax recognized by pre-7.4 releases of PostgreSQL:

- In AREs, `\` followed by an alphanumeric character is either an escape or an error, while in previous releases, it was just another way of writing the alphanumeric. This should not be much of a problem because there was no reason to write such a sequence in earlier releases.
- In AREs, `\` remains a special character within `[]`, so a literal `\` within a bracket expression must be written `\\`.

9.7.3.7. Basic Regular Expressions

BREs differ from EREs in several respects. In BREs, `|`, `+`, and `?` are ordinary characters and there is no equivalent for their functionality. The delimiters for bounds are `\{` and `\}`, with `{` and `}` by themselves ordinary characters. The parentheses for nested subexpressions are `\(` and `\)`, with `(` and `)` by themselves ordinary characters. `^` is an ordinary character except at the beginning of the RE or the beginning of a parenthesized subexpression, `$` is an ordinary character except at the end of the RE or the end of a parenthesized subexpression, and `*` is an ordinary character if it appears at the beginning of the RE or the beginning of a parenthesized subexpression (after a possible leading `^`). Finally, single-digit back references are available, and `\<` and `\>` are synonyms for `[[:<:]]` and `[[:>:]]` respectively; no other escapes are available in BREs.

9.8. Data Type Formatting Functions

The PostgreSQL formatting functions provide a powerful set of tools for converting various data types (date/time, integer, floating point, numeric) to formatted strings and for converting from formatted strings to specific data types. Table 9-21 lists them. These functions all follow a common calling convention: the first argument is the value to be formatted and the second argument is a template that defines the output or input format.

A single-argument `to_timestamp` function is also available; it accepts a `double precision` argument and converts from Unix epoch (seconds since 1970-01-01 00:00:00+00) to `timestamp with time zone`. (`Integer` Unix epochs are implicitly cast to `double precision`.)

Table 9-21. Formatting Functions

Function	Return Type	Description	Example
`to_char(timestamp, text)`	`text`	convert time stamp to string	`to_char(current_timestamp, 'HH12:MI:SS')`
`to_char(interval, text)`	`text`	convert interval to string	`to_char(interval '15h 2m 12s', 'HH24:MI:SS')`
`to_char(int, text)`	`text`	convert integer to string	`to_char(125, '999')`
`to_char(double precision, text)`	`text`	convert real/double precision to string	`to_char(125.8::real, '999D9')`
`to_char(numeric, text)`	`text`	convert numeric to string	`to_char(-125.8, '999D99S')`
`to_date(text, text)`	`date`	convert string to date	`to_date('05 Dec 2000', 'DD Mon YYYY')`
`to_number(text, text)`	`numeric`	convert string to numeric	`to_number('12,454.8-', '99G999D9S')`
`to_timestamp(text, text)`	`timestamp with time zone`	convert string to time stamp	`to_timestamp('05 Dec 2000', 'DD Mon YYYY')`
`to_timestamp(double precision)`	`timestamp with time zone`	convert Unix epoch to time stamp	`to_timestamp(1284352323)`

In a `to_char` output template string, there are certain patterns that are recognized and replaced with appropriately-formatted data based on the given value. Any text that is not a template pattern is simply copied verbatim. Similarly, in an input template string (for the other functions), template patterns identify the values to be supplied by the input data string.

Table 9-22 shows the template patterns available for formatting date and time values.

Table 9-22. Template Patterns for Date/Time Formatting

Pattern	Description
`HH`	hour of day (01-12)
`HH12`	hour of day (01-12)
`HH24`	hour of day (00-23)
`MI`	minute (00-59)
`SS`	second (00-59)
`MS`	millisecond (000-999)
`US`	microsecond (000000-999999)
`SSSS`	seconds past midnight (0-86399)

Pattern	Description
AM, am, PM or pm	meridiem indicator (without periods)
A.M., a.m., P.M. or p.m.	meridiem indicator (with periods)
Y,YYY	year (4 or more digits) with comma
YYYY	year (4 or more digits)
YYY	last 3 digits of year
YY	last 2 digits of year
Y	last digit of year
IYYY	ISO 8601 week-numbering year (4 or more digits)
IYY	last 3 digits of ISO 8601 week-numbering year
IY	last 2 digits of ISO 8601 week-numbering year
I	last digit of ISO 8601 week-numbering year
BC, bc, AD or ad	era indicator (without periods)
B.C., b.c., A.D. or a.d.	era indicator (with periods)
MONTH	full upper case month name (blank-padded to 9 chars)
Month	full capitalized month name (blank-padded to 9 chars)
month	full lower case month name (blank-padded to 9 chars)
MON	abbreviated upper case month name (3 chars in English, localized lengths vary)
Mon	abbreviated capitalized month name (3 chars in English, localized lengths vary)
mon	abbreviated lower case month name (3 chars in English, localized lengths vary)
MM	month number (01-12)
DAY	full upper case day name (blank-padded to 9 chars)
Day	full capitalized day name (blank-padded to 9 chars)
day	full lower case day name (blank-padded to 9 chars)
DY	abbreviated upper case day name (3 chars in English, localized lengths vary)
Dy	abbreviated capitalized day name (3 chars in English, localized lengths vary)
dy	abbreviated lower case day name (3 chars in English, localized lengths vary)
DDD	day of year (001-366)
IDDD	day of ISO 8601 week-numbering year (001-371; day 1 of the year is Monday of the first ISO week)

Pattern	Description
DD	day of month (01-31)
D	day of the week, Sunday (1) to Saturday (7)
ID	ISO 8601 day of the week, Monday (1) to Sunday (7)
W	week of month (1-5) (the first week starts on the first day of the month)
WW	week number of year (1-53) (the first week starts on the first day of the year)
IW	week number of ISO 8601 week-numbering year (01-53; the first Thursday of the year is in week 1)
CC	century (2 digits) (the twenty-first century starts on 2001-01-01)
J	Julian Day (integer days since November 24, 4714 BC at midnight UTC)
Q	quarter (ignored by to_date and to_timestamp)
RM	month in upper case Roman numerals (I-XII; I=January)
rm	month in lower case Roman numerals (i-xii; i=January)
TZ	upper case time-zone abbreviation (only supported in to_char)
tz	lower case time-zone abbreviation (only supported in to_char)
OF	time-zone offset from UTC (only supported in to_char)

Modifiers can be applied to any template pattern to alter its behavior. For example, FMMonth is the Month pattern with the FM modifier. Table 9-23 shows the modifier patterns for date/time formatting.

Table 9-23. Template Pattern Modifiers for Date/Time Formatting

Modifier	Description	Example
FM prefix	fill mode (suppress leading zeroes and padding blanks)	FMMonth
TH suffix	upper case ordinal number suffix	DDTH, e.g., 12TH
th suffix	lower case ordinal number suffix	DDth, e.g., 12th
FX prefix	fixed format global option (see usage notes)	FX Month DD Day
TM prefix	translation mode (print localized day and month names based on lc_time)	TMMonth

Modifier	Description	Example
SP suffix	spell mode (not implemented)	DDSP

Usage notes for date/time formatting:

- FM suppresses leading zeroes and trailing blanks that would otherwise be added to make the output of a pattern be fixed-width. In PostgreSQL, FM modifies only the next specification, while in Oracle FM affects all subsequent specifications, and repeated FM modifiers toggle fill mode on and off.
- TM does not include trailing blanks. to_timestamp and to_date ignore the TM modifier.
- to_timestamp and to_date skip multiple blank spaces in the input string unless the FX option is used. For example, to_timestamp('2000 JUN', 'YYYY MON') works, but to_timestamp('2000 JUN', 'FXYYYY MON') returns an error because to_timestamp expects one space only. FX must be specified as the first item in the template.
- to_timestamp and to_date exist to handle input formats that cannot be converted by simple casting. These functions interpret input liberally, with minimal error checking. While they produce valid output, the conversion can yield unexpected results. For example, input to these functions is not restricted by normal ranges, thus to_date('20096040','YYYYMMDD') returns 2014-01-17 rather than causing an error. Casting does not have this behavior.
- Ordinary text is allowed in to_char templates and will be output literally. You can put a substring in double quotes to force it to be interpreted as literal text even if it contains pattern key words. For example, in '"Hello Year "YYYY', the YYYY will be replaced by the year data, but the single Y in Year will not be. In to_date, to_number, and to_timestamp, double-quoted strings skip the number of input characters contained in the string, e.g., "XX" skips two input characters.
- If you want to have a double quote in the output you must precede it with a backslash, for example '\"YYYY Month\"'.
- If the year format specification is less than four digits, e.g., YYY, and the supplied year is less than four digits, the year will be adjusted to be nearest to the year 2020, e.g., 95 becomes 1995.
- In to_timestamp and to_date, negative years are treated as signifying BC. If you write both a negative year and an explicit BC field, you get AD again. An input of year zero is treated as 1 BC.
- In to_timestamp and to_date, the YYYY conversion has a restriction when processing years with more than 4 digits. You must use some non-digit character or template after YYYY, otherwise the year is always interpreted as 4 digits. For example (with the year 20000): to_date('200001131', 'YYYYMMDD') will be interpreted as a 4-digit year; instead use a non-digit separator after the year, like to_date('20000-1131', 'YYYY-MMDD') or to_date('20000Nov31', 'YYYYMonDD').
- In conversions from string to timestamp or date, the CC (century) field is ignored if there is a YYY, YYYY or Y,YYY field. If CC is used with YY or Y then the year is computed as the year in the specified century. If the century is specified but the year is not, the first year of the century is assumed.
- An ISO 8601 week-numbering date (as distinct from a Gregorian date) can be specified to to_timestamp and to_date in one of two ways:
  - Year, week number, and weekday: for example to_date('2006-42-4', 'IYYY-IW-ID') returns the date 2006-10-19. If you omit the weekday it is assumed to be 1 (Monday).

- Year and day of year: for example `to_date('2006-291', 'IYYY-IDDD')` also returns `2006-10-19`.

Attempting to enter a date using a mixture of ISO 8601 week-numbering fields and Gregorian date fields is nonsensical, and will cause an error. In the context of an ISO 8601 week-numbering year, the concept of a "month" or "day of month" has no meaning. In the context of a Gregorian year, the ISO week has no meaning.

Caution

While `to_date` will reject a mixture of Gregorian and ISO week-numbering date fields, `to_char` will not, since output format specifications like `YYYY-MM-DD (IYYY-IDDD)` can be useful. But avoid writing something like `IYYY-MM-DD`; that would yield surprising results near the start of the year. (See Section 9.9.1 for more information.)

- In a conversion from string to `timestamp`, millisecond (`MS`) or microsecond (`US`) values are used as the seconds digits after the decimal point. For example `to_timestamp('12:3', 'SS:MS')` is not 3 milliseconds, but 300, because the conversion counts it as 12 + 0.3 seconds. This means for the format `SS:MS`, the input values `12:3`, `12:30`, and `12:300` specify the same number of milliseconds. To get three milliseconds, one must use `12:003`, which the conversion counts as 12 + 0.003 = 12.003 seconds.

  Here is a more complex example: `to_timestamp('15:12:02.020.001230', 'HH24:MI:SS.MS.US')` is 15 hours, 12 minutes, and 2 seconds + 20 milliseconds + 1230 microseconds = 2.021230 seconds.

- `to_char(..., 'ID')`'s day of the week numbering matches the `extract(isodow from ...)` function, but `to_char(..., 'D')`'s does not match `extract(dow from ...)`'s day numbering.

- `to_char(interval)` formats `HH` and `HH12` as shown on a 12-hour clock, i.e., zero hours and 36 hours output as `12`, while `HH24` outputs the full hour value, which can exceed 23 for intervals.

Table 9-24 shows the template patterns available for formatting numeric values.

Table 9-24. Template Patterns for Numeric Formatting

Pattern	Description
`9`	digit position (can be dropped if insignificant)
`0`	digit position (will not be dropped, even if insignificant)
`.` (period)	decimal point
`,` (comma)	group (thousands) separator
`PR`	negative value in angle brackets
`S`	sign anchored to number (uses locale)
`L`	currency symbol (uses locale)
`D`	decimal point (uses locale)

Pattern	Description
G	group separator (uses locale)
MI	minus sign in specified position (if number < 0)
PL	plus sign in specified position (if number > 0)
SG	plus/minus sign in specified position
RN	Roman numeral (input between 1 and 3999)
TH or th	ordinal number suffix
V	shift specified number of digits (see notes)
EEEE	exponent for scientific notation

Usage notes for numeric formatting:

- 0 specifies a digit position that will always be printed, even if it contains a leading/trailing zero. 9 also specifies a digit position, but if it is a leading zero then it will be replaced by a space, while if it is a trailing zero and fill mode is specified then it will be deleted. (For to_number(), these two pattern characters are equivalent.)
- The pattern characters S, L, D, and G represent the sign, currency symbol, decimal point, and thousands separator characters defined by the current locale (see lc_monetary and lc_numeric). The pattern characters period and comma represent those exact characters, with the meanings of decimal point and thousands separator, regardless of locale.
- If no explicit provision is made for a sign in to_char()'s pattern, one column will be reserved for the sign, and it will be anchored to (appear just left of) the number. If S appears just left of some 9's, it will likewise be anchored to the number.
- A sign formatted using SG, PL, or MI is not anchored to the number; for example, to_char(-12, 'MI9999') produces '- 12' but to_char(-12, 'S9999') produces ' -12'. (The Oracle implementation does not allow the use of MI before 9, but rather requires that 9 precede MI.)
- TH does not convert values less than zero and does not convert fractional numbers.
- PL, SG, and TH are PostgreSQL extensions.
- V effectively multiplies the input values by 10^*n*, where *n* is the number of digits following V. to_char does not support the use of V combined with a decimal point (e.g., 99.9V99 is not allowed).
- EEEE (scientific notation) cannot be used in combination with any of the other formatting patterns or modifiers other than digit and decimal point patterns, and must be at the end of the format string (e.g., 9.99EEEE is a valid pattern).

Certain modifiers can be applied to any template pattern to alter its behavior. For example, FM99.99 is the 99.99 pattern with the FM modifier. Table 9-25 shows the modifier patterns for numeric formatting.

Table 9-25. Template Pattern Modifiers for Numeric Formatting

Modifier	Description	Example

Modifier	Description	Example
FM prefix	fill mode (suppress trailing zeroes and padding blanks)	FM99.99
TH suffix	upper case ordinal number suffix	999TH
th suffix	lower case ordinal number suffix	999th

Table 9-26 shows some examples of the use of the to_char function.

Table 9-26. to_char Examples

Expression	Result
to_char(current_timestamp, 'Day, DD HH12:MI:SS')	'Tuesday , 06 05:39:18'
to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS')	'Tuesday, 6 05:39:18'
to_char(-0.1, '99.99')	' -.10'
to_char(-0.1, 'FM9.99')	'-.1'
to_char(-0.1, 'FM90.99')	'-0.1'
to_char(0.1, '0.9')	' 0.1'
to_char(12, '9990999.9')	' 0012.0'
to_char(12, 'FM9990999.9')	'0012.'
to_char(485, '999')	' 485'
to_char(-485, '999')	'-485'
to_char(485, '9 9 9')	' 4 8 5'
to_char(1485, '9,999')	' 1,485'
to_char(1485, '9G999')	' 1 485'
to_char(148.5, '999.999')	' 148.500'
to_char(148.5, 'FM999.999')	'148.5'
to_char(148.5, 'FM999.990')	'148.500'
to_char(148.5, '999D999')	' 148,500'
to_char(3148.5, '9G999D999')	' 3 148,500'
to_char(-485, '999S')	'485-'
to_char(-485, '999MI')	'485-'
to_char(485, '999MI')	'485 '
to_char(485, 'FM999MI')	'485'
to_char(485, 'PL999')	'+485'
to_char(485, 'SG999')	'+485'
to_char(-485, 'SG999')	'-485'
to_char(-485, '9SG99')	'4-85'
to_char(-485, '999PR')	'<485>'
to_char(485, 'L999')	'DM 485'

Expression	Result
to_char(485, 'RN')	' CDLXXXV'
to_char(485, 'FMRN')	'CDLXXXV'
to_char(5.2, 'FMRN')	'V'
to_char(482, '999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'
to_char(485.8, '"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'
to_char(12, '99V999')	' 12000'
to_char(12.4, '99V999')	' 12400'
to_char(12.45, '99V9')	' 125'
to_char(0.0004859, '9.99EEEE')	' 4.86e-04'

9.9. Date/Time Functions and Operators

Table 9-28 shows the available functions for date/time value processing, with details appearing in the following subsections. Table 9-27 illustrates the behaviors of the basic arithmetic operators (+, *, etc.). For formatting functions, refer to Section 9.8. You should be familiar with the background information on date/time data types from Section 8.5.

All the functions and operators described below that take time or timestamp inputs actually come in two variants: one that takes time with time zone or timestamp with time zone, and one that takes time without time zone or timestamp without time zone. For brevity, these variants are not shown separately. Also, the + and * operators come in commutative pairs (for example both date + integer and integer + date); we show only one of each such pair.

Table 9-27. Date/Time Operators

Operator	Example	Result
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00:00'
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00:00'
+	time '01:00' + interval '3 hours'	time '04:00:00'

Operator	Example	Result
-	- interval '23 hours'	interval '-23:00:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3' (days)
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00:00'
-	time '05:00' - time '03:00'	interval '02:00:00'
-	time '05:00' - interval '2 hours'	time '03:00:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00:00'
-	interval '1 day' - interval '1 hour'	interval '1 day -01:00:00'
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00:00'
*	900 * interval '1 second'	interval '00:15:00'
*	21 * interval '1 day'	interval '21 days'
*	double precision '3.5' * interval '1 hour'	interval '03:30:00'
/	interval '1 hour' / double precision '1.5'	interval '00:40:00'

Table 9-28. Date/Time Functions

Function	Return Type	Description	Example	Result
age(timestamp, timestamp)	interval	Subtract arguments, producing a "symbolic" result that uses years and months, rather than just days	age(timestamp '2001-04-10', timestamp '1957-06-13')	43 years 9 mons 27 days
age(timestamp)	interval	Subtract from current_date (at midnight)	age(timestamp '1957-06-13')	43 years 8 mons 3 days

Function	Return Type	Description	Example	Result
clock_timestamp()	timestamp with time zone	Current date and time (changes during statement execution); see Section 9.9.4		
current_date	date	Current date; see Section 9.9.4		
current_time	time with time zone	Current time of day; see Section 9.9.4		
current_timestamp	timestamp with time zone	Current date and time (start of current transaction); see Section 9.9.4		
date_part(text, timestamp)	double precision	Get subfield (equivalent to extract); see Section 9.9.1	date_part('hour', timestamp '2001-02-16 20:38:40')	20
date_part(text, interval)	double precision	Get subfield (equivalent to extract); see Section 9.9.1	date_part('month', interval '2 years 3 months')	3
date_trunc(text, timestamp)	timestamp	Truncate to specified precision; see also Section 9.9.2	date_trunc('hour', timestamp '2001-02-16 20:38:40')	2001-02-16 20:00:00
date_trunc(text, interval)	interval	Truncate to specified precision; see also Section 9.9.2	date_trunc('hour', interval '2 days 3 hours 40 minutes')	2 days 03:00:00
extract(field from timestamp)	double precision	Get subfield; see Section 9.9.1	extract(hour from timestamp '2001-02-16 20:38:40')	20
extract(field from interval)	double precision	Get subfield; see Section 9.9.1	extract(month from interval '2 years 3 months')	3
isfinite(date)	boolean	Test for finite date (not +/-infinity)	isfinite(date '2001-02-16')	true
isfinite(timestamp)	boolean	Test for finite time stamp (not +/-infinity)	isfinite(timestamp '2001-02-16 21:28:30')	true

Function	Return Type	Description	Example	Result
isfinite(interval)	boolean	Test for finite interval	isfinite(interval '4 hours')	true
justify_days(interval)	interval	Adjust interval so 30-day time periods are represented as months	justify_days(interval '35 days')	1 mon 5 days
justify_hours(interval)	interval	Adjust interval so 24-hour time periods are represented as days	justify_hours(interval '27 hours')	1 day 03:00:00
justify_interval(interval)	interval	Adjust interval using justify_days and justify_hours, with additional sign adjustments	justify_interval(interval '1 mon -1 hour')	29 days 23:00:00
localtime	time	Current time of day; see Section 9.9.4		
localtimestamp	timestamp	Current date and time (start of current transaction); see Section 9.9.4		
make_date(year int, month int, day int)	date	Create date from year, month and day fields	make_date(2013, 7, 15)	2013-07-15
make_interval(years int DEFAULT 0, months int DEFAULT 0, weeks int DEFAULT 0, days int DEFAULT 0, hours int DEFAULT 0, mins int DEFAULT 0, secs double precision DEFAULT 0.0)	interval	Create interval from years, months, weeks, days, hours, minutes and seconds fields	make_interval(days => 10)	10 days

Function	Return Type	Description	Example	Result
make_time(hour int, min int, sec double precision)	time	Create time from hour, minute and seconds fields	make_time(8, 15, 23.5)	08:15:23.5
make_timestamp(year int, month int, day int, hour int, min int, sec double precision)	timestamp	Create timestamp from year, month, day, hour, minute and seconds fields	make_timestamp(2013, 7, 15, 8, 15, 23.5)	2013-07-15 08:15:23.5
make_timestamptz(year int, month int, day int, hour int, min int, sec double precision, [timezone text])	timestamp with time zone	Create timestamp with time zone from year, month, day, hour, minute and seconds fields. When timezone is not specified, then current time zone is used.	make_timestamptz(2013, 7, 15, 8, 15, 23.5)	2013-07-15 08:15:23.5+01
now()	timestamp with time zone	Current date and time (start of current transaction); see Section 9.9.4		
statement_timestamp()	timestamp with time zone	Current date and time (start of current statement); see Section 9.9.4		
timeofday()	text	Current date and time (like clock_timestamp, but as a text string); see Section 9.9.4		
transaction_timestamp()	timestamp with time zone	Current date and time (start of current transaction); see Section 9.9.4		

In addition to these functions, the SQL OVERLAPS operator is supported:

```
(start1, end1) OVERLAPS (start2, end2)
(start1, length1) OVERLAPS (start2, length2)
```

This expression yields true when two time periods (defined by their endpoints) overlap, false when they do not overlap. The endpoints can be specified as pairs of dates, times, or time stamps; or as a date, time, or time stamp followed by an interval. When a pair of values is provided, either the start or the end can be written first; `OVERLAPS` automatically takes the earlier value of the pair as the start. Each time period is considered to represent the half-open interval *start* <= *time* < *end*, unless *start* and *end* are equal in which case it represents that single time instant. This means for instance that two time periods with only an endpoint in common do not overlap.

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Result: true
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Result: false
SELECT (DATE '2001-10-29', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Result: false
SELECT (DATE '2001-10-30', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Result: true
```

When adding an `interval` value to (or subtracting an `interval` value from) a `timestamp with time zone` value, the days component advances or decrements the date of the `timestamp with time zone` by the indicated number of days, keeping the time of day the same. Across daylight saving time changes (when the session time zone is set to a time zone that recognizes DST), this means `interval '1 day'` does not necessarily equal `interval '24 hours'`. For example, with the session time zone set to `America/Denver`:

```
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval '1 day';
Result: 2005-04-03 12:00:00-06
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval '24 hours';
Result: 2005-04-03 13:00:00-06
```

This happens because an hour was skipped due to a change in daylight saving time at `2005-04-03 02:00:00` in time zone `America/Denver`.

Note there can be ambiguity in the `months` field returned by `age` because different months have different numbers of days. PostgreSQL's approach uses the month from the earlier of the two dates when calculating partial months. For example, `age('2004-06-01', '2004-04-30')` uses April to yield `1 mon 1 day`, while using May would yield `1 mon 2 days` because May has 31 days, while April has only 30.

Subtraction of dates and timestamps can also be complex. One conceptually simple way to perform subtraction is to convert each value to a number of seconds using `EXTRACT(EPOCH FROM ...)`, then subtract the results; this produces the number of *seconds* between the two values. This will adjust for the number of days in each month, timezone changes, and daylight saving time adjustments. Subtraction of date or timestamp values with the "-" operator returns the number of days (24-hours) and hours/minutes/seconds between the values, making the same adjustments. The `age` function returns years, months, days, and hours/minutes/seconds, performing field-by-field subtraction and then adjusting for negative field values. The following queries illustrate the differences in these approaches. The sample results were produced with `timezone = 'US/Eastern'`; there is a daylight saving time change between the two dates used:

```
SELECT EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
```

```
       EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00');
Result: 10537200
SELECT (EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
        EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00'))
        / 60 / 60 / 24;
Result: 121.958333333333
SELECT timestamptz '2013-07-01 12:00:00' - timestamptz '2013-03-01 12:00:00';
Result: 121 days 23:00:00
SELECT age(timestamptz '2013-07-01 12:00:00', timestamptz '2013-03-01 12:00:00');
Result: 4 mons
```

9.9.1. EXTRACT, date_part

EXTRACT(*field* FROM *source*)

The `extract` function retrieves subfields such as year or hour from date/time values. *source* must be a value expression of type `timestamp`, `time`, or `interval`. (Expressions of type `date` are cast to `timestamp` and can therefore be used as well.) *field* is an identifier or string that selects what field to extract from the source value. The `extract` function returns values of type `double precision`. The following are valid field names:

`century`

The century

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
Result: 20
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 21
```

The first century starts at 0001-01-01 00:00:00 AD, although they did not know it at the time. This definition applies to all Gregorian calendar countries. There is no century number 0, you go from -1 century to 1 century. If you disagree with this, please write your complaint to: Pope, Cathedral Saint-Peter of Roma, Vatican.

`day`

For `timestamp` values, the day (of the month) field (1 - 31) ; for `interval` values, the number of days

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 16

SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
Result: 40
```

`decade`

The year field divided by 10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 200
```

dow

The day of the week as Sunday (0) to Saturday (6)

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 5
```

Note that extract's day of the week numbering differs from that of the to_char(..., 'D') function.

doy

The day of the year (1 - 365/366)

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 47
```

epoch

For timestamp with time zone values, the number of seconds since 1970-01-01 00:00:00 UTC (can be negative); for date and timestamp values, the number of seconds since 1970-01-01 00:00:00 local time; for interval values, the total number of seconds in the interval

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40.12-08');
Result: 982384720.12

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Result: 442800
```

Here is how you can convert an epoch value back to a time stamp:

```
SELECT TIMESTAMP WITH TIME ZONE 'epoch' + 982384720.12 * INTERVAL '1 second';
```

(The to_timestamp function encapsulates the above conversion.)

hour

The hour field (0 - 23)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 20
```

isodow

The day of the week as Monday (1) to Sunday (7)

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
Result: 7
```

This is identical to dow except for Sunday. This matches the ISO 8601 day of the week numbering.

isoyear

The ISO 8601 week-numbering year that the date falls in (not applicable to intervals)

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
Result: 2005
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
Result: 2006
```

Each ISO 8601 week-numbering year begins with the Monday of the week containing the 4th of January, so in early January or late December the ISO year may be different from the Gregorian year. See the week field for more information.

This field is not available in PostgreSQL releases prior to 8.3.

`microseconds`

The seconds field, including fractional parts, multiplied by 1 000 000; note that this includes full seconds

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Result: 28500000
```

`millennium`

The millennium

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 3
```

Years in the 1900s are in the second millennium. The third millennium started January 1, 2001.

`milliseconds`

The seconds field, including fractional parts, multiplied by 1000. Note that this includes full seconds.

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
Result: 28500
```

`minute`

The minutes field (0 - 59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 38
```

`month`

For `timestamp` values, the number of the month within the year (1 - 12) ; for `interval` values, the number of months, modulo 12 (0 - 11)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
Result: 3

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
Result: 1
```

`quarter`

The quarter of the year (1 - 4) that the date is in

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 1
```

`second`

The seconds field, including fractional parts (0 - 59[1])

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
```

60 if leap seconds are implemented by the operating system

Result: 28.5

timezone

The time zone offset from UTC, measured in seconds. Positive values correspond to time zones east of UTC, negative values to zones west of UTC. (Technically, PostgreSQL uses UT1 because leap seconds are not handled.)

timezone_hour

The hour component of the time zone offset

timezone_minute

The minute component of the time zone offset

week

The number of the ISO 8601 week-numbering week of the year. By definition, ISO weeks start on Mondays and the first week of a year contains January 4 of that year. In other words, the first Thursday of a year is in week 1 of that year.

In the ISO week-numbering system, it is possible for early-January dates to be part of the 52nd or 53rd week of the previous year, and for late-December dates to be part of the first week of the next year. For example, 2005-01-01 is part of the 53rd week of year 2004, and 2006-01-01 is part of the 52nd week of year 2005, while 2012-12-31 is part of the first week of 2013. It's recommended to use the isoyear field together with week to get consistent results.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 7
```

year

The year field. Keep in mind there is no 0 AD, so subtracting BC years from AD years should be done with care.

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2001
```

The extract function is primarily intended for computational processing. For formatting date/time values for display, see Section 9.8.

The date_part function is modeled on the traditional Ingres equivalent to the SQL-standard function extract:

```
date_part('field', source)
```

Note that here the *field* parameter needs to be a string value, not a name. The valid field names for date_part are the same as for extract.

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Result: 16

SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Result: 4
```

9.9.2. date_trunc

The function date_trunc is conceptually similar to the trunc function for numbers.

```
date_trunc('field', source)
```

source is a value expression of type timestamp or interval. (Values of type date and time are cast automatically to timestamp or interval, respectively.) *field* selects to which precision to truncate the input value. The return value is of type timestamp or interval with all fields that are less significant than the selected one set to zero (or one, for day and month).

Valid values for *field* are:

```
microseconds
milliseconds
second
minute
hour
day
week
month
quarter
year
decade
century
millennium
```

Examples:

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-02-16 20:00:00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-01-01 00:00:00
```

9.9.3. AT TIME ZONE

The AT TIME ZONE converts time stamp *without time zone* to/from time stamp *with time zone*, and *time* values to different time zones. Table 9-29 shows its variants.

Table 9-29. AT TIME ZONE Variants

Expression	Return Type	Description
timestamp without time zone AT TIME ZONE *zone*	timestamp with time zone	Treat given time stamp *without time zone* as located in the specified time zone

Expression	Return Type	Description
`timestamp with time zone AT TIME ZONE` *zone*	`timestamp without time zone`	Convert given time stamp *with time zone* to the new time zone, with no time zone designation
`time with time zone AT TIME ZONE` *zone*	`time with time zone`	Convert given time *with time zone* to the new time zone

In these expressions, the desired time zone *zone* can be specified either as a text string (e.g., `'America/Los_Angeles'`) or as an interval (e.g., `INTERVAL '-08:00'`). In the text case, a time zone name can be specified in any of the ways described in Section 8.5.3.

Examples (assuming the local time zone is `America/Los_Angeles`):

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'America/Denver';
Result: 2001-02-16 19:38:40-08

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'America/Denver';
Result: 2001-02-16 18:38:40

SELECT TIMESTAMP '2001-02-16 20:38:40-05' AT TIME ZONE 'Asia/Tokyo' AT TIME ZONE 'America/C
Result: 2001-02-16 05:38:40
```

The first example adds a time zone to a value that lacks it, and displays the value using the current `TimeZone` setting. The second example shifts the time stamp with time zone value to the specified time zone, and returns the value without a time zone. This allows storage and display of values different from the current `TimeZone` setting. The third example converts Tokyo time to Chicago time. Converting *time* values to other time zones uses the currently active time zone rules since no date is supplied.

The function `timezone(`*zone*`,` *timestamp*`)` is equivalent to the SQL-conforming construct *timestamp* `AT TIME ZONE` *zone*.

9.9.4. Current Date/Time

PostgreSQL provides a number of functions that return values related to the current date and time. These SQL-standard functions all return values based on the start time of the current transaction:

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(precision)
CURRENT_TIMESTAMP(precision)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME(precision)
LOCALTIMESTAMP(precision)
```

`CURRENT_TIME` and `CURRENT_TIMESTAMP` deliver values with time zone; `LOCALTIME` and `LOCALTIMESTAMP` deliver values without time zone.

`CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME`, and `LOCALTIMESTAMP` can optionally take a precision parameter, which causes the result to be rounded to that many fractional digits in the seconds field. Without a precision parameter, the result is given to the full available precision.

Some examples:

```
SELECT CURRENT_TIME;
Result: 14:39:53.662522-05

SELECT CURRENT_DATE;
Result: 2001-12-23

SELECT CURRENT_TIMESTAMP;
Result: 2001-12-23 14:39:53.662522-05

SELECT CURRENT_TIMESTAMP(2);
Result: 2001-12-23 14:39:53.66-05

SELECT LOCALTIMESTAMP;
Result: 2001-12-23 14:39:53.662522
```

Since these functions return the start time of the current transaction, their values do not change during the transaction. This is considered a feature: the intent is to allow a single transaction to have a consistent notion of the "current" time, so that multiple modifications within the same transaction bear the same time stamp.

> **Note:** Other database systems might advance these values more frequently.

PostgreSQL also provides functions that return the start time of the current statement, as well as the actual current time at the instant the function is called. The complete list of non-SQL-standard time functions is:

```
transaction_timestamp()
statement_timestamp()
clock_timestamp()
timeofday()
now()
```

`transaction_timestamp()` is equivalent to `CURRENT_TIMESTAMP`, but is named to clearly reflect what it returns. `statement_timestamp()` returns the start time of the current statement (more specifically, the time of receipt of the latest command message from the client). `statement_timestamp()` and `transaction_timestamp()` return the same value during the first command of a transaction, but might differ during subsequent commands. `clock_timestamp()` returns the actual current time, and therefore its value changes even within a single SQL command. `timeofday()` is a historical PostgreSQL function. Like `clock_timestamp()`, it returns the actual current time, but as a formatted `text` string rather than a `timestamp with time zone` value. `now()` is a traditional PostgreSQL equivalent to `transaction_timestamp()`.

All the date/time data types also accept the special literal value `now` to specify the current date and time (again, interpreted as the transaction start time). Thus, the following three all return the same result:

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';  -- but see tip below
```

> **Tip:** Do not use the third form when specifying a value to be evaluated later, for example in a `DEFAULT` clause for a table column. The system will convert `now` to a `timestamp` as soon as the constant is parsed, so that when the default value is needed, the time of the table creation would be used! The first two forms will not be evaluated until the default value is used, because they are function calls. Thus they will give the desired behavior of defaulting to the time of row insertion. (See also Section 8.5.1.4.)

9.9.5. Delaying Execution

The following functions are available to delay execution of the server process:

```
pg_sleep(seconds)
pg_sleep_for(interval)
pg_sleep_until(timestamp with time zone)
```

`pg_sleep` makes the current session's process sleep until *seconds* seconds have elapsed. *seconds* is a value of type `double precision`, so fractional-second delays can be specified. `pg_sleep_for` is a convenience function for larger sleep times specified as an `interval`. `pg_sleep_until` is a convenience function for when a specific wake-up time is desired. For example:

```
SELECT pg_sleep(1.5);
SELECT pg_sleep_for('5 minutes');
SELECT pg_sleep_until('tomorrow 03:00');
```

> **Note:** The effective resolution of the sleep interval is platform-specific; 0.01 seconds is a common value. The sleep delay will be at least as long as specified. It might be longer depending on factors such as server load. In particular, `pg_sleep_until` is not guaranteed to wake up exactly at the specified time, but it will not wake up any earlier.

> **Warning**
>
> Make sure that your session does not hold more locks than necessary when calling `pg_sleep` or its variants. Otherwise other sessions might have to wait for your sleeping process, slowing down the entire system.

9.10. Enum Support Functions

For enum types (described in Section 8.7), there are several functions that allow cleaner programming without hard-coding particular values of an enum type. These are listed in Table 9-30. The examples assume an enum type created as:

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue', 'purple');
```

Table 9-30. Enum Support Functions

Function	Description	Example	Example Result
`enum_first(anyenum)`	Returns the first value of the input enum type	`enum_first(null::rainbow)`	`red`
`enum_last(anyenum)`	Returns the last value of the input enum type	`enum_last(null::rainbow)`	`purple`
`enum_range(anyenum)`	Returns all values of the input enum type in an ordered array	`enum_range(null::rainbow)`	`{red,orange,yellow,green,blue,pu`
`enum_range(anyenum, anyenum)`	Returns the range between the two given enum values, as an ordered array. The values must be from the same enum type. If the first parameter is null, the result will start with the first value of the enum type. If the second parameter is null, the result will end with the last value of the enum type.	`enum_range('orange'::rainbow, 'green'::rainbow)`	`{orange,yellow,green}`
		`enum_range(NULL, 'green'::rainbow)`	`{red,orange,yellow,green}`
		`enum_range('orange'::rainbow, NULL)`	`{orange,yellow,green,blue,purple`

Notice that except for the two-argument form of `enum_range`, these functions disregard the specific value passed to them; they care only about its declared data type. Either null or a specific value of the type can be passed, with the same result. It is more common to apply these functions to a table column or function argument than to a hardwired type name as suggested by the examples.

9.11. Geometric Functions and Operators

The geometric types `point`, `box`, `lseg`, `line`, `path`, `polygon`, and `circle` have a large set of native

support functions and operators, shown in Table 9-31, Table 9-32, and Table 9-33.

> **Caution**
>
> Note that the "same as" operator, `~=`, represents the usual notion of equality for the `point`, `box`, `polygon`, and `circle` types. Some of these types also have an `=` operator, but `=` compares for equal *areas* only. The other scalar comparison operators (`<=` and so on) likewise compare areas for these types.

Table 9-31. Geometric Operators

Operator	Description	Example
`+`	Translation	`box '((0,0),(1,1))' + point '(2.0,0)'`
`-`	Translation	`box '((0,0),(1,1))' - point '(2.0,0)'`
`*`	Scaling/rotation	`box '((0,0),(1,1))' * point '(2.0,0)'`
`/`	Scaling/rotation	`box '((0,0),(2,2))' / point '(2.0,0)'`
`#`	Point or box of intersection	`box '((1,-1),(-1,1))' # box '((1,1),(-2,-2))'`
`#`	Number of points in path or polygon	`# path '((1,0),(0,1),(-1,0))'`
`@-@`	Length or circumference	`@-@ path '((0,0),(1,0))'`
`@@`	Center	`@@ circle '((0,0),10)'`
`##`	Closest point to first operand on second operand	`point '(0,0)' ## lseg '((2,0),(0,2))'`
`<->`	Distance between	`circle '((0,0),1)' <-> circle '((5,0),1)'`
`&&`	Overlaps? (One point in common makes this true.)	`box '((0,0),(1,1))' && box '((0,0),(2,2))'`
`<<`	Is strictly left of?	`circle '((0,0),1)' << circle '((5,0),1)'`
`>>`	Is strictly right of?	`circle '((5,0),1)' >> circle '((0,0),1)'`
`&<`	Does not extend to the right of?	`box '((0,0),(1,1))' &< box '((0,0),(2,2))'`
`&>`	Does not extend to the left of?	`box '((0,0),(3,3))' &> box '((0,0),(2,2))'`
`<<|`	Is strictly below?	`box '((0,0),(3,3))' <<| box '((3,4),(5,5))'`
`|>>`	Is strictly above?	`box '((3,4),(5,5))' |>> box '((0,0),(3,3))'`

Operator	Description	Example
&<|	Does not extend above?	box '((0,0),(1,1))' &<| box '((0,0),(2,2))'
|&>	Does not extend below?	box '((0,0),(3,3))' |&> box '((0,0),(2,2))'
<^	Is below (allows touching)?	circle '((0,0),1)' <^ circle '((0,5),1)'
>^	Is above (allows touching)?	circle '((0,5),1)' >^ circle '((0,0),1)'
?#	Intersects?	lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'
?-	Is horizontal?	?- lseg '((-1,0),(1,0))'
?-	Are horizontally aligned?	point '(1,0)' ?- point '(0,0)'
?|	Is vertical?	?| lseg '((-1,0),(1,0))'
?|	Are vertically aligned?	point '(0,1)' ?| point '(0,0)'
?-|	Is perpendicular?	lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'
?||	Are parallel?	lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'
@>	Contains?	circle '((0,0),2)' @> point '(1,1)'
<@	Contained in or on?	point '(1,1)' <@ circle '((0,0),2)'
~=	Same as?	polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'

Note: Before PostgreSQL 8.2, the containment operators @> and <@ were respectively called ~ and @. These names are still available, but are deprecated and will eventually be removed.

Table 9-32. Geometric Functions

Function	Return Type	Description	Example
area(*object*)	double precision	area	area(box '((0,0),(1,1))')
center(*object*)	point	center	center(box '((0,0),(1,2))')
diameter(circle)	double precision	diameter of circle	diameter(circle '((0,0),2.0)')

Function	Return Type	Description	Example
height(box)	double precision	vertical size of box	height(box '((0,0),(1,1))')
isclosed(path)	boolean	a closed path?	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	boolean	an open path?	isopen(path '[(0,0),(1,1),(2,0)]')
length(*object*)	double precision	length	length(path '((-1,0),(1,0))')
npoints(path)	int	number of points	npoints(path '[(0,0),(1,1),(2,0)]')
npoints(polygon)	int	number of points	npoints(polygon '((1,1),(0,0))')
pclose(path)	path	convert path to closed	pclose(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	convert path to open	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	radius of circle	radius(circle '((0,0),2.0)')
width(box)	double precision	horizontal size of box	width(box '((0,0),(1,1))')

Table 9-33. Geometric Type Conversion Functions

Function	Return Type	Description	Example
box(circle)	box	circle to box	box(circle '((0,0),2.0)')
box(point)	box	point to empty box	box(point '(0,0)')
box(point, point)	box	points to box	box(point '(0,0)', point '(1,1)')
box(polygon)	box	polygon to box	box(polygon '((0,0),(1,1),(2,0))')
bound_box(box, box)	box	boxes to bounding box	bound_box(box '((0,0),(1,1))', box '((3,3),(4,4))')

Function	Return Type	Description	Example
circle(box)	circle	box to circle	circle(box '((0,0),(1,1))')
circle(point, double precision)	circle	center and radius to circle	circle(point '(0,0)', 2.0)
circle(polygon)	circle	polygon to circle	circle(polygon '((0,0),(1,1),(2,0))')
line(point, point)	line	points to line	line(point '(-1,0)', point '(1,0)')
lseg(box)	lseg	box diagonal to line segment	lseg(box '((-1,0),(1,0))')
lseg(point, point)	lseg	points to line segment	lseg(point '(-1,0)', point '(1,0)')
path(polygon)	path	polygon to path	path(polygon '((0,0),(1,1),(2,0))')
point(double precision, double precision)	point	construct point	point(23.4, -44.5)
point(box)	point	center of box	point(box '((-1,0),(1,0))')
point(circle)	point	center of circle	point(circle '((0,0),2.0)')
point(lseg)	point	center of line segment	point(lseg '((-1,0),(1,0))')
point(polygon)	point	center of polygon	point(polygon '((0,0),(1,1),(2,0))')
polygon(box)	polygon	box to 4-point polygon	polygon(box '((0,0),(1,1))')
polygon(circle)	polygon	circle to 12-point polygon	polygon(circle '((0,0),2.0)')
polygon(*npts*, circle)	polygon	circle to *npts*-point polygon	polygon(12, circle '((0,0),2.0)')
polygon(path)	polygon	path to polygon	polygon(path '((0,0),(1,1),(2,0))')

It is possible to access the two component numbers of a `point` as though the point were an array with indexes 0 and 1. For example, if `t.p` is a `point` column then `SELECT p[0] FROM t` retrieves the X coordinate and `UPDATE t SET p[1] = ...` changes the Y coordinate. In the same way, a value of type `box` or `lseg` can be treated as an array of two `point` values.

The area function works for the types box, circle, and path. The area function only works on the path data type if the points in the path are non-intersecting. For example, the path '((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))'::PATH will not work; however, the following visually identical path '((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))'::PATH will work. If the concept of an intersecting versus non-intersecting path is confusing, draw both of the above paths side by side on a piece of graph paper.

9.12. Network Address Functions and Operators

Table 9-34 shows the operators available for the cidr and inet types. The operators <<, <<=, >>, >>=, and && test for subnet inclusion. They consider only the network parts of the two addresses (ignoring any host part) and determine whether one network is identical to or a subnet of the other.

Table 9-34. cidr and inet Operators

Operator	Description	Example
<	is less than	inet '192.168.1.5' < inet '192.168.1.6'
<=	is less than or equal	inet '192.168.1.5' <= inet '192.168.1.5'
=	equals	inet '192.168.1.5' = inet '192.168.1.5'
>=	is greater or equal	inet '192.168.1.5' >= inet '192.168.1.5'
>	is greater than	inet '192.168.1.5' > inet '192.168.1.4'
<>	is not equal	inet '192.168.1.5' <> inet '192.168.1.4'
<<	is contained by	inet '192.168.1.5' << inet '192.168.1/24'
<<=	is contained by or equals	inet '192.168.1/24' <<= inet '192.168.1/24'
>>	contains	inet '192.168.1/24' >> inet '192.168.1.5'
>>=	contains or equals	inet '192.168.1/24' >>= inet '192.168.1/24'
&&	contains or is contained by	inet '192.168.1/24' && inet '192.168.1.80/28'
~	bitwise NOT	~ inet '192.168.1.6'
&	bitwise AND	inet '192.168.1.6' & inet '0.0.0.255'

Operator	Description	Example
`|`	bitwise OR	`inet '192.168.1.6' | inet '0.0.0.255'`
`+`	addition	`inet '192.168.1.6' + 25`
`-`	subtraction	`inet '192.168.1.43' - 36`
`-`	subtraction	`inet '192.168.1.43' - inet '192.168.1.19'`

Table 9-35 shows the functions available for use with the `cidr` and `inet` types. The `abbrev`, `host`, and `text` functions are primarily intended to offer alternative display formats.

Table 9-35. `cidr` and `inet` Functions

Function	Return Type	Description	Example	Result
`abbrev(inet)`	`text`	abbreviated display format as text	`abbrev(inet '10.1.0.0/16')`	`10.1.0.0/16`
`abbrev(cidr)`	`text`	abbreviated display format as text	`abbrev(cidr '10.1.0.0/16')`	`10.1/16`
`broadcast(inet)`	`inet`	broadcast address for network	`broadcast('192.168.1.5/24')`	`192.168.1.255/24`
`family(inet)`	`int`	extract family of address; `4` for IPv4, `6` for IPv6	`family('::1')`	`6`
`host(inet)`	`text`	extract IP address as text	`host('192.168.1.5/24')`	`192.168.1.5`
`hostmask(inet)`	`inet`	construct host mask for network	`hostmask('192.168.23.20/30')`	`0.0.0.3`
`masklen(inet)`	`int`	extract netmask length	`masklen('192.168.1.5/24')`	`24`
`netmask(inet)`	`inet`	construct netmask for network	`netmask('192.168.1.5/24')`	`255.255.255.0`
`network(inet)`	`cidr`	extract network part of address	`network('192.168.1.5/24')`	`192.168.1.0/24`
`set_masklen(inet, int)`	`inet`	set netmask length for `inet` value	`set_masklen('192.168.1.5/24', 16)`	`192.168.1.5/16`
`set_masklen(cidr, int)`	`cidr`	set netmask length for `cidr` value	`set_masklen('192.168.0.0/24', 16)`	`192.168.0.0/16`
`text(inet)`	`text`	extract IP address and netmask length as text	`text(inet '192.168.1.5')`	`192.168.1.5/32`
`inet_same_family(inet, inet)`	`boolean`	are the addresses from the same family?	`inet_same_family('192.168.1.5/24', '::1')`	`f`

Function	Return Type	Description	Example	Result
inet_merge(inet, inet)	cidr	the smallest network which includes both of the given networks	inet_merge('192.168.1.5/24', '192.168.2.5/24')	192.168.0.0/22

Any cidr value can be cast to inet implicitly or explicitly; therefore, the functions shown above as operating on inet also work on cidr values. (Where there are separate functions for inet and cidr, it is because the behavior should be different for the two cases.) Also, it is permitted to cast an inet value to cidr. When this is done, any bits to the right of the netmask are silently zeroed to create a valid cidr value. In addition, you can cast a text value to inet or cidr using normal casting syntax: for example, inet(*expression*) or *colname*::cidr.

Table 9-36 shows the functions available for use with the macaddr type. The function trunc(macaddr) returns a MAC address with the last 3 bytes set to zero. This can be used to associate the remaining prefix with a manufacturer.

Table 9-36. macaddr Functions

Function	Return Type	Description	Example	Result
trunc(macaddr)	macaddr	set last 3 bytes to zero	trunc(macaddr '12:34:56:78:90:ab')	12:34:56:00:00:00

The macaddr type also supports the standard relational operators (>, <=, etc.) for lexicographical ordering, and the bitwise arithmetic operators (~, & and |) for NOT, AND and OR.

9.13. Text Search Functions and Operators

Table 9-37, Table 9-38 and Table 9-39 summarize the functions and operators that are provided for full text searching. See Chapter 12 for a detailed explanation of PostgreSQL's text search facility.

Table 9-37. Text Search Operators

Operator	Description	Example	Result
@@	tsvector matches tsquery ?	to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat')	t
@@@	deprecated synonym for @@	to_tsvector('fat cats ate rats') @@@ to_tsquery('cat & rat')	t

Operator	Description	Example	Result
`||`	concatenate `tsvectors`	`'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector`	`'a':1 'b':2,5 'c':3 'd':4`
`&&`	AND `tsquerys` together	`'fat | rat'::tsquery && 'cat'::tsquery`	`( 'fat' | 'rat' ) & 'cat'`
`||`	OR `tsquerys` together	`'fat | rat'::tsquery || 'cat'::tsquery`	`( 'fat' | 'rat' ) | 'cat'`
`!!`	negate a `tsquery`	`!! 'cat'::tsquery`	`!'cat'`
`@>`	`tsquery` contains another ?	`'cat'::tsquery @> 'cat & rat'::tsquery`	`f`
`<@`	`tsquery` is contained in ?	`'cat'::tsquery <@ 'cat & rat'::tsquery`	`t`

Note: The `tsquery` containment operators consider only the lexemes listed in the two queries, ignoring the combining operators.

In addition to the operators shown in the table, the ordinary B-tree comparison operators (=, <, etc) are defined for types `tsvector` and `tsquery`. These are not very useful for text searching but allow, for example, unique indexes to be built on columns of these types.

Table 9-38. Text Search Functions

Function	Return Type	Description	Example	Result
`get_current_ts_config()`	`regconfig`	get default text search configuration	`get_current_ts_config()`	`english`
`length(tsvector)`	`integer`	number of lexemes in `tsvector`	`length('fat:2,4 cat:3 rat:5A'::tsvector)`	`3`
`numnode(tsquery)`	`integer`	number of lexemes plus operators in `tsquery`	`numnode('(fat & rat) | cat'::tsquery)`	`5`
`plainto_tsquery([ config regconfig , ] query text)`	`tsquery`	produce `tsquery` ignoring punctuation	`plainto_tsquery('The Fat Rats')`	`'fat' & 'rat'`

Function	Return Type	Description	Example	Result
querytree(*query* tsquery)	text	get indexable part of a tsquery	querytree('foo & ! bar'::tsquery)	'foo'
setweight(tsvector, "char")	tsvector	assign weight to each element of tsvector	setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A')	'cat':3A 'fat':2A,4A 'rat':5A
strip(tsvector)	tsvector	remove positions and weights from tsvector	strip('fat:2,4 cat:3 rat:5A'::tsvector)	'cat' 'fat' 'rat'
to_tsquery([*config* regconfig ,] *query* text)	tsquery	normalize words and convert to tsquery	to_tsquery('english', 'The & Fat & Rats')	'fat' & 'rat'
to_tsvector([*config* regconfig ,] *document* text)	tsvector	reduce document text to tsvector	to_tsvector('english', 'The Fat Rats')	'fat':2 'rat':3
ts_headline([*config* regconfig,] *document* text, *query* tsquery [, *options* text])	text	display a query match	ts_headline('x y z', 'z'::tsquery)	x y z
ts_rank([*weights* float4[],] *vector* tsvector, *query* tsquery [, *normalization* integer])	float4	rank document for query	ts_rank(textsearch, query)	0.818
ts_rank_cd([*weights* float4[],] *vector* tsvector, *query* tsquery [, *normalization* integer])	float4	rank document for query using cover density	ts_rank_cd('{0.1, 0.2, 0.4, 1.0}', textsearch, query)	2.01317
ts_rewrite(*query* tsquery, *target* tsquery, *substitute* tsquery)	tsquery	replace target with substitute within query	ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery)	'b' & ('foo' | 'bar')

Function	Return Type	Description	Example	Result
ts_rewrite(*query* tsquery, *select* text)	tsquery	replace using targets and substitutes from a SELECT command	SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases')	'b' & ('foo' | 'bar')
tsvector_update_trigger()	trigger	trigger function for automatic tsvector column update	CREATE TRIGGER ... tsvector_update_trigger(tsvcol, 'pg_catalog.swedish', title, body)	
tsvector_update_trigger_column()	trigger	trigger function for automatic tsvector column update	CREATE TRIGGER ... tsvector_update_trigger_column(tsvcol, configcol, title, body)	

Note: All the text search functions that accept an optional regconfig argument will use the configuration specified by default_text_search_config when that argument is omitted.

The functions in Table 9-39 are listed separately because they are not usually used in everyday text searching operations. They are helpful for development and debugging of new text search configurations.

Table 9-39. Text Search Debugging Functions

Function	Return Type	Description	Example	Result
ts_debug([*config* regconfig,] *document* text, OUT *alias* text, OUT *description* text, OUT *token* text, OUT *dictionaries* regdictionary[], OUT *dictionary* regdictionary, OUT *lexemes* text[])	setof record	test a configuration	ts_debug('english', 'The Brightest supernovaes')	(asciiword,"Word, all ASCII",The,{english_stem},en ...
ts_lexize(*dict* regdictionary, *token* text)	text[]	test a dictionary	ts_lexize('english_stem', 'stars')	{star}

Function	Return Type	Description	Example	Result
ts_parse(*parser_name* text, *document* text, OUT *tokid* integer, OUT *token* text)	setof record	test a parser	ts_parse('default', 'foo - bar')	(1,foo) ...
ts_parse(*parser_oid* oid, *document* text, OUT *tokid* integer, OUT *token* text)	setof record	test a parser	ts_parse(3722, 'foo - bar')	(1,foo) ...
ts_token_type(*parser_name* text, OUT *tokid* integer, OUT *alias* text, OUT *description* text)	setof record	get token types defined by parser	ts_token_type('default')	(1,asciiword,"Word, all ASCII") ...
ts_token_type(*parser_oid* oid, OUT *tokid* integer, OUT *alias* text, OUT *description* text)	setof record	get token types defined by parser	ts_token_type(3722)	(1,asciiword,"Word, all ASCII") ...
ts_stat(*sqlquery* text, [*weights* text,] OUT *word* text, OUT *ndoc* integer, OUT *nentry* integer)	setof record	get statistics of a tsvector column	ts_stat('SELECT vector from apod')	(foo,10,15) ...

9.14. XML Functions

The functions and function-like expressions described in this section operate on values of type xml. Check Section 8.13 for information about the xml type. The function-like expressions xmlparse and xmlserialize for converting to and from type xml are not repeated here. Use of most of these functions requires the installation to have been built with configure --with-libxml.

9.14.1. Producing XML Content

A set of functions and function-like expressions are available for producing XML content from SQL data. As such, they are particularly suitable for formatting query results into XML documents for processing in client applications.

9.14.1.1. xmlcomment

```
xmlcomment(text)
```

The function xmlcomment creates an XML value containing an XML comment with the specified text as content. The text cannot contain "--" or end with a "-" so that the resulting construct is a valid XML comment. If the argument is null, the result is null.

Example:

```
SELECT xmlcomment('hello');

  xmlcomment
--------------
 <!--hello-->
```

9.14.1.2. xmlconcat

```
xmlconcat(xml[, ...])
```

The function xmlconcat concatenates a list of individual XML values to create a single value containing an XML content fragment. Null values are omitted; the result is only null if there are no nonnull arguments.

Example:

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');

      xmlconcat
----------------------
 <abc/><bar>foo</bar>
```

XML declarations, if present, are combined as follows. If all argument values have the same XML version declaration, that version is used in the result, else no version is used. If all argument values have the standalone declaration value "yes", then that value is used in the result. If all argument values have a standalone declaration value and at least one is "no", then that is used in the result. Else the result will have no standalone declaration. If the result is determined to require a standalone declaration but no version declaration, a version declaration with version 1.0 will be used because XML requires an XML declaration to contain a version declaration. Encoding declarations are ignored and removed in all cases.

Example:

```
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1" standalone="no"?><bar/

             xmlconcat
-----------------------------------
 <?xml version="1.1"?><foo/><bar/>
```

9.14.1.3. xmlelement

```
xmlelement(name name [, xmlattributes(value [AS attname] [, ... ])] [, content, ...])
```

The xmlelement expression produces an XML element with the given name, attributes, and content.

Examples:

```
SELECT xmlelement(name foo);

 xmlelement
------------
 <foo/>

SELECT xmlelement(name foo, xmlattributes('xyz' as bar));

    xmlelement
------------------
 <foo bar="xyz"/>

SELECT xmlelement(name foo, xmlattributes(current_date as bar), 'cont', 'ent');

             xmlelement
-------------------------------------
 <foo bar="2007-01-26">content</foo>
```

Element and attribute names that are not valid XML names are escaped by replacing the offending characters by the sequence _x*HHHH*_, where *HHHH* is the character's Unicode codepoint in hexadecimal notation. For example:

```
SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));

            xmlelement
----------------------------------
 <foo_x0024_bar a_x0026_b="xyz"/>
```

An explicit attribute name need not be specified if the attribute value is a column reference, in which case the column's name will be used as the attribute name by default. In other cases, the attribute must be given an explicit name. So this example is valid:

```
CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

But these are not:

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```

Element content, if specified, will be formatted according to its data type. If the content is itself of type `xml`, complex XML documents can be constructed. For example:

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
                            xmlelement(name abc),
                            xmlcomment('test'),
                            xmlelement(name xyz));

                  xmlelement
----------------------------------------------
 <foo bar="xyz"><abc/><!--test--><xyz/></foo>
```

Content of other types will be formatted into valid XML character data. This means in particular that the characters <, >, and & will be converted to entities. Binary data (data type `bytea`) will be represented in base64 or hex encoding, depending on the setting of the configuration parameter xmlbinary. The particular behavior for individual data types is expected to evolve in order to align the SQL and PostgreSQL data types with the XML Schema specification, at which point a more precise description will appear.

9.14.1.4. `xmlforest`

```
xmlforest(content [AS name] [, ...])
```

The `xmlforest` expression produces an XML forest (sequence) of elements using the given names and content.

Examples:

```
SELECT xmlforest('abc' AS foo, 123 AS bar);

          xmlforest
------------------------------
 <foo>abc</foo><bar>123</bar>


SELECT xmlforest(table_name, column_name)
FROM information_schema.columns
WHERE table_schema = 'pg_catalog';

                                        xmlforest
-------------------------------------------------------------------------------------------
 <table_name>pg_authid</table_name><column_name>rolname</column_name>
 <table_name>pg_authid</table_name><column_name>rolsuper</column_name>
 ...
```

As seen in the second example, the element name can be omitted if the content value is a column reference, in which case the column name is used by default. Otherwise, a name must be specified.

Element names that are not valid XML names are escaped as shown for `xmlelement` above. Similarly, content data is escaped to make valid XML content, unless it is already of type `xml`.

Note that XML forests are not valid XML documents if they consist of more than one element, so it might be useful to wrap `xmlforest` expressions in `xmlelement`.

9.14.1.5. xmlpi

```
xmlpi(name target [, content])
```

The xmlpi expression creates an XML processing instruction. The content, if present, must not contain the character sequence ?>.

Example:

```
SELECT xmlpi(name php, 'echo "hello world";');

            xmlpi
-----------------------------
 <?php echo "hello world";?>
```

9.14.1.6. xmlroot

```
xmlroot(xml, version text | no value [, standalone yes|no|no value])
```

The xmlroot expression alters the properties of the root node of an XML value. If a version is specified, it replaces the value in the root node's version declaration; if a standalone setting is specified, it replaces the value in the root node's standalone declaration.

```
SELECT xmlroot(xmlparse(document '<?xml version="1.1"?><content>abc</content>'),
               version '1.0', standalone yes);

                xmlroot
----------------------------------------
 <?xml version="1.0" standalone="yes"?>
 <content>abc</content>
```

9.14.1.7. xmlagg

```
xmlagg(xml)
```

The function xmlagg is, unlike the other functions described here, an aggregate function. It concatenates the input values to the aggregate function call, much like xmlconcat does, except that concatenation occurs across rows rather than across expressions in a single row. See Section 9.20 for additional information about aggregate functions.

Example:

```
CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;
        xmlagg
```

```
----------------------
 <foo>abc</foo><bar/>
```

To determine the order of the concatenation, an `ORDER BY` clause may be added to the aggregate call as described in Section 4.2.7. For example:

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

The following non-standard approach used to be recommended in previous versions, and may still be useful in specific cases:

```
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y DESC) AS tab;
        xmlagg
----------------------
 <bar/><foo>abc</foo>
```

9.14.2. XML Predicates

The expressions described in this section check properties of `xml` values.

9.14.2.1. IS DOCUMENT

```
xml IS DOCUMENT
```

The expression `IS DOCUMENT` returns true if the argument XML value is a proper XML document, false if it is not (that is, it is a content fragment), or null if the argument is null. See Section 8.13 about the difference between documents and content fragments.

9.14.2.2. IS NOT DOCUMENT

```
xml IS NOT DOCUMENT
```

The expression `IS NOT DOCUMENT` returns false if the argument XML value is a proper XML document, true if it is not (that is, it is a content fragment), or null if the argument is null.

9.14.2.3. XMLEXISTS

```
XMLEXISTS(text PASSING [BY REF] xml [BY REF])
```

The function xmlexists returns true if the XPath expression in the first argument returns any nodes, and false otherwise. (If either argument is null, the result is null.)

Example:

```
SELECT xmlexists('//town[text() = "Toronto"]' PASSING BY REF '<towns><town>Toronto</town><t

 xmlexists
------------
 t
(1 row)
```

The BY REF clauses have no effect in PostgreSQL, but are allowed for SQL conformance and compatibility with other implementations. Per SQL standard, the first BY REF is required, the second is optional. Also note that the SQL standard specifies the xmlexists construct to take an XQuery expression as first argument, but PostgreSQL currently only supports XPath, which is a subset of XQuery.

9.14.2.4. xml_is_well_formed

```
xml_is_well_formed(text)
xml_is_well_formed_document(text)
xml_is_well_formed_content(text)
```

These functions check whether a text string is well-formed XML, returning a Boolean result. xml_is_well_formed_document checks for a well-formed document, while xml_is_well_formed_content checks for well-formed content. xml_is_well_formed does the former if the xmloption configuration parameter is set to DOCUMENT, or the latter if it is set to CONTENT. This means that xml_is_well_formed is useful for seeing whether a simple cast to type xml will succeed, whereas the other two functions are useful for seeing whether the corresponding variants of XMLPARSE will succeed.

Examples:

```
SET xmloption TO DOCUMENT;
SELECT xml_is_well_formed('<>');
 xml_is_well_formed
--------------------
 f
(1 row)

SELECT xml_is_well_formed('<abc/>');
 xml_is_well_formed
--------------------
 t
(1 row)

SET xmloption TO CONTENT;
SELECT xml_is_well_formed('abc');
 xml_is_well_formed
--------------------
```

```
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/stuff">bar</pg:
 xml_is_well_formed_document
-----------------------------
 t
(1 row)

SELECT xml_is_well_formed_document('<pg:foo xmlns:pg="http://postgresql.org/stuff">bar</my:
 xml_is_well_formed_document
-----------------------------
 f
(1 row)
```

The last example shows that the checks include whether namespaces are correctly matched.

9.14.3. Processing XML

To process values of data type `xml`, PostgreSQL offers the functions `xpath` and `xpath_exists`, which evaluate XPath 1.0 expressions.

`xpath(`*`xpath`*`, `*`xml`*` [, `*`nsarray`*`])`

The function `xpath` evaluates the XPath expression *`xpath`* (a `text` value) against the XML value *`xml`*. It returns an array of XML values corresponding to the node set produced by the XPath expression. If the XPath expression returns a scalar value rather than a node set, a single-element array is returned.

The second argument must be a well formed XML document. In particular, it must have a single root node element.

The optional third argument of the function is an array of namespace mappings. This array should be a two-dimensional `text` array with the length of the second axis being equal to 2 (i.e., it should be an array of arrays, each of which consists of exactly 2 elements). The first element of each array entry is the namespace name (alias), the second the namespace URI. It is not required that aliases provided in this array be the same as those being used in the XML document itself (in other words, both in the XML document and in the `xpath` function context, aliases are *local*).

Example:

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
             ARRAY[ARRAY['my', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

To deal with default (anonymous) namespaces, do something like this:

```
SELECT xpath('//mydefns:b/text()', '<a xmlns="http://example.com"><b>test</b></a>',
             ARRAY[ARRAY['mydefns', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

```
xpath_exists(xpath, xml [, nsarray])
```

The function `xpath_exists` is a specialized form of the `xpath` function. Instead of returning the individual XML values that satisfy the XPath, this function returns a Boolean indicating whether the query was satisfied or not. This function is equivalent to the standard `XMLEXISTS` predicate, except that it also offers support for a namespace mapping argument.

Example:

```
SELECT xpath_exists('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
                    ARRAY[ARRAY['my', 'http://example.com']]);

 xpath_exists
--------------
 t
(1 row)
```

9.14.4. Mapping Tables to XML

The following functions map the contents of relational tables to XML values. They can be thought of as XML export functionality:

```
table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xml(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xml(cursor refcursor, count int, nulls boolean,
              tableforest boolean, targetns text)
```

The return type of each function is `xml`.

`table_to_xml` maps the content of the named table, passed as parameter `tbl`. The `regclass` type accepts strings identifying tables using the usual notation, including optional schema qualifications and double quotes. `query_to_xml` executes the query whose text is passed as parameter `query` and maps the result set. `cursor_to_xml` fetches the indicated number of rows from the cursor specified by the parameter `cursor`. This variant is recommended if large tables have to be mapped, because the result value is built up in memory by each function.

If `tableforest` is false, then the resulting XML document looks like this:

```
<tablename>
  <row>
    <columnname1>data</columnname1>
```

```
    <columnname2>data</columnname2>
  </row>

  <row>
    ...
  </row>

  ...
</tablename>
```

If `tableforest` is true, the result is an XML content fragment that looks like this:

```
<tablename>
  <columnname1>data</columnname1>
  <columnname2>data</columnname2>
</tablename>

<tablename>
  ...
</tablename>

...
```

If no table name is available, that is, when mapping a query or a cursor, the string `table` is used in the first format, `row` in the second format.

The choice between these formats is up to the user. The first format is a proper XML document, which will be important in many applications. The second format tends to be more useful in the `cursor_to_xml` function if the result values are to be reassembled into one document later on. The functions for producing XML content discussed above, in particular `xmlelement`, can be used to alter the results to taste.

The data values are mapped in the same way as described for the function `xmlelement` above.

The parameter `nulls` determines whether null values should be included in the output. If true, null values in columns are represented as:

```
<columnname xsi:nil="true"/>
```

where `xsi` is the XML namespace prefix for XML Schema Instance. An appropriate namespace declaration will be added to the result value. If false, columns containing null values are simply omitted from the output.

The parameter `targetns` specifies the desired XML namespace of the result. If no particular namespace is wanted, an empty string should be passed.

The following functions return XML Schema documents describing the mappings performed by the corresponding functions above:

```
table_to_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xmlschema(cursor refcursor, nulls boolean, tableforest boolean, targetns text)
```

It is essential that the same parameters are passed in order to obtain matching XML data mappings and XML Schema documents.

The following functions produce XML data mappings and the corresponding XML Schema in one document (or forest), linked together. They can be useful where self-contained and self-describing results are wanted:

```
table_to_xml_and_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xml_and_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)
```

In addition, the following functions are available to produce analogous mappings of entire schemas or the entire current database:

```
schema_to_xml(schema name, nulls boolean, tableforest boolean, targetns text)
schema_to_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)
schema_to_xml_and_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)

database_to_xml(nulls boolean, tableforest boolean, targetns text)
database_to_xmlschema(nulls boolean, tableforest boolean, targetns text)
database_to_xml_and_xmlschema(nulls boolean, tableforest boolean, targetns text)
```

Note that these potentially produce a lot of data, which needs to be built up in memory. When requesting content mappings of large schemas or databases, it might be worthwhile to consider mapping the tables separately instead, possibly even through a cursor.

The result of a schema content mapping looks like this:

```
<schemaname>

table1-mapping

table2-mapping

...

</schemaname>
```

where the format of a table mapping depends on the `tableforest` parameter as explained above.

The result of a database content mapping looks like this:

```
<dbname>

<schema1name>
  ...
</schema1name>

<schema2name>
  ...
</schema2name>

...

</dbname>
```

where the schema mapping is as above.

As an example of using the output produced by these functions, Example 9-1 shows an XSLT stylesheet that converts the output of `table_to_xml_and_xmlschema` to an HTML document containing a tabular rendition of the table data. In a similar manner, the results from these functions can be converted into other XML-based formats.

Example 9-1. XSLT Stylesheet for Converting SQL/XML Output to HTML

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.w3.org/1999/xhtml"
>

  <xsl:output method="xml"
      doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
      doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"
      indent="yes"/>

  <xsl:template match="/*">
    <xsl:variable name="schema" select="//xsd:schema"/>
    <xsl:variable name="tabletypename"
                  select="$schema/xsd:element[@name=name(current())]/@type"/>
    <xsl:variable name="rowtypename"
                  select="$schema/xsd:complexType[@name=$tabletypename]/xsd:sequence/xsd:el

    <html>
      <head>
        <title><xsl:value-of select="name(current())"/></title>
      </head>
      <body>
        <table>
          <tr>
            <xsl:for-each select="$schema/xsd:complexType[@name=$rowtypename]/xsd:sequence/
              <th><xsl:value-of select="."/></th>
            </xsl:for-each>
          </tr>

          <xsl:for-each select="row">
            <tr>
              <xsl:for-each select="*">
                <td><xsl:value-of select="."/></td>
              </xsl:for-each>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

9.15. JSON Functions and Operators

Table 9-40 shows the operators that are available for use with the two JSON data types (see Section 8.14).

Table 9-40. `json` and `jsonb` Operators

Operator	Right Operand Type	Description	Example	Example Result
`->`	`int`	Get JSON array element (indexed from zero, negative integers count from the end)	`'[{"a":"foo"},{"b":"bar"},{"c":"baz"}]'::json->2`	`{"c":"baz"}`
`->`	`text`	Get JSON object field by key	`'{"a": {"b":"foo"}}'::json->'a'`	`{"b":"foo"}`
`->>`	`int`	Get JSON array element as `text`	`'[1,2,3]'::json->>2`	`3`
`->>`	`text`	Get JSON object field as `text`	`'{"a":1,"b":2}'::json->>'b'`	`2`
`#>`	`text[]`	Get JSON object at specified path	`'{"a": {"b":{"c": "foo"}}}'::json#>'{a,b}'`	`{"c": "foo"}`
`#>>`	`text[]`	Get JSON object at specified path as `text`	`'{"a":[1,2,3],"b":[4,5,6]}'::json#>>'{a,2}'`	`3`

Note: There are parallel variants of these operators for both the `json` and `jsonb` types. The field/element/path extraction operators return the same type as their left-hand input (either `json` or `jsonb`), except for those specified as returning `text`, which coerce the value to text. The field/element/path extraction operators return NULL, rather than failing, if the JSON input does not have the right structure to match the request; for example if no such element exists. The field/element/path extraction operators that accept integer JSON array subscripts all support negative subscripting from the end of arrays.

The standard comparison operators shown in Table 9-1 are available for `jsonb`, but not for `json`. They follow the ordering rules for B-tree operations outlined at Section 8.14.4.

Some further operators also exist only for `jsonb`, as shown in Table 9-41. Many of these operators can be indexed by `jsonb` operator classes. For a full description of `jsonb` containment and existence semantics, see Section 8.14.3. Section 8.14.4 describes how these operators can be used to effectively index `jsonb` data.

Table 9-41. Additional `jsonb` Operators

Operator	Right Operand Type	Description	Example
`@>`	`jsonb`	Does the left JSON value contain the right JSON path/value entries at the top level?	`'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb`
`<@`	`jsonb`	Are the left JSON path/value entries contained at the top level within the right JSON value?	`'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb`
`?`	`text`	Does the *string* exist as a top-level key within the JSON value?	`'{"a":1, "b":2}'::jsonb ? 'b'`
`?|`	`text[]`	Do any of these array *strings* exist as top-level keys?	`'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'c']`
`?&`	`text[]`	Do all of these array *strings* exist as top-level keys?	`'["a", "b"]'::jsonb ?& array['a', 'b']`
`||`	`jsonb`	Concatenate two `jsonb` values into a new `jsonb` value	`'["a", "b"]'::jsonb || '["c", "d"]'::jsonb`
`-`	`text`	Delete key/value pair or *string* element from left operand. Key/value pairs are matched based on their key value.	`'{"a": "b"}'::jsonb - 'a'`
`-`	`integer`	Delete the array element with specified index (Negative integers count from the end). Throws an error if top level container is not an array.	`'["a", "b"]'::jsonb - 1`
`#-`	`text[]`	Delete the field or element with specified path (for JSON arrays, negative integers count from the end)	`'["a", {"b":1}]'::jsonb #- '{1,b}'`

Note: The `||` operator concatenates the elements at the top level of each of its operands. It does not operate recursively. For example, if both operands are objects with a common key field name, the value of the field in the result will just be the value from the right hand operand.

Table 9-42 shows the functions that are available for creating `json` and `jsonb` values. (There are no equivalent functions for `jsonb`, of the `row_to_json` and `array_to_json` functions. However, the `to_jsonb` function supplies much the same functionality as these functions would.)

Table 9-42. JSON Creation Functions

Function	Description	Example	Example Result
`to_json(anyelement)` `to_jsonb(anyelement)`	Returns the value as `json` or `jsonb`. Arrays and composites are converted (recursively) to arrays and objects; otherwise, if there is a cast from the type to `json`, the cast function will be used to perform the conversion; otherwise, a scalar value is produced. For any scalar type other than a number, a Boolean, or a null value, the text representation will be used, in such a fashion that it is a valid `json` or `jsonb` value.	`to_json('Fred said "Hi."'::text)`	`"Fred said \"Hi.\""`
`array_to_json(anyarray [, pretty_bool])`	Returns the array as a JSON array. A PostgreSQL multidimensional array becomes a JSON array of arrays. Line feeds will be added between dimension-1 elements if `pretty_bool` is true.	`array_to_json('{{1,5},{99,100}}'::int[])`	`[[1,5],[99,100]]`
`row_to_json(record [, pretty_bool])`	Returns the row as a JSON object. Line feeds will be added between level-1 elements if `pretty_bool` is true.	`row_to_json(row(1,'foo'))`	`{"f1":1,"f2":"foo"}`
`json_build_array(VARIADIC "any")` `jsonb_build_array(VARIADIC "any")`	Builds a possibly-heterogeneously-typed JSON array out of a variadic argument list.	`json_build_array(1,2,'3',4,'5')`	`[1, 2, "3", 4, "5"]`

Function	Description	Example	Example Result
`json_build_object(VARIADIC "any")` `jsonb_build_object(VARIADIC "any")`	Builds a JSON object out of a variadic argument list. By convention, the argument list consists of alternating keys and values.	`json_build_object('foo',1,'bar',2)`	`{"foo": 1, "bar": 2}`
`json_object(text[])` `jsonb_object(text[])`	Builds a JSON object out of a text array. The array must have either exactly one dimension with an even number of members, in which case they are taken as alternating key/value pairs, or two dimensions such that each inner array has exactly two elements, which are taken as a key/value pair.	`json_object('{a, 1, b, "def", c, 3.5}')` `json_object('{{a, 1},{b, "def"},{c, 3.5}}')`	`{"a": "1", "b": "def", "c": "3.5"}`
`json_object(keys text[], values text[])` `jsonb_object(keys text[], values text[])`	This form of `json_object` takes keys and values pairwise from two separate arrays. In all other respects it is identical to the one-argument form.	`json_object('{a, b}', '{1,2}')`	`{"a": "1", "b": "2"}`

Note: `array_to_json` and `row_to_json` have the same behavior as `to_json` except for offering a pretty-printing option. The behavior described for `to_json` likewise applies to each individual value converted by the other JSON creation functions.

Note: The hstore extension has a cast from `hstore` to `json`, so that `hstore` values converted via the JSON creation functions will be represented as JSON objects, not as primitive string values.

Table 9-43 shows the functions that are available for processing `json` and `jsonb` values.

Table 9-43. JSON Processing Functions

Function	Return Type	Description	Example	Example Result

Function	Return Type	Description	Example	Example Result
`json_array_length(json)` `jsonb_array_length(jsonb)`	`int`	Returns the number of elements in the outermost JSON array.	`json_array_length('[1,2,3,{"f1":1,"f2":[5,6]`	`5`

`json_each(json)`
`jsonb_each(jsonb)`
	`setof key text, value json`
	`setof key text, value jsonb`
		Expands the outermost JSON object into a set of key/value pairs.	`select * from json_each('{"a":"foo",`
			`"b":"bar"}')`
				`key | value`
				`-------`
				`a | "foo"`
				`b | "bar"`

`json_each_text(json)`
`jsonb_each_text(jsonb)`
	`setof key text, value text`	Expands the outermost JSON object into a set of key/value pairs. The returned values will be of type `text`.	`select * from json_each_text('{"a":"foo",`
			`"b":"bar"}')`
				`key | value`
				`-----`
				`a | foo`
				`b | bar`

`json_extract_path(from_json json, VARIADIC path_elems text[])` `jsonb_extract_path(from_json jsonb, VARIADIC path_elems text[])`	`json` `jsonb`	Returns JSON value pointed to by *path_elems* (equivalent to `#>` operator).	`json_extract_path('{"f2":{"f3":1},"f4":{"f5"`	`99`
`json_extract_path_text(from_json json, VARIADIC path_elems text[])` `jsonb_extract_path_text(from_json jsonb, VARIADIC path_elems text[])`	`text`	Returns JSON value pointed to by *path_elems* as `text` (equivalent to `#>>` operator).	`json_extract_path_text('{"f2":{"f3":1},"f4":` `'f6')`	`foo`

Function	Return Type	Description	Example	Example Result

`json_object_keys(json)`
`jsonb_object_keys(jsonb)`
	`setof text`	Returns set of keys in the outermost JSON object.	`json_object_keys('{"f1":"abc","f2":{"f3":"a","f4":"b"}}')`	`json_object_keys`
				`------------------`
				`f1`
				`f2`

`json_populate_record(base anyelement, from_json json)`
`jsonb_populate_record(base anyelement, from_json jsonb)`
	`anyelement`	Expands the object in *from_json* to a row whose columns match the record type defined by *base* (see note below).	`select * from json_populate_record(null::myrowtype, '{"a":1,"b":2}')`	`a | b`
				`---+---`
				`1 | 2`

`json_populate_recordset(base anyelement, from_json json)`
`jsonb_populate_recordset(base anyelement, from_json jsonb)`
	`setof anyelement`	Expands the outermost array of objects in *from_json* to a set of rows whose columns match the record type defined by *base* (see note below).	`select * from json_populate_recordset(null::myrowtype, '[{"a":1,"b":2},{"a":3,"b":4}]')`	`a | b`
				`---+---`
				`1 | 2`
				`3 | 4`

`json_array_elements(json)`
`jsonb_array_elements(jsonb)`
	`setof json`
	`setof jsonb`
		Expands a JSON array to a set of JSON values.	`select * from json_array_elements('[1,true,[2,false]]')`	`value`
				`-----------`
				`1`
				`true`
				`[2,false]`

`json_array_elements_text(json)`
`jsonb_array_elements_text(jsonb)`
	`setof text`	Expands a JSON array to a set of `text` values.	`select * from json_array_elements_text('["foo", "bar"]')`	`value`
				`-----`
				`foo`
				`bar`

`json_typeof(json)` `jsonb_typeof(jsonb)`	`text`	Returns the type of the outermost JSON value as a text string. Possible types are `object`, `array`, `string`, `number`, `boolean`, and `null`.	`json_typeof('-123.4')`	`number`

Function	Return Type	Description	Example	Example Result
`json_to_record(json)` `jsonb_to_record(jsonb)`	`record`	Builds an arbitrary record from a JSON object (see note below). As with all functions returning `record`, the caller must explicitly define the structure of the record with an `AS` clause.	`select * from json_to_record('{"a":1,"b":[1,2,3],"c":"bar" as x(a int, b text, d text)`	`a | b | d` `1 | [1,2,3] |`

`json_to_recordset(json)`
`jsonb_to_recordset(jsonb)`
	`setof record`	Builds an arbitrary set of records from a JSON array of objects (see note below). As with all functions returning `record`, the caller must explicitly define the structure of the record with an `AS` clause.	`select * from json_to_recordset('[{"a":1,"b":"foo"},{"a":" as x(a int, b text);`	`a | b`
				`1 | foo`
				`2 | `

`json_strip_nulls(from_json json)` `jsonb_strip_nulls(from_json jsonb)`	*from_json*	Returns *from_json* with all object fields that have null values omitted. Other null values are untouched.	`json_strip_nulls('[{"f1":1,"f2":null},2,null,3]')`	`[{"f1":1},2,null,3]`

Function	Return Type	Description	Example	Example Result
`jsonb_set(target jsonb, path text[], new_value jsonb [, create_missing boolean])`	`jsonb`	Returns *target* with the section designated by *path* replaced by *new_value*, or with *new_value* added if *create_missing* is true (default is `true`) and the item designated by *path* does not exist. As with the path oriented operators, negative integers that appear in *path* count from the end of JSON arrays.	`jsonb_set('[{"f1":1,"f2":null},2,null,3]', '{0,f1}','[2,3,4]', false)` `jsonb_set('[{"f1":1,"f2":null},2]', '{0,f3}','[2,3,4]')`	`[{"f1": [2, 3, 4], "f2": null}, 2, null, 3]` `[{"f1": 1, "f2": null, "f3": [2, 3, 4]}, 2]`
`jsonb_pretty(from_json jsonb)`	`text`	Returns *from_json* as indented JSON text.	`jsonb_pretty('[{"f1":1,"f2":null},2,null,3]')`	`[ { "f1": 1, "f2": null }, 2, null, 3 ]`

Note: Many of these functions and operators will convert Unicode escapes in JSON strings to the appropriate single character. This is a non-issue if the input is type `jsonb`, because the conversion was already done; but for `json` input, this may result in throwing an error, as noted in Section 8.14.

Note: While the examples for the functions `json_populate_record`, `json_populate_recordset`, `json_to_record` and `json_to_recordset` use constants, the typical use would be to reference a table in the `FROM` clause and use one of its `json` or `jsonb` columns as an argument to the function. Extracted key values can then be referenced in other parts of the query, like `WHERE` clauses and target lists. Extracting multiple values in this way can improve performance over extracting them separately with per-key operators.

JSON keys are matched to identical column names in the target row type. JSON type coercion for these functions is "best effort" and may not result in desired values for some types. JSON fields that do not appear in the target row type will be omitted from the output, and target columns that do not match any JSON field will simply be NULL.

Note: All the items of the `path` parameter of `jsonb_set` must be present in the `target`, unless `create_missing` is true, in which case all but the last item must be present. If these conditions are not met the `target` is returned unchanged.

If the last path item is an object key, it will be created if it is absent and given the new value. If the last path item is an array index, if it is positive the item to set is found by counting from the left, and if negative by counting from the right - `-1` designates the rightmost element, and so on. If the item is out of the range -array_length .. array_length -1, and create_missing is true, the new value is added at the beginning of the array if the item is negative, and at the end of the array if it is positive.

Note: The `json_typeof` function's `null` return value should not be confused with a SQL NULL. While calling `json_typeof('null'::json)` will return `null`, calling `json_typeof(NULL::json)` will return a SQL NULL.

Note: If the argument to `json_strip_nulls` contains duplicate field names in any object, the result could be semantically somewhat different, depending on the order in which they occur. This is not an issue for `jsonb_strip_nulls` since `jsonb` values never have duplicate object field names.

See also Section 9.20 for the aggregate function `json_agg` which aggregates record values as JSON, and the aggregate function `json_object_agg` which aggregates pairs of values into a JSON object, and their `jsonb` equivalents, `jsonb_agg` and `jsonb_object_agg`.

9.16. Sequence Manipulation Functions

This section describes functions for operating on *sequence objects*, also called sequence generators or just sequences. Sequence objects are special single-row tables created with CREATE SEQUENCE. Sequence objects are commonly used to generate unique identifiers for rows of a table. The sequence functions, listed in Table 9-44, provide simple, multiuser-safe methods for obtaining successive sequence values from sequence objects.

Table 9-44. Sequence Functions

Function	Return Type	Description
`currval(regclass)`	`bigint`	Return value most recently obtained with `nextval` for specified sequence
`lastval()`	`bigint`	Return value most recently obtained with `nextval` for any sequence
`nextval(regclass)`	`bigint`	Advance sequence and return new value
`setval(regclass, bigint)`	`bigint`	Set sequence's current value
`setval(regclass, bigint, boolean)`	`bigint`	Set sequence's current value and `is_called` flag

The sequence to be operated on by a sequence function is specified by a `regclass` argument, which is

simply the OID of the sequence in the pg_class system catalog. You do not have to look up the OID by hand, however, since the regclass data type's input converter will do the work for you. Just write the sequence name enclosed in single quotes so that it looks like a literal constant. For compatibility with the handling of ordinary SQL names, the string will be converted to lower case unless it contains double quotes around the sequence name. Thus:

```
nextval('foo')      operates on sequence foo
nextval('FOO')      operates on sequence foo
nextval('"Foo"')    operates on sequence Foo
```

The sequence name can be schema-qualified if necessary:

```
nextval('myschema.foo')     operates on myschema.foo
nextval('"myschema".foo')   same as above
nextval('foo')              searches search path for foo
```

See Section 8.18 for more information about regclass.

> **Note:** Before PostgreSQL 8.1, the arguments of the sequence functions were of type text, not regclass, and the above-described conversion from a text string to an OID value would happen at run time during each call. For backward compatibility, this facility still exists, but internally it is now handled as an implicit coercion from text to regclass before the function is invoked.
>
> When you write the argument of a sequence function as an unadorned literal string, it becomes a constant of type regclass. Since this is really just an OID, it will track the originally identified sequence despite later renaming, schema reassignment, etc. This "early binding" behavior is usually desirable for sequence references in column defaults and views. But sometimes you might want "late binding" where the sequence reference is resolved at run time. To get late-binding behavior, force the constant to be stored as a text constant instead of regclass:
>
> ```
> nextval('foo'::text)      foo is looked up at runtime
> ```
>
> Note that late binding was the only behavior supported in PostgreSQL releases before 8.1, so you might need to do this to preserve the semantics of old applications.
>
> Of course, the argument of a sequence function can be an expression as well as a constant. If it is a text expression then the implicit coercion will result in a run-time lookup.

The available sequence functions are:

nextval

Advance the sequence object to its next value and return that value. This is done atomically: even if multiple sessions execute nextval concurrently, each will safely receive a distinct sequence value.

If a sequence object has been created with default parameters, successive nextval calls will return successive values beginning with 1. Other behaviors can be obtained by using special parameters in the CREATE SEQUENCE command; see its command reference page for more information.

> **Important:** To avoid blocking concurrent transactions that obtain numbers from the same sequence, a nextval operation is never rolled back; that is, once a value has been fetched it is considered used and will not be returned again. This is true even if the surrounding transaction

later aborts, or if the calling query ends up not using the value. For example an INSERT with an ON CONFLICT clause will compute the to-be-inserted tuple, including doing any required nextval calls, before detecting any conflict that would cause it to follow the ON CONFLICT rule instead. Such cases will leave unused "holes" in the sequence of assigned values. Thus, PostgreSQL sequence objects *cannot be used to obtain "gapless" sequences*.

currval

Return the value most recently obtained by nextval for this sequence in the current session. (An error is reported if nextval has never been called for this sequence in this session.) Because this is returning a session-local value, it gives a predictable answer whether or not other sessions have executed nextval since the current session did.

lastval

Return the value most recently returned by nextval in the current session. This function is identical to currval, except that instead of taking the sequence name as an argument it refers to whichever sequence nextval was most recently applied to in the current session. It is an error to call lastval if nextval has not yet been called in the current session.

setval

Reset the sequence object's counter value. The two-parameter form sets the sequence's last_value field to the specified value and sets its is_called field to true, meaning that the next nextval will advance the sequence before returning a value. The value reported by currval is also set to the specified value. In the three-parameter form, is_called can be set to either true or false. true has the same effect as the two-parameter form. If it is set to false, the next nextval will return exactly the specified value, and sequence advancement commences with the following nextval. Furthermore, the value reported by currval is not changed in this case. For example,

```
SELECT setval('foo', 42);           Next nextval will return 43
SELECT setval('foo', 42, true);     Same as above
SELECT setval('foo', 42, false);    Next nextval will return 42
```

The result returned by setval is just the value of its second argument.

Important: Because sequences are non-transactional, changes made by setval are not undone if the transaction rolls back.

9.17. Conditional Expressions

This section describes the SQL-compliant conditional expressions available in PostgreSQL.

Tip: If your needs go beyond the capabilities of these conditional expressions, you might want to consider writing a stored procedure in a more expressive programming language.

9.17.1. CASE

The SQL CASE expression is a generic conditional expression, similar to if/else statements in other programming languages:

```
CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

CASE clauses can be used wherever an expression is valid. Each *condition* is an expression that returns a boolean result. If the condition's result is true, the value of the CASE expression is the *result* that follows the condition, and the remainder of the CASE expression is not processed. If the condition's result is not true, any subsequent WHEN clauses are examined in the same manner. If no WHEN *condition* yields true, the value of the CASE expression is the *result* of the ELSE clause. If the ELSE clause is omitted and no condition is true, the result is null.

An example:

```
SELECT * FROM test;

 a
---
 1
 2
 3


SELECT a,
       CASE WHEN a=1 THEN 'one'
            WHEN a=2 THEN 'two'
            ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

The data types of all the *result* expressions must be convertible to a single output type. See Section 10.5 for more details.

There is a "simple" form of CASE expression that is a variant of the general form above:

```
CASE expression
    WHEN value THEN result
    [WHEN ...]
    [ELSE result]
END
```

The first *expression* is computed, then compared to each of the *value* expressions in the WHEN clauses until one is found that is equal to it. If no match is found, the *result* of the ELSE clause (or a null value) is returned. This is similar to the switch statement in C.

The example above can be written using the simple CASE syntax:

```
SELECT a,
       CASE a WHEN 1 THEN 'one'
              WHEN 2 THEN 'two'
              ELSE 'other'
       END
    FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

A CASE expression does not evaluate any subexpressions that are not needed to determine the result. For example, this is a possible way of avoiding a division-by-zero failure:

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

> **Note:** As described in Section 4.2.14, there are various situations in which subexpressions of an expression are evaluated at different times, so that the principle that "CASE evaluates only necessary subexpressions" is not ironclad. For example a constant 1/0 subexpression will usually result in a division-by-zero failure at planning time, even if it's within a CASE arm that would never be entered at run time.

9.17.2. COALESCE

```
COALESCE(value [, ...])
```

The COALESCE function returns the first of its arguments that is not null. Null is returned only if all arguments are null. It is often used to substitute a default value for null values when data is retrieved for display, for example:

```
SELECT COALESCE(description, short_description, '(none)') ...
```

This returns description if it is not null, otherwise short_description if it is not null, otherwise (none).

The arguments must all be convertible to a common data type, which will be the type of the result (see Section 10.5 for details).

Like a CASE expression, COALESCE only evaluates the arguments that are needed to determine the result; that is, arguments to the right of the first non-null argument are not evaluated. This SQL-standard function provides capabilities similar to NVL and IFNULL, which are used in some other database systems.

9.17.3. NULLIF

```
NULLIF(value1, value2)
```

The NULLIF function returns a null value if *value1* equals *value2*; otherwise it returns *value1*. This can be used to perform the inverse operation of the COALESCE example given above:

```
SELECT NULLIF(value, '(none)') ...
```

In this example, if value is (none), null is returned, otherwise the value of value is returned.

The two arguments must be of comparable types. To be specific, they are compared exactly as if you had written *value1 = value2*, so there must be a suitable = operator available.

The result has the same type as the first argument — but there is a subtlety. What is actually returned is the first argument of the implied = operator, and in some cases that will have been promoted to match the second argument's type. For example, NULLIF(1, 2.2) yields numeric, because there is no integer = numeric operator, only numeric = numeric.

9.17.4. GREATEST and LEAST

```
GREATEST(value [, ...])

LEAST(value [, ...])
```

The GREATEST and LEAST functions select the largest or smallest value from a list of any number of expressions. The expressions must all be convertible to a common data type, which will be the type of the result (see Section 10.5 for details). NULL values in the list are ignored. The result will be NULL only if all the expressions evaluate to NULL.

Note that GREATEST and LEAST are not in the SQL standard, but are a common extension. Some other databases make them return NULL if any argument is NULL, rather than only when all are NULL.

9.18. Array Functions and Operators

Table 9-45 shows the operators available for array types.

Table 9-45. Array Operators

Operator	Description	Example	Result
=	equal	ARRAY[1.1,2.1,3.1]::int[] = ARRAY[1,2,3]	t

Operator	Description	Example	Result
<>	not equal	ARRAY[1,2,3] <> ARRAY[1,2,4]	t
<	less than	ARRAY[1,2,3] < ARRAY[1,2,4]	t
>	greater than	ARRAY[1,4,3] > ARRAY[1,2,4]	t
<=	less than or equal	ARRAY[1,2,3] <= ARRAY[1,2,3]	t
>=	greater than or equal	ARRAY[1,4,3] >= ARRAY[1,4,3]	t
@>	contains	ARRAY[1,4,3] @> ARRAY[3,1,3]	t
<@	is contained by	ARRAY[2,2,7] <@ ARRAY[1,7,4,2,6]	t
&&	overlap (have elements in common)	ARRAY[1,4,3] && ARRAY[2,1]	t
||	array-to-array concatenation	ARRAY[1,2,3] || ARRAY[4,5,6]	{1,2,3,4,5,6}
||	array-to-array concatenation	ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]]	{{1,2,3},{4,5,6},{7,8,9}}
||	element-to-array concatenation	3 || ARRAY[4,5,6]	{3,4,5,6}
||	array-to-element concatenation	ARRAY[4,5,6] || 7	{4,5,6,7}

The array ordering operators (<, >=, etc) compare the array contents element-by-element, using the default B-tree comparison function for the element data type, and sort based on the first difference. In multidimensional arrays the elements are visited in row-major order (last subscript varies most rapidly). If the contents of two arrays are equal but the dimensionality is different, the first difference in the dimensionality information determines the sort order. (This is a change from versions of PostgreSQL prior to 8.2: older versions would claim that two arrays with the same contents were equal, even if the number of dimensions or subscript ranges were different.)

The array containment operators (<@ and @>) consider one array to be contained in another one if each of its elements appears in the other one. Duplicates are not treated specially, thus ARRAY[1] and ARRAY[1,1] are each considered to contain the other.

See Section 8.15 for more details about array operator behavior. See Section 11.2 for more details about which operators support indexed operations.

Table 9-46 shows the functions available for use with array types. See Section 8.15 for more information and examples of the use of these functions.

Table 9-46. Array Functions

Function	Return Type	Description	Example	Result
`array_append(anyarray, anyelement)`	`anyarray`	append an element to the end of an array	`array_append(ARRAY[1,2], 3)`	`{1,2,3}`
`array_cat(anyarray, anyarray)`	`anyarray`	concatenate two arrays	`array_cat(ARRAY[1,2,3], ARRAY[4,5])`	`{1,2,3,4,5}`
`array_ndims(anyarray)`	`int`	returns the number of dimensions of the array	`array_ndims(ARRAY[[1,2,3], [4,5,6]])`	`2`
`array_dims(anyarray)`	`text`	returns a text representation of array's dimensions	`array_dims(ARRAY[[1,2,3], [4,5,6]])`	`[1:2][1:3]`
`array_fill(anyelement, int[] [, int[]])`	`anyarray`	returns an array initialized with supplied value and dimensions, optionally with lower bounds other than 1	`array_fill(7, ARRAY[3], ARRAY[2])`	`[2:4]={7,7,7}`
`array_length(anyarray, int)`	`int`	returns the length of the requested array dimension	`array_length(array[1,2,3], 1)`	`3`
`array_lower(anyarray, int)`	`int`	returns lower bound of the requested array dimension	`array_lower('[0:2]={1,2,3}'::int[], 1)`	`0`
`array_position(anyarray, anyelement [, int])`	`int`	returns the subscript of the first occurrence of the second argument in the array, starting at the element indicated by the third argument or at the first element (array must be one-dimensional)	`array_position(ARRAY['sun','mon','tue','wed', 'mon')`	`2`

Function	Return Type	Description	Example	Result
`array_positions(anyarray, anyelement)`	`int[]`	returns an array of subscripts of all occurrences of the second argument in the array given as first argument (array must be one-dimensional)	`array_positions(ARRAY['A','A','B','A'], 'A')`	`{1,2,4}`
`array_prepend(anyelement, anyarray)`	`anyarray`	append an element to the beginning of an array	`array_prepend(1,ARRAY[2,3])`	`{1,2,3}`
`array_remove(anyarray, anyelement)`	`anyarray`	remove all elements equal to the given value from the array (array must be one-dimensional)	`array_remove(ARRAY[1,2,3,2], 2)`	`{1,3}`
`array_replace(anyarray, anyelement, anyelement)`	`anyarray`	replace each array element equal to the given value with a new value	`array_replace(ARRAY[1,2,5,4], 5, 3)`	`{1,2,3,4}`
`array_to_string(anyarray, text [, text])`	`text`	concatenates array elements using supplied delimiter and optional null string	`array_to_string(ARRAY[1, 2, 3, NULL, 5], ',', '*')`	`1,2,3,*,5`
`array_upper(anyarray, int)`	`int`	returns upper bound of the requested array dimension	`array_upper(ARRAY[1,8,3,7], 1)`	`4`
`cardinality(anyarray)`	`int`	returns the total number of elements in the array, or 0 if the array is empty	`cardinality(ARRAY[[1,2],[3,4]])`	`4`
`string_to_array(text, text [, text])`	`text[]`	splits string into array elements using supplied delimiter and optional null string	`string_to_array('xx~^~yy~^~zz', '~^~', 'yy')`	`{xx,NULL,zz}`

`unnest(anyarray)`	`setof anyelement`	expand an array to a set of rows	`unnest(ARRAY[1,2])`	`1`
				`2`
				(2 rows)

Function	Return Type	Description	Example	Result
unnest(anyarray, anyarray [, ...])	setof anyelement, anyelement [, ...]	expand multiple arrays (possibly of different types) to a set of rows. This is only allowed in the FROM clause; see Section 7.2.1.4	unnest(ARRAY[1,2],ARRAY['foo','bar','baz'])	1 foo 2 bar NULL baz (3 rows)

In array_position and array_positions, each array element is compared to the searched value using IS NOT DISTINCT FROM semantics.

In array_position, NULL is returned if the value is not found.

In array_positions, NULL is returned only if the array is NULL; if the value is not found in the array, an empty array is returned instead.

In string_to_array, if the delimiter parameter is NULL, each character in the input string will become a separate element in the resulting array. If the delimiter is an empty string, then the entire input string is returned as a one-element array. Otherwise the input string is split at each occurrence of the delimiter string.

In string_to_array, if the null-string parameter is omitted or NULL, none of the substrings of the input will be replaced by NULL. In array_to_string, if the null-string parameter is omitted or NULL, any null elements in the array are simply skipped and not represented in the output string.

> **Note:** There are two differences in the behavior of string_to_array from pre-9.1 versions of PostgreSQL. First, it will return an empty (zero-element) array rather than NULL when the input string is of zero length. Second, if the delimiter string is NULL, the function splits the input into individual characters, rather than returning NULL as before.

See also Section 9.20 about the aggregate function array_agg for use with arrays.

9.19. Range Functions and Operators

See Section 8.17 for an overview of range types.

Table 9-47 shows the operators available for range types.

Table 9-47. Range Operators

Operator	Description	Example	Result
=	equal	int4range(1,5) = '[1,4]'::int4range	t

Operator	Description	Example	Result
<>	not equal	numrange(1.1,2.2) <> numrange(1.1,2.3)	t
<	less than	int4range(1,10) < int4range(2,3)	t
>	greater than	int4range(1,10) > int4range(1,5)	t
<=	less than or equal	numrange(1.1,2.2) <= numrange(1.1,2.2)	t
>=	greater than or equal	numrange(1.1,2.2) >= numrange(1.1,2.0)	t
@>	contains range	int4range(2,4) @> int4range(2,3)	t
@>	contains element	'[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp	t
<@	range is contained by	int4range(2,4) <@ int4range(1,7)	t
<@	element is contained by	42 <@ int4range(1,7)	f
&&	overlap (have points in common)	int8range(3,7) && int8range(4,12)	t
<<	strictly left of	int8range(1,10) << int8range(100,110)	t
>>	strictly right of	int8range(50,60) >> int8range(20,30)	t
&<	does not extend to the right of	int8range(1,20) &< int8range(18,20)	t
&>	does not extend to the left of	int8range(7,20) &> int8range(5,10)	t
-|-	is adjacent to	numrange(1.1,2.2) -|- numrange(2.2,3.3)	t
+	union	numrange(5,15) + numrange(10,20)	[5,20)
*	intersection	int8range(5,15) * int8range(10,20)	[10,15)

Operator	Description	Example	Result
-	difference	int8range(5,15) - int8range(10,20)	[5,10)

The simple comparison operators <, >, <=, and >= compare the lower bounds first, and only if those are equal, compare the upper bounds. These comparisons are not usually very useful for ranges, but are provided to allow B-tree indexes to be constructed on ranges.

The left-of/right-of/adjacent operators always return false when an empty range is involved; that is, an empty range is not considered to be either before or after any other range.

The union and difference operators will fail if the resulting range would need to contain two disjoint sub-ranges, as such a range cannot be represented.

Table 9-48 shows the functions available for use with range types.

Table 9-48. Range Functions

Function	Return Type	Description	Example	Result
lower(anyrange)	range's element type	lower bound of range	lower(numrange(1.1,2.2))	1.1
upper(anyrange)	range's element type	upper bound of range	upper(numrange(1.1,2.2))	2.2
isempty(anyrange)	boolean	is the range empty?	isempty(numrange(1.1,2.2))	false
lower_inc(anyrange)	boolean	is the lower bound inclusive?	lower_inc(numrange(1.1,2.2))	true
upper_inc(anyrange)	boolean	is the upper bound inclusive?	upper_inc(numrange(1.1,2.2))	false
lower_inf(anyrange)	boolean	is the lower bound infinite?	lower_inf('(,)'::daterange)	true
upper_inf(anyrange)	boolean	is the upper bound infinite?	upper_inf('(,)'::daterange)	true
range_merge(anyrange, anyrange)	anyrange	the smallest range which includes both of the given ranges	range_merge('[1,2)'::int4range, '[3,4)'::int4range)	[1,4)

The lower and upper functions return null if the range is empty or the requested bound is infinite. The lower_inc, upper_inc, lower_inf, and upper_inf functions all return false for an empty range.

9.20. Aggregate Functions

Aggregate functions compute a single result from a set of input values. The built-in normal aggregate functions are listed in Table 9-49 and Table 9-50. The built-in ordered-set aggregate functions are listed in Table 9-51 and Table 9-52. Grouping operations, which are closely related to aggregate functions, are listed in Table 9-53. The special syntax considerations for aggregate functions are explained in Section 4.2.7. Consult Section 2.7 for additional introductory information.

Table 9-49. General-Purpose Aggregate Functions

Function	Argument Type(s)	Return Type	Description
`array_agg(`*`expression`*`)`	any non-array type	array of the argument type	input values, including nulls, concatenated into an array
`array_agg(`*`expression`*`)`	any array type	same as argument data type	input arrays concatenated into array of one higher dimension (inputs must all have same dimensionality, and cannot be empty or null)
`avg(`*`expression`*`)`	`smallint`, `int`, `bigint`, `real`, `double precision`, `numeric`, or `interval`	`numeric` for any integer-type argument, `double precision` for a floating-point argument, otherwise the same as the argument data type	the average (arithmetic mean) of all non-null input values
`bit_and(`*`expression`*`)`	`smallint`, `int`, `bigint`, or `bit`	same as argument data type	the bitwise AND of all non-null input values, or null if none
`bit_or(`*`expression`*`)`	`smallint`, `int`, `bigint`, or `bit`	same as argument data type	the bitwise OR of all non-null input values, or null if none
`bool_and(`*`expression`*`)`	`bool`	`bool`	true if all input values are true, otherwise false
`bool_or(`*`expression`*`)`	`bool`	`bool`	true if at least one input value is true, otherwise false
`count(*)`		`bigint`	number of input rows
`count(`*`expression`*`)`	any	`bigint`	number of input rows for which the value of *`expression`* is not null
`every(`*`expression`*`)`	`bool`	`bool`	equivalent to `bool_and`

Function	Argument Type(s)	Return Type	Description
json_agg(*expression*)	any	json	aggregates values, including nulls, as a JSON array
jsonb_agg(*expression*)	any	jsonb	aggregates values, including nulls, as a JSON array
json_object_agg(*name*, *value*)	(any, any)	json	aggregates name/value pairs as a JSON object; values can be null, but not names
jsonb_object_agg(*name*, *value*)	(any, any)	jsonb	aggregates name/value pairs as a JSON object; values can be null, but not names
max(*expression*)	any numeric, string, date/time, network, or enum type, or arrays of these types	same as argument type	maximum value of *expression* across all non-null input values
min(*expression*)	any numeric, string, date/time, network, or enum type, or arrays of these types	same as argument type	minimum value of *expression* across all non-null input values
string_agg(*expression*, *delimiter*)	(text, text) or (bytea, bytea)	same as argument types	non-null input values concatenated into a string, separated by delimiter
sum(*expression*)	smallint, int, bigint, real, double precision, numeric, interval, or money	bigint for smallint or int arguments, numeric for bigint arguments, otherwise the same as the argument data type	sum of *expression* across all non-null input values
xmlagg(*expression*)	xml	xml	concatenation of non-null XML values (see also Section 9.14.1.7)

It should be noted that except for count, these functions return a null value when no rows are selected. In particular, sum of no rows returns null, not zero as one might expect, and array_agg returns null rather than an empty array when there are no input rows. The coalesce function can be used to substitute zero or an empty array for null when necessary.

Note: Boolean aggregates bool_and and bool_or correspond to standard SQL aggregates every and any or some. As for any and some, it seems that there is an ambiguity built into the standard syntax:

```
SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;
```

Here `ANY` can be considered either as introducing a subquery, or as being an aggregate function, if the subquery returns one row with a Boolean value. Thus the standard name cannot be given to these aggregates.

Note: Users accustomed to working with other SQL database management systems might be disappointed by the performance of the `count` aggregate when it is applied to the entire table. A query like:

```
SELECT count(*) FROM sometable;
```

will require effort proportional to the size of the table: PostgreSQL will need to scan either the entire table or the entirety of an index which includes all rows in the table.

The aggregate functions `array_agg`, `json_agg`, `jsonb_agg`, `json_object_agg`, `jsonb_object_agg`, `string_agg`, and `xmlagg`, as well as similar user-defined aggregate functions, produce meaningfully different result values depending on the order of the input values. This ordering is unspecified by default, but can be controlled by writing an `ORDER BY` clause within the aggregate call, as shown in Section 4.2.7. Alternatively, supplying the input values from a sorted subquery will usually work. For example:

```
SELECT xmlagg(x) FROM (SELECT x FROM test ORDER BY y DESC) AS tab;
```

But this syntax is not allowed in the SQL standard, and is not portable to other database systems.

Table 9-50 shows aggregate functions typically used in statistical analysis. (These are separated out merely to avoid cluttering the listing of more-commonly-used aggregates.) Where the description mentions *N*, it means the number of input rows for which all the input expressions are non-null. In all cases, null is returned if the computation is meaningless, for example when *N* is zero.

Table 9-50. Aggregate Functions for Statistics

Function	Argument Type	Return Type	Description
corr(*Y*, *X*)	double precision	double precision	correlation coefficient
covar_pop(*Y*, *X*)	double precision	double precision	population covariance
covar_samp(*Y*, *X*)	double precision	double precision	sample covariance
regr_avgx(*Y*, *X*)	double precision	double precision	average of the independent variable (sum(*X*)/*N*)
regr_avgy(*Y*, *X*)	double precision	double precision	average of the dependent variable (sum(*Y*)/*N*)
regr_count(*Y*, *X*)	double precision	bigint	number of input rows in which both expressions are nonnull

Function	Argument Type	Return Type	Description
regr_intercept(*Y*, *X*)	double precision	double precision	y-intercept of the least-squares-fit linear equation determined by the (*X*, *Y*) pairs
regr_r2(*Y*, *X*)	double precision	double precision	square of the correlation coefficient
regr_slope(*Y*, *X*)	double precision	double precision	slope of the least-squares-fit linear equation determined by the (*X*, *Y*) pairs
regr_sxx(*Y*, *X*)	double precision	double precision	sum(*X*^2) - sum(*X*)^2/*N* ("sum of squares" of the independent variable)
regr_sxy(*Y*, *X*)	double precision	double precision	sum(*X***Y*) - sum(*X*) * sum(*Y*)/*N* ("sum of products" of independent times dependent variable)
regr_syy(*Y*, *X*)	double precision	double precision	sum(*Y*^2) - sum(*Y*)^2/*N* ("sum of squares" of the dependent variable)
stddev(*expression*)	smallint, int, bigint, real, double precision, or numeric	double precision for floating-point arguments, otherwise numeric	historical alias for stddev_samp
stddev_pop(*expression*)	smallint, int, bigint, real, double precision, or numeric	double precision for floating-point arguments, otherwise numeric	population standard deviation of the input values
stddev_samp(*expression*)	smallint, int, bigint, real, double precision, or numeric	double precision for floating-point arguments, otherwise numeric	sample standard deviation of the input values
variance(*expression*)	smallint, int, bigint, real, double precision, or numeric	double precision for floating-point arguments, otherwise numeric	historical alias for var_samp
var_pop(*expression*)	smallint, int, bigint, real, double precision, or numeric	double precision for floating-point arguments, otherwise numeric	population variance of the input values (square of the population standard deviation)

Function	Argument Type	Return Type	Description
var_samp(*expression*)	smallint, int, bigint, real, double precision, or numeric	double precision for floating-point arguments, otherwise numeric	sample variance of the input values (square of the sample standard deviation)

Table 9-51 shows some aggregate functions that use the *ordered-set aggregate* syntax. These functions are sometimes referred to as "inverse distribution" functions.

Table 9-51. Ordered-Set Aggregate Functions

Function	Direct Argument Type(s)	Aggregated Argument Type(s)	Return Type	Description
mode() WITHIN GROUP (ORDER BY *sort_expression*)		any sortable type	same as sort expression	returns the most frequent input value (arbitrarily choosing the first one if there are multiple equally-frequent results)
percentile_cont(*fraction*) WITHIN GROUP (ORDER BY *sort_expression*)	double precision	double precision or interval	same as sort expression	continuous percentile: returns a value corresponding to the specified fraction in the ordering, interpolating between adjacent input items if needed
percentile_cont(*fractions*) WITHIN GROUP (ORDER BY *sort_expression*)	double precision[]	double precision or interval	array of sort expression's type	multiple continuous percentile: returns an array of results matching the shape of the fractions parameter, with each non-null element replaced by the value corresponding to that percentile

Function	Direct Argument Type(s)	Aggregated Argument Type(s)	Return Type	Description
percentile_disc(*fraction*) WITHIN GROUP (ORDER BY *sort_expression*)	double precision	any sortable type	same as sort expression	discrete percentile: returns the first input value whose position in the ordering equals or exceeds the specified fraction
percentile_disc(*fractions*) WITHIN GROUP (ORDER BY *sort_expression*)	double precision[]	any sortable type	array of sort expression's type	multiple discrete percentile: returns an array of results matching the shape of the `fractions` parameter, with each non-null element replaced by the input value corresponding to that percentile

All the aggregates listed in Table 9-51 ignore null values in their sorted input. For those that take a *fraction* parameter, the fraction value must be between 0 and 1; an error is thrown if not. However, a null fraction value simply produces a null result.

Each of the aggregates listed in Table 9-52 is associated with a window function of the same name defined in Section 9.21. In each case, the aggregate result is the value that the associated window function would have returned for the "hypothetical" row constructed from *args*, if such a row had been added to the sorted group of rows computed from the *sorted_args*.

Table 9-52. Hypothetical-Set Aggregate Functions

Function	Direct Argument Type(s)	Aggregated Argument Type(s)	Return Type	Description
rank(*args*) WITHIN GROUP (ORDER BY *sorted_args*)	VARIADIC "any"	VARIADIC "any"	bigint	rank of the hypothetical row, with gaps for duplicate rows
dense_rank(*args*) WITHIN GROUP (ORDER BY *sorted_args*)	VARIADIC "any"	VARIADIC "any"	bigint	rank of the hypothetical row, without gaps

Function	Direct Argument Type(s)	Aggregated Argument Type(s)	Return Type	Description
percent_rank(*args*) WITHIN GROUP (ORDER BY *sorted_args*)	VARIADIC "any"	VARIADIC "any"	double precision	relative rank of the hypothetical row, ranging from 0 to 1
cume_dist(*args*) WITHIN GROUP (ORDER BY *sorted_args*)	VARIADIC "any"	VARIADIC "any"	double precision	relative rank of the hypothetical row, ranging from 1/*N* to 1

For each of these hypothetical-set aggregates, the list of direct arguments given in *args* must match the number and types of the aggregated arguments given in *sorted_args*. Unlike most built-in aggregates, these aggregates are not strict, that is they do not drop input rows containing nulls. Null values sort according to the rule specified in the ORDER BY clause.

Table 9-53. Grouping Operations

Function	Return Type	Description
GROUPING(*args...*)	integer	Integer bit mask indicating which arguments are not being included in the current grouping set

Grouping operations are used in conjunction with grouping sets (see Section 7.2.4) to distinguish result rows. The arguments to the GROUPING operation are not actually evaluated, but they must match exactly expressions given in the GROUP BY clause of the associated query level. Bits are assigned with the rightmost argument being the least-significant bit; each bit is 0 if the corresponding expression is included in the grouping criteria of the grouping set generating the result row, and 1 if it is not. For example:

```
=> SELECT * FROM items_sold;
 make  | model | sales
-------+-------+-------
 Foo   | GT    |  10
 Foo   | Tour  |  20
 Bar   | City  |  15
 Bar   | Sport |  5
(4 rows)

=> SELECT make, model, GROUPING(make,model), sum(sales) FROM items_sold GROUP BY ROLLUP(make,model);
 make  | model | grouping | sum
-------+-------+----------+-----
 Foo   | GT    |        0 | 10
 Foo   | Tour  |        0 | 20
 Bar   | City  |        0 | 15
 Bar   | Sport |        0 | 5
```

```
 Foo    |        |         1 | 30
 Bar    |        |         1 | 20
        |        |         3 | 50
(7 rows)
```

9.21. Window Functions

Window functions provide the ability to perform calculations across sets of rows that are related to the current query row. See Section 3.5 for an introduction to this feature, and Section 4.2.8 for syntax details.

The built-in window functions are listed in Table 9-54. Note that these functions *must* be invoked using window function syntax; that is an `OVER` clause is required.

In addition to these functions, any built-in or user-defined normal aggregate function (but not ordered-set or hypothetical-set aggregates) can be used as a window function; see Section 9.20 for a list of the built-in aggregates. Aggregate functions act as window functions only when an `OVER` clause follows the call; otherwise they act as regular aggregates.

Table 9-54. General-Purpose Window Functions

Function	Return Type	Description
`row_number()`	`bigint`	number of the current row within its partition, counting from 1
`rank()`	`bigint`	rank of the current row with gaps; same as `row_number` of its first peer
`dense_rank()`	`bigint`	rank of the current row without gaps; this function counts peer groups
`percent_rank()`	`double precision`	relative rank of the current row: (`rank` - 1) / (total rows - 1)
`cume_dist()`	`double precision`	relative rank of the current row: (number of rows preceding or peer with current row) / (total rows)
`ntile(`*`num_buckets`* `integer)`	`integer`	integer ranging from 1 to the argument value, dividing the partition as equally as possible

Function	Return Type	Description
lag(*value* anyelement [, *offset* integer [, *default* anyelement]])	same type as *value*	returns *value* evaluated at the row that is *offset* rows before the current row within the partition; if there is no such row, instead return *default* (which must be of the same type as *value*). Both *offset* and *default* are evaluated with respect to the current row. If omitted, *offset* defaults to 1 and *default* to null
lead(*value* anyelement [, *offset* integer [, *default* anyelement]])	same type as *value*	returns *value* evaluated at the row that is *offset* rows after the current row within the partition; if there is no such row, instead return *default* (which must be of the same type as *value*). Both *offset* and *default* are evaluated with respect to the current row. If omitted, *offset* defaults to 1 and *default* to null
first_value(*value* any)	same type as *value*	returns *value* evaluated at the row that is the first row of the window frame
last_value(*value* any)	same type as *value*	returns *value* evaluated at the row that is the last row of the window frame
nth_value(*value* any, *nth* integer)	same type as *value*	returns *value* evaluated at the row that is the *nth* row of the window frame (counting from 1); null if no such row

All of the functions listed in Table 9-54 depend on the sort ordering specified by the ORDER BY clause of the associated window definition. Rows that are not distinct in the ORDER BY ordering are said to be *peers*; the four ranking functions are defined so that they give the same answer for any two peer rows.

Note that first_value, last_value, and nth_value consider only the rows within the "window frame", which by default contains the rows from the start of the partition through the last peer of the current row. This is likely to give unhelpful results for last_value and sometimes also nth_value. You can redefine the frame by adding a suitable frame specification (RANGE or ROWS) to the OVER clause. See Section 4.2.8 for more information about frame specifications.

When an aggregate function is used as a window function, it aggregates over the rows within the current row's window frame. An aggregate used with ORDER BY and the default window frame definition produces a "running sum" type of behavior, which may or may not be what's wanted. To obtain aggregation over the whole partition, omit ORDER BY or use ROWS BETWEEN UNBOUNDED PRECEDING AND

UNBOUNDED FOLLOWING. Other frame specifications can be used to obtain other effects.

> **Note:** The SQL standard defines a RESPECT NULLS or IGNORE NULLS option for lead, lag, first_value, last_value, and nth_value. This is not implemented in PostgreSQL: the behavior is always the same as the standard's default, namely RESPECT NULLS. Likewise, the standard's FROM FIRST or FROM LAST option for nth_value is not implemented: only the default FROM FIRST behavior is supported. (You can achieve the result of FROM LAST by reversing the ORDER BY ordering.)

9.22. Subquery Expressions

This section describes the SQL-compliant subquery expressions available in PostgreSQL. All of the expression forms documented in this section return Boolean (true/false) results.

9.22.1. EXISTS

```
EXISTS (subquery)
```

The argument of EXISTS is an arbitrary SELECT statement, or *subquery*. The subquery is evaluated to determine whether it returns any rows. If it returns at least one row, the result of EXISTS is "true"; if the subquery returns no rows, the result of EXISTS is "false".

The subquery can refer to variables from the surrounding query, which will act as constants during any one evaluation of the subquery.

The subquery will generally only be executed long enough to determine whether at least one row is returned, not all the way to completion. It is unwise to write a subquery that has side effects (such as calling sequence functions); whether the side effects occur might be unpredictable.

Since the result depends only on whether any rows are returned, and not on the contents of those rows, the output list of the subquery is normally unimportant. A common coding convention is to write all EXISTS tests in the form EXISTS(SELECT 1 WHERE ...). There are exceptions to this rule however, such as subqueries that use INTERSECT.

This simple example is like an inner join on col2, but it produces at most one output row for each tab1 row, even if there are several matching tab2 rows:

```
SELECT col1
FROM tab1
WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

9.22.2. IN

```
expression IN (subquery)
```

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of IN is "true" if any equal subquery row is found. The result is "false" if no equal row is found (including the case where the subquery returns no rows).

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the IN construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

row_constructor IN (*subquery*)

The left-hand side of this form of IN is a row constructor, as described in Section 4.2.13. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result. The result of IN is "true" if any equal subquery row is found. The result is "false" if no equal row is found (including the case where the subquery returns no rows).

As usual, null values in the rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is unknown (null). If all the per-row results are either unequal or null, with at least one null, then the result of IN is null.

9.22.3. NOT IN

expression NOT IN (*subquery*)

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result. The result of NOT IN is "true" if only unequal subquery rows are found (including the case where the subquery returns no rows). The result is "false" if any equal row is found.

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand row yields null, the result of the NOT IN construct will be null, not true. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

row_constructor NOT IN (*subquery*)

The left-hand side of this form of NOT IN is a row constructor, as described in Section 4.2.13. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result. The result of NOT IN is "true" if only unequal subquery rows are found (including the case where the subquery returns no rows). The result is "false" if any equal row is found.

As usual, null values in the rows are combined per the normal rules of SQL Boolean expressions. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of that row comparison is

unknown (null). If all the per-row results are either unequal or null, with at least one null, then the result of NOT IN is null.

9.22.4. ANY/SOME

```
expression operator ANY (subquery)
expression operator SOME (subquery)
```

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given *operator*, which must yield a Boolean result. The result of ANY is "true" if any true result is obtained. The result is "false" if no true result is found (including the case where the subquery returns no rows).

SOME is a synonym for ANY. IN is equivalent to = ANY.

Note that if there are no successes and at least one right-hand row yields null for the operator's result, the result of the ANY construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

```
row_constructor operator ANY (subquery)
row_constructor operator SOME (subquery)
```

The left-hand side of this form of ANY is a row constructor, as described in Section 4.2.13. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given *operator*. The result of ANY is "true" if the comparison returns true for any subquery row. The result is "false" if the comparison returns false for every subquery row (including the case where the subquery returns no rows). The result is NULL if no comparison with a subquery row returns true, and at least one comparison returns NULL.

See Section 9.23.5 for details about the meaning of a row constructor comparison.

9.22.5. ALL

```
expression operator ALL (subquery)
```

The right-hand side is a parenthesized subquery, which must return exactly one column. The left-hand expression is evaluated and compared to each row of the subquery result using the given *operator*, which must yield a Boolean result. The result of ALL is "true" if all rows yield true (including the case where the subquery returns no rows). The result is "false" if any false result is found. The result is NULL if no comparison with a subquery row returns false, and at least one comparison returns NULL.

NOT IN is equivalent to <> ALL.

As with EXISTS, it's unwise to assume that the subquery will be evaluated completely.

```
row_constructor operator ALL (subquery)
```

The left-hand side of this form of ALL is a row constructor, as described in Section 4.2.13. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. The left-hand expressions are evaluated and compared row-wise to each row of the subquery result, using the given *operator*. The result of ALL is "true" if the comparison returns true for all subquery rows (including the case where the subquery returns no rows). The result is "false" if the comparison returns false for any subquery row. The result is NULL if no comparison with a subquery row returns false, and at least one comparison returns NULL.

See Section 9.23.5 for details about the meaning of a row constructor comparison.

9.22.6. Single-row Comparison

```
row_constructor operator (subquery)
```

The left-hand side is a row constructor, as described in Section 4.2.13. The right-hand side is a parenthesized subquery, which must return exactly as many columns as there are expressions in the left-hand row. Furthermore, the subquery cannot return more than one row. (If it returns zero rows, the result is taken to be null.) The left-hand side is evaluated and compared row-wise to the single subquery result row.

See Section 9.23.5 for details about the meaning of a row constructor comparison.

9.23. Row and Array Comparisons

This section describes several specialized constructs for making multiple comparisons between groups of values. These forms are syntactically related to the subquery forms of the previous section, but do not involve subqueries. The forms involving array subexpressions are PostgreSQL extensions; the rest are SQL-compliant. All of the expression forms documented in this section return Boolean (true/false) results.

9.23.1. IN

```
expression IN (value [, ...])
```

The right-hand side is a parenthesized list of scalar expressions. The result is "true" if the left-hand expression's result is equal to any of the right-hand expressions. This is a shorthand notation for

```
expression = value1
OR
expression = value2
OR
...
```

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand expression yields null, the result of the IN construct will be null, not false. This is in accordance with SQL's normal rules for Boolean combinations of null values.

9.23.2. NOT IN

```
expression NOT IN (value [, ...])
```

The right-hand side is a parenthesized list of scalar expressions. The result is "true" if the left-hand expression's result is unequal to all of the right-hand expressions. This is a shorthand notation for

```
expression <> value1
AND
expression <> value2
AND
...
```

Note that if the left-hand expression yields null, or if there are no equal right-hand values and at least one right-hand expression yields null, the result of the NOT IN construct will be null, not true as one might naively expect. This is in accordance with SQL's normal rules for Boolean combinations of null values.

> **Tip:** x NOT IN y is equivalent to NOT (x IN y) in all cases. However, null values are much more likely to trip up the novice when working with NOT IN than when working with IN. It is best to express your condition positively if possible.

9.23.3. ANY/SOME (array)

```
expression operator ANY (array expression)
expression operator SOME (array expression)
```

The right-hand side is a parenthesized expression, which must yield an array value. The left-hand expression is evaluated and compared to each element of the array using the given *operator*, which must yield a Boolean result. The result of ANY is "true" if any true result is obtained. The result is "false" if no true result is found (including the case where the array has zero elements).

If the array expression yields a null array, the result of ANY will be null. If the left-hand expression yields null, the result of ANY is ordinarily null (though a non-strict comparison operator could possibly yield a different result). Also, if the right-hand array contains any null elements and no true comparison result is obtained, the result of ANY will be null, not false (again, assuming a strict comparison operator). This is in accordance with SQL's normal rules for Boolean combinations of null values.

SOME is a synonym for ANY.

9.23.4. ALL (array)

```
expression operator ALL (array expression)
```

The right-hand side is a parenthesized expression, which must yield an array value. The left-hand expression is evaluated and compared to each element of the array using the given *operator*, which must yield

a Boolean result. The result of ALL is "true" if all comparisons yield true (including the case where the array has zero elements). The result is "false" if any false result is found.

If the array expression yields a null array, the result of ALL will be null. If the left-hand expression yields null, the result of ALL is ordinarily null (though a non-strict comparison operator could possibly yield a different result). Also, if the right-hand array contains any null elements and no false comparison result is obtained, the result of ALL will be null, not true (again, assuming a strict comparison operator). This is in accordance with SQL's normal rules for Boolean combinations of null values.

9.23.5. Row Constructor Comparison

row_constructor operator row_constructor

Each side is a row constructor, as described in Section 4.2.13. The two row values must have the same number of fields. Each side is evaluated and they are compared row-wise. Row constructor comparisons are allowed when the *operator* is =, <>, <, <=, > or >=. Every row element must be of a type which has a default B-tree operator class or the attempted comparison may generate an error.

> **Note:** Errors related to the number or types of elements might not occur if the comparison is resolved using earlier columns.

The = and <> cases work slightly differently from the others. Two rows are considered equal if all their corresponding members are non-null and equal; the rows are unequal if any corresponding members are non-null and unequal; otherwise the result of the row comparison is unknown (null).

For the <, <=, > and >= cases, the row elements are compared left-to-right, stopping as soon as an unequal or null pair of elements is found. If either of this pair of elements is null, the result of the row comparison is unknown (null); otherwise comparison of this pair of elements determines the result. For example, `ROW(1,2,NULL) < ROW(1,3,0)` yields true, not null, because the third pair of elements are not considered.

> **Note:** Prior to PostgreSQL 8.2, the <, <=, > and >= cases were not handled per SQL specification. A comparison like `ROW(a,b) < ROW(c,d)` was implemented as `a < c AND b < d` whereas the correct behavior is equivalent to `a < c OR (a = c AND b < d)`.

row_constructor IS DISTINCT FROM *row_constructor*

This construct is similar to a <> row comparison, but it does not yield null for null inputs. Instead, any null value is considered unequal to (distinct from) any non-null value, and any two nulls are considered equal (not distinct). Thus the result will either be true or false, never null.

row_constructor IS NOT DISTINCT FROM *row_constructor*

This construct is similar to a = row comparison, but it does not yield null for null inputs. Instead, any null value is considered unequal to (distinct from) any non-null value, and any two nulls are considered equal (not distinct). Thus the result will always be either true or false, never null.

9.23.6. Composite Type Comparison

record operator record

The SQL specification requires row-wise comparison to return NULL if the result depends on comparing two NULL values or a NULL and a non-NULL. PostgreSQL does this only when comparing the results of two row constructors (as in Section 9.23.5) or comparing a row constructor to the output of a subquery (as in Section 9.22). In other contexts where two composite-type values are compared, two NULL field values are considered equal, and a NULL is considered larger than a non-NULL. This is necessary in order to have consistent sorting and indexing behavior for composite types.

Each side is evaluated and they are compared row-wise. Composite type comparisons are allowed when the *operator* is =, <>, <, <=, > or >=, or has semantics similar to one of these. (To be specific, an operator can be a row comparison operator if it is a member of a B-tree operator class, or is the negator of the = member of a B-tree operator class.) The default behavior of the above operators is the same as for `IS [ NOT ] DISTINCT FROM` for row constructors (see Section 9.23.5).

To support matching of rows which include elements without a default B-tree operator class, the following operators are defined for composite type comparison: *=, *<>, *<, *<=, *>, and *>=. These operators compare the internal binary representation of the two rows. Two rows might have a different binary representation even though comparisons of the two rows with the equality operator is true. The ordering of rows under these comparison operators is deterministic but not otherwise meaningful. These operators are used internally for materialized views and might be useful for other specialized purposes such as replication but are not intended to be generally useful for writing queries.

9.24. Set Returning Functions

This section describes functions that possibly return more than one row. The most widely used functions in this class are series generating functions, as detailed in Table 9-55 and Table 9-56. Other, more specialized set-returning functions are described elsewhere in this manual. See Section 7.2.1.4 for ways to combine multiple set-returning functions.

Table 9-55. Series Generating Functions

Function	Argument Type	Return Type	Description
`generate_series(start, stop)`	`int`, `bigint` or `numeric`	`setof int`, `setof bigint`, or `setof numeric` (same as argument type)	Generate a series of values, from `start` to `stop` with a step size of one
`generate_series(start, stop, step)`	`int`, `bigint` or `numeric`	`setof int`, `setof bigint` or `setof numeric` (same as argument type)	Generate a series of values, from `start` to `stop` with a step size of `step`

Function	Argument Type	Return Type	Description
generate_series(start, stop, step interval)	timestamp or timestamp with time zone	setof timestamp or setof timestamp with time zone (same as argument type)	Generate a series of values, from start to stop with a step size of step

When step is positive, zero rows are returned if start is greater than stop. Conversely, when step is negative, zero rows are returned if start is less than stop. Zero rows are also returned for NULL inputs. It is an error for step to be zero. Some examples follow:

```
SELECT * FROM generate_series(2,4);
 generate_series
-----------------
               2
               3
               4
(3 rows)

SELECT * FROM generate_series(5,1,-2);
 generate_series
-----------------
               5
               3
               1
(3 rows)

SELECT * FROM generate_series(4,3);
 generate_series
-----------------
(0 rows)

SELECT generate_series(1.1, 4, 1.3);
 generate_series
-----------------
             1.1
             2.4
             3.7
(3 rows)

-- this example relies on the date-plus-integer operator
SELECT current_date + s.a AS dates FROM generate_series(0,14,7) AS s(a);
   dates
------------
 2004-02-05
 2004-02-12
 2004-02-19
(3 rows)

SELECT * FROM generate_series('2008-03-01 00:00'::timestamp,
                              '2008-03-04 12:00', '10 hours');
   generate_series
```

```
---------------------
 2008-03-01 00:00:00
 2008-03-01 10:00:00
 2008-03-01 20:00:00
 2008-03-02 06:00:00
 2008-03-02 16:00:00
 2008-03-03 02:00:00
 2008-03-03 12:00:00
 2008-03-03 22:00:00
 2008-03-04 08:00:00
(9 rows)
```

Table 9-56. Subscript Generating Functions

Function	Return Type	Description
`generate_subscripts(array anyarray, dim int)`	`setof int`	Generate a series comprising the given array's subscripts.
`generate_subscripts(array anyarray, dim int, reverse boolean)`	`setof int`	Generate a series comprising the given array's subscripts. When `reverse` is true, the series is returned in reverse order.

`generate_subscripts` is a convenience function that generates the set of valid subscripts for the specified dimension of the given array. Zero rows are returned for arrays that do not have the requested dimension, or for NULL arrays (but valid subscripts are returned for NULL array elements). Some examples follow:

```
-- basic usage
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;
 s
---
 1
 2
 3
 4
(4 rows)

-- presenting an array, the subscript and the subscripted
-- value requires a subquery
SELECT * FROM arrays;
         a
--------------------
 {-1,-2}
 {100,200,300}
(2 rows)

SELECT a AS array, s AS subscript, a[s] AS value
FROM (SELECT generate_subscripts(a, 1) AS s, a FROM arrays) foo;
     array     | subscript | value
```

```
---------------+-----------+-------
 {-1,-2}       |         1 |    -1
 {-1,-2}       |         2 |    -2
 {100,200,300} |         1 |   100
 {100,200,300} |         2 |   200
 {100,200,300} |         3 |   300
(5 rows)

-- unnest a 2D array
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
select $1[i][j]
   from generate_subscripts($1,1) g1(i),
        generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
 unnest2
---------
       1
       2
       3
       4
(4 rows)
```

When a function in the `FROM` clause is suffixed by `WITH ORDINALITY`, a `bigint` column is appended to the output which starts from 1 and increments by 1 for each row of the function's output. This is most useful in the case of set returning functions such as `unnest()`.

```
-- set returning function WITH ORDINALITY
SELECT * FROM pg_ls_dir('.') WITH ORDINALITY AS t(ls,n);
       ls        | n
-----------------+----
 pg_serial       |  1
 pg_twophase     |  2
 postmaster.opts |  3
 pg_notify       |  4
 postgresql.conf |  5
 pg_tblspc       |  6
 logfile         |  7
 base            |  8
 postmaster.pid  |  9
 pg_ident.conf   | 10
 global          | 11
 pg_clog         | 12
 pg_snapshots    | 13
 pg_multixact    | 14
 PG_VERSION      | 15
 pg_xlog         | 16
 pg_hba.conf     | 17
 pg_stat_tmp     | 18
```

```
 pg_subtrans     | 19
(19 rows)
```

9.25. System Information Functions

Table 9-57 shows several functions that extract session and system information.

In addition to the functions listed in this section, there are a number of functions related to the statistics system that also provide system information. See Section 27.2.2 for more information.

Table 9-57. Session Information Functions

Name	Return Type	Description
`current_catalog`	`name`	name of current database (called "catalog" in the SQL standard)
`current_database()`	`name`	name of current database
`current_query()`	`text`	text of the currently executing query, as submitted by the client (might contain more than one statement)
`current_role`	`name`	equivalent to `current_user`
`current_schema[()]`	`name`	name of current schema
`current_schemas(boolean)`	`name[]`	names of schemas in search path, optionally including implicit schemas
`current_user`	`name`	user name of current execution context
`inet_client_addr()`	`inet`	address of the remote connection
`inet_client_port()`	`int`	port of the remote connection
`inet_server_addr()`	`inet`	address of the local connection
`inet_server_port()`	`int`	port of the local connection
`pg_backend_pid()`	`int`	Process ID of the server process attached to the current session
`pg_conf_load_time()`	`timestamp with time zone`	configuration load time
`pg_is_other_temp_schema(oid)`	`boolean`	is schema another session's temporary schema?
`pg_listening_channels()`	`setof text`	channel names that the session is currently listening on
`pg_my_temp_schema()`	`oid`	OID of session's temporary schema, or 0 if none
`pg_postmaster_start_time()`	`timestamp with time zone`	server start time

Name	Return Type	Description
pg_trigger_depth()	int	current nesting level of PostgreSQL triggers (0 if not called, directly or indirectly, from inside a trigger)
session_user	name	session user name
user	name	equivalent to current_user
version()	text	PostgreSQL version information. See also server_version_num for a machine-readable version.

Note: current_catalog, current_role, current_schema, current_user, session_user, and user have special syntactic status in SQL: they must be called without trailing parentheses. (In PostgreSQL, parentheses can optionally be used with current_schema, but not with the others.)

The session_user is normally the user who initiated the current database connection; but superusers can change this setting with SET SESSION AUTHORIZATION. The current_user is the user identifier that is applicable for permission checking. Normally it is equal to the session user, but it can be changed with SET ROLE. It also changes during the execution of functions with the attribute SECURITY DEFINER. In Unix parlance, the session user is the "real user" and the current user is the "effective user". current_role and user are synonyms for current_user. (The SQL standard draws a distinction between current_role and current_user, but PostgreSQL does not, since it unifies users and roles into a single kind of entity.)

current_schema returns the name of the schema that is first in the search path (or a null value if the search path is empty). This is the schema that will be used for any tables or other named objects that are created without specifying a target schema. current_schemas(boolean) returns an array of the names of all schemas presently in the search path. The Boolean option determines whether or not implicitly included system schemas such as pg_catalog are included in the returned search path.

Note: The search path can be altered at run time. The command is:

```
SET search_path TO schema [, schema, ...]
```

pg_listening_channels returns a set of names of channels that the current session is listening to. See LISTEN for more information.

inet_client_addr returns the IP address of the current client, and inet_client_port returns the port number. inet_server_addr returns the IP address on which the server accepted the current connection, and inet_server_port returns the port number. All these functions return NULL if the current connection is via a Unix-domain socket.

pg_my_temp_schema returns the OID of the current session's temporary schema, or zero if it has none (because it has not created any temporary tables). pg_is_other_temp_schema returns true if the given

OID is the OID of another session's temporary schema. (This can be useful, for example, to exclude other sessions' temporary tables from a catalog display.)

`pg_postmaster_start_time` returns the `timestamp with time zone` when the server started.

`pg_conf_load_time` returns the `timestamp with time zone` when the server configuration files were last loaded. (If the current session was alive at the time, this will be the time when the session itself re-read the configuration files, so the reading will vary a little in different sessions. Otherwise it is the time when the postmaster process re-read the configuration files.)

`version` returns a string describing the PostgreSQL server's version. You can also get this information from server_version or for a machine-readable version, server_version_num. Software developers should use `server_version_num` (available since 8.2) or *`PQserverVersion`* instead of parsing the text version.

Table 9-58 lists functions that allow the user to query object access privileges programmatically. See Section 5.6 for more information about privileges.

Table 9-58. Access Privilege Inquiry Functions

Name	Return Type	Description
`has_any_column_privilege(user, table, privilege)`	`boolean`	does user have privilege for any column of table
`has_any_column_privilege(table, privilege)`	`boolean`	does current user have privilege for any column of table
`has_column_privilege(user, table, column, privilege)`	`boolean`	does user have privilege for column
`has_column_privilege(table, column, privilege)`	`boolean`	does current user have privilege for column
`has_database_privilege(user, database, privilege)`	`boolean`	does user have privilege for database
`has_database_privilege(database, privilege)`	`boolean`	does current user have privilege for database
`has_foreign_data_wrapper_privilege(user, fdw, privilege)`	`boolean`	does user have privilege for foreign-data wrapper
`has_foreign_data_wrapper_privilege(fdw, privilege)`	`boolean`	does current user have privilege for foreign-data wrapper
`has_function_privilege(user, function, privilege)`	`boolean`	does user have privilege for function
`has_function_privilege(function, privilege)`	`boolean`	does current user have privilege for function
`has_language_privilege(user, language, privilege)`	`boolean`	does user have privilege for language
`has_language_privilege(language, privilege)`	`boolean`	does current user have privilege for language
`has_schema_privilege(user, schema, privilege)`	`boolean`	does user have privilege for schema

Name	Return Type	Description
has_schema_privilege(schema, privilege)	boolean	does current user have privilege for schema
has_sequence_privilege(user, sequence, privilege)	boolean	does user have privilege for sequence
has_sequence_privilege(sequence, privilege)	boolean	does current user have privilege for sequence
has_server_privilege(user, server, privilege)	boolean	does user have privilege for foreign server
has_server_privilege(server, privilege)	boolean	does current user have privilege for foreign server
has_table_privilege(user, table, privilege)	boolean	does user have privilege for table
has_table_privilege(table, privilege)	boolean	does current user have privilege for table
has_tablespace_privilege(user, tablespace, privilege)	boolean	does user have privilege for tablespace
has_tablespace_privilege(tablespace, privilege)	boolean	does current user have privilege for tablespace
has_type_privilege(user, type, privilege)	boolean	does user have privilege for type
has_type_privilege(type, privilege)	boolean	does current user have privilege for type
pg_has_role(user, role, privilege)	boolean	does user have privilege for role
pg_has_role(role, privilege)	boolean	does current user have privilege for role
row_security_active(table)	boolean	does current user have row level security active for table

has_table_privilege checks whether a user can access a table in a particular way. The user can be specified by name, by OID (pg_authid.oid), public to indicate the PUBLIC pseudo-role, or if the argument is omitted current_user is assumed. The table can be specified by name or by OID. (Thus, there are actually six variants of has_table_privilege, which can be distinguished by the number and types of their arguments.) When specifying by name, the name can be schema-qualified if necessary. The desired access privilege type is specified by a text string, which must evaluate to one of the values SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, or TRIGGER. Optionally, WITH GRANT OPTION can be added to a privilege type to test whether the privilege is held with grant option. Also, multiple privilege types can be listed separated by commas, in which case the result will be true if any of the listed privileges is held. (Case of the privilege string is not significant, and extra whitespace is allowed between but not within privilege names.) Some examples:

```
SELECT has_table_privilege('myschema.mytable', 'select');
SELECT has_table_privilege('joe', 'mytable', 'INSERT, SELECT WITH GRANT OPTION');
```

has_sequence_privilege checks whether a user can access a sequence in a particular way. The possibilities for its arguments are analogous to has_table_privilege. The desired access privilege type must evaluate to one of USAGE, SELECT, or UPDATE.

has_any_column_privilege checks whether a user can access any column of a table in a particular way. Its argument possibilities are analogous to has_table_privilege, except that the desired access privilege type must evaluate to some combination of SELECT, INSERT, UPDATE, or REFERENCES. Note that having any of these privileges at the table level implicitly grants it for each column of the table, so has_any_column_privilege will always return true if has_table_privilege does for the same arguments. But has_any_column_privilege also succeeds if there is a column-level grant of the privilege for at least one column.

has_column_privilege checks whether a user can access a column in a particular way. Its argument possibilities are analogous to has_table_privilege, with the addition that the column can be specified either by name or attribute number. The desired access privilege type must evaluate to some combination of SELECT, INSERT, UPDATE, or REFERENCES. Note that having any of these privileges at the table level implicitly grants it for each column of the table.

has_database_privilege checks whether a user can access a database in a particular way. Its argument possibilities are analogous to has_table_privilege. The desired access privilege type must evaluate to some combination of CREATE, CONNECT, TEMPORARY, or TEMP (which is equivalent to TEMPORARY).

has_function_privilege checks whether a user can access a function in a particular way. Its argument possibilities are analogous to has_table_privilege. When specifying a function by a text string rather than by OID, the allowed input is the same as for the regprocedure data type (see Section 8.18). The desired access privilege type must evaluate to EXECUTE. An example is:

```
SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');
```

has_foreign_data_wrapper_privilege checks whether a user can access a foreign-data wrapper in a particular way. Its argument possibilities are analogous to has_table_privilege. The desired access privilege type must evaluate to USAGE.

has_language_privilege checks whether a user can access a procedural language in a particular way. Its argument possibilities are analogous to has_table_privilege. The desired access privilege type must evaluate to USAGE.

has_schema_privilege checks whether a user can access a schema in a particular way. Its argument possibilities are analogous to has_table_privilege. The desired access privilege type must evaluate to some combination of CREATE or USAGE.

has_server_privilege checks whether a user can access a foreign server in a particular way. Its argument possibilities are analogous to has_table_privilege. The desired access privilege type must evaluate to USAGE.

has_tablespace_privilege checks whether a user can access a tablespace in a particular way. Its argument possibilities are analogous to has_table_privilege. The desired access privilege type must evaluate to CREATE.

has_type_privilege checks whether a user can access a type in a particular way. Its argument possibilities are analogous to has_table_privilege. When specifying a type by a text string rather than by

OID, the allowed input is the same as for the `regtype` data type (see Section 8.18). The desired access privilege type must evaluate to `USAGE`.

`pg_has_role` checks whether a user can access a role in a particular way. Its argument possibilities are analogous to `has_table_privilege`, except that `public` is not allowed as a user name. The desired access privilege type must evaluate to some combination of `MEMBER` or `USAGE`. `MEMBER` denotes direct or indirect membership in the role (that is, the right to do `SET ROLE`), while `USAGE` denotes whether the privileges of the role are immediately available without doing `SET ROLE`.

`row_security_active` checks whether row level security is active for the specified table in the context of the `current_user` and environment. The table can be specified by name or by OID.

Table 9-59 shows functions that determine whether a certain object is *visible* in the current schema search path. For example, a table is said to be visible if its containing schema is in the search path and no table of the same name appears earlier in the search path. This is equivalent to the statement that the table can be referenced by name without explicit schema qualification. To list the names of all visible tables:

```
SELECT relname FROM pg_class WHERE pg_table_is_visible(oid);
```

Table 9-59. Schema Visibility Inquiry Functions

Name	Return Type	Description
`pg_collation_is_visible(collation_oid)`	`boolean`	is collation visible in search path
`pg_conversion_is_visible(conversion_oid)`	`boolean`	is conversion visible in search path
`pg_function_is_visible(function_oid)`	`boolean`	is function visible in search path
`pg_opclass_is_visible(opclass_oid)`	`boolean`	is operator class visible in search path
`pg_operator_is_visible(operator_oid)`	`boolean`	is operator visible in search path
`pg_opfamily_is_visible(opclass_oid)`	`boolean`	is operator family visible in search path
`pg_table_is_visible(table_oid)`	`boolean`	is table visible in search path
`pg_ts_config_is_visible(config_oid)`	`boolean`	is text search configuration visible in search path
`pg_ts_dict_is_visible(dict_oid)`	`boolean`	is text search dictionary visible in search path
`pg_ts_parser_is_visible(parser_oid)`	`boolean`	is text search parser visible in search path
`pg_ts_template_is_visible(template_oid)`	`boolean`	is text search template visible in search path

Name	Return Type	Description
pg_type_is_visible(type_oid)	boolean	is type (or domain) visible in search path

Each function performs the visibility check for one type of database object. Note that pg_table_is_visible can also be used with views, materialized views, indexes, sequences and foreign tables; pg_type_is_visible can also be used with domains. For functions and operators, an object in the search path is visible if there is no object of the same name *and argument data type(s)* earlier in the path. For operator classes, both name and associated index access method are considered.

All these functions require object OIDs to identify the object to be checked. If you want to test an object by name, it is convenient to use the OID alias types (regclass, regtype, regprocedure, regoperator, regconfig, or regdictionary), for example:

```
SELECT pg_type_is_visible('myschema.widget'::regtype);
```

Note that it would not make much sense to test a non-schema-qualified type name in this way — if the name can be recognized at all, it must be visible.

Table 9-60 lists functions that extract information from the system catalogs.

Table 9-60. System Catalog Information Functions

Name	Return Type	Description
format_type(type_oid, typemod)	text	get SQL name of a data type
pg_get_constraintdef(constraint_oid)	text	get definition of a constraint
pg_get_constraintdef(constraint_oid, pretty_bool)	text	get definition of a constraint
pg_get_expr(pg_node_tree, relation_oid)	text	decompile internal form of an expression, assuming that any Vars in it refer to the relation indicated by the second parameter
pg_get_expr(pg_node_tree, relation_oid, pretty_bool)	text	decompile internal form of an expression, assuming that any Vars in it refer to the relation indicated by the second parameter
pg_get_functiondef(func_oid)	text	get definition of a function
pg_get_function_arguments(func_oid)	text	get argument list of function's definition (with default values)
pg_get_function_identity_arguments(func_oid)	text	get argument list to identify a function (without default values)

Name	Return Type	Description
pg_get_function_result(func_oid)	text	get RETURNS clause for function
pg_get_indexdef(index_oid)	text	get CREATE INDEX command for index
pg_get_indexdef(index_oid, column_no, pretty_bool)	text	get CREATE INDEX command for index, or definition of just one index column when column_no is not zero
pg_get_keywords()	setof record	get list of SQL keywords and their categories
pg_get_ruledef(rule_oid)	text	get CREATE RULE command for rule
pg_get_ruledef(rule_oid, pretty_bool)	text	get CREATE RULE command for rule
pg_get_serial_sequence(table_name, column_name)	text	get name of the sequence that a serial, smallserial or bigserial column uses
pg_get_triggerdef(trigger_oid)	text	get CREATE [CONSTRAINT] TRIGGER command for trigger
pg_get_triggerdef(trigger_oid, pretty_bool)	text	get CREATE [CONSTRAINT] TRIGGER command for trigger
pg_get_userbyid(role_oid)	name	get role name with given OID
pg_get_viewdef(view_name)	text	get underlying SELECT command for view or materialized view (*deprecated*)
pg_get_viewdef(view_name, pretty_bool)	text	get underlying SELECT command for view or materialized view (*deprecated*)
pg_get_viewdef(view_oid)	text	get underlying SELECT command for view or materialized view
pg_get_viewdef(view_oid, pretty_bool)	text	get underlying SELECT command for view or materialized view
pg_get_viewdef(view_oid, wrap_column_int)	text	get underlying SELECT command for view or materialized view; lines with fields are wrapped to specified number of columns, pretty-printing is implied
pg_options_to_table(reloptions)	setof record	get the set of storage option name/value pairs

Name	Return Type	Description
pg_tablespace_databases(tablespace_oid)	setof oid	get the set of database OIDs that have objects in the tablespace
pg_tablespace_location(tablespace_oid)	text	get the path in the file system that this tablespace is located in
pg_typeof(any)	regtype	get the data type of any value
collation for (any)	text	get the collation of the argument
to_regclass(rel_name)	regclass	get the OID of the named relation
to_regproc(func_name)	regproc	get the OID of the named function
to_regprocedure(func_name)	regprocedure	get the OID of the named function
to_regoper(operator_name)	regoper	get the OID of the named operator
to_regoperator(operator_name)	regoperator	get the OID of the named operator
to_regtype(type_name)	regtype	get the OID of the named type
to_regnamespace(schema_name)	regnamespace	get the OID of the named schema
to_regrole(role_name)	regrole	get the OID of the named role

format_type returns the SQL name of a data type that is identified by its type OID and possibly a type modifier. Pass NULL for the type modifier if no specific modifier is known.

pg_get_keywords returns a set of records describing the SQL keywords recognized by the server. The word column contains the keyword. The catcode column contains a category code: U for unreserved, C for column name, T for type or function name, or R for reserved. The catdesc column contains a possibly-localized string describing the category.

pg_get_constraintdef, pg_get_indexdef, pg_get_ruledef, and pg_get_triggerdef, respectively reconstruct the creating command for a constraint, index, rule, or trigger. (Note that this is a decompiled reconstruction, not the original text of the command.) pg_get_expr decompiles the internal form of an individual expression, such as the default value for a column. It can be useful when examining the contents of system catalogs. If the expression might contain Vars, specify the OID of the relation they refer to as the second parameter; if no Vars are expected, zero is sufficient. pg_get_viewdef reconstructs the SELECT query that defines a view. Most of these functions come in two variants, one of which can optionally "pretty-print" the result. The pretty-printed format is more readable, but the default format is more likely to be interpreted the same way by future versions of PostgreSQL; avoid using pretty-printed output for dump purposes. Passing false for the pretty-print parameter yields the same result as the variant that does not have the parameter at all.

pg_get_functiondef returns a complete CREATE OR REPLACE FUNCTION statement for a function. pg_get_function_arguments returns the argument list of a function, in the form it would need to appear in within CREATE FUNCTION. pg_get_function_result similarly returns the appropriate RETURNS clause for the function. pg_get_function_identity_arguments returns the argument list necessary to identify a function, in the form it would need to appear in within ALTER FUNCTION, for

instance. This form omits default values.

pg_get_serial_sequence returns the name of the sequence associated with a column, or NULL if no sequence is associated with the column. The first input parameter is a table name with optional schema, and the second parameter is a column name. Because the first parameter is potentially a schema and table, it is not treated as a double-quoted identifier, meaning it is lower cased by default, while the second parameter, being just a column name, is treated as double-quoted and has its case preserved. The function returns a value suitably formatted for passing to sequence functions (see Section 9.16). This association can be modified or removed with ALTER SEQUENCE OWNED BY. (The function probably should have been called pg_get_owned_sequence; its current name reflects the fact that it's typically used with serial or bigserial columns.)

pg_get_userbyid extracts a role's name given its OID.

pg_options_to_table returns the set of storage option name/value pairs (option_name/option_value) when passed pg_class.reloptions or pg_attribute.attoptions.

pg_tablespace_databases allows a tablespace to be examined. It returns the set of OIDs of databases that have objects stored in the tablespace. If this function returns any rows, the tablespace is not empty and cannot be dropped. To display the specific objects populating the tablespace, you will need to connect to the databases identified by pg_tablespace_databases and query their pg_class catalogs.

pg_typeof returns the OID of the data type of the value that is passed to it. This can be helpful for troubleshooting or dynamically constructing SQL queries. The function is declared as returning regtype, which is an OID alias type (see Section 8.18); this means that it is the same as an OID for comparison purposes but displays as a type name. For example:

```
SELECT pg_typeof(33);

 pg_typeof
-----------
 integer
(1 row)

SELECT typlen FROM pg_type WHERE oid = pg_typeof(33);
 typlen
--------
      4
(1 row)
```

The expression collation for returns the collation of the value that is passed to it. Example:

```
SELECT collation for (description) FROM pg_description LIMIT 1;
 pg_collation_for
------------------
 "default"
(1 row)

SELECT collation for ('foo' COLLATE "de_DE");
 pg_collation_for
------------------
```

```
 "de_DE"
(1 row)
```

The value might be quoted and schema-qualified. If no collation is derived for the argument expression, then a null value is returned. If the argument is not of a collatable data type, then an error is raised.

The `to_regclass`, `to_regproc`, `to_regprocedure`, `to_regoper`, `to_regoperator`, `to_regtype`, `to_regnamespace`, and `to_regrole` functions translate relation, function, operator, type, schema, and role names to objects of type `regclass`, `regproc`, `regprocedure`, `regoper`, `regoperator`, `regtype`, `regnamespace`, and `regrole` respectively. These functions differ from a cast from text in that they don't accept a numeric OID, and that they return null rather than throwing an error if the name is not found (or, for `to_regproc` and `to_regoper`, if the given name matches multiple objects).

Table 9-61 lists functions related to database object identification and addressing.

Table 9-61. Object Information and Addressing Functions

Name	Return Type	Description
`pg_describe_object(classid oid, objid oid, objsubid integer)`	`text`	get description of a database object
`pg_identify_object(classid oid, objid oid, objsubid integer)`	`type text, schema text, name text, identity text`	get identity of a database object
`pg_identify_object_as_address(classid oid, objid oid, objsubid integer)`	`type text, object_names text[], object_args text[]`	get external representation of a database object's address
`pg_get_object_address(type text, name text[], args text[])`	`classid oid, objid oid, objsubid integer`	get address of a database object from its external representation

`pg_describe_object` returns a textual description of a database object specified by catalog OID, object OID, and sub-object ID (such as a column number within a table; the sub-object ID is zero when referring to a whole object). This description is intended to be human-readable, and might be translated, depending on server configuration. This is useful to determine the identity of an object as stored in the `pg_depend` catalog.

`pg_identify_object` returns a row containing enough information to uniquely identify the database object specified by catalog OID, object OID and sub-object ID. This information is intended to be machine-readable, and is never translated. `type` identifies the type of database object; `schema` is the schema name that the object belongs in, or `NULL` for object types that do not belong to schemas; `name` is the name of the object, quoted if necessary, if the name (along with schema name, if pertinent) is sufficient to uniquely identify the object, otherwise `NULL`; `identity` is the complete object identity, with the precise format depending on object type, and each name within the format being schema-qualified and quoted as necessary.

`pg_identify_object_as_address` returns a row containing enough information to uniquely identify the database object specified by catalog OID, object OID and sub-object ID. The returned information is independent of the current server, that is, it could be used to identify an identically named object in another server. `type` identifies the type of database object; `object_names` and `object_args` are text arrays that

together form a reference to the object. These three values can be passed to pg_get_object_address to obtain the internal address of the object. This function is the inverse of pg_get_object_address.

pg_get_object_address returns a row containing enough information to uniquely identify the database object specified by its type and object name and argument arrays. The returned values are the ones that would be used in system catalogs such as pg_depend and can be passed to other system functions such as pg_identify_object or pg_describe_object. classid is the OID of the system catalog containing the object; objid is the OID of the object itself, and objsubid is the sub-object ID, or zero if none. This function is the inverse of pg_identify_object_as_address.

The functions shown in Table 9-62 extract comments previously stored with the COMMENT command. A null value is returned if no comment could be found for the specified parameters.

Table 9-62. Comment Information Functions

Name	Return Type	Description
col_description(table_oid, column_number)	text	get comment for a table column
obj_description(object_oid, catalog_name)	text	get comment for a database object
obj_description(object_oid)	text	get comment for a database object (*deprecated*)
shobj_description(object_oid, catalog_name)	text	get comment for a shared database object

col_description returns the comment for a table column, which is specified by the OID of its table and its column number. (obj_description cannot be used for table columns since columns do not have OIDs of their own.)

The two-parameter form of obj_description returns the comment for a database object specified by its OID and the name of the containing system catalog. For example, obj_description(123456,'pg_class') would retrieve the comment for the table with OID 123456. The one-parameter form of obj_description requires only the object OID. It is deprecated since there is no guarantee that OIDs are unique across different system catalogs; therefore, the wrong comment might be returned.

shobj_description is used just like obj_description except it is used for retrieving comments on shared objects. Some system catalogs are global to all databases within each cluster, and the descriptions for objects in them are stored globally as well.

The functions shown in Table 9-63 provide server transaction information in an exportable form. The main use of these functions is to determine which transactions were committed between two snapshots.

Table 9-63. Transaction IDs and Snapshots

Name	Return Type	Description
txid_current()	bigint	get current transaction ID, assigning a new one if the current transaction does not have one
txid_current_snapshot()	txid_snapshot	get current snapshot

Name	Return Type	Description
txid_snapshot_xip(txid_snapshot)	setof bigint	get in-progress transaction IDs in snapshot
txid_snapshot_xmax(txid_snapshot)	bigint	get xmax of snapshot
txid_snapshot_xmin(txid_snapshot)	bigint	get xmin of snapshot
txid_visible_in_snapshot(bigint, txid_snapshot)	boolean	is transaction ID visible in snapshot? (do not use with subtransaction ids)

The internal transaction ID type (xid) is 32 bits wide and wraps around every 4 billion transactions. However, these functions export a 64-bit format that is extended with an "epoch" counter so it will not wrap around during the life of an installation. The data type used by these functions, txid_snapshot, stores information about transaction ID visibility at a particular moment in time. Its components are described in Table 9-64.

Table 9-64. Snapshot Components

Name	Description
xmin	Earliest transaction ID (txid) that is still active. All earlier transactions will either be committed and visible, or rolled back and dead.
xmax	First as-yet-unassigned txid. All txids greater than or equal to this are not yet started as of the time of the snapshot, and thus invisible.
xip_list	Active txids at the time of the snapshot. The list includes only those active txids between xmin and xmax; there might be active txids higher than xmax. A txid that is xmin <= txid < xmax and not in this list was already completed at the time of the snapshot, and thus either visible or dead according to its commit status. The list does not include txids of subtransactions.

txid_snapshot's textual representation is *xmin*:*xmax*:*xip_list*. For example 10:20:10,14,15 means xmin=10, xmax=20, xip_list=10, 14, 15.

The functions shown in Table 9-65 provide information about transactions that have been already committed. These functions mainly provide information about when the transactions were committed. They only provide useful data when track_commit_timestamp configuration option is enabled and only for transactions that were committed after it was enabled.

Table 9-65. Committed transaction information

Name	Return Type	Description

Name	Return Type	Description
pg_xact_commit_timestamp(xid)	timestamp with time zone	get commit timestamp of a transaction
pg_last_committed_xact()	xid xid, timestamp timestamp with time zone	get transaction ID and commit timestamp of latest committed transaction

9.26. System Administration Functions

The functions described in this section are used to control and monitor a PostgreSQL installation.

9.26.1. Configuration Settings Functions

Table 9-66 shows the functions available to query and alter run-time configuration parameters.

Table 9-66. Configuration Settings Functions

Name	Return Type	Description
current_setting(setting_name)	text	get current value of setting
set_config(setting_name, new_value, is_local)	text	set parameter and return new value

The function current_setting yields the current value of the setting setting_name. It corresponds to the SQL command SHOW. An example:

```
SELECT current_setting('datestyle');

 current_setting
-----------------
 ISO, MDY
(1 row)
```

set_config sets the parameter setting_name to new_value. If is_local is true, the new value will only apply to the current transaction. If you want the new value to apply for the current session, use false instead. The function corresponds to the SQL command SET. An example:

```
SELECT set_config('log_statement_stats', 'off', false);

 set_config
------------
 off
(1 row)
```

9.26.2. Server Signaling Functions

The functions shown in Table 9-67 send control signals to other server processes. Use of these functions is usually restricted to superusers, with noted exceptions.

Table 9-67. Server Signaling Functions

Name	Return Type	Description
pg_cancel_backend(pid int)	boolean	Cancel a backend's current query. This is also allowed if the calling role is a member of the role whose backend is being canceled, however only superusers can cancel superuser backends.
pg_reload_conf()	boolean	Cause server processes to reload their configuration files
pg_rotate_logfile()	boolean	Rotate server's log file
pg_terminate_backend(pid int)	boolean	Terminate a backend. This is also allowed if the calling role is a member of the role whose backend is being terminated, however only superusers can terminate superuser backends.

Each of these functions returns `true` if successful and `false` otherwise.

`pg_cancel_backend` and `pg_terminate_backend` send signals (SIGINT or SIGTERM respectively) to backend processes identified by process ID. The process ID of an active backend can be found from the `pid` column of the `pg_stat_activity` view, or by listing the `postgres` processes on the server (using ps on Unix or the Task Manager on Windows). The role of an active backend can be found from the `usename` column of the `pg_stat_activity` view.

`pg_reload_conf` sends a SIGHUP signal to the server, causing configuration files to be reloaded by all server processes.

`pg_rotate_logfile` signals the log-file manager to switch to a new output file immediately. This works only when the built-in log collector is running, since otherwise there is no log-file manager subprocess.

9.26.3. Backup Control Functions

The functions shown in Table 9-68 assist in making on-line backups. These functions cannot be executed during recovery (except `pg_is_in_backup`, `pg_backup_start_time` and `pg_xlog_location_diff`).

Table 9-68. Backup Control Functions

Name	Return Type	Description
pg_create_restore_point(name text)	pg_lsn	Create a named point for performing restore (restricted to superusers)
pg_current_xlog_insert_location()	pg_lsn	Get current transaction log insert location
pg_current_xlog_location()	pg_lsn	Get current transaction log write location
pg_start_backup(label text [, fast boolean])	pg_lsn	Prepare for performing on-line backup (restricted to superusers or replication roles)
pg_stop_backup()	pg_lsn	Finish performing on-line backup (restricted to superusers or replication roles)
pg_is_in_backup()	bool	True if an on-line exclusive backup is still in progress.
pg_backup_start_time()	timestamp with time zone	Get start time of an on-line exclusive backup in progress.
pg_switch_xlog()	pg_lsn	Force switch to a new transaction log file (restricted to superusers)
pg_xlogfile_name(location pg_lsn)	text	Convert transaction log location string to file name
pg_xlogfile_name_offset(location pg_lsn)	text, integer	Convert transaction log location string to file name and decimal byte offset within file
pg_xlog_location_diff(location pg_lsn, location pg_lsn)	numeric	Calculate the difference between two transaction log locations

pg_start_backup accepts an arbitrary user-defined label for the backup. (Typically this would be the name under which the backup dump file will be stored.) The function writes a backup label file (backup_label) and, if there are any links in the pg_tblspc/ directory, a tablespace map file (tablespace_map) into the database cluster's data directory, performs a checkpoint, and then returns the backup's starting transaction log location as text. The user can ignore this result value, but it is provided in case it is useful.

```
postgres=# select pg_start_backup('label_goes_here');
 pg_start_backup
-----------------
 0/D4445B8
(1 row)
```

There is an optional second parameter of type boolean. If true, it specifies executing pg_start_backup as quickly as possible. This forces an immediate checkpoint which will cause a

spike in I/O operations, slowing any concurrently executing queries.

pg_stop_backup removes the label file and, if it exists, the tablespace_map file created by pg_start_backup, and creates a backup history file in the transaction log archive area. The history file includes the label given to pg_start_backup, the starting and ending transaction log locations for the backup, and the starting and ending times of the backup. The return value is the backup's ending transaction log location (which again can be ignored). After recording the ending location, the current transaction log insertion point is automatically advanced to the next transaction log file, so that the ending transaction log file can be archived immediately to complete the backup.

pg_switch_xlog moves to the next transaction log file, allowing the current file to be archived (assuming you are using continuous archiving). The return value is the ending transaction log location + 1 within the just-completed transaction log file. If there has been no transaction log activity since the last transaction log switch, pg_switch_xlog does nothing and returns the start location of the transaction log file currently in use.

pg_create_restore_point creates a named transaction log record that can be used as recovery target, and returns the corresponding transaction log location. The given name can then be used with recovery_target_name to specify the point up to which recovery will proceed. Avoid creating multiple restore points with the same name, since recovery will stop at the first one whose name matches the recovery target.

pg_current_xlog_location displays the current transaction log write location in the same format used by the above functions. Similarly, pg_current_xlog_insert_location displays the current transaction log insertion point. The insertion point is the "logical" end of the transaction log at any instant, while the write location is the end of what has actually been written out from the server's internal buffers. The write location is the end of what can be examined from outside the server, and is usually what you want if you are interested in archiving partially-complete transaction log files. The insertion point is made available primarily for server debugging purposes. These are both read-only operations and do not require superuser permissions.

You can use pg_xlogfile_name_offset to extract the corresponding transaction log file name and byte offset from the results of any of the above functions. For example:

```
postgres=# SELECT * FROM pg_xlogfile_name_offset(pg_stop_backup());
         file_name          | file_offset
----------------------------+-------------
 00000001000000000000000D |     4039624
(1 row)
```

Similarly, pg_xlogfile_name extracts just the transaction log file name. When the given transaction log location is exactly at a transaction log file boundary, both these functions return the name of the preceding transaction log file. This is usually the desired behavior for managing transaction log archiving behavior, since the preceding file is the last one that currently needs to be archived.

pg_xlog_location_diff calculates the difference in bytes between two transaction log locations. It can be used with pg_stat_replication or some functions shown in Table 9-68 to get the replication lag.

For details about proper usage of these functions, see Section 24.3.

9.26.4. Recovery Control Functions

The functions shown in Table 9-69 provide information about the current status of the standby. These functions may be executed both during recovery and in normal running.

Table 9-69. Recovery Information Functions

Name	Return Type	Description
`pg_is_in_recovery()`	`bool`	True if recovery is still in progress.
`pg_last_xlog_receive_location()`	`pg_lsn`	Get last transaction log location received and synced to disk by streaming replication. While streaming replication is in progress this will increase monotonically. If recovery has completed this will remain static at the value of the last WAL record received and synced to disk during recovery. If streaming replication is disabled, or if it has not yet started, the function returns NULL.
`pg_last_xlog_replay_location()`	`pg_lsn`	Get last transaction log location replayed during recovery. If recovery is still in progress this will increase monotonically. If recovery has completed then this value will remain static at the value of the last WAL record applied during that recovery. When the server has been started normally without recovery the function returns NULL.

Name	Return Type	Description
pg_last_xact_replay_timestamp()	timestamp with time zone	Get time stamp of last transaction replayed during recovery. This is the time at which the commit or abort WAL record for that transaction was generated on the primary. If no transactions have been replayed during recovery, this function returns NULL. Otherwise, if recovery is still in progress this will increase monotonically. If recovery has completed then this value will remain static at the value of the last transaction applied during that recovery. When the server has been started normally without recovery the function returns NULL.

The functions shown in Table 9-70 control the progress of recovery. These functions may be executed only during recovery.

Table 9-70. Recovery Control Functions

Name	Return Type	Description
pg_is_xlog_replay_paused()	bool	True if recovery is paused.
pg_xlog_replay_pause()	void	Pauses recovery immediately (restricted to superusers).
pg_xlog_replay_resume()	void	Restarts recovery if it was paused (restricted to superusers).

While recovery is paused no further database changes are applied. If in hot standby, all new queries will see the same consistent snapshot of the database, and no further query conflicts will be generated until recovery is resumed.

If streaming replication is disabled, the paused state may continue indefinitely without problem. While streaming replication is in progress WAL records will continue to be received, which will eventually fill available disk space, depending upon the duration of the pause, the rate of WAL generation and available disk space.

9.26.5. Snapshot Synchronization Functions

PostgreSQL allows database sessions to synchronize their snapshots. A *snapshot* determines which data is visible to the transaction that is using the snapshot. Synchronized snapshots are necessary when two or more sessions need to see identical content in the database. If two sessions just start their transactions independently, there is always a possibility that some third transaction commits between the executions

of the two START TRANSACTION commands, so that one session sees the effects of that transaction and the other does not.

To solve this problem, PostgreSQL allows a transaction to *export* the snapshot it is using. As long as the exporting transaction remains open, other transactions can *import* its snapshot, and thereby be guaranteed that they see exactly the same view of the database that the first transaction sees. But note that any database changes made by any one of these transactions remain invisible to the other transactions, as is usual for changes made by uncommitted transactions. So the transactions are synchronized with respect to pre-existing data, but act normally for changes they make themselves.

Snapshots are exported with the pg_export_snapshot function, shown in Table 9-71, and imported with the SET TRANSACTION command.

Table 9-71. Snapshot Synchronization Functions

Name	Return Type	Description
pg_export_snapshot()	text	Save the current snapshot and return its identifier

The function pg_export_snapshot saves the current snapshot and returns a text string identifying the snapshot. This string must be passed (outside the database) to clients that want to import the snapshot. The snapshot is available for import only until the end of the transaction that exported it. A transaction can export more than one snapshot, if needed. Note that doing so is only useful in READ COMMITTED transactions, since in REPEATABLE READ and higher isolation levels, transactions use the same snapshot throughout their lifetime. Once a transaction has exported any snapshots, it cannot be prepared with PREPARE TRANSACTION.

See SET TRANSACTION for details of how to use an exported snapshot.

9.26.6. Replication Functions

The functions shown in Table 9-72 are for controlling and interacting with replication features. See Section 25.2.5, Section 25.2.6, and Chapter 47 for information about the underlying features. Use of functions for replication origin is restricted to superusers. Use of functions for replication slot is restricted to superusers and users having REPLICATION privilege.

Many of these functions have equivalent commands in the replication protocol; see Section 50.3.

The functions described in Section 9.26.3, Section 9.26.4, and Section 9.26.5 are also relevant for replication.

Table 9-72. Replication SQL Functions

Function	Return Type	Description

Function	Return Type	Description
pg_create_physical_replication_slot(slot_name name)	(slot_name name, xlog_position pg_lsn)	Creates a new physical replication slot named slot_name. Streaming changes from a physical slot is only possible with the streaming-replication protocol - see Section 50.3. Corresponds to the replication protocol command CREATE_REPLICATION_SLOT ... PHYSICAL.
pg_drop_replication_slot(slot_name name)	void	Drops the physical or logical replication slot named slot_name. Same as replication protocol command DROP_REPLICATION_SLOT.
pg_create_logical_replication_slot(slot_name name, plugin name)	(slot_name name, xlog_position pg_lsn)	Creates a new logical (decoding) replication slot named slot_name using the output plugin plugin. A call to this function has the same effect as the replication protocol command CREATE_REPLICATION_SLOT ... LOGICAL.

Function	Return Type	Description
pg_logical_slot_get_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])	(location pg_lsn, xid xid, data text)	Returns changes in the slot slot_name, starting from the point at which since changes have been consumed last. If upto_lsn and upto_nchanges are NULL, logical decoding will continue until end of WAL. If upto_lsn is non-NULL, decoding will include only those transactions which commit prior to the specified LSN. If upto_nchanges is non-NULL, decoding will stop when the number of rows produced by decoding exceeds the specified value. Note, however, that the actual number of rows returned may be larger, since this limit is only checked after adding the rows produced when decoding each new transaction commit.
pg_logical_slot_peek_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])	(location text, xid xid, data text)	Behaves just like the pg_logical_slot_get_changes() function, except that changes are not consumed; that is, they will be returned again on future calls.
pg_logical_slot_get_binary_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])	(location pg_lsn, xid xid, data bytea)	Behaves just like the pg_logical_slot_get_changes() function, except that changes are returned as bytea.
pg_logical_slot_peek_binary_changes(slot_name name, upto_lsn pg_lsn, upto_nchanges int, VARIADIC options text[])	(location pg_lsn, xid xid, data bytea)	Behaves just like the pg_logical_slot_get_changes() function, except that changes are returned as bytea and that changes are not consumed; that is, they will be returned again on future calls.
pg_replication_origin_create(node_name text)	oid	Create a replication origin with the given external name, and return the internal id assigned to it.

Function	Return Type	Description
pg_replication_origin_drop(node_name text)	void	Delete a previously created replication origin, including any associated replay progress.
pg_replication_origin_oid(node_name text)	oid	Lookup a replication origin by name and return the internal id. If no corresponding replication origin is found an error is thrown.
pg_replication_origin_session_setup(node_name text)	void	Mark the current session as replaying from the given origin, allowing replay progress to be tracked. Use pg_replication_origin_session_reset to revert. Can only be used if no previous origin is configured.
pg_replication_origin_session_reset()	void	Cancel the effects of pg_replication_origin_session_setup().
pg_replication_origin_session_is_setup()	bool	Has a replication origin been configured in the current session?
pg_replication_origin_session_progress(flush bool)	pg_lsn	Return the replay position for the replication origin configured in the current session. The parameter flush determines whether the corresponding local transaction will be guaranteed to have been flushed to disk or not.
pg_replication_origin_xact_setup(origin_lsn pg_lsn, origin_timestamp timestamptz)	void	Mark the current transaction as replaying a transaction that has committed at the given LSN and timestamp. Can only be called when a replication origin has previously been configured using pg_replication_origin_session_setup().

Function	Return Type	Description
pg_replication_origin_xact_reset(origin_lsn pg_lsn, origin_timestamp timestamptz)	void	Cancel the effects of pg_replication_origin_xact_setup(). Note that two arguments were introduced *by mistake* during the PostgreSQL 9.5 development cycle while pg_replication_origin_xact_reset() actually doesn't use them at all. Therefore, any dummy values except NULL can be safely specified as the arguments. This mistake will be fixed in a future release.
pg_replication_origin_advance(node_name text, pos pg_lsn)	void	Set replication progress for the given node to the given position. This primarily is useful for setting up the initial position or a new position after configuration changes and similar. Be aware that careless use of this function can lead to inconsistently replicated data.
pg_replication_origin_progress(node_name text, flush bool)	pg_lsn	Return the replay position for the given replication origin. The parameter flush determines whether the corresponding local transaction will be guaranteed to have been flushed to disk or not.

9.26.7. Database Object Management Functions

The functions shown in Table 9-73 calculate the disk space usage of database objects.

Table 9-73. Database Object Size Functions

Name	Return Type	Description
pg_column_size(any)	int	Number of bytes used to store a particular value (possibly compressed)
pg_database_size(oid)	bigint	Disk space used by the database with the specified OID
pg_database_size(name)	bigint	Disk space used by the database with the specified name

Name	Return Type	Description
pg_indexes_size(regclass)	bigint	Total disk space used by indexes attached to the specified table
pg_relation_size(relation regclass, fork text)	bigint	Disk space used by the specified fork ('main', 'fsm', 'vm', or 'init') of the specified table or index
pg_relation_size(relation regclass)	bigint	Shorthand for pg_relation_size(..., 'main')
pg_size_pretty(bigint)	text	Converts a size in bytes expressed as a 64-bit integer into a human-readable format with size units
pg_size_pretty(numeric)	text	Converts a size in bytes expressed as a numeric value into a human-readable format with size units
pg_table_size(regclass)	bigint	Disk space used by the specified table, excluding indexes (but including TOAST, free space map, and visibility map)
pg_tablespace_size(oid)	bigint	Disk space used by the tablespace with the specified OID
pg_tablespace_size(name)	bigint	Disk space used by the tablespace with the specified name
pg_total_relation_size(regclass)	bigint	Total disk space used by the specified table, including all indexes and TOAST data

pg_column_size shows the space used to store any individual data value.

pg_total_relation_size accepts the OID or name of a table or toast table, and returns the total on-disk space used for that table, including all associated indexes. This function is equivalent to pg_table_size + pg_indexes_size.

pg_table_size accepts the OID or name of a table and returns the disk space needed for that table, exclusive of indexes. (TOAST space, free space map, and visibility map are included.)

pg_indexes_size accepts the OID or name of a table and returns the total disk space used by all the indexes attached to that table.

pg_database_size and pg_tablespace_size accept the OID or name of a database or tablespace, and return the total disk space used therein. To use pg_database_size, you must have CONNECT permission on the specified database (which is granted by default). To use pg_tablespace_size, you must have CREATE permission on the specified tablespace, unless it is the default tablespace for the current

database.

pg_relation_size accepts the OID or name of a table, index or toast table, and returns the on-disk size in bytes of one fork of that relation. (Note that for most purposes it is more convenient to use the higher-level functions pg_total_relation_size or pg_table_size, which sum the sizes of all forks.) With one argument, it returns the size of the main data fork of the relation. The second argument can be provided to specify which fork to examine:

- 'main' returns the size of the main data fork of the relation.
- 'fsm' returns the size of the Free Space Map (see Section 63.3) associated with the relation.
- 'vm' returns the size of the Visibility Map (see Section 63.4) associated with the relation.
- 'init' returns the size of the initialization fork, if any, associated with the relation.

pg_size_pretty can be used to format the result of one of the other functions in a human-readable way, using kB, MB, GB or TB as appropriate.

The functions above that operate on tables or indexes accept a regclass argument, which is simply the OID of the table or index in the pg_class system catalog. You do not have to look up the OID by hand, however, since the regclass data type's input converter will do the work for you. Just write the table name enclosed in single quotes so that it looks like a literal constant. For compatibility with the handling of ordinary SQL names, the string will be converted to lower case unless it contains double quotes around the table name.

If an OID that does not represent an existing object is passed as argument to one of the above functions, NULL is returned.

The functions shown in Table 9-74 assist in identifying the specific disk files associated with database objects.

Table 9-74. Database Object Location Functions

Name	Return Type	Description
pg_relation_filenode(relation regclass)	oid	Filenode number of the specified relation
pg_relation_filepath(relation regclass)	text	File path name of the specified relation
pg_filenode_relation(tablespace oid, filenode oid)	regclass	Find the relation associated with a given tablespace and filenode

pg_relation_filenode accepts the OID or name of a table, index, sequence, or toast table, and returns the "filenode" number currently assigned to it. The filenode is the base component of the file name(s) used for the relation (see Section 63.1 for more information). For most tables the result is the same as pg_class.relfilenode, but for certain system catalogs relfilenode is zero and this function must be used to get the correct value. The function returns NULL if passed a relation that does not have storage, such as a view.

pg_relation_filepath is similar to pg_relation_filenode, but it returns the entire file path name (relative to the database cluster's data directory PGDATA) of the relation.

pg_filenode_relation is the reverse of pg_relation_filenode. Given a "tablespace" OID and a "filenode", it returns the associated relation's OID. For a table in the database's default tablespace, the tablespace can be specified as 0.

9.26.8. Index Maintenance Functions

Table 9-75 shows the functions available for index maintenance tasks.

Table 9-75. Index Maintenance Functions

Name	Return Type	Description
brin_summarize_new_values(index_oid regclass)	integer	summarize page ranges not already summarized

brin_summarize_new_values receives a BRIN index OID as argument and inspects the index to find page ranges in the base table that are not currently summarized by the index; for any such range it creates a new summary index tuple by scanning the table pages. It returns the number of new page range summaries that were inserted into the index.

9.26.9. Generic File Access Functions

The functions shown in Table 9-76 provide native access to files on the machine hosting the server. Only files within the database cluster directory and the log_directory can be accessed. Use a relative path for files in the cluster directory, and a path matching the log_directory configuration setting for log files. Use of these functions is restricted to superusers.

Table 9-76. Generic File Access Functions

Name	Return Type	Description
pg_ls_dir(dirname text [, missing_ok boolean, include_dot_dirs boolean])	setof text	List the contents of a directory.
pg_read_file(filename text [, offset bigint, length bigint [, missing_ok boolean]])	text	Return the contents of a text file.
pg_read_binary_file(filename text [, offset bigint, length bigint [, missing_ok boolean]])	bytea	Return the contents of a file.

Name	Return Type	Description
pg_stat_file(filename text[, missing_ok boolean])	record	Return information about a file.

All of these functions take an optional missing_ok parameter, which specifies the behavior when the file or directory does not exist. If true, the function returns NULL (except pg_ls_dir, which returns an empty result set). If false, an error is raised. The default is false.

pg_ls_dir returns the names of all files (and directories and other special files) in the specified directory. The include_dot_dirs indicates whether "." and ".." are included in the result set. The default is to exclude them (false), but including them can be useful when missing_ok is true, to distinguish an empty directory from an non-existent directory.

pg_read_file returns part of a text file, starting at the given offset, returning at most length bytes (less if the end of file is reached first). If offset is negative, it is relative to the end of the file. If offset and length are omitted, the entire file is returned. The bytes read from the file are interpreted as a string in the server encoding; an error is thrown if they are not valid in that encoding.

pg_read_binary_file is similar to pg_read_file, except that the result is a bytea value; accordingly, no encoding checks are performed. In combination with the convert_from function, this function can be used to read a file in a specified encoding:

```
SELECT convert_from(pg_read_binary_file('file_in_utf8.txt'), 'UTF8');
```

pg_stat_file returns a record containing the file size, last accessed time stamp, last modified time stamp, last file status change time stamp (Unix platforms only), file creation time stamp (Windows only), and a boolean indicating if it is a directory. Typical usages include:

```
SELECT * FROM pg_stat_file('filename');
SELECT (pg_stat_file('filename')).modification;
```

9.26.10. Advisory Lock Functions

The functions shown in Table 9-77 manage advisory locks. For details about proper use of these functions, see Section 13.3.5.

Table 9-77. Advisory Lock Functions

Name	Return Type	Description
pg_advisory_lock(key bigint)	void	Obtain exclusive session level advisory lock
pg_advisory_lock(key1 int, key2 int)	void	Obtain exclusive session level advisory lock
pg_advisory_lock_shared(key bigint)	void	Obtain shared session level advisory lock

Name	Return Type	Description
pg_advisory_lock_shared(key1 int, key2 int)	void	Obtain shared session level advisory lock
pg_advisory_unlock(key bigint)	boolean	Release an exclusive session level advisory lock
pg_advisory_unlock(key1 int, key2 int)	boolean	Release an exclusive session level advisory lock
pg_advisory_unlock_all()	void	Release all session level advisory locks held by the current session
pg_advisory_unlock_shared(key bigint)	boolean	Release a shared session level advisory lock
pg_advisory_unlock_shared(key1 int, key2 int)	boolean	Release a shared session level advisory lock
pg_advisory_xact_lock(key bigint)	void	Obtain exclusive transaction level advisory lock
pg_advisory_xact_lock(key1 int, key2 int)	void	Obtain exclusive transaction level advisory lock
pg_advisory_xact_lock_shared(key bigint)	void	Obtain shared transaction level advisory lock
pg_advisory_xact_lock_shared(key1 int, key2 int)	void	Obtain shared transaction level advisory lock
pg_try_advisory_lock(key bigint)	boolean	Obtain exclusive session level advisory lock if available
pg_try_advisory_lock(key1 int, key2 int)	boolean	Obtain exclusive session level advisory lock if available
pg_try_advisory_lock_shared(key bigint)	boolean	Obtain shared session level advisory lock if available
pg_try_advisory_lock_shared(key1 int, key2 int)	boolean	Obtain shared session level advisory lock if available
pg_try_advisory_xact_lock(key bigint)	boolean	Obtain exclusive transaction level advisory lock if available
pg_try_advisory_xact_lock(key1 int, key2 int)	boolean	Obtain exclusive transaction level advisory lock if available

Name	Return Type	Description
pg_try_advisory_xact_lock_shared(key bigint)	boolean	Obtain shared transaction level advisory lock if available
pg_try_advisory_xact_lock_shared(key1 int, key2 int)	boolean	Obtain shared transaction level advisory lock if available

pg_advisory_lock locks an application-defined resource, which can be identified either by a single 64-bit key value or two 32-bit key values (note that these two key spaces do not overlap). If another session already holds a lock on the same resource identifier, this function will wait until the resource becomes available. The lock is exclusive. Multiple lock requests stack, so that if the same resource is locked three times it must then be unlocked three times to be released for other sessions' use.

pg_advisory_lock_shared works the same as pg_advisory_lock, except the lock can be shared with other sessions requesting shared locks. Only would-be exclusive lockers are locked out.

pg_try_advisory_lock is similar to pg_advisory_lock, except the function will not wait for the lock to become available. It will either obtain the lock immediately and return true, or return false if the lock cannot be acquired immediately.

pg_try_advisory_lock_shared works the same as pg_try_advisory_lock, except it attempts to acquire a shared rather than an exclusive lock.

pg_advisory_unlock will release a previously-acquired exclusive session level advisory lock. It returns true if the lock is successfully released. If the lock was not held, it will return false, and in addition, an SQL warning will be reported by the server.

pg_advisory_unlock_shared works the same as pg_advisory_unlock, except it releases a shared session level advisory lock.

pg_advisory_unlock_all will release all session level advisory locks held by the current session. (This function is implicitly invoked at session end, even if the client disconnects ungracefully.)

pg_advisory_xact_lock works the same as pg_advisory_lock, except the lock is automatically released at the end of the current transaction and cannot be released explicitly.

pg_advisory_xact_lock_shared works the same as pg_advisory_lock_shared, except the lock is automatically released at the end of the current transaction and cannot be released explicitly.

pg_try_advisory_xact_lock works the same as pg_try_advisory_lock, except the lock, if acquired, is automatically released at the end of the current transaction and cannot be released explicitly.

pg_try_advisory_xact_lock_shared works the same as pg_try_advisory_lock_shared, except the lock, if acquired, is automatically released at the end of the current transaction and cannot be released explicitly.

9.27. Trigger Functions

Currently PostgreSQL provides one built in trigger function, suppress_redundant_updates_trigger, which will prevent any update that

does not actually change the data in the row from taking place, in contrast to the normal behavior which always performs the update regardless of whether or not the data has changed. (This normal behavior makes updates run faster, since no checking is required, and is also useful in certain cases.)

Ideally, you should normally avoid running updates that don't actually change the data in the record. Redundant updates can cost considerable unnecessary time, especially if there are lots of indexes to alter, and space in dead rows that will eventually have to be vacuumed. However, detecting such situations in client code is not always easy, or even possible, and writing expressions to detect them can be error-prone. An alternative is to use `suppress_redundant_updates_trigger`, which will skip updates that don't change the data. You should use this with care, however. The trigger takes a small but non-trivial time for each record, so if most of the records affected by an update are actually changed, use of this trigger will actually make the update run slower.

The `suppress_redundant_updates_trigger` function can be added to a table like this:

```
CREATE TRIGGER z_min_update
BEFORE UPDATE ON tablename
FOR EACH ROW EXECUTE PROCEDURE suppress_redundant_updates_trigger();
```

In most cases, you would want to fire this trigger last for each row. Bearing in mind that triggers fire in name order, you would then choose a trigger name that comes after the name of any other trigger you might have on the table.

For more information about creating triggers, see CREATE TRIGGER.

9.28. Event Trigger Functions

PostgreSQL provides these helper functions to retrieve information from event triggers.

For more information about event triggers, see Chapter 37.

9.28.1. Capturing Changes at Command End

`pg_event_trigger_ddl_commands` returns a list of DDL commands executed by each user action, when invoked in a function attached to a `ddl_command_end` event trigger. If called in any other context, an error is raised. `pg_event_trigger_ddl_commands` returns one row for each base command executed; some commands that are a single SQL sentence may return more than one row. This function returns the following columns:

Name	Type	Description
`classid`	`oid`	OID of catalog the object belongs in
`objid`	`oid`	OID of the object itself
`objsubid`	`integer`	Sub-object ID (e.g., attribute number for a column)
`command_tag`	`text`	Command tag
`object_type`	`text`	Type of the object

Name	Type	Description
schema_name	text	Name of the schema the object belongs in, if any; otherwise NULL. No quoting is applied.
object_identity	text	Text rendering of the object identity, schema-qualified. Each identifier included in the identity is quoted if necessary.
in_extension	bool	True if the command is part of an extension script
command	pg_ddl_command	A complete representation of the command, in internal format. This cannot be output directly, but it can be passed to other functions to obtain different pieces of information about the command.

9.28.2. Processing Objects Dropped by a DDL Command

pg_event_trigger_dropped_objects returns a list of all objects dropped by the command in whose sql_drop event it is called. If called in any other context, pg_event_trigger_dropped_objects raises an error. pg_event_trigger_dropped_objects returns the following columns:

Name	Type	Description
classid	oid	OID of catalog the object belonged in
objid	oid	OID of the object itself
objsubid	integer	Sub-object ID (e.g., attribute number for a column)
original	bool	True if this was one of the root object(s) of the deletion
normal	bool	True if there was a normal dependency relationship in the dependency graph leading to this object
is_temporary	bool	True if this was a temporary object
object_type	text	Type of the object
schema_name	text	Name of the schema the object belonged in, if any; otherwise NULL. No quoting is applied.

Name	Type	Description
object_name	text	Name of the object, if the combination of schema and name can be used as a unique identifier for the object; otherwise NULL. No quoting is applied, and name is never schema-qualified.
object_identity	text	Text rendering of the object identity, schema-qualified. Each identifier included in the identity is quoted if necessary.
address_names	text[]	An array that, together with object_type and address_args, can be used by the pg_get_object_address() function to recreate the object address in a remote server containing an identically named object of the same kind
address_args	text[]	Complement for address_names

The pg_event_trigger_dropped_objects function can be used in an event trigger like this:

```
CREATE FUNCTION test_event_trigger_for_drops()
        RETURNS event_trigger LANGUAGE plpgsql AS $$
DECLARE
    obj record;
BEGIN
    FOR obj IN SELECT * FROM pg_event_trigger_dropped_objects()
    LOOP
        RAISE NOTICE '% dropped object: % %.% %',
                     tg_tag,
                     obj.object_type,
                     obj.schema_name,
                     obj.object_name,
                     obj.object_identity;
    END LOOP;
END;
$$;
CREATE EVENT TRIGGER test_event_trigger_for_drops
   ON sql_drop
   EXECUTE PROCEDURE test_event_trigger_for_drops();
```

9.28.3. Handling a Table Rewrite Event

The functions shown in Table 9-78 provide information about a table for which a `table_rewrite` event has just been called. If called in any other context, an error is raised.

Table 9-78. Table Rewrite information

Name	Return Type	Description
`pg_event_trigger_table_rewrite_oid()`	`Oid`	The OID of the table about to be rewritten.
`pg_event_trigger_table_rewrite_reason()`	`int`	The reason code(s) explaining the reason for rewriting. The exact meaning of the codes is release dependent.

The `pg_event_trigger_table_rewrite_oid` function can be used in an event trigger like this:

```
CREATE FUNCTION test_event_trigger_table_rewrite_oid()
 RETURNS event_trigger
 LANGUAGE plpgsql AS
$$
BEGIN
  RAISE NOTICE 'rewriting table % for reason %',
                pg_event_trigger_table_rewrite_oid()::regclass,
                pg_event_trigger_table_rewrite_reason();
END;
$$;

CREATE EVENT TRIGGER test_table_rewrite_oid
                  ON table_rewrite
   EXECUTE PROCEDURE test_event_trigger_table_rewrite_oid();
```

Chapter 10. Type Conversion

SQL statements can, intentionally or not, require the mixing of different data types in the same expression. PostgreSQL has extensive facilities for evaluating mixed-type expressions.

In many cases a user does not need to understand the details of the type conversion mechanism. However, implicit conversions done by PostgreSQL can affect the results of a query. When necessary, these results can be tailored by using *explicit* type conversion.

This chapter introduces the PostgreSQL type conversion mechanisms and conventions. Refer to the relevant sections in Chapter 8 and Chapter 9 for more information on specific data types and allowed functions and operators.

10.1. Overview

SQL is a strongly typed language. That is, every data item has an associated data type which determines its behavior and allowed usage. PostgreSQL has an extensible type system that is more general and flexible than other SQL implementations. Hence, most type conversion behavior in PostgreSQL is governed by general rules rather than by *ad hoc* heuristics. This allows the use of mixed-type expressions even with user-defined types.

The PostgreSQL scanner/parser divides lexical elements into five fundamental categories: integers, non-integer numbers, strings, identifiers, and key words. Constants of most non-numeric types are first classified as strings. The SQL language definition allows specifying type names with strings, and this mechanism can be used in PostgreSQL to start the parser down the correct path. For example, the query:

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";

 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

has two literal constants, of type `text` and `point`. If a type is not specified for a string literal, then the placeholder type `unknown` is assigned initially, to be resolved in later stages as described below.

There are four fundamental SQL constructs requiring distinct type conversion rules in the PostgreSQL parser:

Function calls

Much of the PostgreSQL type system is built around a rich set of functions. Functions can have one or more arguments. Since PostgreSQL permits function overloading, the function name alone does not uniquely identify the function to be called; the parser must select the right function based on the data types of the supplied arguments.

Operators

PostgreSQL allows expressions with prefix and postfix unary (one-argument) operators, as well as binary (two-argument) operators. Like functions, operators can be overloaded, so the same problem of selecting the right operator exists.

Value Storage

SQL INSERT and UPDATE statements place the results of expressions into a table. The expressions in the statement must be matched up with, and perhaps converted to, the types of the target columns.

UNION, CASE, and related constructs

Since all query results from a unionized SELECT statement must appear in a single set of columns, the types of the results of each SELECT clause must be matched up and converted to a uniform set. Similarly, the result expressions of a CASE construct must be converted to a common type so that the CASE expression as a whole has a known output type. Some other constructs, such as ARRAY[] and the GREATEST and LEAST functions, likewise require determination of a common type for several subexpressions.

The system catalogs store information about which conversions, or *casts*, exist between which data types, and how to perform those conversions. Additional casts can be added by the user with the CREATE CAST command. (This is usually done in conjunction with defining new data types. The set of casts between built-in types has been carefully crafted and is best not altered.)

An additional heuristic provided by the parser allows improved determination of the proper casting behavior among groups of types that have implicit casts. Data types are divided into several basic *type categories*, including boolean, numeric, string, bitstring, datetime, timespan, geometric, network, and user-defined. (For a list see Table 49-55; but note it is also possible to create custom type categories.) Within each category there can be one or more *preferred types*, which are preferred when there is a choice of possible types. With careful selection of preferred types and available implicit casts, it is possible to ensure that ambiguous expressions (those with multiple candidate parsing solutions) can be resolved in a useful way.

All type conversion rules are designed with several principles in mind:

- Implicit conversions should never have surprising or unpredictable outcomes.
- There should be no extra overhead in the parser or executor if a query does not need implicit type conversion. That is, if a query is well-formed and the types already match, then the query should execute without spending extra time in the parser and without introducing unnecessary implicit conversion calls in the query.
- Additionally, if a query usually requires an implicit conversion for a function, and if then the user defines a new function with the correct argument types, the parser should use this new function and no longer do implicit conversion to use the old function.

10.2. Operators

The specific operator that is referenced by an operator expression is determined using the following procedure. Note that this procedure is indirectly affected by the precedence of the operators involved, since that will determine which sub-expressions are taken to be the inputs of which operators. See Section 4.1.6 for more information.

Operator Type Resolution

1. Select the operators to be considered from the `pg_operator` system catalog. If a non-schema-qualified operator name was used (the usual case), the operators considered are those with the matching name and argument count that are visible in the current search path (see Section 5.8.3). If a qualified operator name was given, only operators in the specified schema are considered.

    a. If the search path finds multiple operators with identical argument types, only the one appearing earliest in the path is considered. Operators with different argument types are considered on an equal footing regardless of search path position.

2. Check for an operator accepting exactly the input argument types. If one exists (there can be only one exact match in the set of operators considered), use it. Lack of an exact match creates a security hazard when calling, via qualified name [1] (not typical), any operator found in a schema that permits untrusted users to create objects. In such situations, cast arguments to force an exact match.

    a. If one argument of a binary operator invocation is of the `unknown` type, then assume it is the same type as the other argument for this check. Invocations involving two `unknown` inputs, or a unary operator with an `unknown` input, will never find a match at this step.

    b. If one argument of a binary operator invocation is of the `unknown` type and the other is of a domain type, next check to see if there is an operator accepting exactly the domain's base type on both sides; if so, use it.

3. Look for the best match.

    a. Discard candidate operators for which the input types do not match and cannot be converted (using an implicit conversion) to match. `unknown` literals are assumed to be convertible to anything for this purpose. If only one candidate remains, use it; else continue to the next step.

    b. If any input argument is of a domain type, treat it as being of the domain's base type for all subsequent steps. This ensures that domains act like their base types for purposes of ambiguous-operator resolution.

    c. Run through all candidates and keep those with the most exact matches on input types. Keep all candidates if none have exact matches. If only one candidate remains, use it; else continue to the next step.

    d. Run through all candidates and keep those that accept preferred types (of the input data type's type category) at the most positions where type conversion will be required. Keep all candidates if none accept preferred types. If only one candidate remains, use it; else continue to the next step.

    e. If any input arguments are `unknown`, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the `string` category if any candidate accepts that category. (This bias towards string is appropriate since an unknown-type literal looks like a string.) Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Now discard candidates that do not accept the selected type category. Furthermore, if any candidate accepts a preferred type in that category, discard

1. The hazard does not arise with a non-schema-qualified name, because a search path containing schemas that permit untrusted users to create objects is not a secure schema usage pattern.

candidates that accept non-preferred types for that argument. Keep all candidates if none survive these tests. If only one candidate remains, use it; else continue to the next step.

f. If there are both `unknown` and known-type arguments, and all the known-type arguments have the same type, assume that the `unknown` arguments are also of that type, and check which candidates can accept that type at the `unknown`-argument positions. If exactly one candidate passes this test, use it. Otherwise, fail.

Some examples follow.

Example 10-1. Factorial Operator Type Resolution

There is only one factorial operator (postfix `!`) defined in the standard catalog, and it takes an argument of type `bigint`. The scanner assigns an initial type of `integer` to the argument in this query expression:

```
SELECT 40 ! AS "40 factorial";

                    40 factorial
-----------------------------------------------
 815915283247897734345611269596115894272000000000
(1 row)
```

So the parser does a type conversion on the operand and the query is equivalent to:

```
SELECT CAST(40 AS bigint) ! AS "40 factorial";
```

Example 10-2. String Concatenation Operator Type Resolution

A string-like syntax is used for working with string types and for working with complex extension types. Strings with unspecified type are matched with likely operator candidates.

An example with one unspecified argument:

```
SELECT text 'abc' || 'def' AS "text and unknown";

 text and unknown
------------------
 abcdef
(1 row)
```

In this case the parser looks to see if there is an operator taking `text` for both arguments. Since there is, it assumes that the second argument should be interpreted as type `text`.

Here is a concatenation of two values of unspecified types:

```
SELECT 'abc' || 'def' AS "unspecified";

 unspecified
-------------
 abcdef
(1 row)
```

In this case there is no initial hint for which type to use, since no types are specified in the query. So, the parser looks for all candidate operators and finds that there are candidates accepting both string-category and bit-string-category inputs. Since string category is preferred when available, that category is selected,

and then the preferred type for strings, `text`, is used as the specific type to resolve the unknown-type literals as.

Example 10-3. Absolute-Value and Negation Operator Type Resolution

The PostgreSQL operator catalog has several entries for the prefix operator `@`, all of which implement absolute-value operations for various numeric data types. One of these entries is for type `float8`, which is the preferred type in the numeric category. Therefore, PostgreSQL will use that entry when faced with an `unknown` input:

```
SELECT @ '-4.5' AS "abs";
 abs
-----
 4.5
(1 row)
```

Here the system has implicitly resolved the unknown-type literal as type `float8` before applying the chosen operator. We can verify that `float8` and not some other type was used:

```
SELECT @ '-4.5e500' AS "abs";

ERROR:  "-4.5e500" is out of range for type double precision
```

On the other hand, the prefix operator ~ (bitwise negation) is defined only for integer data types, not for `float8`. So, if we try a similar case with ~, we get:

```
SELECT ~ '20' AS "negation";

ERROR:  operator is not unique: ~ "unknown"
HINT:  Could not choose a best candidate operator. You might need to add
explicit type casts.
```

This happens because the system cannot decide which of the several possible ~ operators should be preferred. We can help it out with an explicit cast:

```
SELECT ~ CAST('20' AS int8) AS "negation";

 negation
----------
      -21
(1 row)
```

Example 10-4. Array Inclusion Operator Type Resolution

Here is another example of resolving an operator with one known and one unknown input:

```
SELECT array[1,2] <@ '{1,2,3}' as "is subset";

 is subset
-----------
 t
(1 row)
```

The PostgreSQL operator catalog has several entries for the infix operator <@, but the only two that could possibly accept an integer array on the left-hand side are array inclusion (anyarray <@ anyarray) and range inclusion (anyelement <@ anyrange). Since none of these polymorphic pseudo-types (see Section 8.20) are considered preferred, the parser cannot resolve the ambiguity on that basis. However, step 3.f tells it to assume that the unknown-type literal is of the same type as the other input, that is, integer array. Now only one of the two operators can match, so array inclusion is selected. (Had range inclusion been selected, we would have gotten an error, because the string does not have the right format to be a range literal.)

Example 10-5. Custom Operator on a Domain Type

Users sometimes try to declare operators applying just to a domain type. This is possible but is not nearly as useful as it might seem, because the operator resolution rules are designed to select operators applying to the domain's base type. As an example consider

```
CREATE DOMAIN mytext AS text CHECK(...);
CREATE FUNCTION mytext_eq_text (mytext, text) RETURNS boolean AS ...;
CREATE OPERATOR = (procedure=mytext_eq_text, leftarg=mytext, rightarg=text);
CREATE TABLE mytable (val mytext);

SELECT * FROM mytable WHERE val = 'foo';
```

This query will not use the custom operator. The parser will first see if there is a mytext = mytext operator (step 2.a), which there is not; then it will consider the domain's base type text, and see if there is a text = text operator (step 2.b), which there is; so it resolves the unknown-type literal as text and uses the text = text operator. The only way to get the custom operator to be used is to explicitly cast the literal:

```
SELECT * FROM mytable WHERE val = text 'foo';
```

so that the mytext = text operator is found immediately according to the exact-match rule. If the best-match rules are reached, they actively discriminate against operators on domain types. If they did not, such an operator would create too many ambiguous-operator failures, because the casting rules always consider a domain as castable to or from its base type, and so the domain operator would be considered usable in all the same cases as a similarly-named operator on the base type.

10.3. Functions

The specific function that is referenced by a function call is determined using the following procedure.

Function Type Resolution

1. Select the functions to be considered from the pg_proc system catalog. If a non-schema-qualified function name was used, the functions considered are those with the matching name and argument count that are visible in the current search path (see Section 5.8.3). If a qualified function name was given, only functions in the specified schema are considered.

a. If the search path finds multiple functions of identical argument types, only the one appearing earliest in the path is considered. Functions of different argument types are considered on an equal footing regardless of search path position.

b. If a function is declared with a `VARIADIC` array parameter, and the call does not use the `VARIADIC` keyword, then the function is treated as if the array parameter were replaced by one or more occurrences of its element type, as needed to match the call. After such expansion the function might have effective argument types identical to some non-variadic function. In that case the function appearing earlier in the search path is used, or if the two functions are in the same schema, the non-variadic one is preferred.

   This creates a security hazard when calling, via qualified name [2], a variadic function found in a schema that permits untrusted users to create objects. A malicious user can take control and execute arbitrary SQL functions as though you executed them. Substitute a call bearing the `VARIADIC` keyword, which bypasses this hazard. Calls populating `VARIADIC "any"` parameters often have no equivalent formulation containing the `VARIADIC` keyword. To issue those calls safely, the function's schema must permit only trusted users to create objects.

c. Functions that have default values for parameters are considered to match any call that omits zero or more of the defaultable parameter positions. If more than one such function matches a call, the one appearing earliest in the search path is used. If there are two or more such functions in the same schema with identical parameter types in the non-defaulted positions (which is possible if they have different sets of defaultable parameters), the system will not be able to determine which to prefer, and so an "ambiguous function call" error will result if no better match to the call can be found.

   This creates an availability hazard when calling, via qualified name[2], any function found in a schema that permits untrusted users to create objects. A malicious user can create a function with the name of an existing function, replicating that function's parameters and appending novel parameters having default values. This precludes new calls to the original function. To forestall this hazard, place functions in schemas that permit only trusted users to create objects.

2. Check for a function accepting exactly the input argument types. If one exists (there can be only one exact match in the set of functions considered), use it. Lack of an exact match creates a security hazard when calling, via qualified name[2], a function found in a schema that permits untrusted users to create objects. In such situations, cast arguments to force an exact match. (Cases involving `unknown` will never find a match at this step.)

3. If no exact match is found, see if the function call appears to be a special type conversion request. This happens if the function call has just one argument and the function name is the same as the (internal) name of some data type. Furthermore, the function argument must be either an unknown-type literal, or a type that is binary-coercible to the named data type, or a type that could be converted to the named data type by applying that type's I/O functions (that is, the conversion is either to or from one

2. The hazard does not arise with a non-schema-qualified name, because a search path containing schemas that permit untrusted users to create objects is not a secure schema usage pattern.
2. The hazard does not arise with a non-schema-qualified name, because a search path containing schemas that permit untrusted users to create objects is not a secure schema usage pattern.
2. The hazard does not arise with a non-schema-qualified name, because a search path containing schemas that permit untrusted users to create objects is not a secure schema usage pattern.

of the standard string types). When these conditions are met, the function call is treated as a form of `CAST` specification. [3]

4. Look for the best match.

    a. Discard candidate functions for which the input types do not match and cannot be converted (using an implicit conversion) to match. `unknown` literals are assumed to be convertible to anything for this purpose. If only one candidate remains, use it; else continue to the next step.

    b. If any input argument is of a domain type, treat it as being of the domain's base type for all subsequent steps. This ensures that domains act like their base types for purposes of ambiguous-function resolution.

    c. Run through all candidates and keep those with the most exact matches on input types. Keep all candidates if none have exact matches. If only one candidate remains, use it; else continue to the next step.

    d. Run through all candidates and keep those that accept preferred types (of the input data type's type category) at the most positions where type conversion will be required. Keep all candidates if none accept preferred types. If only one candidate remains, use it; else continue to the next step.

    e. If any input arguments are `unknown`, check the type categories accepted at those argument positions by the remaining candidates. At each position, select the `string` category if any candidate accepts that category. (This bias towards string is appropriate since an unknown-type literal looks like a string.) Otherwise, if all the remaining candidates accept the same type category, select that category; otherwise fail because the correct choice cannot be deduced without more clues. Now discard candidates that do not accept the selected type category. Furthermore, if any candidate accepts a preferred type in that category, discard candidates that accept non-preferred types for that argument. Keep all candidates if none survive these tests. If only one candidate remains, use it; else continue to the next step.

    f. If there are both `unknown` and known-type arguments, and all the known-type arguments have the same type, assume that the `unknown` arguments are also of that type, and check which candidates can accept that type at the `unknown`-argument positions. If exactly one candidate passes this test, use it. Otherwise, fail.

Note that the "best match" rules are identical for operator and function type resolution. Some examples follow.

Example 10-6. Rounding Function Argument Type Resolution

There is only one `round` function that takes two arguments; it takes a first argument of type `numeric` and a second argument of type `integer`. So the following query automatically converts the first argument of type `integer` to `numeric`:

```
SELECT round(4, 4);
```

3. The reason for this step is to support function-style cast specifications in cases where there is not an actual cast function. If there is a cast function, it is conventionally named after its output type, and so there is no need to have a special case. See CREATE CAST for additional commentary.

```
 round
--------
 4.0000
(1 row)
```

That query is actually transformed by the parser to:

```
SELECT round(CAST (4 AS numeric), 4);
```

Since numeric constants with decimal points are initially assigned the type `numeric`, the following query will require no type conversion and therefore might be slightly more efficient:

```
SELECT round(4.0, 4);
```

Example 10-7. Variadic Function Resolution

```
CREATE FUNCTION public.variadic_example(VARIADIC numeric[]) RETURNS int
  LANGUAGE sql AS 'SELECT 1';
CREATE FUNCTION
```

This function accepts, but does not require, the VARIADIC keyword. It tolerates both integer and numeric arguments:

```
SELECT public.variadic_example(0),
       public.variadic_example(0.0),
       public.variadic_example(VARIADIC array[0.0]);
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                1 |                1 |                1
(1 row)
```

However, the first and second calls will prefer more-specific functions, if available:

```
CREATE FUNCTION public.variadic_example(numeric) RETURNS int
  LANGUAGE sql AS 'SELECT 2';
CREATE FUNCTION

CREATE FUNCTION public.variadic_example(int) RETURNS int
  LANGUAGE sql AS 'SELECT 3';
CREATE FUNCTION

SELECT public.variadic_example(0),
       public.variadic_example(0.0),
       public.variadic_example(VARIADIC array[0.0]);
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                3 |                2 |                1
(1 row)
```

Given the default configuration and only the first function existing, the first and second calls are insecure. Any user could intercept them by creating the second or third function. By matching the argument type exactly and using the `VARIADIC` keyword, the third call is secure.

Example 10-8. Substring Function Type Resolution

There are several `substr` functions, one of which takes types `text` and `integer`. If called with a string constant of unspecified type, the system chooses the candidate function that accepts an argument of the preferred category `string` (namely of type `text`).

```
SELECT substr('1234', 3);

 substr
--------
     34
(1 row)
```

If the string is declared to be of type `varchar`, as might be the case if it comes from a table, then the parser will try to convert it to become `text`:

```
SELECT substr(varchar '1234', 3);

 substr
--------
     34
(1 row)
```

This is transformed by the parser to effectively become:

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

> **Note:** The parser learns from the `pg_cast` catalog that `text` and `varchar` are binary-compatible, meaning that one can be passed to a function that accepts the other without doing any physical conversion. Therefore, no type conversion call is really inserted in this case.

And, if the function is called with an argument of type `integer`, the parser will try to convert that to `text`:

```
SELECT substr(1234, 3);
ERROR:  function substr(integer, integer) does not exist
HINT:  No function matches the given name and argument types. You might need
to add explicit type casts.
```

This does not work because `integer` does not have an implicit cast to `text`. An explicit cast will work, however:

```
SELECT substr(CAST (1234 AS text), 3);

 substr
--------
     34
(1 row)
```

10.4. Value Storage

Values to be inserted into a table are converted to the destination column's data type according to the following steps.

Value Storage Type Conversion

1. Check for an exact match with the target.
2. Otherwise, try to convert the expression to the target type. This is possible if an *assignment cast* between the two types is registered in the `pg_cast` catalog (see CREATE CAST). Alternatively, if the expression is an unknown-type literal, the contents of the literal string will be fed to the input conversion routine for the target type.
3. Check to see if there is a sizing cast for the target type. A sizing cast is a cast from that type to itself. If one is found in the `pg_cast` catalog, apply it to the expression before storing into the destination column. The implementation function for such a cast always takes an extra parameter of type `integer`, which receives the destination column's `atttypmod` value (typically its declared length, although the interpretation of `atttypmod` varies for different data types), and it may take a third `boolean` parameter that says whether the cast is explicit or implicit. The cast function is responsible for applying any length-dependent semantics such as size checking or truncation.

Example 10-9. `character` Storage Type Conversion

For a target column declared as `character(20)` the following statement shows that the stored value is sized correctly:

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, octet_length(v) FROM vv;

          v           | octet_length
----------------------+--------------
 abcdef               |           20
(1 row)
```

What has really happened here is that the two unknown literals are resolved to `text` by default, allowing the `||` operator to be resolved as `text` concatenation. Then the `text` result of the operator is converted to `bpchar` ("blank-padded char", the internal name of the `character` data type) to match the target column type. (Since the conversion from `text` to `bpchar` is binary-coercible, this conversion does not insert any real function call.) Finally, the sizing function `bpchar(bpchar, integer, boolean)` is found in the system catalog and applied to the operator's result and the stored column length. This type-specific function performs the required length check and addition of padding spaces.

10.5. `UNION`, `CASE`, and Related Constructs

SQL `UNION` constructs must match up possibly dissimilar types to become a single result set. The resolution algorithm is applied separately to each output column of a union query. The `INTERSECT` and `EXCEPT`

constructs resolve dissimilar types in the same way as UNION. Some other constructs, including CASE, ARRAY, VALUES, and the GREATEST and LEAST functions, use the identical algorithm to match up their component expressions and select a result data type.

Type Resolution for UNION, CASE, and Related Constructs

1. If all inputs are of the same type, and it is not unknown, resolve as that type.
2. If any input is of a domain type, treat it as being of the domain's base type for all subsequent steps. [4]
3. If all inputs are of type unknown, resolve as type text (the preferred type of the string category). Otherwise, unknown inputs are ignored.
4. If the non-unknown inputs are not all of the same type category, fail.
5. Select the first non-unknown input type as the candidate type, then consider each other non-unknown input type, left to right. [5] If the candidate type can be implicitly converted to the other type, but not vice-versa, select the other type as the new candidate type. Then continue considering the remaining inputs. If, at any stage of this process, a preferred type is selected, stop considering additional inputs.
6. Convert all inputs to the final candidate type. Fail if there is not an implicit conversion from a given input type to the candidate type.

Some examples follow.

Example 10-10. Type Resolution with Underspecified Types in a Union

```
SELECT text 'a' AS "text" UNION SELECT 'b';

 text
------
 a
 b
(2 rows)
```

Here, the unknown-type literal 'b' will be resolved to type text.

Example 10-11. Type Resolution in a Simple Union

```
SELECT 1.2 AS "numeric" UNION SELECT 1;

 numeric
---------
       1
     1.2
(2 rows)
```

4. Somewhat like the treatment of domain inputs for operators and functions, this behavior allows a domain type to be preserved through a UNION or similar construct, so long as the user is careful to ensure that all inputs are implicitly or explicitly of that exact type. Otherwise the domain's base type will be used.
5. For historical reasons, CASE treats its ELSE clause (if any) as the "first" input, with the THEN clauses(s) considered after that. In all other cases, "left to right" means the order in which the expressions appear in the query text.

The literal `1.2` is of type `numeric`, and the `integer` value `1` can be cast implicitly to `numeric`, so that type is used.

Example 10-12. Type Resolution in a Transposed Union

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

 real
------
    1
  2.2
(2 rows)
```

Here, since type `real` cannot be implicitly cast to `integer`, but `integer` can be implicitly cast to `real`, the union result type is resolved as `real`.

Example 10-13. Type Resolution in a Nested Union

```
SELECT NULL UNION SELECT NULL UNION SELECT 1;

ERROR:  UNION types text and integer cannot be matched
```

This failure occurs because PostgreSQL treats multiple `UNION`s as a nest of pairwise operations; that is, this input is the same as

```
(SELECT NULL UNION SELECT NULL) UNION SELECT 1;
```

The inner `UNION` is resolved as emitting type `text`, according to the rules given above. Then the outer `UNION` has inputs of types `text` and `integer`, leading to the observed error. The problem can be fixed by ensuring that the leftmost `UNION` has at least one input of the desired result type.

`INTERSECT` and `EXCEPT` operations are likewise resolved pairwise. However, the other constructs described in this section consider all of their inputs in one resolution step.

Chapter 11. Indexes

Indexes are a common way to enhance database performance. An index allows the database server to find and retrieve specific rows much faster than it could do without an index. But indexes also add overhead to the database system as a whole, so they should be used sensibly.

11.1. Introduction

Suppose we have a table similar to this:

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

and the application issues many queries of the form:

```
SELECT content FROM test1 WHERE id = constant;
```

With no advance preparation, the system would have to scan the entire `test1` table, row by row, to find all matching entries. If there are many rows in `test1` and only a few rows (perhaps zero or one) that would be returned by such a query, this is clearly an inefficient method. But if the system has been instructed to maintain an index on the `id` column, it can use a more efficient method for locating matching rows. For instance, it might only have to walk a few levels deep into a search tree.

A similar approach is used in most non-fiction books: terms and concepts that are frequently looked up by readers are collected in an alphabetic index at the end of the book. The interested reader can scan the index relatively quickly and flip to the appropriate page(s), rather than having to read the entire book to find the material of interest. Just as it is the task of the author to anticipate the items that readers are likely to look up, it is the task of the database programmer to foresee which indexes will be useful.

The following command can be used to create an index on the `id` column, as discussed:

```
CREATE INDEX test1_id_index ON test1 (id);
```

The name `test1_id_index` can be chosen freely, but you should pick something that enables you to remember later what the index was for.

To remove an index, use the `DROP INDEX` command. Indexes can be added to and removed from tables at any time.

Once an index is created, no further intervention is required: the system will update the index when the table is modified, and it will use the index in queries when it thinks doing so would be more efficient than a sequential table scan. But you might have to run the `ANALYZE` command regularly to update statistics to allow the query planner to make educated decisions. See Chapter 14 for information about how to find out whether an index is used and when and why the planner might choose *not* to use an index.

Indexes can also benefit `UPDATE` and `DELETE` commands with search conditions. Indexes can moreover be used in join searches. Thus, an index defined on a column that is part of a join condition can also significantly speed up queries with joins.

Creating an index on a large table can take a long time. By default, PostgreSQL allows reads (SELECT statements) to occur on the table in parallel with index creation, but writes (INSERT, UPDATE, DELETE) are blocked until the index build is finished. In production environments this is often unacceptable. It is possible to allow writes to occur in parallel with index creation, but there are several caveats to be aware of — for more information see *Building Indexes Concurrently*.

After an index is created, the system has to keep it synchronized with the table. This adds overhead to data manipulation operations. Therefore indexes that are seldom or never used in queries should be removed.

11.2. Index Types

PostgreSQL provides several index types: B-tree, Hash, GiST, SP-GiST, GIN and BRIN. Each index type uses a different algorithm that is best suited to different types of queries. By default, the CREATE INDEX command creates B-tree indexes, which fit the most common situations.

B-trees can handle equality and range queries on data that can be sorted into some ordering. In particular, the PostgreSQL query planner will consider using a B-tree index whenever an indexed column is involved in a comparison using one of these operators:

```
<
<=
=
>=
>
```

Constructs equivalent to combinations of these operators, such as BETWEEN and IN, can also be implemented with a B-tree index search. Also, an IS NULL or IS NOT NULL condition on an index column can be used with a B-tree index.

The optimizer can also use a B-tree index for queries involving the pattern matching operators LIKE and ~ *if* the pattern is a constant and is anchored to the beginning of the string — for example, `col LIKE 'foo%'` or `col ~ '^foo'`, but not `col LIKE '%bar'`. However, if your database does not use the C locale you will need to create the index with a special operator class to support indexing of pattern-matching queries; see Section 11.9 below. It is also possible to use B-tree indexes for ILIKE and ~*, but only if the pattern starts with non-alphabetic characters, i.e., characters that are not affected by upper/lower case conversion.

B-tree indexes can also be used to retrieve data in sorted order. This is not always faster than a simple scan and sort, but it is often helpful.

Hash indexes can only handle simple equality comparisons. The query planner will consider using a hash index whenever an indexed column is involved in a comparison using the = operator. The following command is used to create a hash index:

```
CREATE INDEX name ON table USING HASH (column);
```

> **Caution**
>
> Hash index operations are not presently WAL-logged, so hash indexes might need to be rebuilt with `REINDEX` after a database crash if there were unwritten changes. Also, changes to hash indexes are not replicated over streaming or file-based replication after the initial base backup, so they give wrong answers to queries that subsequently use them. For these reasons, hash index use is presently discouraged.

GiST indexes are not a single kind of index, but rather an infrastructure within which many different indexing strategies can be implemented. Accordingly, the particular operators with which a GiST index can be used vary depending on the indexing strategy (the *operator class*). As an example, the standard distribution of PostgreSQL includes GiST operator classes for several two-dimensional geometric data types, which support indexed queries using these operators:

```
<<
&<
&>
>>
<<|
&<|
|&>
|>>
@>
<@
~=
&&
```

(See Section 9.11 for the meaning of these operators.) The GiST operator classes included in the standard distribution are documented in Table 59-1. Many other GiST operator classes are available in the `contrib` collection or as separate projects. For more information see Chapter 59.

GiST indexes are also capable of optimizing "nearest-neighbor" searches, such as

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

which finds the ten places closest to a given target point. The ability to do this is again dependent on the particular operator class being used. In Table 59-1, operators that can be used in this way are listed in the column "Ordering Operators".

SP-GiST indexes, like GiST indexes, offer an infrastructure that supports various kinds of searches. SP-GiST permits implementation of a wide range of different non-balanced disk-based data structures, such as quadtrees, k-d trees, and radix trees (tries). As an example, the standard distribution of PostgreSQL includes SP-GiST operator classes for two-dimensional points, which support indexed queries using these operators:

```
<<
>>
~=
<@
```

```
<^
>^
```

(See Section 9.11 for the meaning of these operators.) The SP-GiST operator classes included in the standard distribution are documented in Table 60-1. For more information see Chapter 60.

GIN indexes are inverted indexes which can handle values that contain more than one key, arrays for example. Like GiST and SP-GiST, GIN can support many different user-defined indexing strategies and the particular operators with which a GIN index can be used vary depending on the indexing strategy. As an example, the standard distribution of PostgreSQL includes GIN operator classes for one-dimensional arrays, which support indexed queries using these operators:

```
<@
@>
=
&&
```

(See Section 9.18 for the meaning of these operators.) The GIN operator classes included in the standard distribution are documented in Table 61-1. Many other GIN operator classes are available in the `contrib` collection or as separate projects. For more information see Chapter 61.

BRIN indexes (a shorthand for Block Range indexes) store summaries about the values stored in consecutive table physical block ranges. Like GiST, SP-GiST and GIN, BRIN can support many different indexing strategies, and the particular operators with which a BRIN index can be used vary depending on the indexing strategy. For data types that have a linear sort order, the indexed data corresponds to the minimum and maximum values of the values in the column for each block range, which support indexed queries using these operators:

```
<
<=
=
>=
>
```

The BRIN operator classes included in the standard distribution are documented in Table 62-1. For more information see Chapter 62.

11.3. Multicolumn Indexes

An index can be defined on more than one column of a table. For example, if you have a table of this form:

```
CREATE TABLE test2 (
  major int,
  minor int,
  name varchar
);
```

(say, you keep your `/dev` directory in a database...) and you frequently issue queries like:

```
SELECT name FROM test2 WHERE major = constant AND minor = constant;
```

then it might be appropriate to define an index on the columns `major` and `minor` together, e.g.:

```
CREATE INDEX test2_mm_idx ON test2 (major, minor);
```

Currently, only the B-tree, GiST, GIN, and BRIN index types support multicolumn indexes. Up to 32 columns can be specified. (This limit can be altered when building PostgreSQL; see the file `pg_config_manual.h`.)

A multicolumn B-tree index can be used with query conditions that involve any subset of the index's columns, but the index is most efficient when there are constraints on the leading (leftmost) columns. The exact rule is that equality constraints on leading columns, plus any inequality constraints on the first column that does not have an equality constraint, will be used to limit the portion of the index that is scanned. Constraints on columns to the right of these columns are checked in the index, so they save visits to the table proper, but they do not reduce the portion of the index that has to be scanned. For example, given an index on `(a, b, c)` and a query condition `WHERE a = 5 AND b >= 42 AND c < 77`, the index would have to be scanned from the first entry with `a` = 5 and `b` = 42 up through the last entry with `a` = 5. Index entries with `c` >= 77 would be skipped, but they'd still have to be scanned through. This index could in principle be used for queries that have constraints on `b` and/or `c` with no constraint on `a` — but the entire index would have to be scanned, so in most cases the planner would prefer a sequential table scan over using the index.

A multicolumn GiST index can be used with query conditions that involve any subset of the index's columns. Conditions on additional columns restrict the entries returned by the index, but the condition on the first column is the most important one for determining how much of the index needs to be scanned. A GiST index will be relatively ineffective if its first column has only a few distinct values, even if there are many distinct values in additional columns.

A multicolumn GIN index can be used with query conditions that involve any subset of the index's columns. Unlike B-tree or GiST, index search effectiveness is the same regardless of which index column(s) the query conditions use.

A multicolumn BRIN index can be used with query conditions that involve any subset of the index's columns. Like GIN and unlike B-tree or GiST, index search effectiveness is the same regardless of which index column(s) the query conditions use. The only reason to have multiple BRIN indexes instead of one multicolumn BRIN index on a single table is to have a different `pages_per_range` storage parameter.

Of course, each column must be used with operators appropriate to the index type; clauses that involve other operators will not be considered.

Multicolumn indexes should be used sparingly. In most situations, an index on a single column is sufficient and saves space and time. Indexes with more than three columns are unlikely to be helpful unless the usage of the table is extremely stylized. See also Section 11.5 for some discussion of the merits of different index configurations.

11.4. Indexes and `ORDER BY`

In addition to simply finding the rows to be returned by a query, an index may be able to deliver them in a specific sorted order. This allows a query's `ORDER BY` specification to be honored without a separate

sorting step. Of the index types currently supported by PostgreSQL, only B-tree can produce sorted output — the other index types return matching rows in an unspecified, implementation-dependent order.

The planner will consider satisfying an ORDER BY specification either by scanning an available index that matches the specification, or by scanning the table in physical order and doing an explicit sort. For a query that requires scanning a large fraction of the table, an explicit sort is likely to be faster than using an index because it requires less disk I/O due to following a sequential access pattern. Indexes are more useful when only a few rows need be fetched. An important special case is ORDER BY in combination with LIMIT *n*: an explicit sort will have to process all the data to identify the first *n* rows, but if there is an index matching the ORDER BY, the first *n* rows can be retrieved directly, without scanning the remainder at all.

By default, B-tree indexes store their entries in ascending order with nulls last. This means that a forward scan of an index on column x produces output satisfying ORDER BY x (or more verbosely, ORDER BY x ASC NULLS LAST). The index can also be scanned backward, producing output satisfying ORDER BY x DESC (or more verbosely, ORDER BY x DESC NULLS FIRST, since NULLS FIRST is the default for ORDER BY DESC).

You can adjust the ordering of a B-tree index by including the options ASC, DESC, NULLS FIRST, and/or NULLS LAST when creating the index; for example:

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

An index stored in ascending order with nulls first can satisfy either ORDER BY x ASC NULLS FIRST or ORDER BY x DESC NULLS LAST depending on which direction it is scanned in.

You might wonder why bother providing all four options, when two options together with the possibility of backward scan would cover all the variants of ORDER BY. In single-column indexes the options are indeed redundant, but in multicolumn indexes they can be useful. Consider a two-column index on (x, y): this can satisfy ORDER BY x, y if we scan forward, or ORDER BY x DESC, y DESC if we scan backward. But it might be that the application frequently needs to use ORDER BY x ASC, y DESC. There is no way to get that ordering from a plain index, but it is possible if the index is defined as (x ASC, y DESC) or (x DESC, y ASC).

Obviously, indexes with non-default sort orderings are a fairly specialized feature, but sometimes they can produce tremendous speedups for certain queries. Whether it's worth maintaining such an index depends on how often you use queries that require a special sort ordering.

11.5. Combining Multiple Indexes

A single index scan can only use query clauses that use the index's columns with operators of its operator class and are joined with AND. For example, given an index on (a, b) a query condition like WHERE a = 5 AND b = 6 could use the index, but a query like WHERE a = 5 OR b = 6 could not directly use the index.

Fortunately, PostgreSQL has the ability to combine multiple indexes (including multiple uses of the same index) to handle cases that cannot be implemented by single index scans. The system can form AND and OR conditions across several index scans. For example, a query like WHERE x = 42 OR x = 47 OR x = 53 OR x = 99 could be broken down into four separate scans of an index on x, each scan using one of the query clauses. The results of these scans are then ORed together to produce the result. Another example

is that if we have separate indexes on `x` and `y`, one possible implementation of a query like `WHERE x = 5 AND y = 6` is to use each index with the appropriate query clause and then AND together the index results to identify the result rows.

To combine multiple indexes, the system scans each needed index and prepares a *bitmap* in memory giving the locations of table rows that are reported as matching that index's conditions. The bitmaps are then ANDed and ORed together as needed by the query. Finally, the actual table rows are visited and returned. The table rows are visited in physical order, because that is how the bitmap is laid out; this means that any ordering of the original indexes is lost, and so a separate sort step will be needed if the query has an `ORDER BY` clause. For this reason, and because each additional index scan adds extra time, the planner will sometimes choose to use a simple index scan even though additional indexes are available that could have been used as well.

In all but the simplest applications, there are various combinations of indexes that might be useful, and the database developer must make trade-offs to decide which indexes to provide. Sometimes multicolumn indexes are best, but sometimes it's better to create separate indexes and rely on the index-combination feature. For example, if your workload includes a mix of queries that sometimes involve only column `x`, sometimes only column `y`, and sometimes both columns, you might choose to create two separate indexes on `x` and `y`, relying on index combination to process the queries that use both columns. You could also create a multicolumn index on `(x, y)`. This index would typically be more efficient than index combination for queries involving both columns, but as discussed in Section 11.3, it would be almost useless for queries involving only `y`, so it should not be the only index. A combination of the multicolumn index and a separate index on `y` would serve reasonably well. For queries involving only `x`, the multicolumn index could be used, though it would be larger and hence slower than an index on `x` alone. The last alternative is to create all three indexes, but this is probably only reasonable if the table is searched much more often than it is updated and all three types of query are common. If one of the types of query is much less common than the others, you'd probably settle for creating just the two indexes that best match the common types.

11.6. Unique Indexes

Indexes can also be used to enforce uniqueness of a column's value, or the uniqueness of the combined values of more than one column.

```
CREATE UNIQUE INDEX name ON table (column [, ...]);
```

Currently, only B-tree indexes can be declared unique.

When an index is declared unique, multiple table rows with equal indexed values are not allowed. Null values are not considered equal. A multicolumn unique index will only reject cases where all indexed columns are equal in multiple rows.

PostgreSQL automatically creates a unique index when a unique constraint or primary key is defined for a table. The index covers the columns that make up the primary key or unique constraint (a multicolumn index, if appropriate), and is the mechanism that enforces the constraint.

> **Note:** There's no need to manually create indexes on unique columns; doing so would just duplicate the automatically-created index.

11.7. Indexes on Expressions

An index column need not be just a column of the underlying table, but can be a function or scalar expression computed from one or more columns of the table. This feature is useful to obtain fast access to tables based on the results of computations.

For example, a common way to do case-insensitive comparisons is to use the `lower` function:

```
SELECT * FROM test1 WHERE lower(col1) = 'value';
```

This query can use an index if one has been defined on the result of the `lower(col1)` function:

```
CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));
```

If we were to declare this index `UNIQUE`, it would prevent creation of rows whose `col1` values differ only in case, as well as rows whose `col1` values are actually identical. Thus, indexes on expressions can be used to enforce constraints that are not definable as simple unique constraints.

As another example, if one often does queries like:

```
SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';
```

then it might be worth creating an index like this:

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

The syntax of the `CREATE INDEX` command normally requires writing parentheses around index expressions, as shown in the second example. The parentheses can be omitted when the expression is just a function call, as in the first example.

Index expressions are relatively expensive to maintain, because the derived expression(s) must be computed for each row upon insertion and whenever it is updated. However, the index expressions are *not* recomputed during an indexed search, since they are already stored in the index. In both examples above, the system sees the query as just `WHERE indexedcolumn = 'constant'` and so the speed of the search is equivalent to any other simple index query. Thus, indexes on expressions are useful when retrieval speed is more important than insertion and update speed.

11.8. Partial Indexes

A *partial index* is an index built over a subset of a table; the subset is defined by a conditional expression (called the *predicate* of the partial index). The index contains entries only for those table rows that satisfy the predicate. Partial indexes are a specialized feature, but there are several situations in which they are useful.

One major reason for using a partial index is to avoid indexing common values. Since a query searching for a common value (one that accounts for more than a few percent of all the table rows) will not use the index anyway, there is no point in keeping those rows in the index at all. This reduces the size of the index, which will speed up those queries that do use the index. It will also speed up many table update operations

because the index does not need to be updated in all cases. Example 11-1 shows a possible application of this idea.

Example 11-1. Setting up a Partial Index to Exclude Common Values

Suppose you are storing web server access logs in a database. Most accesses originate from the IP address range of your organization but some are from elsewhere (say, employees on dial-up connections). If your searches by IP are primarily for outside accesses, you probably do not need to index the IP range that corresponds to your organization's subnet.

Assume a table like this:

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

To create a partial index that suits our example, use a command such as this:

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
WHERE NOT (client_ip > inet '192.168.100.0' AND
           client_ip < inet '192.168.100.255');
```

A typical query that can use this index would be:

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

A query that cannot use this index is:

```
SELECT *
FROM access_log
WHERE client_ip = inet '192.168.100.23';
```

Observe that this kind of partial index requires that the common values be predetermined, so such partial indexes are best used for data distributions that do not change. The indexes can be recreated occasionally to adjust for new data distributions, but this adds maintenance effort.

Another possible use for a partial index is to exclude values from the index that the typical query workload is not interested in; this is shown in Example 11-2. This results in the same advantages as listed above, but it prevents the "uninteresting" values from being accessed via that index, even if an index scan might be profitable in that case. Obviously, setting up partial indexes for this kind of scenario will require a lot of care and experimentation.

Example 11-2. Setting up a Partial Index to Exclude Uninteresting Values

If you have a table that contains both billed and unbilled orders, where the unbilled orders take up a small fraction of the total table and yet those are the most-accessed rows, you can improve performance by creating an index on just the unbilled rows. The command to create the index would look like this:

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
    WHERE billed is not true;
```

A possible query to use this index would be:

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```
However, the index can also be used in queries that do not involve `order_nr` at all, e.g.:
```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```
This is not as efficient as a partial index on the `amount` column would be, since the system has to scan the entire index. Yet, if there are relatively few unbilled orders, using this partial index just to find the unbilled orders could be a win.

Note that this query cannot use this index:

```
SELECT * FROM orders WHERE order_nr = 3501;
```
The order 3501 might be among the billed or unbilled orders.

Example 11-2 also illustrates that the indexed column and the column used in the predicate do not need to match. PostgreSQL supports partial indexes with arbitrary predicates, so long as only columns of the table being indexed are involved. However, keep in mind that the predicate must match the conditions used in the queries that are supposed to benefit from the index. To be precise, a partial index can be used in a query only if the system can recognize that the `WHERE` condition of the query mathematically implies the predicate of the index. PostgreSQL does not have a sophisticated theorem prover that can recognize mathematically equivalent expressions that are written in different forms. (Not only is such a general theorem prover extremely difficult to create, it would probably be too slow to be of any real use.) The system can recognize simple inequality implications, for example "x < 1" implies "x < 2"; otherwise the predicate condition must exactly match part of the query's `WHERE` condition or the index will not be recognized as usable. Matching takes place at query planning time, not at run time. As a result, parameterized query clauses do not work with a partial index. For example a prepared query with a parameter might specify "x < ?" which will never imply "x < 2" for all possible values of the parameter.

A third possible use for partial indexes does not require the index to be used in queries at all. The idea here is to create a unique index over a subset of a table, as in Example 11-3. This enforces uniqueness among the rows that satisfy the index predicate, without constraining those that do not.

Example 11-3. Setting up a Partial Unique Index

Suppose that we have a table describing test outcomes. We wish to ensure that there is only one "successful" entry for a given subject and target combination, but there might be any number of "unsuccessful" entries. Here is one way to do it:

```
CREATE TABLE tests (
    subject text,
    target text,
    success boolean,
    ...
);

CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
    WHERE success;
```
This is a particularly efficient approach when there are few successful tests and many unsuccessful ones. It is also possible to allow only one null in a column by creating a unique partial index with an `IS NULL` restriction.

Finally, a partial index can also be used to override the system's query plan choices. Also, data sets with peculiar distributions might cause the system to use an index when it really should not. In that case the index can be set up so that it is not available for the offending query. Normally, PostgreSQL makes reasonable choices about index usage (e.g., it avoids them when retrieving common values, so the earlier example really only saves index size, it is not required to avoid index usage), and grossly incorrect plan choices are cause for a bug report.

Keep in mind that setting up a partial index indicates that you know at least as much as the query planner knows, in particular you know when an index might be profitable. Forming this knowledge requires experience and understanding of how indexes in PostgreSQL work. In most cases, the advantage of a partial index over a regular index will be minimal.

More information about partial indexes can be found in *The case for partial indexes* , *Partial indexing in POSTGRES: research project*, and *Generalized Partial Indexes (cached version)* .

11.9. Operator Classes and Operator Families

An index definition can specify an *operator class* for each column of an index.

```
CREATE INDEX name ON table (column opclass [sort options] [, ...]);
```

The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on the type `int4` would use the `int4_ops` class; this operator class includes comparison functions for values of type `int4`. In practice the default operator class for the column's data type is usually sufficient. The main reason for having operator classes is that for some data types, there could be more than one meaningful index behavior. For example, we might want to sort a complex-number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index. The operator class determines the basic sort ordering (which can then be modified by adding sort options `COLLATE`, `ASC`/`DESC` and/or `NULLS FIRST`/`NULLS LAST`).

There are also some built-in operator classes besides the default ones:

- The operator classes `text_pattern_ops`, `varchar_pattern_ops`, and `bpchar_pattern_ops` support B-tree indexes on the types `text`, `varchar`, and `char` respectively. The difference from the default operator classes is that the values are compared strictly character by character rather than according to the locale-specific collation rules. This makes these operator classes suitable for use by queries involving pattern matching expressions (`LIKE` or POSIX regular expressions) when the database does not use the standard "C" locale. As an example, you might index a `varchar` column like this:

  ```
  CREATE INDEX test_index ON test_table (col varchar_pattern_ops);
  ```

  Note that you should also create an index with the default operator class if you want queries involving ordinary <, <=, >, or >= comparisons to use an index. Such queries cannot use the *xxx*`_pattern_ops` operator classes. (Ordinary equality comparisons can use these operator classes, however.) It is possible to create multiple indexes on the same column with different operator classes. If you do use the C locale, you do not need the *xxx*`_pattern_ops` operator classes, because an index with the default operator class is usable for pattern-matching queries in the C locale.

The following query shows all defined operator classes:

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opc.opcintype::regtype AS indexed_type,
       opc.opcdefault AS is_default
    FROM pg_am am, pg_opclass opc
    WHERE opc.opcmethod = am.oid
    ORDER BY index_method, opclass_name;
```

An operator class is actually just a subset of a larger structure called an *operator family*. In cases where several data types have similar behaviors, it is frequently useful to define cross-data-type operators and allow these to work with indexes. To do this, the operator classes for each of the types must be grouped into the same operator family. The cross-type operators are members of the family, but are not associated with any single class within the family.

This expanded version of the previous query shows the operator family each operator class belongs to:

```
SELECT am.amname AS index_method,
       opc.opcname AS opclass_name,
       opf.opfname AS opfamily_name,
       opc.opcintype::regtype AS indexed_type,
       opc.opcdefault AS is_default
    FROM pg_am am, pg_opclass opc, pg_opfamily opf
    WHERE opc.opcmethod = am.oid AND
          opc.opcfamily = opf.oid
    ORDER BY index_method, opclass_name;
```

This query shows all defined operator families and all the operators included in each family:

```
SELECT am.amname AS index_method,
       opf.opfname AS opfamily_name,
       amop.amopopr::regoperator AS opfamily_operator
    FROM pg_am am, pg_opfamily opf, pg_amop amop
    WHERE opf.opfmethod = am.oid AND
          amop.amopfamily = opf.oid
    ORDER BY index_method, opfamily_name, opfamily_operator;
```

11.10. Indexes and Collations

An index can support only one collation per index column. If multiple collations are of interest, multiple indexes may be needed.

Consider these statements:

```
CREATE TABLE test1c (
    id integer,
```

```
    content varchar COLLATE "x"
);

CREATE INDEX test1c_content_index ON test1c (content);
```

The index automatically uses the collation of the underlying column. So a query of the form

```
SELECT * FROM test1c WHERE content > constant;
```

could use the index, because the comparison will by default use the collation of the column. However, this index cannot accelerate queries that involve some other collation. So if queries of the form, say,

```
SELECT * FROM test1c WHERE content > constant COLLATE "y";
```

are also of interest, an additional index could be created that supports the `"y"` collation, like this:

```
CREATE INDEX test1c_content_y_index ON test1c (content COLLATE "y");
```

11.11. Examining Index Usage

Although indexes in PostgreSQL do not need maintenance or tuning, it is still important to check which indexes are actually used by the real-life query workload. Examining index usage for an individual query is done with the EXPLAIN command; its application for this purpose is illustrated in Section 14.1. It is also possible to gather overall statistics about index usage in a running server, as described in Section 27.2.

It is difficult to formulate a general procedure for determining which indexes to create. There are a number of typical cases that have been shown in the examples throughout the previous sections. A good deal of experimentation is often necessary. The rest of this section gives some tips for that:

- Always run ANALYZE first. This command collects statistics about the distribution of the values in the table. This information is required to estimate the number of rows returned by a query, which is needed by the planner to assign realistic costs to each possible query plan. In absence of any real statistics, some default values are assumed, which are almost certain to be inaccurate. Examining an application's index usage without having run `ANALYZE` is therefore a lost cause. See Section 23.1.3 and Section 23.1.6 for more information.
- Use real data for experimentation. Using test data for setting up indexes will tell you what indexes you need for the test data, but that is all.

  It is especially fatal to use very small test data sets. While selecting 1000 out of 100000 rows could be a candidate for an index, selecting 1 out of 100 rows will hardly be, because the 100 rows probably fit within a single disk page, and there is no plan that can beat sequentially fetching 1 disk page.

  Also be careful when making up test data, which is often unavoidable when the application is not yet in production. Values that are very similar, completely random, or inserted in sorted order will skew the statistics away from the distribution that real data would have.

- When indexes are not used, it can be useful for testing to force their use. There are run-time parameters that can turn off various plan types (see Section 18.7.1). For instance, turning off sequential scans (`enable_seqscan`) and nested-loop joins (`enable_nestloop`), which are the most basic plans, will force the system to use a different plan. If the system still chooses a sequential scan or nested-loop join then there is probably a more fundamental reason why the index is not being used; for example, the query condition does not match the index. (What kind of query can use what kind of index is explained in the previous sections.)
- If forcing index usage does use the index, then there are two possibilities: Either the system is right and using the index is indeed not appropriate, or the cost estimates of the query plans are not reflecting reality. So you should time your query with and without indexes. The `EXPLAIN ANALYZE` command can be useful here.
- If it turns out that the cost estimates are wrong, there are, again, two possibilities. The total cost is computed from the per-row costs of each plan node times the selectivity estimate of the plan node. The costs estimated for the plan nodes can be adjusted via run-time parameters (described in Section 18.7.2). An inaccurate selectivity estimate is due to insufficient statistics. It might be possible to improve this by tuning the statistics-gathering parameters (see ALTER TABLE).

  If you do not succeed in adjusting the costs to be more appropriate, then you might have to resort to forcing index usage explicitly. You might also want to contact the PostgreSQL developers to examine the issue.

Chapter 12. Full Text Search

12.1. Introduction

Full Text Searching (or just *text search*) provides the capability to identify natural-language *documents* that satisfy a *query*, and optionally to sort them by relevance to the query. The most common type of search is to find all documents containing given *query terms* and return them in order of their *similarity* to the query. Notions of `query` and `similarity` are very flexible and depend on the specific application. The simplest search considers `query` as a set of words and `similarity` as the frequency of query words in the document.

Textual search operators have existed in databases for years. PostgreSQL has `~`, `~*`, `LIKE`, and `ILIKE` operators for textual data types, but they lack many essential properties required by modern information systems:

- There is no linguistic support, even for English. Regular expressions are not sufficient because they cannot easily handle derived words, e.g., `satisfies` and `satisfy`. You might miss documents that contain `satisfies`, although you probably would like to find them when searching for `satisfy`. It is possible to use `OR` to search for multiple derived forms, but this is tedious and error-prone (some words can have several thousand derivatives).
- They provide no ordering (ranking) of search results, which makes them ineffective when thousands of matching documents are found.
- They tend to be slow because there is no index support, so they must process all documents for every search.

Full text indexing allows documents to be *preprocessed* and an index saved for later rapid searching. Preprocessing includes:

Parsing documents into tokens. It is useful to identify various classes of tokens, e.g., numbers, words, complex words, email addresses, so that they can be processed differently. In principle token classes depend on the specific application, but for most purposes it is adequate to use a predefined set of classes. PostgreSQL uses a *parser* to perform this step. A standard parser is provided, and custom parsers can be created for specific needs.

Converting tokens into lexemes. A lexeme is a string, just like a token, but it has been *normalized* so that different forms of the same word are made alike. For example, normalization almost always includes folding upper-case letters to lower-case, and often involves removal of suffixes (such as `s` or `es` in English). This allows searches to find variant forms of the same word, without tediously entering all the possible variants. Also, this step typically eliminates *stop words*, which are words that are so common that they are useless for searching. (In short, then, tokens are raw fragments of the document text, while lexemes are words that are believed useful for indexing and searching.) PostgreSQL uses *dictionaries* to perform this step. Various standard dictionaries are provided, and custom ones can be created for specific needs.

Storing preprocessed documents optimized for searching. For example, each document can be represented as a sorted array of normalized lexemes. Along with the lexemes it is often desirable to store positional information to use for *proximity ranking*, so that a document that contains a more "dense" region of query words is assigned a higher rank than one with scattered query words.

Dictionaries allow fine-grained control over how tokens are normalized. With appropriate dictionaries, you can:

- Define stop words that should not be indexed.
- Map synonyms to a single word using Ispell.
- Map phrases to a single word using a thesaurus.
- Map different variations of a word to a canonical form using an Ispell dictionary.
- Map different variations of a word to a canonical form using Snowball stemmer rules.

A data type `tsvector` is provided for storing preprocessed documents, along with a type `tsquery` for representing processed queries (Section 8.11). There are many functions and operators available for these data types (Section 9.13), the most important of which is the match operator `@@`, which we introduce in Section 12.1.2. Full text searches can be accelerated using indexes (Section 12.9).

12.1.1. What Is a Document?

A *document* is the unit of searching in a full text search system; for example, a magazine article or email message. The text search engine must be able to parse documents and store associations of lexemes (key words) with their parent document. Later, these associations are used to search for documents that contain query words.

For searches within PostgreSQL, a document is normally a textual field within a row of a database table, or possibly a combination (concatenation) of such fields, perhaps stored in several tables or obtained dynamically. In other words, a document can be constructed from different parts for indexing and it might not be stored anywhere as a whole. For example:

```
SELECT title || ' ' ||  author || ' ' ||  abstract || ' ' || body AS document
FROM messages
WHERE mid = 12;

SELECT m.title || ' ' || m.author || ' ' || m.abstract || ' ' || d.body AS document
FROM messages m, docs d
WHERE m.mid = d.did AND m.mid = 12;
```

> **Note:** Actually, in these example queries, `coalesce` should be used to prevent a single `NULL` attribute from causing a `NULL` result for the whole document.

Another possibility is to store the documents as simple text files in the file system. In this case, the database can be used to store the full text index and to execute searches, and some unique identifier can be used to retrieve the document from the file system. However, retrieving files from outside the database requires superuser permissions or special function support, so this is usually less convenient than keeping all the data inside PostgreSQL. Also, keeping everything inside the database allows easy access to document metadata to assist in indexing and display.

For text search purposes, each document must be reduced to the preprocessed `tsvector` format. Searching and ranking are performed entirely on the `tsvector` representation of a document — the original text need only be retrieved when the document has been selected for display to a user. We therefore often

speak of the tsvector as being the document, but of course it is only a compact representation of the full document.

12.1.2. Basic Text Matching

Full text searching in PostgreSQL is based on the match operator @@, which returns true if a tsvector (document) matches a tsquery (query). It doesn't matter which data type is written first:

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat & rat'::tsquery;
 ?column?
----------
 t

SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat rat'::tsvector;
 ?column?
----------
 f
```

As the above example suggests, a tsquery is not just raw text, any more than a tsvector is. A tsquery contains search terms, which must be already-normalized lexemes, and may combine multiple terms using AND, OR, and NOT operators. (For details see Section 8.11.) There are functions to_tsquery and plainto_tsquery that are helpful in converting user-written text into a proper tsquery, for example by normalizing words appearing in the text. Similarly, to_tsvector is used to parse and normalize a document string. So in practice a text search match would look more like this:

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat');
 ?column?
----------
 t
```

Observe that this match would not succeed if written as

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat');
 ?column?
----------
 f
```

since here no normalization of the word rats will occur. The elements of a tsvector are lexemes, which are assumed already normalized, so rats does not match rat.

The @@ operator also supports text input, allowing explicit conversion of a text string to tsvector or tsquery to be skipped in simple cases. The variants available are:

```
tsvector @@ tsquery
tsquery  @@ tsvector
text @@ tsquery
text @@ text
```

The first two of these we saw already. The form `text @@ tsquery` is equivalent to `to_tsvector(x) @@ y`. The form `text @@ text` is equivalent to `to_tsvector(x) @@ plainto_tsquery(y)`.

12.1.3. Configurations

The above are all simple text search examples. As mentioned before, full text search functionality includes the ability to do many more things: skip indexing certain words (stop words), process synonyms, and use sophisticated parsing, e.g., parse based on more than just white space. This functionality is controlled by *text search configurations*. PostgreSQL comes with predefined configurations for many languages, and you can easily create your own configurations. (psql's `\dF` command shows all available configurations.)

During installation an appropriate configuration is selected and default_text_search_config is set accordingly in `postgresql.conf`. If you are using the same text search configuration for the entire cluster you can use the value in `postgresql.conf`. To use different configurations throughout the cluster but the same configuration within any one database, use `ALTER DATABASE ... SET`. Otherwise, you can set `default_text_search_config` in each session.

Each text search function that depends on a configuration has an optional `regconfig` argument, so that the configuration to use can be specified explicitly. `default_text_search_config` is used only when this argument is omitted.

To make it easier to build custom text search configurations, a configuration is built up from simpler database objects. PostgreSQL's text search facility provides four types of configuration-related database objects:

- *Text search parsers* break documents into tokens and classify each token (for example, as words or numbers).
- *Text search dictionaries* convert tokens to normalized form and reject stop words.
- *Text search templates* provide the functions underlying dictionaries. (A dictionary simply specifies a template and a set of parameters for the template.)
- *Text search configurations* select a parser and a set of dictionaries to use to normalize the tokens produced by the parser.

Text search parsers and templates are built from low-level C functions; therefore it requires C programming ability to develop new ones, and superuser privileges to install one into a database. (There are examples of add-on parsers and templates in the `contrib/` area of the PostgreSQL distribution.) Since dictionaries and configurations just parameterize and connect together some underlying parsers and templates, no special privilege is needed to create a new dictionary or configuration. Examples of creating custom dictionaries and configurations appear later in this chapter.

12.2. Tables and Indexes

The examples in the previous section illustrated full text matching using simple constant strings. This section shows how to search table data, optionally using indexes.

12.2.1. Searching a Table

It is possible to do a full text search without an index. A simple query to print the `title` of each row that contains the word `friend` in its `body` field is:

```
SELECT title
FROM pgweb
WHERE to_tsvector('english', body) @@ to_tsquery('english', 'friend');
```

This will also find related words such as `friends` and `friendly`, since all these are reduced to the same normalized lexeme.

The query above specifies that the `english` configuration is to be used to parse and normalize the strings. Alternatively we could omit the configuration parameters:

```
SELECT title
FROM pgweb
WHERE to_tsvector(body) @@ to_tsquery('friend');
```

This query will use the configuration set by default_text_search_config.

A more complex example is to select the ten most recent documents that contain `create` and `table` in the `title` or `body`:

```
SELECT title
FROM pgweb
WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

For clarity we omitted the `coalesce` function calls which would be needed to find rows that contain `NULL` in one of the two fields.

Although these queries will work without an index, most applications will find this approach too slow, except perhaps for occasional ad-hoc searches. Practical use of text searching usually requires creating an index.

12.2.2. Creating Indexes

We can create a GIN index (Section 12.9) to speed up text searches:

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector('english', body));
```

Notice that the 2-argument version of `to_tsvector` is used. Only text search functions that specify a configuration name can be used in expression indexes (Section 11.7). This is because the index contents must be unaffected by default_text_search_config. If they were affected, the index contents might be inconsistent because different entries could contain `tsvector`s that were created with different text search configurations, and there would be no way to guess which was which. It would be impossible to dump and restore such an index correctly.

Because the two-argument version of `to_tsvector` was used in the index above, only a query reference that uses the 2-argument version of `to_tsvector` with the same configuration name will use that index. That is, `WHERE to_tsvector('english', body) @@ 'a & b'` can use the index, but `WHERE`

`to_tsvector(body) @@ 'a & b'` cannot. This ensures that an index will be used only with the same configuration used to create the index entries.

It is possible to set up more complex expression indexes wherein the configuration name is specified by another column, e.g.:

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector(config_name, body));
```

where `config_name` is a column in the `pgweb` table. This allows mixed configurations in the same index while recording which configuration was used for each index entry. This would be useful, for example, if the document collection contained documents in different languages. Again, queries that are meant to use the index must be phrased to match, e.g., `WHERE to_tsvector(config_name, body) @@ 'a & b'`.

Indexes can even concatenate columns:

```
CREATE INDEX pgweb_idx ON pgweb USING GIN (to_tsvector('english', title || ' ' || body));
```

Another approach is to create a separate `tsvector` column to hold the output of `to_tsvector`. This example is a concatenation of `title` and `body`, using `coalesce` to ensure that one field will still be indexed when the other is `NULL`:

```
ALTER TABLE pgweb ADD COLUMN textsearchable_index_col tsvector;
UPDATE pgweb SET textsearchable_index_col =
     to_tsvector('english', coalesce(title,'') || ' ' || coalesce(body,''));
```

Then we create a GIN index to speed up the search:

```
CREATE INDEX textsearch_idx ON pgweb USING GIN (textsearchable_index_col);
```

Now we are ready to perform a fast full text search:

```
SELECT title
FROM pgweb
WHERE textsearchable_index_col @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

When using a separate column to store the `tsvector` representation, it is necessary to create a trigger to keep the `tsvector` column current anytime `title` or `body` changes. Section 12.4.3 explains how to do that.

One advantage of the separate-column approach over an expression index is that it is not necessary to explicitly specify the text search configuration in queries in order to make use of the index. As shown in the example above, the query can depend on `default_text_search_config`. Another advantage is that searches will be faster, since it will not be necessary to redo the `to_tsvector` calls to verify index matches. (This is more important when using a GiST index than a GIN index; see Section 12.9.) The expression-index approach is simpler to set up, however, and it requires less disk space since the `tsvector` representation is not stored explicitly.

12.3. Controlling Text Search

To implement full text searching there must be a function to create a tsvector from a document and a tsquery from a user query. Also, we need to return results in a useful order, so we need a function that compares documents with respect to their relevance to the query. It's also important to be able to display the results nicely. PostgreSQL provides support for all of these functions.

12.3.1. Parsing Documents

PostgreSQL provides the function to_tsvector for converting a document to the tsvector data type.

```
to_tsvector([ config regconfig, ] document text) returns tsvector
```

to_tsvector parses a textual document into tokens, reduces the tokens to lexemes, and returns a tsvector which lists the lexemes together with their positions in the document. The document is processed according to the specified or default text search configuration. Here is a simple example:

```
SELECT to_tsvector('english', 'a fat  cat sat on a mat - it ate a fat rats');
                  to_tsvector
-----------------------------------------------------
 'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

In the example above we see that the resulting tsvector does not contain the words a, on, or it, the word rats became rat, and the punctuation sign - was ignored.

The to_tsvector function internally calls a parser which breaks the document text into tokens and assigns a type to each token. For each token, a list of dictionaries (Section 12.6) is consulted, where the list can vary depending on the token type. The first dictionary that *recognizes* the token emits one or more normalized *lexemes* to represent the token. For example, rats became rat because one of the dictionaries recognized that the word rats is a plural form of rat. Some words are recognized as *stop words* (Section 12.6.1), which causes them to be ignored since they occur too frequently to be useful in searching. In our example these are a, on, and it. If no dictionary in the list recognizes the token then it is also ignored. In this example that happened to the punctuation sign - because there are in fact no dictionaries assigned for its token type (Space symbols), meaning space tokens will never be indexed. The choices of parser, dictionaries and which types of tokens to index are determined by the selected text search configuration (Section 12.7). It is possible to have many different configurations in the same database, and predefined configurations are available for various languages. In our example we used the default configuration english for the English language.

The function setweight can be used to label the entries of a tsvector with a given *weight*, where a weight is one of the letters A, B, C, or D. This is typically used to mark entries coming from different parts of a document, such as title versus body. Later, this information can be used for ranking of search results.

Because to_tsvector(NULL) will return NULL, it is recommended to use coalesce whenever a field might be null. Here is the recommended method for creating a tsvector from a structured document:

```
UPDATE tt SET ti =
    setweight(to_tsvector(coalesce(title,'')), 'A')    ||
    setweight(to_tsvector(coalesce(keyword,'')), 'B')  ||
    setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
```

```
    setweight(to_tsvector(coalesce(body,'')), 'D');
```

Here we have used `setweight` to label the source of each lexeme in the finished `tsvector`, and then merged the labeled `tsvector` values using the `tsvector` concatenation operator `||`. (Section 12.4.1 gives details about these operations.)

12.3.2. Parsing Queries

PostgreSQL provides the functions `to_tsquery` and `plainto_tsquery` for converting a query to the `tsquery` data type. `to_tsquery` offers access to more features than `plainto_tsquery`, but is less forgiving about its input.

```
to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

`to_tsquery` creates a `tsquery` value from *querytext*, which must consist of single tokens separated by the Boolean operators `&` (AND), `|` (OR) and `!` (NOT). These operators can be grouped using parentheses. In other words, the input to `to_tsquery` must already follow the general rules for `tsquery` input, as described in Section 8.11. The difference is that while basic `tsquery` input takes the tokens at face value, `to_tsquery` normalizes each token to a lexeme using the specified or default configuration, and discards any tokens that are stop words according to the configuration. For example:

```
SELECT to_tsquery('english', 'The & Fat & Rats');
  to_tsquery
---------------
 'fat' & 'rat'
```

As in basic `tsquery` input, weight(s) can be attached to each lexeme to restrict it to match only `tsvector` lexemes of those weight(s). For example:

```
SELECT to_tsquery('english', 'Fat | Rats:AB');
    to_tsquery
------------------
 'fat' | 'rat':AB
```

Also, `*` can be attached to a lexeme to specify prefix matching:

```
SELECT to_tsquery('supern:*A & star:A*B');
        to_tsquery
--------------------------
 'supern':*A & 'star':*AB
```

Such a lexeme will match any word in a `tsvector` that begins with the given string.

`to_tsquery` can also accept single-quoted phrases. This is primarily useful when the configuration includes a thesaurus dictionary that may trigger on such phrases. In the example below, a thesaurus contains the rule `supernovae stars : sn`:

```
SELECT to_tsquery('''supernovae stars'' & !crab');
  to_tsquery
---------------
 'sn' & !'crab'
```

Without quotes, to_tsquery will generate a syntax error for tokens that are not separated by an AND or OR operator.

```
plainto_tsquery([ config regconfig, ] querytext text) returns tsquery
```

plainto_tsquery transforms unformatted text *querytext* to tsquery. The text is parsed and normalized much as for to_tsvector, then the & (AND) Boolean operator is inserted between surviving words.

Example:

```
SELECT plainto_tsquery('english', 'The Fat Rats');
 plainto_tsquery
-----------------
 'fat' & 'rat'
```

Note that plainto_tsquery cannot recognize Boolean operators, weight labels, or prefix-match labels in its input:

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');
   plainto_tsquery
---------------------
 'fat' & 'rat' & 'c'
```

Here, all the input punctuation was discarded as being space symbols.

12.3.3. Ranking Search Results

Ranking attempts to measure how relevant documents are to a particular query, so that when there are many matches the most relevant ones can be shown first. PostgreSQL provides two predefined ranking functions, which take into account lexical, proximity, and structural information; that is, they consider how often the query terms appear in the document, how close together the terms are in the document, and how important is the part of the document where they occur. However, the concept of relevancy is vague and very application-specific. Different applications might require additional information for ranking, e.g., document modification time. The built-in ranking functions are only examples. You can write your own ranking functions and/or combine their results with additional factors to fit your specific needs.

The two ranking functions currently available are:

```
ts_rank([ weights float4[], ] vector tsvector, query tsquery [, normalization
integer ]) returns float4
```

Ranks vectors based on the frequency of their matching lexemes.

```
ts_rank_cd([ weights float4[], ] vector tsvector, query tsquery [, normalization
integer ]) returns float4
```

This function computes the *cover density* ranking for the given document vector and query, as described in Clarke, Cormack, and Tudhope's "Relevance Ranking for One to Three Term Queries" in the journal "Information Processing and Management", 1999. Cover density is similar to ts_rank ranking except that the proximity of matching lexemes to each other is taken into consideration.

This function requires lexeme positional information to perform its calculation. Therefore, it ignores any "stripped" lexemes in the `tsvector`. If there are no unstripped lexemes in the input, the result will be zero. (See Section 12.4.1 for more information about the `strip` function and positional information in `tsvector`s.)

For both these functions, the optional *`weights`* argument offers the ability to weigh word instances more or less heavily depending on how they are labeled. The weight arrays specify how heavily to weigh each category of word, in the order:

```
{D-weight, C-weight, B-weight, A-weight}
```

If no *`weights`* are provided, then these defaults are used:

```
{0.1, 0.2, 0.4, 1.0}
```

Typically weights are used to mark words from special areas of the document, like the title or an initial abstract, so they can be treated with more or less importance than words in the document body.

Since a longer document has a greater chance of containing a query term it is reasonable to take into account document size, e.g., a hundred-word document with five instances of a search word is probably more relevant than a thousand-word document with five instances. Both ranking functions take an integer *`normalization`* option that specifies whether and how a document's length should impact its rank. The integer option controls several behaviors, so it is a bit mask: you can specify one or more behaviors using `|` (for example, `2|4`).

- 0 (the default) ignores the document length
- 1 divides the rank by 1 + the logarithm of the document length
- 2 divides the rank by the document length
- 4 divides the rank by the mean harmonic distance between extents (this is implemented only by `ts_rank_cd`)
- 8 divides the rank by the number of unique words in document
- 16 divides the rank by 1 + the logarithm of the number of unique words in document
- 32 divides the rank by itself + 1

If more than one flag bit is specified, the transformations are applied in the order listed.

It is important to note that the ranking functions do not use any global information, so it is impossible to produce a fair normalization to 1% or 100% as sometimes desired. Normalization option 32 (`rank/(rank+1)`) can be applied to scale all ranks into the range zero to one, but of course this is just a cosmetic change; it will not affect the ordering of the search results.

Here is an example that selects only the ten highest-ranked matches:

```
SELECT title, ts_rank_cd(textsearch, query) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |   rank
-----------------------------------------------+----------
 Neutrinos in the Sun                          |      3.1
 The Sudbury Neutrino Detector                 |      2.4
```

```
 A MACHO View of Galactic Dark Matter          |  2.01317
 Hot Gas and Dark Matter                       |  1.91171
 The Virgo Cluster: Hot Plasma and Dark Matter |  1.90953
 Rafting for Solar Neutrinos                   |      1.9
 NGC 4650A: Strange Galaxy and Dark Matter     |  1.85774
 Hot Gas and Dark Matter                       |   1.6123
 Ice Fishing for Cosmic Neutrinos              |      1.6
 Weak Lensing Distorts the Universe            | 0.818218
```

This is the same example using normalized ranking:

```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1) */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE  query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                     title                     |        rank
-----------------------------------------------+-------------------
 Neutrinos in the Sun                          | 0.756097569485493
 The Sudbury Neutrino Detector                 | 0.705882361190954
 A MACHO View of Galactic Dark Matter          | 0.668123210574724
 Hot Gas and Dark Matter                       |  0.65655958650282
 The Virgo Cluster: Hot Plasma and Dark Matter | 0.656301290640973
 Rafting for Solar Neutrinos                   | 0.655172410958162
 NGC 4650A: Strange Galaxy and Dark Matter     | 0.650072921219637
 Hot Gas and Dark Matter                       | 0.617195790024749
 Ice Fishing for Cosmic Neutrinos              | 0.615384618911517
 Weak Lensing Distorts the Universe            | 0.450010798361481
```

Ranking can be expensive since it requires consulting the `tsvector` of each matching document, which can be I/O bound and therefore slow. Unfortunately, it is almost impossible to avoid since practical queries often result in large numbers of matches.

12.3.4. Highlighting Results

To present search results it is ideal to show a part of each document and how it is related to the query. Usually, search engines show fragments of the document with marked search terms. PostgreSQL provides a function `ts_headline` that implements this functionality.

```
ts_headline([ config regconfig, ] document text, query tsquery [, options text ]) returns text
```

`ts_headline` accepts a document along with a query, and returns an excerpt from the document in which terms from the query are highlighted. The configuration to be used to parse the document can be specified by *`config`*; if *`config`* is omitted, the `default_text_search_config` configuration is used.

If an *`options`* string is specified it must consist of a comma-separated list of one or more *`option=value`* pairs. The available options are:

- `StartSel`, `StopSel`: the strings with which to delimit query words appearing in the document, to distinguish them from other excerpted words. You must double-quote these strings if they contain spaces or commas.
- `MaxWords`, `MinWords`: these numbers determine the longest and shortest headlines to output.
- `ShortWord`: words of this length or less will be dropped at the start and end of a headline. The default value of three eliminates common English articles.
- `HighlightAll`: Boolean flag; if `true` the whole document will be used as the headline, ignoring the preceding three parameters.
- `MaxFragments`: maximum number of text excerpts or fragments to display. The default value of zero selects a non-fragment-oriented headline generation method. A value greater than zero selects fragment-based headline generation. This method finds text fragments with as many query words as possible and stretches those fragments around the query words. As a result query words are close to the middle of each fragment and have words on each side. Each fragment will be of at most `MaxWords` and words of length `ShortWord` or less are dropped at the start and end of each fragment. If not all query words are found in the document, then a single fragment of the first `MinWords` in the document will be displayed.
- `FragmentDelimiter`: When more than one fragment is displayed, the fragments will be separated by this string.

Any unspecified options receive these defaults:

```
StartSel=<b>, StopSel=</b>,
MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE,
MaxFragments=0, FragmentDelimiter=" ... "
```

For example:

```
SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('query & similarity'));
                        ts_headline
------------------------------------------------------------
 containing given <b>query</b> terms
 and return them in order of their <b>similarity</b> to the
 <b>query</b>.

SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('query & similarity'),
  'StartSel = <, StopSel = >');
                       ts_headline
-------------------------------------------------------
 containing given <query> terms
 and return them in order of their <similarity> to the
 <query>.
```

ts_headline uses the original document, not a tsvector summary, so it can be slow and should be used with care. A typical mistake is to call ts_headline for *every* matching document when only ten documents are to be shown. SQL subqueries can help; here is an example:

```
SELECT id, ts_headline(body, q), rank
FROM (SELECT id, body, q, ts_rank_cd(ti, q) AS rank
      FROM apod, to_tsquery('stars') q
      WHERE ti @@ q
      ORDER BY rank DESC
      LIMIT 10) AS foo;
```

12.4. Additional Features

This section describes additional functions and operators that are useful in connection with text search.

12.4.1. Manipulating Documents

Section 12.3.1 showed how raw textual documents can be converted into tsvector values. PostgreSQL also provides functions and operators that can be used to manipulate documents that are already in tsvector form.

`tsvector || tsvector`

The tsvector concatenation operator returns a vector which combines the lexemes and positional information of the two vectors given as arguments. Positions and weight labels are retained during the concatenation. Positions appearing in the right-hand vector are offset by the largest position mentioned in the left-hand vector, so that the result is nearly equivalent to the result of performing to_tsvector on the concatenation of the two original document strings. (The equivalence is not exact, because any stop-words removed from the end of the left-hand argument will not affect the result, whereas they would have affected the positions of the lexemes in the right-hand argument if textual concatenation were used.)

One advantage of using concatenation in the vector form, rather than concatenating text before applying to_tsvector, is that you can use different configurations to parse different sections of the document. Also, because the setweight function marks all lexemes of the given vector the same way, it is necessary to parse the text and do setweight before concatenating if you want to label different parts of the document with different weights.

`setweight(`*`vector`* `tsvector,` *`weight`* `"char") returns tsvector`

setweight returns a copy of the input vector in which every position has been labeled with the given *weight*, either A, B, C, or D. (D is the default for new vectors and as such is not displayed on output.) These labels are retained when vectors are concatenated, allowing words from different parts of a document to be weighted differently by ranking functions.

Note that weight labels apply to *positions*, not *lexemes*. If the input vector has been stripped of positions then `setweight` does nothing.

`length(`*vector* `tsvector) returns integer`

Returns the number of lexemes stored in the vector.

`strip(`*vector* `tsvector) returns tsvector`

Returns a vector which lists the same lexemes as the given vector, but which lacks any position or weight information. While the returned vector is much less useful than an unstripped vector for relevance ranking, it will usually be much smaller.

12.4.2. Manipulating Queries

Section 12.3.2 showed how raw textual queries can be converted into `tsquery` values. PostgreSQL also provides functions and operators that can be used to manipulate queries that are already in `tsquery` form.

`tsquery && tsquery`

Returns the AND-combination of the two given queries.

`tsquery || tsquery`

Returns the OR-combination of the two given queries.

`!! tsquery`

Returns the negation (NOT) of the given query.

`numnode(`*query* `tsquery) returns integer`

Returns the number of nodes (lexemes plus operators) in a `tsquery`. This function is useful to determine if the *query* is meaningful (returns > 0), or contains only stop words (returns 0). Examples:

```
SELECT numnode(plainto_tsquery('the any'));
NOTICE:  query contains only stopword(s) or doesn't contain lexeme(s), ignored
 numnode
---------
       0

SELECT numnode('foo & bar'::tsquery);
 numnode
---------
       3
```

`querytree(`*query* `tsquery) returns text`

Returns the portion of a `tsquery` that can be used for searching an index. This function is useful for detecting unindexable queries, for example those containing only stop words or only negated terms. For example:

```
SELECT querytree(to_tsquery('!defined'));
 querytree
-----------
```

12.4.2.1. Query Rewriting

The ts_rewrite family of functions search a given tsquery for occurrences of a target subquery, and replace each occurrence with a substitute subquery. In essence this operation is a tsquery-specific version of substring replacement. A target and substitute combination can be thought of as a *query rewrite rule*. A collection of such rewrite rules can be a powerful search aid. For example, you can expand the search using synonyms (e.g., new york, big apple, nyc, gotham) or narrow the search to direct the user to some hot topic. There is some overlap in functionality between this feature and thesaurus dictionaries (Section 12.6.4). However, you can modify a set of rewrite rules on-the-fly without reindexing, whereas updating a thesaurus requires reindexing to be effective.

ts_rewrite (*query* tsquery, *target* tsquery, *substitute* tsquery) returns tsquery

This form of ts_rewrite simply applies a single rewrite rule: *target* is replaced by *substitute* wherever it appears in *query*. For example:

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
 ts_rewrite
------------
 'b' & 'c'
```

ts_rewrite (*query* tsquery, *select* text) returns tsquery

This form of ts_rewrite accepts a starting *query* and a SQL *select* command, which is given as a text string. The *select* must yield two columns of tsquery type. For each row of the *select* result, occurrences of the first column value (the target) are replaced by the second column value (the substitute) within the current *query* value. For example:

```
CREATE TABLE aliases (t tsquery PRIMARY KEY, s tsquery);
INSERT INTO aliases VALUES('a', 'c');

SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');
 ts_rewrite
------------
 'b' & 'c'
```

Note that when multiple rewrite rules are applied in this way, the order of application can be important; so in practice you will want the source query to ORDER BY some ordering key.

Let's consider a real-life astronomical example. We'll expand query supernovae using table-driven rewriting rules:

```
CREATE TABLE aliases (t tsquery primary key, s tsquery);
INSERT INTO aliases VALUES(to_tsquery('supernovae'), to_tsquery('supernovae|sn'));

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
           ts_rewrite
---------------------------------
 'crab' & ( 'supernova' | 'sn' )
```

We can change the rewriting rules just by updating the table:

```
UPDATE aliases
SET s = to_tsquery('supernovae|sn & !nebulae')
WHERE t = to_tsquery('supernovae');
```

```
SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
                ts_rewrite
-------------------------------------------
 'crab' & ( 'supernova' | 'sn' & !'nebula' )
```

Rewriting can be slow when there are many rewriting rules, since it checks every rule for a possible match. To filter out obvious non-candidate rules we can use the containment operators for the `tsquery` type. In the example below, we select only those rules which might match the original query:

```
SELECT ts_rewrite('a & b'::tsquery,
                  'SELECT t,s FROM aliases WHERE "a & b"::tsquery @> t');
 ts_rewrite
------------
 'b' & 'c'
```

12.4.3. Triggers for Automatic Updates

When using a separate column to store the `tsvector` representation of your documents, it is necessary to create a trigger to update the `tsvector` column when the document content columns change. Two built-in trigger functions are available for this, or you can write your own.

```
tsvector_update_trigger(tsvector_column_name, config_name, text_column_name [, ... ])
tsvector_update_trigger_column(tsvector_column_name, config_column_name, text_column_name [, ...
```

These trigger functions automatically compute a `tsvector` column from one or more textual columns, under the control of parameters specified in the `CREATE TRIGGER` command. An example of their use is:

```
CREATE TABLE messages (
    title       text,
    body        text,
    tsv         tsvector
);

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE PROCEDURE
tsvector_update_trigger(tsv, 'pg_catalog.english', title, body);

INSERT INTO messages VALUES('title here', 'the body text is here');

SELECT * FROM messages;
   title    |         body          |            tsv
------------+-----------------------+----------------------------
 title here | the body text is here | 'bodi':4 'text':5 'titl':1

SELECT title, body FROM messages WHERE tsv @@ to_tsquery('title & body');
   title    |         body
------------+-----------------------
```

```
 title here | the body text is here
```

Having created this trigger, any change in `title` or `body` will automatically be reflected into `tsv`, without the application having to worry about it.

The first trigger argument must be the name of the `tsvector` column to be updated. The second argument specifies the text search configuration to be used to perform the conversion. For `tsvector_update_trigger`, the configuration name is simply given as the second trigger argument. It must be schema-qualified as shown above, so that the trigger behavior will not change with changes in `search_path`. For `tsvector_update_trigger_column`, the second trigger argument is the name of another table column, which must be of type `regconfig`. This allows a per-row selection of configuration to be made. The remaining argument(s) are the names of textual columns (of type `text`, `varchar`, or `char`). These will be included in the document in the order given. NULL values will be skipped (but the other columns will still be indexed).

A limitation of these built-in triggers is that they treat all the input columns alike. To process columns differently — for example, to weight title differently from body — it is necessary to write a custom trigger. Here is an example using PL/pgSQL as the trigger language:

```
CREATE FUNCTION messages_trigger() RETURNS trigger AS $$
begin
  new.tsv :=
     setweight(to_tsvector('pg_catalog.english', coalesce(new.title,'')), 'A') ||
     setweight(to_tsvector('pg_catalog.english', coalesce(new.body,'')), 'D');
  return new;
end
$$ LANGUAGE plpgsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
    ON messages FOR EACH ROW EXECUTE PROCEDURE messages_trigger();
```

Keep in mind that it is important to specify the configuration name explicitly when creating `tsvector` values inside triggers, so that the column's contents will not be affected by changes to `default_text_search_config`. Failure to do this is likely to lead to problems such as search results changing after a dump and reload.

12.4.4. Gathering Document Statistics

The function `ts_stat` is useful for checking your configuration and for finding stop-word candidates.

```
ts_stat(sqlquery text, [ weights text, ]
        OUT word text, OUT ndoc integer,
        OUT nentry integer) returns setof record
```

sqlquery is a text value containing an SQL query which must return a single `tsvector` column. `ts_stat` executes the query and returns statistics about each distinct lexeme (word) contained in the `tsvector` data. The columns returned are

- *word* text — the value of a lexeme
- *ndoc* integer — number of documents (tsvectors) the word occurred in
- *nentry* integer — total number of occurrences of the word

If *weights* is supplied, only occurrences having one of those weights are counted.

For example, to find the ten most frequent words in a document collection:

```
SELECT * FROM ts_stat('SELECT vector FROM apod')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

The same, but counting only word occurrences with weight A or B:

```
SELECT * FROM ts_stat('SELECT vector FROM apod', 'ab')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

12.5. Parsers

Text search parsers are responsible for splitting raw document text into *tokens* and identifying each token's type, where the set of possible types is defined by the parser itself. Note that a parser does not modify the text at all — it simply identifies plausible word boundaries. Because of this limited scope, there is less need for application-specific custom parsers than there is for custom dictionaries. At present PostgreSQL provides just one built-in parser, which has been found to be useful for a wide range of applications.

The built-in parser is named pg_catalog.default. It recognizes 23 token types, shown in Table 12-1.

Table 12-1. Default Parser's Token Types

Alias	Description	Example
asciiword	Word, all ASCII letters	elephant
word	Word, all letters	mañana
numword	Word, letters and digits	beta1
asciihword	Hyphenated word, all ASCII	up-to-date
hword	Hyphenated word, all letters	lógico-matemática
numhword	Hyphenated word, letters and digits	postgresql-beta1
hword_asciipart	Hyphenated word part, all ASCII	postgresql in the context postgresql-beta1
hword_part	Hyphenated word part, all letters	lógico or matemática in the context lógico-matemática
hword_numpart	Hyphenated word part, letters and digits	beta1 in the context postgresql-beta1

Alias	Description	Example
email	Email address	foo@example.com
protocol	Protocol head	http://
url	URL	example.com/stuff/index.html
host	Host	example.com
url_path	URL path	/stuff/index.html, in the context of a URL
file	File or path name	/usr/local/foo.txt, if not within a URL
sfloat	Scientific notation	-1.234e56
float	Decimal notation	-1.234
int	Signed integer	-1234
uint	Unsigned integer	1234
version	Version number	8.3.0
tag	XML tag	<a href="dictionaries.html">
entity	XML entity	&
blank	Space symbols	(any whitespace or punctuation not otherwise recognized)

> **Note:** The parser's notion of a "letter" is determined by the database's locale setting, specifically lc_ctype. Words containing only the basic ASCII letters are reported as a separate token type, since it is sometimes useful to distinguish them. In most European languages, token types word and asciiword should be treated alike.
>
> email does not support all valid email characters as defined by RFC 5322. Specifically, the only non-alphanumeric characters supported for email user names are period, dash, and underscore.

It is possible for the parser to produce overlapping tokens from the same piece of text. As an example, a hyphenated word will be reported both as the entire word and as each component:

```
SELECT alias, description, token FROM ts_debug('foo-bar-beta1');
      alias      |               description                |     token
-----------------+------------------------------------------+---------------
 numhword        | Hyphenated word, letters and digits      | foo-bar-beta1
 hword_asciipart | Hyphenated word part, all ASCII          | foo
 blank           | Space symbols                            | -
 hword_asciipart | Hyphenated word part, all ASCII          | bar
 blank           | Space symbols                            | -
 hword_numpart   | Hyphenated word part, letters and digits | beta1
```

This behavior is desirable since it allows searches to work for both the whole compound word and for components. Here is another instructive example:

```
SELECT alias, description, token FROM ts_debug('http://example.com/stuff/index.html');
  alias   |  description  |            token
----------+---------------+------------------------------
 protocol | Protocol head | http://
 url      | URL           | example.com/stuff/index.html
 host     | Host          | example.com
 url_path | URL path      | /stuff/index.html
```

12.6. Dictionaries

Dictionaries are used to eliminate words that should not be considered in a search (*stop words*), and to *normalize* words so that different derived forms of the same word will match. A successfully normalized word is called a *lexeme*. Aside from improving search quality, normalization and removal of stop words reduce the size of the `tsvector` representation of a document, thereby improving performance. Normalization does not always have linguistic meaning and usually depends on application semantics.

Some examples of normalization:

- Linguistic - Ispell dictionaries try to reduce input words to a normalized form; stemmer dictionaries remove word endings
- URL locations can be canonicalized to make equivalent URLs match:
  - http://www.pgsql.ru/db/mw/index.html
  - http://www.pgsql.ru/db/mw/
  - http://www.pgsql.ru/db/../db/mw/index.html
- Color names can be replaced by their hexadecimal values, e.g., `red, green, blue, magenta -> FF0000, 00FF00, 0000FF, FF00FF`
- If indexing numbers, we can remove some fractional digits to reduce the range of possible numbers, so for example *3.14*159265359, *3.14*15926, *3.14* will be the same after normalization if only two digits are kept after the decimal point.

A dictionary is a program that accepts a token as input and returns:

- an array of lexemes if the input token is known to the dictionary (notice that one token can produce more than one lexeme)
- a single lexeme with the `TSL_FILTER` flag set, to replace the original token with a new token to be passed to subsequent dictionaries (a dictionary that does this is called a *filtering dictionary*)
- an empty array if the dictionary knows the token, but it is a stop word
- `NULL` if the dictionary does not recognize the input token

PostgreSQL provides predefined dictionaries for many languages. There are also several predefined templates that can be used to create new dictionaries with custom parameters. Each predefined dictionary template is described below. If no existing template is suitable, it is possible to create new ones; see the `contrib/` area of the PostgreSQL distribution for examples.

A text search configuration binds a parser together with a set of dictionaries to process the parser's output tokens. For each token type that the parser can return, a separate list of dictionaries is specified by the configuration. When a token of that type is found by the parser, each dictionary in the list is consulted in turn, until some dictionary recognizes it as a known word. If it is identified as a stop word, or if no dictionary recognizes the token, it will be discarded and not indexed or searched for. Normally, the first dictionary that returns a non-`NULL` output determines the result, and any remaining dictionaries are not consulted; but a filtering dictionary can replace the given word with a modified word, which is then passed to subsequent dictionaries.

The general rule for configuring a list of dictionaries is to place first the most narrow, most specific dictionary, then the more general dictionaries, finishing with a very general dictionary, like a Snowball stemmer or `simple`, which recognizes everything. For example, for an astronomy-specific search (`astro_en` configuration) one could bind token type `asciiword` (ASCII word) to a synonym dictionary of astronomical terms, a general English dictionary and a Snowball English stemmer:

```
ALTER TEXT SEARCH CONFIGURATION astro_en
    ADD MAPPING FOR asciiword WITH astrosyn, english_ispell, english_stem;
```

A filtering dictionary can be placed anywhere in the list, except at the end where it'd be useless. Filtering dictionaries are useful to partially normalize words to simplify the task of later dictionaries. For example, a filtering dictionary could be used to remove accents from accented letters, as is done by the unaccent module.

12.6.1. Stop Words

Stop words are words that are very common, appear in almost every document, and have no discrimination value. Therefore, they can be ignored in the context of full text searching. For example, every English text contains words like `a` and `the`, so it is useless to store them in an index. However, stop words do affect the positions in `tsvector`, which in turn affect ranking:

```
SELECT to_tsvector('english', 'in the list of stop words');
        to_tsvector
----------------------------
 'list':3 'stop':5 'word':6
```

The missing positions 1,2,4 are because of stop words. Ranks calculated for documents with and without stop words are quite different:

```
SELECT ts_rank_cd (to_tsvector('english', 'in the list of stop words'), to_tsquery('list &
 ts_rank_cd
------------
       0.05

SELECT ts_rank_cd (to_tsvector('english', 'list stop words'), to_tsquery('list & stop'));
 ts_rank_cd
------------
        0.1
```

It is up to the specific dictionary how it treats stop words. For example, `ispell` dictionaries first normalize words and then look at the list of stop words, while `Snowball` stemmers first check the list of stop words. The reason for the different behavior is an attempt to decrease noise.

12.6.2. Simple Dictionary

The `simple` dictionary template operates by converting the input token to lower case and checking it against a file of stop words. If it is found in the file then an empty array is returned, causing the token to be discarded. If not, the lower-cased form of the word is returned as the normalized lexeme. Alternatively, the dictionary can be configured to report non-stop-words as unrecognized, allowing them to be passed on to the next dictionary in the list.

Here is an example of a dictionary definition using the `simple` template:

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
    TEMPLATE = pg_catalog.simple,
    STOPWORDS = english
);
```

Here, `english` is the base name of a file of stop words. The file's full name will be `$SHAREDIR/tsearch_data/english.stop`, where `$SHAREDIR` means the PostgreSQL installation's shared-data directory, often `/usr/local/share/postgresql` (use `pg_config --sharedir` to determine it if you're not sure). The file format is simply a list of words, one per line. Blank lines and trailing spaces are ignored, and upper case is folded to lower case, but no other processing is done on the file contents.

Now we can test our dictionary:

```
SELECT ts_lexize('public.simple_dict', 'YeS');
 ts_lexize
-----------
 {yes}

SELECT ts_lexize('public.simple_dict', 'The');
 ts_lexize
-----------
 {}
```

We can also choose to return `NULL`, instead of the lower-cased word, if it is not found in the stop words file. This behavior is selected by setting the dictionary's `Accept` parameter to `false`. Continuing the example:

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );

SELECT ts_lexize('public.simple_dict', 'YeS');
 ts_lexize
-----------


SELECT ts_lexize('public.simple_dict', 'The');
```

```
 ts_lexize
-----------
 {}
```

With the default setting of `Accept = true`, it is only useful to place a `simple` dictionary at the end of a list of dictionaries, since it will never pass on any token to a following dictionary. Conversely, `Accept = false` is only useful when there is at least one following dictionary.

Caution

Most types of dictionaries rely on configuration files, such as files of stop words. These files *must* be stored in UTF-8 encoding. They will be translated to the actual database encoding, if that is different, when they are read into the server.

Caution

Normally, a database session will read a dictionary configuration file only once, when it is first used within the session. If you modify a configuration file and want to force existing sessions to pick up the new contents, issue an `ALTER TEXT SEARCH DICTIONARY` command on the dictionary. This can be a "dummy" update that doesn't actually change any parameter values.

12.6.3. Synonym Dictionary

This dictionary template is used to create dictionaries that replace a word with a synonym. Phrases are not supported (use the thesaurus template (Section 12.6.4) for that). A synonym dictionary can be used to overcome linguistic problems, for example, to prevent an English stemmer dictionary from reducing the word "Paris" to "pari". It is enough to have a `Paris paris` line in the synonym dictionary and put it before the `english_stem` dictionary. For example:

```
SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | Paris | {english_stem} | english_stem | {pari}

CREATE TEXT SEARCH DICTIONARY my_synonym (
    TEMPLATE = synonym,
    SYNONYMS = my_synonyms
);

ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword
    WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |       dictionaries        | dictionary | lexemes
-----------+-----------------+-------+---------------------------+------------+---------
```

```
 asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {paris}
```

The only parameter required by the `synonym` template is `SYNONYMS`, which is the base name of its configuration file — `my_synonyms` in the above example. The file's full name will be `$SHAREDIR/tsearch_data/my_synonyms.syn` (where `$SHAREDIR` means the PostgreSQL installation's shared-data directory). The file format is just one line per word to be substituted, with the word followed by its synonym, separated by white space. Blank lines and trailing spaces are ignored.

The `synonym` template also has an optional parameter `CaseSensitive`, which defaults to `false`. When `CaseSensitive` is `false`, words in the synonym file are folded to lower case, as are input tokens. When it is `true`, words and tokens are not folded to lower case, but are compared as-is.

An asterisk (`*`) can be placed at the end of a synonym in the configuration file. This indicates that the synonym is a prefix. The asterisk is ignored when the entry is used in `to_tsvector()`, but when it is used in `to_tsquery()`, the result will be a query item with the prefix match marker (see Section 12.3.2). For example, suppose we have these entries in `$SHAREDIR/tsearch_data/synonym_sample.syn`:

```
postgres        pgsql
postgresql      pgsql
postgre pgsql
gogle   googl
indices index*
```

Then we will get these results:

```
mydb=# CREATE TEXT SEARCH DICTIONARY syn (template=synonym, synonyms='synonym_sample');
mydb=# SELECT ts_lexize('syn', 'indices');
 ts_lexize
-----------
 {index}
(1 row)

mydb=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);
mydb=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH syn;
mydb=# SELECT to_tsvector('tst', 'indices');
 to_tsvector
-------------
 'index':1
(1 row)

mydb=# SELECT to_tsquery('tst', 'indices');
 to_tsquery
------------
 'index':*
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector;
            tsvector
---------------------------------
 'are' 'indexes' 'useful' 'very'
(1 row)
```

```
mydb=# SELECT 'indexes are very useful'::tsvector @@ to_tsquery('tst', 'indices');
 ?column?
----------
 t
(1 row)
```

12.6.4. Thesaurus Dictionary

A thesaurus dictionary (sometimes abbreviated as TZ) is a collection of words that includes information about the relationships of words and phrases, i.e., broader terms (BT), narrower terms (NT), preferred terms, non-preferred terms, related terms, etc.

Basically a thesaurus dictionary replaces all non-preferred terms by one preferred term and, optionally, preserves the original terms for indexing as well. PostgreSQL's current implementation of the thesaurus dictionary is an extension of the synonym dictionary with added *phrase* support. A thesaurus dictionary requires a configuration file of the following format:

```
# this is a comment
sample word(s) : indexed word(s)
more sample word(s) : more indexed word(s)
...
```

where the colon (`:`) symbol acts as a delimiter between a phrase and its replacement.

A thesaurus dictionary uses a *subdictionary* (which is specified in the dictionary's configuration) to normalize the input text before checking for phrase matches. It is only possible to select one subdictionary. An error is reported if the subdictionary fails to recognize a word. In that case, you should remove the use of the word or teach the subdictionary about it. You can place an asterisk (`*`) at the beginning of an indexed word to skip applying the subdictionary to it, but all sample words *must* be known to the subdictionary.

The thesaurus dictionary chooses the longest match if there are multiple phrases matching the input, and ties are broken by using the last definition.

Specific stop words recognized by the subdictionary cannot be specified; instead use `?` to mark the location where any stop word can appear. For example, assuming that `a` and `the` are stop words according to the subdictionary:

```
? one ? two : swsw
```

matches `a one the two` and `the one a two`; both would be replaced by `swsw`.

Since a thesaurus dictionary has the capability to recognize phrases it must remember its state and interact with the parser. A thesaurus dictionary uses these assignments to check if it should handle the next word or stop accumulation. The thesaurus dictionary must be configured carefully. For example, if the thesaurus dictionary is assigned to handle only the `asciiword` token, then a thesaurus dictionary definition like `one 7` will not work since token type `uint` is not assigned to the thesaurus dictionary.

> **Caution**
>
> Thesauruses are used during indexing so any change in the thesaurus dictionary's parameters *requires* reindexing. For most other dictionary types, small changes such as adding or removing stopwords does not force reindexing.

12.6.4.1. Thesaurus Configuration

To define a new thesaurus dictionary, use the `thesaurus` template. For example:

```
CREATE TEXT SEARCH DICTIONARY thesaurus_simple (
    TEMPLATE = thesaurus,
    DictFile = mythesaurus,
    Dictionary = pg_catalog.english_stem
);
```

Here:

- `thesaurus_simple` is the new dictionary's name
- `mythesaurus` is the base name of the thesaurus configuration file. (Its full name will be `$SHAREDIR/tsearch_data/mythesaurus.ths`, where `$SHAREDIR` means the installation shared-data directory.)
- `pg_catalog.english_stem` is the subdictionary (here, a Snowball English stemmer) to use for thesaurus normalization. Notice that the subdictionary will have its own configuration (for example, stop words), which is not shown here.

Now it is possible to bind the thesaurus dictionary `thesaurus_simple` to the desired token types in a configuration, for example:

```
ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_simple;
```

12.6.4.2. Thesaurus Example

Consider a simple astronomical thesaurus `thesaurus_astro`, which contains some astronomical word combinations:

```
supernovae stars : sn
crab nebulae : crab
```

Below we create a dictionary and bind some token types to an astronomical thesaurus and English stemmer:

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (
    TEMPLATE = thesaurus,
    DictFile = thesaurus_astro,
```

```
    Dictionary = english_stem
);

ALTER TEXT SEARCH CONFIGURATION russian
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
    WITH thesaurus_astro, english_stem;
```

Now we can see how it works. `ts_lexize` is not very useful for testing a thesaurus, because it treats its input as a single token. Instead we can use `plainto_tsquery` and `to_tsvector` which will break their input strings into multiple tokens:

```
SELECT plainto_tsquery('supernova star');
 plainto_tsquery
-----------------
 'sn'

SELECT to_tsvector('supernova star');
 to_tsvector
-------------
 'sn':1
```

In principle, one can use `to_tsquery` if you quote the argument:

```
SELECT to_tsquery('''supernova star''');
 to_tsquery
------------
 'sn'
```

Notice that `supernova star` matches `supernovae stars` in `thesaurus_astro` because we specified the `english_stem` stemmer in the thesaurus definition. The stemmer removed the `e` and `s`.

To index the original phrase as well as the substitute, just include it in the right-hand part of the definition:

```
supernovae stars : sn supernovae stars

SELECT plainto_tsquery('supernova star');
       plainto_tsquery
-----------------------------
 'sn' & 'supernova' & 'star'
```

12.6.5. Ispell Dictionary

The Ispell dictionary template supports *morphological dictionaries*, which can normalize many different linguistic forms of a word into the same lexeme. For example, an English Ispell dictionary can match all declensions and conjugations of the search term `bank`, e.g., `banking`, `banked`, `banks`, `banks'`, and `bank's`.

The standard PostgreSQL distribution does not include any Ispell configuration files. Dictionaries for a large number of languages are available from Ispell[1]. Also, some more modern dictionary file formats are supported — MySpell[2] (OO < 2.0.1) and Hunspell[3] (OO >= 2.0.2). A large list of dictionaries is available on the OpenOffice Wiki[4].

To create an Ispell dictionary, use the built-in `ispell` template and specify several parameters:

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

Here, `DictFile`, `AffFile`, and `StopWords` specify the base names of the dictionary, affixes, and stop-words files. The stop-words file has the same format explained above for the `simple` dictionary type. The format of the other files is not specified here but is available from the above-mentioned web sites.

Ispell dictionaries usually recognize a limited set of words, so they should be followed by another broader dictionary; for example, a Snowball dictionary, which recognizes everything.

Ispell dictionaries support splitting compound words; a useful feature. Notice that the affix file should specify a special flag using the `compoundwords controlled` statement that marks dictionary words that can participate in compound formation:

```
compoundwords  controlled z
```

Here are some examples for the Norwegian language:

```
SELECT ts_lexize('norwegian_ispell', 'overbuljongterningpakkmesterassistent');
   {over,buljong,terning,pakk,mester,assistent}
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
   {sjokoladefabrikk,sjokolade,fabrikk}
```

> **Note:** MySpell does not support compound words. Hunspell has sophisticated support for compound words. At present, PostgreSQL implements only the basic compound word operations of Hunspell.

12.6.6. Snowball Dictionary

The Snowball dictionary template is based on a project by Martin Porter, inventor of the popular Porter's stemming algorithm for the English language. Snowball now provides stemming algorithms for many languages (see the Snowball site[5] for more information). Each algorithm understands how to reduce common variant forms of words to a base, or stem, spelling within its language. A Snowball dictionary requires a

1. http://ficus-www.cs.ucla.edu/geoff/ispell.html
2. http://en.wikipedia.org/wiki/MySpell
3. http://sourceforge.net/projects/hunspell/
4. http://wiki.services.openoffice.org/wiki/Dictionaries
5. https://snowballstem.org/

language parameter to identify which stemmer to use, and optionally can specify a stopword file name that gives a list of words to eliminate. (PostgreSQL's standard stopword lists are also provided by the Snowball project.) For example, there is a built-in definition equivalent to

```
CREATE TEXT SEARCH DICTIONARY english_stem (
    TEMPLATE = snowball,
    Language = english,
    StopWords = english
);
```

The stopword file format is the same as already explained.

A Snowball dictionary recognizes everything, whether or not it is able to simplify the word, so it should be placed at the end of the dictionary list. It is useless to have it before any other dictionary because a token will never pass through it to the next dictionary.

12.7. Configuration Example

A text search configuration specifies all options necessary to transform a document into a tsvector: the parser to use to break text into tokens, and the dictionaries to use to transform each token into a lexeme. Every call of to_tsvector or to_tsquery needs a text search configuration to perform its processing. The configuration parameter default_text_search_config specifies the name of the default configuration, which is the one used by text search functions if an explicit configuration parameter is omitted. It can be set in postgresql.conf, or set for an individual session using the SET command.

Several predefined text search configurations are available, and you can create custom configurations easily. To facilitate management of text search objects, a set of SQL commands is available, and there are several psql commands that display information about text search objects (Section 12.10).

As an example we will create a configuration pg, starting by duplicating the built-in english configuration:

```
CREATE TEXT SEARCH CONFIGURATION public.pg ( COPY = pg_catalog.english );
```

We will use a PostgreSQL-specific synonym list and store it in $SHAREDIR/tsearch_data/pg_dict.syn. The file contents look like:

```
postgres    pg
pgsql       pg
postgresql  pg
```

We define the synonym dictionary like this:

```
CREATE TEXT SEARCH DICTIONARY pg_dict (
    TEMPLATE = synonym,
    SYNONYMS = pg_dict
);
```

Next we register the Ispell dictionary `english_ispell`, which has its own configuration files:

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);
```

Now we can set up the mappings for words in configuration `pg`:

```
ALTER TEXT SEARCH CONFIGURATION pg
    ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                      word, hword, hword_part
    WITH pg_dict, english_ispell, english_stem;
```

We choose not to index or search some token types that the built-in configuration does handle:

```
ALTER TEXT SEARCH CONFIGURATION pg
    DROP MAPPING FOR email, url, url_path, sfloat, float;
```

Now we can test our configuration:

```
SELECT * FROM ts_debug('public.pg', '
PostgreSQL, the highly scalable, SQL compliant, open source object-relational
database management system, is now undergoing beta testing of the next
version of our software.
');
```

The next step is to set the session to use the new configuration, which was created in the `public` schema:

```
=> \dF
   List of text search configurations
 Schema  | Name | Description
---------+------+-------------
 public  | pg   |

SET default_text_search_config = 'public.pg';
SET

SHOW default_text_search_config;
 default_text_search_config
----------------------------
 public.pg
```

12.8. Testing and Debugging Text Search

The behavior of a custom text search configuration can easily become confusing. The functions described in this section are useful for testing text search objects. You can test a complete configuration, or test parsers and dictionaries separately.

12.8.1. Configuration Testing

The function `ts_debug` allows easy testing of a text search configuration.

```
ts_debug([ config regconfig, ] document text,
         OUT alias text,
         OUT description text,
         OUT token text,
         OUT dictionaries regdictionary[],
         OUT dictionary regdictionary,
         OUT lexemes text[])
         returns setof record
```

`ts_debug` displays information about every token of *document* as produced by the parser and processed by the configured dictionaries. It uses the configuration specified by *config*, or `default_text_search_config` if that argument is omitted.

`ts_debug` returns one row for each token identified in the text by the parser. The columns returned are

- *alias* `text` — short name of the token type
- *description* `text` — description of the token type
- *token* `text` — text of the token
- *dictionaries* `regdictionary[]` — the dictionaries selected by the configuration for this token type
- *dictionary* `regdictionary` — the dictionary that recognized the token, or `NULL` if none did
- *lexemes* `text[]` — the lexeme(s) produced by the dictionary that recognized the token, or `NULL` if none did; an empty array (`{}`) means it was recognized as a stop word

Here is a simple example:

```
SELECT * FROM ts_debug('english', 'a fat  cat sat on a mat - it ate a fat rats');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | cat   | {english_stem} | english_stem | {cat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | sat   | {english_stem} | english_stem | {sat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | on    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
```

```
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | mat   | {english_stem} | english_stem | {mat}
 blank     | Space symbols   |       | {}             |              |
 blank     | Space symbols   | -     | {}             |              |
 asciiword | Word, all ASCII | it    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | ate   | {english_stem} | english_stem | {ate}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | rats  | {english_stem} | english_stem | {rat}
```

For a more extensive demonstration, we first create a `public.english` configuration and Ispell dictionary for the English language:

```
CREATE TEXT SEARCH CONFIGURATION public.english ( COPY = pg_catalog.english );

CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);

ALTER TEXT SEARCH CONFIGURATION public.english
   ALTER MAPPING FOR asciiword WITH english_ispell, english_stem;

SELECT * FROM ts_debug('public.english', 'The Brightest supernovaes');
   alias   |   description   |    token    |         dictionaries          |   dictionary   |
-----------+-----------------+-------------+-------------------------------+----------------
 asciiword | Word, all ASCII | The         | {english_ispell,english_stem} | english_ispell
 blank     | Space symbols   |             | {}                            |
 asciiword | Word, all ASCII | Brightest   | {english_ispell,english_stem} | english_ispell
 blank     | Space symbols   |             | {}                            |
 asciiword | Word, all ASCII | supernovaes | {english_ispell,english_stem} | english_stem
```

In this example, the word `Brightest` was recognized by the parser as an `ASCII word` (alias `asciiword`). For this token type the dictionary list is `english_ispell` and `english_stem`. The word was recognized by `english_ispell`, which reduced it to the noun `bright`. The word `supernovaes` is unknown to the `english_ispell` dictionary so it was passed to the next dictionary, and, fortunately, was recognized (in fact, `english_stem` is a Snowball dictionary which recognizes everything; that is why it was placed at the end of the dictionary list).

The word `The` was recognized by the `english_ispell` dictionary as a stop word (Section 12.6.1) and will not be indexed. The spaces are discarded too, since the configuration provides no dictionaries at all for them.

You can reduce the width of the output by explicitly specifying which columns you want to see:

```
SELECT alias, token, dictionary, lexemes
```

```
FROM ts_debug('public.english', 'The Brightest supernovaes');
   alias   |    token    |   dictionary   |   lexemes
-----------+-------------+----------------+-------------
 asciiword | The         | english_ispell | {}
 blank     |             |                |
 asciiword | Brightest   | english_ispell | {bright}
 blank     |             |                |
 asciiword | supernovaes | english_stem   | {supernova}
```

12.8.2. Parser Testing

The following functions allow direct testing of a text search parser.

```
ts_parse(parser_name text, document text,
         OUT tokid integer, OUT token text) returns setof record
ts_parse(parser_oid oid, document text,
         OUT tokid integer, OUT token text) returns setof record
```

`ts_parse` parses the given *document* and returns a series of records, one for each token produced by parsing. Each record includes a `tokid` showing the assigned token type and a `token` which is the text of the token. For example:

```
SELECT * FROM ts_parse('default', '123 - a number');
 tokid | token
-------+--------
    22 | 123
    12 |
    12 | -
     1 | a
    12 |
     1 | number
```

```
ts_token_type(parser_name text, OUT tokid integer,
              OUT alias text, OUT description text) returns setof record
ts_token_type(parser_oid oid, OUT tokid integer,
              OUT alias text, OUT description text) returns setof record
```

`ts_token_type` returns a table which describes each type of token the specified parser can recognize. For each token type, the table gives the integer `tokid` that the parser uses to label a token of that type, the `alias` that names the token type in configuration commands, and a short `description`. For example:

```
SELECT * FROM ts_token_type('default');
 tokid |      alias      |               description
-------+-----------------+-----------------------------------------
     1 | asciiword       | Word, all ASCII
     2 | word            | Word, all letters
     3 | numword         | Word, letters and digits
     4 | email           | Email address
```

```
 5 | url             | URL
 6 | host            | Host
 7 | sfloat          | Scientific notation
 8 | version         | Version number
 9 | hword_numpart   | Hyphenated word part, letters and digits
10 | hword_part      | Hyphenated word part, all letters
11 | hword_asciipart | Hyphenated word part, all ASCII
12 | blank           | Space symbols
13 | tag             | XML tag
14 | protocol        | Protocol head
15 | numhword        | Hyphenated word, letters and digits
16 | asciihword      | Hyphenated word, all ASCII
17 | hword           | Hyphenated word, all letters
18 | url_path        | URL path
19 | file            | File or path name
20 | float           | Decimal notation
21 | int             | Signed integer
22 | uint            | Unsigned integer
23 | entity          | XML entity
```

12.8.3. Dictionary Testing

The `ts_lexize` function facilitates dictionary testing.

```
ts_lexize(dict regdictionary, token text) returns text[]
```

`ts_lexize` returns an array of lexemes if the input *`token`* is known to the dictionary, or an empty array if the token is known to the dictionary but it is a stop word, or `NULL` if it is an unknown word.

Examples:

```
SELECT ts_lexize('english_stem', 'stars');
 ts_lexize
-----------
 {star}

SELECT ts_lexize('english_stem', 'a');
 ts_lexize
-----------
 {}
```

> **Note:** The `ts_lexize` function expects a single *token*, not text. Here is a case where this can be confusing:
>
> ```
> SELECT ts_lexize('thesaurus_astro', 'supernovae stars') is null;
>  ?column?
> ----------
>  t
> ```

The thesaurus dictionary `thesaurus_astro` does know the phrase `supernovae stars`, but `ts_lexize` fails since it does not parse the input text but treats it as a single token. Use `plainto_tsquery` or `to_tsvector` to test thesaurus dictionaries, for example:

```
SELECT plainto_tsquery('supernovae stars');
 plainto_tsquery
-----------------
 'sn'
```

12.9. GIN and GiST Index Types

There are two kinds of indexes that can be used to speed up full text searches. Note that indexes are not mandatory for full text searching, but in cases where a column is searched on a regular basis, an index is usually desirable.

```
CREATE INDEX name ON table USING GIN (column);
```

Creates a GIN (Generalized Inverted Index)-based index. The *column* must be of `tsvector` type.

```
CREATE INDEX name ON table USING GIST (column);
```

Creates a GiST (Generalized Search Tree)-based index. The *column* can be of `tsvector` or `tsquery` type.

GIN indexes are the preferred text search index type. As inverted indexes, they contain an index entry for each word (lexeme), with a compressed list of matching locations. Multi-word searches can find the first match, then use the index to remove rows that are lacking additional words. GIN indexes store only the words (lexemes) of `tsvector` values, and not their weight labels. Thus a table row recheck is needed when using a query that involves weights.

A GiST index is *lossy*, meaning that the index might produce false matches, and it is necessary to check the actual table row to eliminate such false matches. (PostgreSQL does this automatically when needed.) GiST indexes are lossy because each document is represented in the index by a fixed-length signature. The signature is generated by hashing each word into a single bit in an n-bit string, with all these bits OR-ed together to produce an n-bit document signature. When two words hash to the same bit position there will be a false match. If all words in the query have matches (real or false) then the table row must be retrieved to see if the match is correct.

Lossiness causes performance degradation due to unnecessary fetches of table records that turn out to be false matches. Since random access to table records is slow, this limits the usefulness of GiST indexes. The likelihood of false matches depends on several factors, in particular the number of unique words, so using dictionaries to reduce this number is recommended.

Note that GIN index build time can often be improved by increasing maintenance_work_mem, while GiST index build time is not sensitive to that parameter.

Partitioning of big collections and the proper use of GIN and GiST indexes allows the implementation of very fast searches with online update. Partitioning can be done at the database level using table inheritance, or by distributing documents over servers and collecting search results using the dblink module. The latter is possible because ranking functions use only local information.

12.10. psql Support

Information about text search configuration objects can be obtained in psql using a set of commands:

```
\dF{d,p,t}[+] [PATTERN]
```

An optional + produces more details.

The optional parameter `PATTERN` can be the name of a text search object, optionally schema-qualified. If `PATTERN` is omitted then information about all visible objects will be displayed. `PATTERN` can be a regular expression and can provide *separate* patterns for the schema and object names. The following examples illustrate this:

```
=> \dF *fulltext*
       List of text search configurations
 Schema |  Name        | Description
--------+--------------+-------------
 public | fulltext_cfg |

=> \dF *.fulltext*
       List of text search configurations
 Schema   |  Name        | Description
----------+--------------------------
 fulltext | fulltext_cfg |
 public   | fulltext_cfg |
```

The available commands are:

```
\dF[+] [PATTERN]
```

List text search configurations (add + for more detail).

```
=> \dF russian
            List of text search configurations
   Schema   |  Name   |            Description
------------+---------+-----------------------------------
 pg_catalog | russian | configuration for russian language

=> \dF+ russian
Text search configuration "pg_catalog.russian"
Parser: "pg_catalog.default"
      Token      | Dictionaries
-----------------+--------------
 asciihword      | english_stem
 asciiword       | english_stem
 email           | simple
 file            | simple
```

```
 float           | simple
 host            | simple
 hword           | russian_stem
 hword_asciipart | english_stem
 hword_numpart   | simple
 hword_part      | russian_stem
 int             | simple
 numhword        | simple
 numword         | simple
 sfloat          | simple
 uint            | simple
 url             | simple
 url_path        | simple
 version         | simple
 word            | russian_stem
```

`\dFd[+] [PATTERN]`

List text search dictionaries (add + for more detail).

```
=> \dFd
                            List of text search dictionaries
   Schema   |      Name       |                        Description
------------+-----------------+-----------------------------------------------------------
 pg_catalog | danish_stem     | snowball stemmer for danish language
 pg_catalog | dutch_stem      | snowball stemmer for dutch language
 pg_catalog | english_stem    | snowball stemmer for english language
 pg_catalog | finnish_stem    | snowball stemmer for finnish language
 pg_catalog | french_stem     | snowball stemmer for french language
 pg_catalog | german_stem     | snowball stemmer for german language
 pg_catalog | hungarian_stem  | snowball stemmer for hungarian language
 pg_catalog | italian_stem    | snowball stemmer for italian language
 pg_catalog | norwegian_stem  | snowball stemmer for norwegian language
 pg_catalog | portuguese_stem | snowball stemmer for portuguese language
 pg_catalog | romanian_stem   | snowball stemmer for romanian language
 pg_catalog | russian_stem    | snowball stemmer for russian language
 pg_catalog | simple          | simple dictionary: just lower case and check for stopwor
 pg_catalog | spanish_stem    | snowball stemmer for spanish language
 pg_catalog | swedish_stem    | snowball stemmer for swedish language
 pg_catalog | turkish_stem    | snowball stemmer for turkish language
```

`\dFp[+] [PATTERN]`

List text search parsers (add + for more detail).

```
=> \dFp
        List of text search parsers
   Schema   |  Name   |     Description
------------+---------+---------------------
 pg_catalog | default | default word parser
=> \dFp+
    Text search parser "pg_catalog.default"
     Method      |    Function    | Description
-----------------+----------------+-------------
 Start parse     | prsd_start     |
 Get next token  | prsd_nexttoken |
```

```
 End parse       | prsd_end       |
 Get headline    | prsd_headline  |
 Get token types | prsd_lextype   |

        Token types for parser "pg_catalog.default"
   Token name    |               Description
-----------------+------------------------------------------
 asciihword      | Hyphenated word, all ASCII
 asciiword       | Word, all ASCII
 blank           | Space symbols
 email           | Email address
 entity          | XML entity
 file            | File or path name
 float           | Decimal notation
 host            | Host
 hword           | Hyphenated word, all letters
 hword_asciipart | Hyphenated word part, all ASCII
 hword_numpart   | Hyphenated word part, letters and digits
 hword_part      | Hyphenated word part, all letters
 int             | Signed integer
 numhword        | Hyphenated word, letters and digits
 numword         | Word, letters and digits
 protocol        | Protocol head
 sfloat          | Scientific notation
 tag             | XML tag
 uint            | Unsigned integer
 url             | URL
 url_path        | URL path
 version         | Version number
 word            | Word, all letters
(23 rows)
```

```
\dFt[+] [PATTERN]
```

List text search templates (add + for more detail).

```
=> \dFt
                            List of text search templates
   Schema   |   Name    |                        Description
------------+-----------+-----------------------------------------------------------
 pg_catalog | ispell    | ispell dictionary
 pg_catalog | simple    | simple dictionary: just lower case and check for stopword
 pg_catalog | snowball  | snowball stemmer
 pg_catalog | synonym   | synonym dictionary: replace word by its synonym
 pg_catalog | thesaurus | thesaurus dictionary: phrase by phrase substitution
```

12.11. Limitations

The current limitations of PostgreSQL's text search features are:

- The length of each lexeme must be less than 2K bytes

- The length of a `tsvector` (lexemes + positions) must be less than 1 megabyte
- The number of lexemes must be less than 2^{64}
- Position values in `tsvector` must be greater than 0 and no more than 16,383
- No more than 256 positions per lexeme
- The number of nodes (lexemes + operators) in a `tsquery` must be less than 32,768

For comparison, the PostgreSQL 8.1 documentation contained 10,441 unique words, a total of 335,420 words, and the most frequent word "postgresql" was mentioned 6,127 times in 655 documents.

Another example — the PostgreSQL mailing list archives contained 910,989 unique words with 57,491,343 lexemes in 461,020 messages.

12.12. Migration from Pre-8.3 Text Search

Applications that use the tsearch2 module for text searching will need some adjustments to work with the built-in features:

- Some functions have been renamed or had small adjustments in their argument lists, and all of them are now in the `pg_catalog` schema, whereas in a previous installation they would have been in `public` or another non-system schema. There is a new version of tsearch2 that provides a compatibility layer to solve most problems in this area.
- The old tsearch2 functions and other objects *must* be suppressed when loading pg_dump output from a pre-8.3 database. While many of them won't load anyway, a few will and then cause problems. One simple way to deal with this is to load the new tsearch2 module before restoring the dump; then it will block the old objects from being loaded.
- Text search configuration setup is completely different now. Instead of manually inserting rows into configuration tables, search is configured through the specialized SQL commands shown earlier in this chapter. There is no automated support for converting an existing custom configuration for 8.3; you're on your own here.
- Most types of dictionaries rely on some outside-the-database configuration files. These are largely compatible with pre-8.3 usage, but note the following differences:
  - Configuration files now must be placed in a single specified directory (`$SHAREDIR/tsearch_data`), and must have a specific extension depending on the type of file, as noted previously in the descriptions of the various dictionary types. This restriction was added to forestall security problems.
  - Configuration files must be encoded in UTF-8 encoding, regardless of what database encoding is used.
  - In thesaurus configuration files, stop words must be marked with `?`.

Chapter 13. Concurrency Control

This chapter describes the behavior of the PostgreSQL database system when two or more sessions try to access the same data at the same time. The goals in that situation are to allow efficient access for all sessions while maintaining strict data integrity. Every developer of database applications should be familiar with the topics covered in this chapter.

13.1. Introduction

PostgreSQL provides a rich set of tools for developers to manage concurrent access to data. Internally, data consistency is maintained by using a multiversion model (Multiversion Concurrency Control, MVCC). This means that each SQL statement sees a snapshot of data (a *database version*) as it was some time ago, regardless of the current state of the underlying data. This prevents statements from viewing inconsistent data produced by concurrent transactions performing updates on the same data rows, providing *transaction isolation* for each database session. MVCC, by eschewing the locking methodologies of traditional database systems, minimizes lock contention in order to allow for reasonable performance in multiuser environments.

The main advantage of using the MVCC model of concurrency control rather than locking is that in MVCC locks acquired for querying (reading) data do not conflict with locks acquired for writing data, and so reading never blocks writing and writing never blocks reading. PostgreSQL maintains this guarantee even when providing the strictest level of transaction isolation through the use of an innovative *Serializable Snapshot Isolation* (SSI) level.

Table- and row-level locking facilities are also available in PostgreSQL for applications which don't generally need full transaction isolation and prefer to explicitly manage particular points of conflict. However, proper use of MVCC will generally provide better performance than locks. In addition, application-defined advisory locks provide a mechanism for acquiring locks that are not tied to a single transaction.

13.2. Transaction Isolation

The SQL standard defines four levels of transaction isolation. The most strict is Serializable, which is defined by the standard in a paragraph which says that any concurrent execution of a set of Serializable transactions is guaranteed to produce the same effect as running them one at a time in some order. The other three levels are defined in terms of phenomena, resulting from interaction between concurrent transactions, which must not occur at each level. The standard notes that due to the definition of Serializable, none of these phenomena are possible at that level. (This is hardly surprising -- if the effect of the transactions must be consistent with having been run one at a time, how could you see any phenomena caused by interactions?)

The phenomena which are prohibited at various levels are:

dirty read

A transaction reads data written by a concurrent uncommitted transaction.

nonrepeatable read

A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).

phantom read

A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently-committed transaction.

serialization anomaly

The result of successfully committing a group of transactions is inconsistent with all possible orderings of running those transactions one at a time.

The SQL standard and PostgreSQL-implemented transaction isolation levels are described in Table 13-1.

Table 13-1. Transaction Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	Allowed, but not in PG	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

In PostgreSQL, you can request any of the four standard transaction isolation levels, but internally only three distinct isolation levels are implemented, i.e., PostgreSQL's Read Uncommitted mode behaves like Read Committed. This is because it is the only sensible way to map the standard isolation levels to PostgreSQL's multiversion concurrency control architecture.

The table also shows that PostgreSQL's Repeatable Read implementation does not allow phantom reads. Stricter behavior is permitted by the SQL standard: the four isolation levels only define which phenomena must not happen, not which phenomena *must* happen. The behavior of the available isolation levels is detailed in the following subsections.

To set the transaction isolation level of a transaction, use the command SET TRANSACTION.

Important: Some PostgreSQL data types and functions have special rules regarding transactional behavior. In particular, changes made to a sequence (and therefore the counter of a column declared using `serial`) are immediately visible to all other transactions and are not rolled back if the transaction that made the changes aborts. See Section 9.16 and Section 8.1.4.

13.2.1. Read Committed Isolation Level

Read Committed is the default isolation level in PostgreSQL. When a transaction uses this isolation level, a `SELECT` query (without a `FOR UPDATE/SHARE` clause) sees only data committed before the query be-

gan; it never sees either uncommitted data or changes committed during query execution by concurrent transactions. In effect, a SELECT query sees a snapshot of the database as of the instant the query begins to run. However, SELECT does see the effects of previous updates executed within its own transaction, even though they are not yet committed. Also note that two successive SELECT commands can see different data, even though they are within a single transaction, if other transactions commit changes after the first SELECT starts and before the second SELECT starts.

UPDATE, DELETE, SELECT FOR UPDATE, and SELECT FOR SHARE commands behave the same as SELECT in terms of searching for target rows: they will only find target rows that were committed as of the command start time. However, such a target row might have already been updated (or deleted or locked) by another concurrent transaction by the time it is found. In this case, the would-be updater will wait for the first updating transaction to commit or roll back (if it is still in progress). If the first updater rolls back, then its effects are negated and the second updater can proceed with updating the originally found row. If the first updater commits, the second updater will ignore the row if the first updater deleted it, otherwise it will attempt to apply its operation to the updated version of the row. The search condition of the command (the WHERE clause) is re-evaluated to see if the updated version of the row still matches the search condition. If so, the second updater proceeds with its operation using the updated version of the row. In the case of SELECT FOR UPDATE and SELECT FOR SHARE, this means it is the updated version of the row that is locked and returned to the client.

INSERT with an ON CONFLICT DO UPDATE clause behaves similarly. In Read Committed mode, each row proposed for insertion will either insert or update. Unless there are unrelated errors, one of those two outcomes is guaranteed. If a conflict originates in another transaction whose effects are not yet visible to the INSERT , the UPDATE clause will affect that row, even though possibly *no* version of that row is conventionally visible to the command.

INSERT with an ON CONFLICT DO NOTHING clause may have insertion not proceed for a row due to the outcome of another transaction whose effects are not visible to the INSERT snapshot. Again, this is only the case in Read Committed mode.

Because of the above rules, it is possible for an updating command to see an inconsistent snapshot: it can see the effects of concurrent updating commands on the same rows it is trying to update, but it does not see effects of those commands on other rows in the database. This behavior makes Read Committed mode unsuitable for commands that involve complex search conditions; however, it is just right for simpler cases. For example, consider updating bank balances with transactions like:

```
BEGIN;
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 12345;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 7534;
COMMIT;
```

If two such transactions concurrently try to change the balance of account 12345, we clearly want the second transaction to start with the updated version of the account's row. Because each command is affecting only a predetermined row, letting it see the updated version of the row does not create any troublesome inconsistency.

More complex usage can produce undesirable results in Read Committed mode. For example, consider a DELETE command operating on data that is being both added and removed from its restriction criteria by another command, e.g., assume website is a two-row table with website.hits equaling 9 and 10:

```
BEGIN;
UPDATE website SET hits = hits + 1;
```

```
-- run from another session:  DELETE FROM website WHERE hits = 10;
COMMIT;
```

The `DELETE` will have no effect even though there is a `website.hits = 10` row before and after the `UPDATE`. This occurs because the pre-update row value `9` is skipped, and when the `UPDATE` completes and `DELETE` obtains a lock, the new row value is no longer `10` but `11`, which no longer matches the criteria.

Because Read Committed mode starts each command with a new snapshot that includes all transactions committed up to that instant, subsequent commands in the same transaction will see the effects of the committed concurrent transaction in any case. The point at issue above is whether or not a *single* command sees an absolutely consistent view of the database.

The partial transaction isolation provided by Read Committed mode is adequate for many applications, and this mode is fast and simple to use; however, it is not sufficient for all cases. Applications that do complex queries and updates might require a more rigorously consistent view of the database than Read Committed mode provides.

13.2.2. Repeatable Read Isolation Level

The *Repeatable Read* isolation level only sees data committed before the transaction began; it never sees either uncommitted data or changes committed during transaction execution by concurrent transactions. (However, the query does see the effects of previous updates executed within its own transaction, even though they are not yet committed.) This is a stronger guarantee than is required by the SQL standard for this isolation level, and prevents all of the phenomena described in Table 13-1. As mentioned above, this is specifically allowed by the standard, which only describes the *minimum* protections each isolation level must provide.

This level is different from Read Committed in that a query in a repeatable read transaction sees a snapshot as of the start of the first non-transaction-control statement in the *transaction*, not as of the start of the current statement within the transaction. Thus, successive `SELECT` commands within a *single* transaction see the same data, i.e., they do not see changes made by other transactions that committed after their own transaction started.

Applications using this level must be prepared to retry transactions due to serialization failures.

`UPDATE`, `DELETE`, `SELECT FOR UPDATE`, and `SELECT FOR SHARE` commands behave the same as `SELECT` in terms of searching for target rows: they will only find target rows that were committed as of the transaction start time. However, such a target row might have already been updated (or deleted or locked) by another concurrent transaction by the time it is found. In this case, the repeatable read transaction will wait for the first updating transaction to commit or roll back (if it is still in progress). If the first updater rolls back, then its effects are negated and the repeatable read transaction can proceed with updating the originally found row. But if the first updater commits (and actually updated or deleted the row, not just locked it) then the repeatable read transaction will be rolled back with the message

```
ERROR:  could not serialize access due to concurrent update
```

because a repeatable read transaction cannot modify or lock rows changed by other transactions after the repeatable read transaction began.

When an application receives this error message, it should abort the current transaction and retry the whole transaction from the beginning. The second time through, the transaction will see the previously-

committed change as part of its initial view of the database, so there is no logical conflict in using the new version of the row as the starting point for the new transaction's update.

Note that only updating transactions might need to be retried; read-only transactions will never have serialization conflicts.

The Repeatable Read mode provides a rigorous guarantee that each transaction sees a completely stable view of the database. However, this view will not necessarily always be consistent with some serial (one at a time) execution of concurrent transactions of the same level. For example, even a read only transaction at this level may see a control record updated to show that a batch has been completed but *not* see one of the detail records which is logically part of the batch because it read an earlier revision of the control record. Attempts to enforce business rules by transactions running at this isolation level are not likely to work correctly without careful use of explicit locks to block conflicting transactions.

The Repeatable Read isolation level is implemented using a technique known in academic database literature and in some other database products as *Snapshot Isolation*. Differences in behavior and performance may be observed when compared with systems that use a traditional locking technique that reduces concurrency. Some other systems may even offer Repeatable Read and Snapshot Isolation as distinct isolation levels with different behavior. The permitted phenomena that distinguish the two techniques were not formalized by database researchers until after the SQL standard was developed, and are outside the scope of this manual. For a full treatment, please see *A Critique of ANSI SQL Isolation Levels*.

> **Note:** Prior to PostgreSQL version 9.1, a request for the Serializable transaction isolation level provided exactly the same behavior described here. To retain the legacy Serializable behavior, Repeatable Read should now be requested.

13.2.3. Serializable Isolation Level

The *Serializable* isolation level provides the strictest transaction isolation. This level emulates serial transaction execution for all committed transactions; as if transactions had been executed one after another, serially, rather than concurrently. However, like the Repeatable Read level, applications using this level must be prepared to retry transactions due to serialization failures. In fact, this isolation level works exactly the same as Repeatable Read except that it monitors for conditions which could make execution of a concurrent set of serializable transactions behave in a manner inconsistent with all possible serial (one at a time) executions of those transactions. This monitoring does not introduce any blocking beyond that present in repeatable read, but there is some overhead to the monitoring, and detection of the conditions which could cause a *serialization anomaly* will trigger a *serialization failure*.

As an example, consider a table `mytab`, initially containing:

```
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

Suppose that serializable transaction A computes:

```
SELECT SUM(value) FROM mytab WHERE class = 1;
```

and then inserts the result (30) as the `value` in a new row with `class = 2`. Concurrently, serializable transaction B computes:

```
SELECT SUM(value) FROM mytab WHERE class = 2;
```

and obtains the result 300, which it inserts in a new row with `class = 1`. Then both transactions try to commit. If either transaction were running at the Repeatable Read isolation level, both would be allowed to commit; but since there is no serial order of execution consistent with the result, using Serializable transactions will allow one transaction to commit and will roll the other back with this message:

```
ERROR:  could not serialize access due to read/write dependencies among transactions
```

This is because if A had executed before B, B would have computed the sum 330, not 300, and similarly the other order would have resulted in a different sum computed by A.

When relying on Serializable transactions to prevent anomalies, it is important that any data read from a permanent user table not be considered valid until the transaction which read it has successfully committed. This is true even for read-only transactions, except that data read within a *deferrable* read-only transaction is known to be valid as soon as it is read, because such a transaction waits until it can acquire a snapshot guaranteed to be free from such problems before starting to read any data. In all other cases applications must not depend on results read during a transaction that later aborted; instead, they should retry the transaction until it succeeds.

To guarantee true serializability PostgreSQL uses *predicate locking*, which means that it keeps locks which allow it to determine when a write would have had an impact on the result of a previous read from a concurrent transaction, had it run first. In PostgreSQL these locks do not cause any blocking and therefore can *not* play any part in causing a deadlock. They are used to identify and flag dependencies among concurrent Serializable transactions which in certain combinations can lead to serialization anomalies. In contrast, a Read Committed or Repeatable Read transaction which wants to ensure data consistency may need to take out a lock on an entire table, which could block other users attempting to use that table, or it may use `SELECT FOR UPDATE` or `SELECT FOR SHARE` which not only can block other transactions but cause disk access.

Predicate locks in PostgreSQL, like in most other database systems, are based on data actually accessed by a transaction. These will show up in the `pg_locks` system view with a `mode` of `SIReadLock`. The particular locks acquired during execution of a query will depend on the plan used by the query, and multiple finer-grained locks (e.g., tuple locks) may be combined into fewer coarser-grained locks (e.g., page locks) during the course of the transaction to prevent exhaustion of the memory used to track the locks. A `READ ONLY` transaction may be able to release its SIRead locks before completion, if it detects that no conflicts can still occur which could lead to a serialization anomaly. In fact, `READ ONLY` transactions will often be able to establish that fact at startup and avoid taking any predicate locks. If you explicitly request a `SERIALIZABLE READ ONLY DEFERRABLE` transaction, it will block until it can establish this fact. (This is the *only* case where Serializable transactions block but Repeatable Read transactions don't.) On the other hand, SIRead locks often need to be kept past transaction commit, until overlapping read write transactions complete.

Consistent use of Serializable transactions can simplify development. The guarantee that any set of successfully committed concurrent Serializable transactions will have the same effect as if they were run one at a time means that if you can demonstrate that a single transaction, as written, will do the right thing when run by itself, you can have confidence that it will do the right thing in any mix of Serializable

transactions, even without any information about what those other transactions might do, or it will not successfully commit. It is important that an environment which uses this technique have a generalized way of handling serialization failures (which always return with a SQLSTATE value of '40001'), because it will be very hard to predict exactly which transactions might contribute to the read/write dependencies and need to be rolled back to prevent serialization anomalies. The monitoring of read/write dependencies has a cost, as does the restart of transactions which are terminated with a serialization failure, but balanced against the cost and blocking involved in use of explicit locks and `SELECT FOR UPDATE` or `SELECT FOR SHARE`, Serializable transactions are the best performance choice for some environments.

While PostgreSQL's Serializable transaction isolation level only allows concurrent transactions to commit if it can prove there is a serial order of execution that would produce the same effect, it doesn't always prevent errors from being raised that would not occur in true serial execution. In particular, it is possible to see unique constraint violations caused by conflicts with overlapping Serializable transactions even after explicitly checking that the key isn't present before attempting to insert it. This can be avoided by making sure that *all* Serializable transactions that insert potentially conflicting keys explicitly check if they can do so first. For example, imagine an application that asks the user for a new key and then checks that it doesn't exist already by trying to select it first, or generates a new key by selecting the maximum existing key and adding one. If some Serializable transactions insert new keys directly without following this protocol, unique constraints violations might be reported even in cases where they could not occur in a serial execution of the concurrent transactions.

For optimal performance when relying on Serializable transactions for concurrency control, these issues should be considered:

- Declare transactions as `READ ONLY` when possible.
- Control the number of active connections, using a connection pool if needed. This is always an important performance consideration, but it can be particularly important in a busy system using Serializable transactions.
- Don't put more into a single transaction than needed for integrity purposes.
- Don't leave connections dangling "idle in transaction" longer than necessary.
- Eliminate explicit locks, `SELECT FOR UPDATE`, and `SELECT FOR SHARE` where no longer needed due to the protections automatically provided by Serializable transactions.
- When the system is forced to combine multiple page-level predicate locks into a single relation-level predicate lock because the predicate lock table is short of memory, an increase in the rate of serialization failures may occur. You can avoid this by increasing max_pred_locks_per_transaction.
- A sequential scan will always necessitate a relation-level predicate lock. This can result in an increased rate of serialization failures. It may be helpful to encourage the use of index scans by reducing random_page_cost and/or increasing cpu_tuple_cost. Be sure to weigh any decrease in transaction rollbacks and restarts against any overall change in query execution time.

The Serializable isolation level is implemented using a technique known in academic database literature as Serializable Snapshot Isolation, which builds on Snapshot Isolation by adding checks for serialization anomalies. Some differences in behavior and performance may be observed when compared with other systems that use a traditional locking technique. Please see *Serializable Snapshot Isolation in PostgreSQL* for detailed information.

13.3. Explicit Locking

PostgreSQL provides various lock modes to control concurrent access to data in tables. These modes can be used for application-controlled locking in situations where MVCC does not give the desired behavior. Also, most PostgreSQL commands automatically acquire locks of appropriate modes to ensure that referenced tables are not dropped or modified in incompatible ways while the command executes. (For example, TRUNCATE cannot safely be executed concurrently with other operations on the same table, so it obtains an exclusive lock on the table to enforce that.)

To examine a list of the currently outstanding locks in a database server, use the pg_locks system view. For more information on monitoring the status of the lock manager subsystem, refer to Chapter 27.

13.3.1. Table-level Locks

The list below shows the available lock modes and the contexts in which they are used automatically by PostgreSQL. You can also acquire any of these locks explicitly with the command LOCK. Remember that all of these lock modes are table-level locks, even if the name contains the word "row"; the names of the lock modes are historical. To some extent the names reflect the typical usage of each lock mode — but the semantics are all the same. The only real difference between one lock mode and another is the set of lock modes with which each conflicts (see Table 13-2). Two transactions cannot hold locks of conflicting modes on the same table at the same time. (However, a transaction never conflicts with itself. For example, it might acquire ACCESS EXCLUSIVE lock and later acquire ACCESS SHARE lock on the same table.) Non-conflicting lock modes can be held concurrently by many transactions. Notice in particular that some lock modes are self-conflicting (for example, an ACCESS EXCLUSIVE lock cannot be held by more than one transaction at a time) while others are not self-conflicting (for example, an ACCESS SHARE lock can be held by multiple transactions).

Table-level Lock Modes

ACCESS SHARE

Conflicts with the ACCESS EXCLUSIVE lock mode only.

The SELECT command acquires a lock of this mode on referenced tables. In general, any query that only *reads* a table and does not modify it will acquire this lock mode.

ROW SHARE

Conflicts with the EXCLUSIVE and ACCESS EXCLUSIVE lock modes.

The SELECT FOR UPDATE and SELECT FOR SHARE commands acquire a lock of this mode on the target table(s) (in addition to ACCESS SHARE locks on any other tables that are referenced but not selected FOR UPDATE/FOR SHARE).

ROW EXCLUSIVE

Conflicts with the SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes.

The commands UPDATE, DELETE, and INSERT acquire this lock mode on the target table (in addition to ACCESS SHARE locks on any other referenced tables). In general, this lock mode will be acquired by any command that *modifies data* in a table.

SHARE UPDATE EXCLUSIVE

Conflicts with the SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode protects a table against concurrent schema changes and VACUUM runs.

Acquired by VACUUM (without FULL), ANALYZE, CREATE INDEX CONCURRENTLY, and ALTER TABLE VALIDATE and other ALTER TABLE variants (for full details see ALTER TABLE).

SHARE

Conflicts with the ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode protects a table against concurrent data changes.

Acquired by CREATE INDEX (without CONCURRENTLY).

SHARE ROW EXCLUSIVE

Conflicts with the ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode protects a table against concurrent data changes, and is self-exclusive so that only one session can hold it at a time.

Acquired by CREATE TRIGGER and many forms of ALTER TABLE (see ALTER TABLE).

EXCLUSIVE

Conflicts with the ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE lock modes. This mode allows only concurrent ACCESS SHARE locks, i.e., only reads from the table can proceed in parallel with a transaction holding this lock mode.

Acquired by REFRESH MATERIALIZED VIEW CONCURRENTLY.

ACCESS EXCLUSIVE

Conflicts with locks of all modes (ACCESS SHARE, ROW SHARE, ROW EXCLUSIVE, SHARE UPDATE EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE, and ACCESS EXCLUSIVE). This mode guarantees that the holder is the only transaction accessing the table in any way.

Acquired by the DROP TABLE, TRUNCATE, REINDEX, CLUSTER, VACUUM FULL, and REFRESH MATERIALIZED VIEW (without CONCURRENTLY) commands. Many forms of ALTER TABLE also acquire a lock at this level. This is also the default lock mode for LOCK TABLE statements that do not specify a mode explicitly.

Tip: Only an ACCESS EXCLUSIVE lock blocks a SELECT (without FOR UPDATE/SHARE) statement.

Once acquired, a lock is normally held until the end of the transaction. But if a lock is acquired after establishing a savepoint, the lock is released immediately if the savepoint is rolled back to. This is consistent with the principle that ROLLBACK cancels all effects of the commands since the savepoint. The same holds for locks acquired within a PL/pgSQL exception block: an error escape from the block releases locks acquired within it.

Table 13-2. Conflicting Lock Modes

Requested Lock Mode	Current Lock Mode							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

13.3.2. Row-level Locks

In addition to table-level locks, there are row-level locks, which are listed as below with the contexts in which they are used automatically by PostgreSQL. See Table 13-3 for a complete table of row-level lock conflicts. Note that a transaction can hold conflicting locks on the same row, even in different subtransactions; but other than that, two transactions can never hold conflicting locks on the same row. Row-level locks do not affect data querying; they block only *writers and lockers* to the same row. Row-level locks are released at transaction end or during savepoint rollback, just like table-level locks.

Row-level Lock Modes

`FOR UPDATE`

`FOR UPDATE` causes the rows retrieved by the `SELECT` statement to be locked as though for update. This prevents them from being locked, modified or deleted by other transactions until the current transaction ends. That is, other transactions that attempt `UPDATE`, `DELETE`, `SELECT FOR UPDATE`, `SELECT FOR NO KEY UPDATE`, `SELECT FOR SHARE` or `SELECT FOR KEY SHARE` of these rows will be blocked until the current transaction ends; conversely, `SELECT FOR UPDATE` will wait for a concurrent transaction that has run any of those commands on the same row, and will then lock and return the updated row (or no row, if the row was deleted). Within a `REPEATABLE READ` or `SERIALIZABLE` transaction, however, an error will be thrown if a row to be locked has changed since the transaction started. For further discussion see Section 13.4.

The `FOR UPDATE` lock mode is also acquired by any `DELETE` on a row, and also by an `UPDATE` that modifies the values of certain columns. Currently, the set of columns considered for the `UPDATE` case are those that have a unique index on them that can be used in a foreign key (so partial indexes and expressional indexes are not considered), but this may change in the future.

`FOR NO KEY UPDATE`

Behaves similarly to `FOR UPDATE`, except that the lock acquired is weaker: this lock will not block `SELECT FOR KEY SHARE` commands that attempt to acquire a lock on the same rows. This lock mode is also acquired by any `UPDATE` that does not acquire a `FOR UPDATE` lock.

`FOR SHARE`

Behaves similarly to `FOR NO KEY UPDATE`, except that it acquires a shared lock rather than exclusive lock on each retrieved row. A shared lock blocks other transactions from performing `UPDATE`, `DELETE`, `SELECT FOR UPDATE` or `SELECT FOR NO KEY UPDATE` on these rows, but it does not prevent them from performing `SELECT FOR SHARE` or `SELECT FOR KEY SHARE`.

`FOR KEY SHARE`

Behaves similarly to `FOR SHARE`, except that the lock is weaker: `SELECT FOR UPDATE` is blocked, but not `SELECT FOR NO KEY UPDATE`. A key-shared lock blocks other transactions from performing `DELETE` or any `UPDATE` that changes the key values, but not other `UPDATE`, and neither does it prevent `SELECT FOR NO KEY UPDATE`, `SELECT FOR SHARE`, or `SELECT FOR KEY SHARE`.

PostgreSQL doesn't remember any information about modified rows in memory, so there is no limit on the number of rows locked at one time. However, locking a row might cause a disk write, e.g., `SELECT FOR UPDATE` modifies selected rows to mark them locked, and so will result in disk writes.

Table 13-3. Conflicting Row-level Locks

Requested Lock Mode	Current Lock Mode			
	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X

Requested Lock Mode	Current Lock Mode			
	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

13.3.3. Page-level Locks

In addition to table and row locks, page-level share/exclusive locks are used to control read/write access to table pages in the shared buffer pool. These locks are released immediately after a row is fetched or updated. Application developers normally need not be concerned with page-level locks, but they are mentioned here for completeness.

13.3.4. Deadlocks

The use of explicit locking can increase the likelihood of *deadlocks*, wherein two (or more) transactions each hold locks that the other wants. For example, if transaction 1 acquires an exclusive lock on table A and then tries to acquire an exclusive lock on table B, while transaction 2 has already exclusive-locked table B and now wants an exclusive lock on table A, then neither one can proceed. PostgreSQL automatically detects deadlock situations and resolves them by aborting one of the transactions involved, allowing the other(s) to complete. (Exactly which transaction will be aborted is difficult to predict and should not be relied upon.)

Note that deadlocks can also occur as the result of row-level locks (and thus, they can occur even if explicit locking is not used). Consider the case in which two concurrent transactions modify a table. The first transaction executes:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;
```

This acquires a row-level lock on the row with the specified account number. Then, the second transaction executes:

```
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;
```

The first `UPDATE` statement successfully acquires a row-level lock on the specified row, so it succeeds in updating that row. However, the second `UPDATE` statement finds that the row it is attempting to update has already been locked, so it waits for the transaction that acquired the lock to complete. Transaction two is now waiting on transaction one to complete before it continues execution. Now, transaction one executes:

```
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;
```

Transaction one attempts to acquire a row-level lock on the specified row, but it cannot: transaction two already holds such a lock. So it waits for transaction two to complete. Thus, transaction one is blocked

on transaction two, and transaction two is blocked on transaction one: a deadlock condition. PostgreSQL will detect this situation and abort one of the transactions.

The best defense against deadlocks is generally to avoid them by being certain that all applications using a database acquire locks on multiple objects in a consistent order. In the example above, if both transactions had updated the rows in the same order, no deadlock would have occurred. One should also ensure that the first lock acquired on an object in a transaction is the most restrictive mode that will be needed for that object. If it is not feasible to verify this in advance, then deadlocks can be handled on-the-fly by retrying transactions that abort due to deadlocks.

So long as no deadlock situation is detected, a transaction seeking either a table-level or row-level lock will wait indefinitely for conflicting locks to be released. This means it is a bad idea for applications to hold transactions open for long periods of time (e.g., while waiting for user input).

13.3.5. Advisory Locks

PostgreSQL provides a means for creating locks that have application-defined meanings. These are called *advisory locks*, because the system does not enforce their use — it is up to the application to use them correctly. Advisory locks can be useful for locking strategies that are an awkward fit for the MVCC model. For example, a common use of advisory locks is to emulate pessimistic locking strategies typical of so-called "flat file" data management systems. While a flag stored in a table could be used for the same purpose, advisory locks are faster, avoid table bloat, and are automatically cleaned up by the server at the end of the session.

There are two ways to acquire an advisory lock in PostgreSQL: at session level or at transaction level. Once acquired at session level, an advisory lock is held until explicitly released or the session ends. Unlike standard lock requests, session-level advisory lock requests do not honor transaction semantics: a lock acquired during a transaction that is later rolled back will still be held following the rollback, and likewise an unlock is effective even if the calling transaction fails later. A lock can be acquired multiple times by its owning process; for each completed lock request there must be a corresponding unlock request before the lock is actually released. Transaction-level lock requests, on the other hand, behave more like regular lock requests: they are automatically released at the end of the transaction, and there is no explicit unlock operation. This behavior is often more convenient than the session-level behavior for short-term usage of an advisory lock. Session-level and transaction-level lock requests for the same advisory lock identifier will block each other in the expected way. If a session already holds a given advisory lock, additional requests by it will always succeed, even if other sessions are awaiting the lock; this statement is true regardless of whether the existing lock hold and new request are at session level or transaction level.

Like all locks in PostgreSQL, a complete list of advisory locks currently held by any session can be found in the `pg_locks` system view.

Both advisory locks and regular locks are stored in a shared memory pool whose size is defined by the configuration variables max_locks_per_transaction and max_connections. Care must be taken not to exhaust this memory or the server will be unable to grant any locks at all. This imposes an upper limit on the number of advisory locks grantable by the server, typically in the tens to hundreds of thousands depending on how the server is configured.

In certain cases using advisory locking methods, especially in queries involving explicit ordering and `LIMIT` clauses, care must be taken to control the locks acquired because of the order in which SQL expressions are evaluated. For example:

```
SELECT pg_advisory_lock(id) FROM foo WHERE id = 12345; -- ok
SELECT pg_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100; -- danger!
SELECT pg_advisory_lock(q.id) FROM
(
  SELECT id FROM foo WHERE id > 12345 LIMIT 100
) q; -- ok
```

In the above queries, the second form is dangerous because the `LIMIT` is not guaranteed to be applied before the locking function is executed. This might cause some locks to be acquired that the application was not expecting, and hence would fail to release (until it ends the session). From the point of view of the application, such locks would be dangling, although still viewable in `pg_locks`.

The functions provided to manipulate advisory locks are described in Section 9.26.10.

13.4. Data Consistency Checks at the Application Level

It is very difficult to enforce business rules regarding data integrity using Read Committed transactions because the view of the data is shifting with each statement, and even a single statement may not restrict itself to the statement's snapshot if a write conflict occurs.

While a Repeatable Read transaction has a stable view of the data throughout its execution, there is a subtle issue with using MVCC snapshots for data consistency checks, involving something known as *read/write conflicts*. If one transaction writes data and a concurrent transaction attempts to read the same data (whether before or after the write), it cannot see the work of the other transaction. The reader then appears to have executed first regardless of which started first or which committed first. If that is as far as it goes, there is no problem, but if the reader also writes data which is read by a concurrent transaction there is now a transaction which appears to have run before either of the previously mentioned transactions. If the transaction which appears to have executed last actually commits first, it is very easy for a cycle to appear in a graph of the order of execution of the transactions. When such a cycle appears, integrity checks will not work correctly without some help.

As mentioned in Section 13.2.3, Serializable transactions are just Repeatable Read transactions which add nonblocking monitoring for dangerous patterns of read/write conflicts. When a pattern is detected which could cause a cycle in the apparent order of execution, one of the transactions involved is rolled back to break the cycle.

13.4.1. Enforcing Consistency With Serializable Transactions

If the Serializable transaction isolation level is used for all writes and for all reads which need a consistent view of the data, no other effort is required to ensure consistency. Software from other environments which is written to use serializable transactions to ensure consistency should "just work" in this regard in PostgreSQL.

When using this technique, it will avoid creating an unnecessary burden for application programmers if the application software goes through a framework which automatically retries transactions which are rolled back with a serialization failure. It may be a good idea to set `default_transaction_isolation` to `serializable`. It would also be wise to take some action to ensure that no other transaction isolation

level is used, either inadvertently or to subvert integrity checks, through checks of the transaction isolation level in triggers.

See Section 13.2.3 for performance suggestions.

Warning

This level of integrity protection using Serializable transactions does not yet extend to hot standby mode (Section 25.5). Because of that, those using hot standby may want to use Repeatable Read and explicit locking on the master.

13.4.2. Enforcing Consistency With Explicit Blocking Locks

When non-serializable writes are possible, to ensure the current validity of a row and protect it against concurrent updates one must use `SELECT FOR UPDATE`, `SELECT FOR SHARE`, or an appropriate `LOCK TABLE` statement. (`SELECT FOR UPDATE` and `SELECT FOR SHARE` lock just the returned rows against concurrent updates, while `LOCK TABLE` locks the whole table.) This should be taken into account when porting applications to PostgreSQL from other environments.

Also of note to those converting from other environments is the fact that `SELECT FOR UPDATE` does not ensure that a concurrent transaction will not update or delete a selected row. To do that in PostgreSQL you must actually update the row, even if no values need to be changed. `SELECT FOR UPDATE` *temporarily blocks* other transactions from acquiring the same lock or executing an `UPDATE` or `DELETE` which would affect the locked row, but once the transaction holding this lock commits or rolls back, a blocked transaction will proceed with the conflicting operation unless an actual `UPDATE` of the row was performed while the lock was held.

Global validity checks require extra thought under non-serializable MVCC. For example, a banking application might wish to check that the sum of all credits in one table equals the sum of debits in another table, when both tables are being actively updated. Comparing the results of two successive `SELECT sum(...)` commands will not work reliably in Read Committed mode, since the second query will likely include the results of transactions not counted by the first. Doing the two sums in a single repeatable read transaction will give an accurate picture of only the effects of transactions that committed before the repeatable read transaction started — but one might legitimately wonder whether the answer is still relevant by the time it is delivered. If the repeatable read transaction itself applied some changes before trying to make the consistency check, the usefulness of the check becomes even more debatable, since now it includes some but not all post-transaction-start changes. In such cases a careful person might wish to lock all tables needed for the check, in order to get an indisputable picture of current reality. A `SHARE` mode (or higher) lock guarantees that there are no uncommitted changes in the locked table, other than those of the current transaction.

Note also that if one is relying on explicit locking to prevent concurrent changes, one should either use Read Committed mode, or in Repeatable Read mode be careful to obtain locks before performing queries. A lock obtained by a repeatable read transaction guarantees that no other transactions modifying the table are still running, but if the snapshot seen by the transaction predates obtaining the lock, it might predate some now-committed changes in the table. A repeatable read transaction's snapshot is actually frozen at the start of its first query or data-modification command (`SELECT`, `INSERT`, `UPDATE`, or `DELETE`), so it is possible to obtain locks explicitly before the snapshot is frozen.

13.5. Caveats

Some DDL commands, currently only TRUNCATE and the table-rewriting forms of ALTER TABLE, are not MVCC-safe. This means that after the truncation or rewrite commits, the table will appear empty to concurrent transactions, if they are using a snapshot taken before the DDL command committed. This will only be an issue for a transaction that did not access the table in question before the DDL command started — any transaction that has done so would hold at least an `ACCESS SHARE` table lock, which would block the DDL command until that transaction completes. So these commands will not cause any apparent inconsistency in the table contents for successive queries on the target table, but they could cause visible inconsistency between the contents of the target table and other tables in the database.

Support for the Serializable transaction isolation level has not yet been added to Hot Standby replication targets (described in Section 25.5). The strictest isolation level currently supported in hot standby mode is Repeatable Read. While performing all permanent database writes within Serializable transactions on the master will ensure that all standbys will eventually reach a consistent state, a Repeatable Read transaction run on the standby can sometimes see a transient state that is inconsistent with any serial execution of the transactions on the master.

Internal access to the system catalogs is not done using the isolation level of the current transaction. This means that newly created database objects such as tables are visible to concurrent Repeatable Read and Serializable transactions, even though the rows they contain are not. In contrast, queries that explicitly examine the system catalogs don't see rows representing concurrently created database objects, in the higher isolation levels.

13.6. Locking and Indexes

Though PostgreSQL provides nonblocking read/write access to table data, nonblocking read/write access is not currently offered for every index access method implemented in PostgreSQL. The various index types are handled as follows:

B-tree, GiST and SP-GiST indexes

: Short-term share/exclusive page-level locks are used for read/write access. Locks are released immediately after each index row is fetched or inserted. These index types provide the highest concurrency without deadlock conditions.

Hash indexes

: Share/exclusive hash-bucket-level locks are used for read/write access. Locks are released after the whole bucket is processed. Bucket-level locks provide better concurrency than index-level ones, but deadlock is possible since the locks are held longer than one index operation.

GIN indexes

: Short-term share/exclusive page-level locks are used for read/write access. Locks are released immediately after each index row is fetched or inserted. But note that insertion of a GIN-indexed value usually produces several index key insertions per row, so GIN might do substantial work for a single value's insertion.

Currently, B-tree indexes offer the best performance for concurrent applications; since they also have more features than hash indexes, they are the recommended index type for concurrent applications that need to index scalar data. When dealing with non-scalar data, B-trees are not useful, and GiST, SP-GiST or GIN indexes should be used instead.

Chapter 14. Performance Tips

Query performance can be affected by many things. Some of these can be controlled by the user, while others are fundamental to the underlying design of the system. This chapter provides some hints about understanding and tuning PostgreSQL performance.

14.1. Using EXPLAIN

PostgreSQL devises a *query plan* for each query it receives. Choosing the right plan to match the query structure and the properties of the data is absolutely critical for good performance, so the system includes a complex *planner* that tries to choose good plans. You can use the EXPLAIN command to see what query plan the planner creates for any query. Plan-reading is an art that requires some experience to master, but this section attempts to cover the basics.

Examples in this section are drawn from the regression test database after doing a VACUUM ANALYZE, using 9.3 development sources. You should be able to get similar results if you try the examples yourself, but your estimated costs and row counts might vary slightly because ANALYZE's statistics are random samples rather than exact, and because costs are inherently somewhat platform-dependent.

The examples use EXPLAIN's default "text" output format, which is compact and convenient for humans to read. If you want to feed EXPLAIN's output to a program for further analysis, you should use one of its machine-readable output formats (XML, JSON, or YAML) instead.

14.1.1. EXPLAIN Basics

The structure of a query plan is a tree of *plan nodes*. Nodes at the bottom level of the tree are scan nodes: they return raw rows from a table. There are different types of scan nodes for different table access methods: sequential scans, index scans, and bitmap index scans. There are also non-table row sources, such as VALUES clauses and set-returning functions in FROM, which have their own scan node types. If the query requires joining, aggregation, sorting, or other operations on the raw rows, then there will be additional nodes above the scan nodes to perform these operations. Again, there is usually more than one possible way to do these operations, so different node types can appear here too. The output of EXPLAIN has one line for each node in the plan tree, showing the basic node type plus the cost estimates that the planner made for the execution of that plan node. Additional lines might appear, indented from the node's summary line, to show additional properties of the node. The very first line (the summary line for the topmost node) has the estimated total execution cost for the plan; it is this number that the planner seeks to minimize.

Here is a trivial example, just to show what the output looks like:

```
EXPLAIN SELECT * FROM tenk1;

                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

Since this query has no WHERE clause, it must scan all the rows of the table, so the planner has chosen to use a simple sequential scan plan. The numbers that are quoted in parentheses are (left to right):

- Estimated start-up cost. This is the time expended before the output phase can begin, e.g., time to do the sorting in a sort node.
- Estimated total cost. This is stated on the assumption that the plan node is run to completion, i.e., all available rows are retrieved. In practice a node's parent node might stop short of reading all available rows (see the LIMIT example below).
- Estimated number of rows output by this plan node. Again, the node is assumed to be run to completion.
- Estimated average width of rows output by this plan node (in bytes).

The costs are measured in arbitrary units determined by the planner's cost parameters (see Section 18.7.2). Traditional practice is to measure the costs in units of disk page fetches; that is, seq_page_cost is conventionally set to 1.0 and the other cost parameters are set relative to that. The examples in this section are run with the default cost parameters.

It's important to understand that the cost of an upper-level node includes the cost of all its child nodes. It's also important to realize that the cost only reflects things that the planner cares about. In particular, the cost does not consider the time spent transmitting result rows to the client, which could be an important factor in the real elapsed time; but the planner ignores it because it cannot change it by altering the plan. (Every correct plan will output the same row set, we trust.)

The rows value is a little tricky because it is not the number of rows processed or scanned by the plan node, but rather the number emitted by the node. This is often less than the number scanned, as a result of filtering by any WHERE-clause conditions that are being applied at the node. Ideally the top-level rows estimate will approximate the number of rows actually returned, updated, or deleted by the query.

Returning to our example:

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

These numbers are derived very straightforwardly. If you do:

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';
```

you will find that tenk1 has 358 disk pages and 10000 rows. The estimated cost is computed as (disk pages read * seq_page_cost) + (rows scanned * cpu_tuple_cost). By default, seq_page_cost is 1.0 and cpu_tuple_cost is 0.01, so the estimated cost is (358 * 1.0) + (10000 * 0.01) = 458.

Now let's modify the query to add a WHERE condition:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

                         QUERY PLAN
------------------------------------------------------------
```

```
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7001 width=244)
   Filter: (unique1 < 7000)
```

Notice that the EXPLAIN output shows the WHERE clause being applied as a "filter" condition attached to the Seq Scan plan node. This means that the plan node checks the condition for each row it scans, and outputs only the ones that pass the condition. The estimate of output rows has been reduced because of the WHERE clause. However, the scan will still have to visit all 10000 rows, so the cost hasn't decreased; in fact it has gone up a bit (by 10000 * cpu_operator_cost, to be exact) to reflect the extra CPU time spent checking the WHERE condition.

The actual number of rows this query would select is 7000, but the rows estimate is only approximate. If you try to duplicate this experiment, you will probably get a slightly different estimate; moreover, it can change after each ANALYZE command, because the statistics produced by ANALYZE are taken from a randomized sample of the table.

Now, let's make the condition more restrictive:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

                                 QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.07..229.20 rows=101 width=244)
   Recheck Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

Here the planner has decided to use a two-step plan: the child plan node visits an index to find the locations of rows matching the index condition, and then the upper plan node actually fetches those rows from the table itself. Fetching rows separately is much more expensive than reading them sequentially, but because not all the pages of the table have to be visited, this is still cheaper than a sequential scan. (The reason for using two plan levels is that the upper plan node sorts the row locations identified by the index into physical order before reading them, to minimize the cost of separate fetches. The "bitmap" mentioned in the node names is the mechanism that does the sorting.)

Now let's add another condition to the WHERE clause:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringu1 = 'xxx';

                                 QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.04..229.43 rows=1 width=244)
   Recheck Cond: (unique1 < 100)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

The added condition stringu1 = 'xxx' reduces the output row count estimate, but not the cost because we still have to visit the same set of rows. Notice that the stringu1 clause cannot be applied as an index condition, since this index is only on the unique1 column. Instead it is applied as a filter on the rows retrieved by the index. Thus the cost has actually gone up slightly to reflect this extra checking.

In some cases the planner will prefer a "simple" index scan plan:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;

                                 QUERY PLAN
-----------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.29..8.30 rows=1 width=244)
   Index Cond: (unique1 = 42)
```

In this type of plan the table rows are fetched in index order, which makes them even more expensive to read, but there are so few that the extra cost of sorting the row locations is not worth it. You'll most often see this plan type for queries that fetch just a single row. It's also often used for queries that have an `ORDER BY` condition that matches the index order, because then no extra sorting step is needed to satisfy the `ORDER BY`.

If there are separate indexes on several of the columns referenced in `WHERE`, the planner might choose to use an AND or OR combination of the indexes:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
               Index Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0)
               Index Cond: (unique2 > 9000)
```

But this requires visiting both indexes, so it's not necessarily a win compared to using just one index and treating the other condition as a filter. If you vary the ranges involved you'll see the plan change accordingly.

Here is an example showing the effects of `LIMIT`:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                     QUERY PLAN
-------------------------------------------------------------------------------------
 Limit  (cost=0.29..14.48 rows=2 width=244)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..71.27 rows=10 width=244)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
```

This is the same query as above, but we added a `LIMIT` so that not all the rows need be retrieved, and the planner changed its mind about what to do. Notice that the total cost and row count of the Index Scan node are shown as if it were run to completion. However, the Limit node is expected to stop after retrieving only a fifth of those rows, so its total cost is only a fifth as much, and that's the actual estimated cost of the query. This plan is preferred over adding a Limit node to the previous plan because the Limit could not avoid paying the startup cost of the bitmap scan, so the total cost would be something over 25 units with that approach.

Let's try joining two tables, using the columns we have been discussing:

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
--------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1 width=244)
         Index Cond: (unique2 = t1.unique2)
```

In this plan, we have a nested-loop join node with two table scans as inputs, or children. The indentation of the node summary lines reflects the plan tree structure. The join's first, or "outer", child is a bitmap scan similar to those we saw before. Its cost and row count are the same as we'd get from `SELECT ... WHERE unique1 < 10` because we are applying the `WHERE` clause `unique1 < 10` at that node. The `t1.unique2 = t2.unique2` clause is not relevant yet, so it doesn't affect the row count of the outer scan. The nested-loop join node will run its second, or "inner" child once for each row obtained from the outer child. Column values from the current outer row can be plugged into the inner scan; here, the `t1.unique2` value from the outer row is available, so we get a plan and costs similar to what we saw above for a simple `SELECT ... WHERE t2.unique2 = constant` case. (The estimated cost is actually a bit lower than what was seen above, as a result of caching that's expected to occur during the repeated index scans on `t2`.) The costs of the loop node are then set on the basis of the cost of the outer scan, plus one repetition of the inner scan for each outer row (10 * 7.91, here), plus a little CPU time for join processing.

In this example the join's output row count is the same as the product of the two scans' row counts, but that's not true in all cases because there can be additional `WHERE` clauses that mention both tables and so can only be applied at the join point, not to either input scan. Here's an example:

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t2.unique2 < 10 AND t1.hundred < t2.hundred;

                                      QUERY PLAN
--------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..49.46 rows=33 width=488)
   Join Filter: (t1.hundred < t2.hundred)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Materialize  (cost=0.29..8.51 rows=10 width=244)
         ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..8.46 rows=10 width=244
               Index Cond: (unique2 < 10)
```

The condition `t1.hundred < t2.hundred` can't be tested in the `tenk2_unique2` index, so it's applied at the join node. This reduces the estimated output row count of the join node, but does not change either input scan.

Notice that here the planner has chosen to "materialize" the inner relation of the join, by putting a Materialize plan node atop it. This means that the `t2` index scan will be done just once, even though the nested-loop join node needs to read that data ten times, once for each row from the outer relation. The Materialize node saves the data in memory as it's read, and then returns the data from memory on each subsequent pass.

When dealing with outer joins, you might see join plan nodes with both "Join Filter" and plain "Filter" conditions attached. Join Filter conditions come from the outer join's `ON` clause, so a row that fails the Join Filter condition could still get emitted as a null-extended row. But a plain Filter condition is applied after the outer-join rules and so acts to remove rows unconditionally. In an inner join there is no semantic difference between these types of filters.

If we change the query's selectivity a bit, we might get a very different join plan:

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
------------------------------------------------------------------------------------------
 Hash Join  (cost=230.47..713.98 rows=101 width=488)
   Hash Cond: (t2.unique2 = t1.unique2)
   ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244)
   ->  Hash  (cost=229.20..229.20 rows=101 width=244)
         ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101 width=244)
               Recheck Cond: (unique1 < 100)
               ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
                     Index Cond: (unique1 < 100)
```

Here, the planner has chosen to use a hash join, in which rows of one table are entered into an in-memory hash table, after which the other table is scanned and the hash table is probed for matches to each row. Again note how the indentation reflects the plan structure: the bitmap scan on `tenk1` is the input to the Hash node, which constructs the hash table. That's then returned to the Hash Join node, which reads rows from its outer child plan and searches the hash table for each one.

Another possible type of join is a merge join, illustrated here:

```
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
------------------------------------------------------------------------------------------
 Merge Join  (cost=198.11..268.19 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101 width=244)
         Filter: (unique1 < 100)
   ->  Sort  (cost=197.83..200.33 rows=1000 width=244)
```

```
        Sort Key: t2.unique2
        ->  Seq Scan on onek t2  (cost=0.00..148.00 rows=1000 width=244)
```

Merge join requires its input data to be sorted on the join keys. In this plan the `tenk1` data is sorted by using an index scan to visit the rows in the correct order, but a sequential scan and sort is preferred for `onek`, because there are many more rows to be visited in that table. (Sequential-scan-and-sort frequently beats an index scan for sorting many rows, because of the nonsequential disk access required by the index scan.)

One way to look at variant plans is to force the planner to disregard whatever strategy it thought was the cheapest, using the enable/disable flags described in Section 18.7.1. (This is a crude tool, but useful. See also Section 14.3.) For example, if we're unconvinced that sequential-scan-and-sort is the best way to deal with table `onek` in the previous example, we could try

```
SET enable_sort = off;

EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                        QUERY PLAN
-----------------------------------------------------------------------------------------
 Merge Join  (cost=0.56..292.65 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101 width=244)
         Filter: (unique1 < 100)
   ->  Index Scan using onek_unique2 on onek t2  (cost=0.28..224.79 rows=1000 width=244)
```

which shows that the planner thinks that sorting `onek` by index-scanning is about 12% more expensive than sequential-scan-and-sort. Of course, the next question is whether it's right about that. We can investigate that using `EXPLAIN ANALYZE`, as discussed below.

14.1.2. EXPLAIN ANALYZE

It is possible to check the accuracy of the planner's estimates by using `EXPLAIN`'s `ANALYZE` option. With this option, `EXPLAIN` actually executes the query, and then displays the true row counts and true run time accumulated within each plan node, along with the same estimates that a plain `EXPLAIN` shows. For example, we might get a result like this:

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                                          QUERY PLAN
-----------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488) (actual time=0.128..0.377 rows=10 loops
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244) (actual time=0.05
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0) (actual
```

```
              Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1 width=244) (actu
         Index Cond: (unique2 = t1.unique2)
 Planning time: 0.181 ms
 Execution time: 0.501 ms
```

Note that the "actual time" values are in milliseconds of real time, whereas the `cost` estimates are expressed in arbitrary units; so they are unlikely to match up. The thing that's usually most important to look for is whether the estimated row counts are reasonably close to reality. In this example the estimates were all dead-on, but that's quite unusual in practice.

In some query plans, it is possible for a subplan node to be executed more than once. For example, the inner index scan will be executed once per outer row in the above nested-loop plan. In such cases, the `loops` value reports the total number of executions of the node, and the actual time and rows values shown are averages per-execution. This is done to make the numbers comparable with the way that the cost estimates are shown. Multiply by the `loops` value to get the total time actually spent in the node. In the above example, we spent a total of 0.220 milliseconds executing the index scans on `tenk2`.

In some cases `EXPLAIN ANALYZE` shows additional execution statistics beyond the plan node execution times and row counts. For example, Sort and Hash nodes provide extra information:

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY t1.fivethous;

                                                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Sort  (cost=717.34..717.59 rows=101 width=488) (actual time=7.761..7.774 rows=100 loops=1)
   Sort Key: t1.fivethous
   Sort Method: quicksort  Memory: 77kB
   ->  Hash Join  (cost=230.47..713.98 rows=101 width=488) (actual time=0.711..7.427 rows=1
         Hash Cond: (t2.unique2 = t1.unique2)
         ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244) (actual time=0.
         ->  Hash  (cost=229.20..229.20 rows=101 width=244) (actual time=0.659..0.659 rows=
               Buckets: 1024  Batches: 1  Memory Usage: 28kB
               ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101 width=244) (ac
                     Recheck Cond: (unique1 < 100)
                     ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 widt
                           Index Cond: (unique1 < 100)
 Planning time: 0.194 ms
 Execution time: 8.008 ms
```

The Sort node shows the sort method used (in particular, whether the sort was in-memory or on-disk) and the amount of memory or disk space needed. The Hash node shows the number of hash buckets and batches as well as the peak amount of memory used for the hash table. (If the number of batches exceeds one, there will also be disk space usage involved, but that is not shown.)

Another type of extra information is the number of rows removed by a filter condition:

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE ten < 7;

                                                QUERY PLAN
-------------------------------------------------------------------------------------------
```

```
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7000 width=244) (actual time=0.016..5.107 rows=
   Filter: (ten < 7)
   Rows Removed by Filter: 3000
 Planning time: 0.083 ms
 Execution time: 5.905 ms
```

These counts can be particularly valuable for filter conditions applied at join nodes. The "Rows Removed" line only appears when at least one scanned row, or potential join pair in the case of a join node, is rejected by the filter condition.

A case similar to filter conditions occurs with "lossy" index scans. For example, consider this search for polygons containing a specific point:

```
EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                                QUERY PLAN
-------------------------------------------------------------------------------------------
 Seq Scan on polygon_tbl  (cost=0.00..1.05 rows=1 width=32) (actual time=0.044..0.044 rows=
   Filter: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Filter: 4
 Planning time: 0.040 ms
 Execution time: 0.083 ms
```

The planner thinks (quite correctly) that this sample table is too small to bother with an index scan, so we have a plain sequential scan in which all the rows got rejected by the filter condition. But if we force an index scan to be used, we see:

```
SET enable_seqscan TO off;

EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                                      QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using gpolygonind on polygon_tbl  (cost=0.13..8.15 rows=1 width=32) (actual tim
   Index Cond: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Index Recheck: 1
 Planning time: 0.034 ms
 Execution time: 0.144 ms
```

Here we can see that the index returned one candidate row, which was then rejected by a recheck of the index condition. This happens because a GiST index is "lossy" for polygon containment tests: it actually returns the rows with polygons that overlap the target, and then we have to do the exact containment test on those rows.

`EXPLAIN` has a `BUFFERS` option that can be used with `ANALYZE` to get even more run time statistics:

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244) (actual time=0.323..0.342
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   Buffers: shared hit=15
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0) (actual time=0.309..0.309 rows=0 loop
```

```
         Buffers: shared hit=7
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0) (actual
               Index Cond: (unique1 < 100)
               Buffers: shared hit=2
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0) (actua
               Index Cond: (unique2 > 9000)
               Buffers: shared hit=5
 Planning time: 0.088 ms
 Execution time: 0.423 ms
```

The numbers provided by BUFFERS help to identify which parts of the query are the most I/O-intensive.

Keep in mind that because EXPLAIN ANALYZE actually runs the query, any side-effects will happen as usual, even though whatever results the query might output are discarded in favor of printing the EXPLAIN data. If you want to analyze a data-modifying query without changing your tables, you can roll the command back afterwards, for example:

```
BEGIN;

EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE unique1 < 100;

                                                          QUERY PLAN
------------------------------------------------------------------------------------------
 Update on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual time=14.628..14.628 rows=0
   ->  Bitmap Heap Scan on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual time=0.101
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0) (actual
               Index Cond: (unique1 < 100)
 Planning time: 0.079 ms
 Execution time: 14.727 ms

ROLLBACK;
```

As seen in this example, when the query is an INSERT, UPDATE, or DELETE command, the actual work of applying the table changes is done by a top-level Insert, Update, or Delete plan node. The plan nodes underneath this node perform the work of locating the old rows and/or computing the new data. So above, we see the same sort of bitmap table scan we've seen already, and its output is fed to an Update node that stores the updated rows. It's worth noting that although the data-modifying node can take a considerable amount of run time (here, it's consuming the lion's share of the time), the planner does not currently add anything to the cost estimates to account for that work. That's because the work to be done is the same for every correct query plan, so it doesn't affect planning decisions.

When an UPDATE or DELETE command affects an inheritance hierarchy, the output might look like this:

```
EXPLAIN UPDATE parent SET f2 = f2 + 1 WHERE f1 = 101;
                                    QUERY PLAN
-----------------------------------------------------------------------------
 Update on parent  (cost=0.00..24.53 rows=4 width=14)
   Update on parent
   Update on child1
   Update on child2
   Update on child3
```

```
->  Seq Scan on parent  (cost=0.00..0.00 rows=1 width=14)
      Filter: (f1 = 101)
->  Index Scan using child1_f1_key on child1  (cost=0.15..8.17 rows=1 width=14)
      Index Cond: (f1 = 101)
->  Index Scan using child2_f1_key on child2  (cost=0.15..8.17 rows=1 width=14)
      Index Cond: (f1 = 101)
->  Index Scan using child3_f1_key on child3  (cost=0.15..8.17 rows=1 width=14)
      Index Cond: (f1 = 101)
```

In this example the Update node needs to consider three child tables as well as the originally-mentioned parent table. So there are four input scanning subplans, one per table. For clarity, the Update node is annotated to show the specific target tables that will be updated, in the same order as the corresponding subplans. (These annotations are new as of PostgreSQL 9.5; in prior versions the reader had to intuit the target tables by inspecting the subplans.)

The `Planning time` shown by `EXPLAIN ANALYZE` is the time it took to generate the query plan from the parsed query and optimize it. It does not include parsing or rewriting.

The `Execution time` shown by `EXPLAIN ANALYZE` includes executor start-up and shut-down time, as well as the time to run any triggers that are fired, but it does not include parsing, rewriting, or planning time. Time spent executing `BEFORE` triggers, if any, is included in the time for the related Insert, Update, or Delete node; but time spent executing `AFTER` triggers is not counted there because `AFTER` triggers are fired after completion of the whole plan. The total time spent in each trigger (either `BEFORE` or `AFTER`) is also shown separately. Note that deferred constraint triggers will not be executed until end of transaction and are thus not considered at all by `EXPLAIN ANALYZE`.

14.1.3. Caveats

There are two significant ways in which run times measured by `EXPLAIN ANALYZE` can deviate from normal execution of the same query. First, since no output rows are delivered to the client, network transmission costs and I/O conversion costs are not included. Second, the measurement overhead added by `EXPLAIN ANALYZE` can be significant, especially on machines with slow `gettimeofday()` operating-system calls. You can use the pg_test_timing tool to measure the overhead of timing on your system.

`EXPLAIN` results should not be extrapolated to situations much different from the one you are actually testing; for example, results on a toy-sized table cannot be assumed to apply to large tables. The planner's cost estimates are not linear and so it might choose a different plan for a larger or smaller table. An extreme example is that on a table that only occupies one disk page, you'll nearly always get a sequential scan plan whether indexes are available or not. The planner realizes that it's going to take one disk page read to process the table in any case, so there's no value in expending additional page reads to look at an index. (We saw this happening in the `polygon_tbl` example above.)

There are cases in which the actual and estimated values won't match up well, but nothing is really wrong. One such case occurs when plan node execution is stopped short by a `LIMIT` or similar effect. For example, in the `LIMIT` query we used before,

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------
 Limit  (cost=0.29..14.71 rows=2 width=244) (actual time=0.177..0.249 rows=2 loops=1)
```

```
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..72.42 rows=10 width=244) (actua
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
         Rows Removed by Filter: 287
 Planning time: 0.096 ms
 Execution time: 0.336 ms
```

the estimated cost and row count for the Index Scan node are shown as though it were run to completion. But in reality the Limit node stopped requesting rows after it got two, so the actual row count is only 2 and the run time is less than the cost estimate would suggest. This is not an estimation error, only a discrepancy in the way the estimates and true values are displayed.

Merge joins also have measurement artifacts that can confuse the unwary. A merge join will stop reading one input if it's exhausted the other input and the next key value in the one input is greater than the last key value of the other input; in such a case there can be no more matches and so no need to scan the rest of the first input. This results in not reading all of one child, with results like those mentioned for `LIMIT`. Also, if the outer (first) child contains rows with duplicate key values, the inner (second) child is backed up and rescanned for the portion of its rows matching that key value. `EXPLAIN ANALYZE` counts these repeated emissions of the same inner rows as if they were real additional rows. When there are many outer duplicates, the reported actual row count for the inner child plan node can be significantly larger than the number of rows that are actually in the inner relation.

BitmapAnd and BitmapOr nodes always report their actual row counts as zero, due to implementation limitations.

14.2. Statistics Used by the Planner

As we saw in the previous section, the query planner needs to estimate the number of rows retrieved by a query in order to make good choices of query plans. This section provides a quick look at the statistics that the system uses for these estimates.

One component of the statistics is the total number of entries in each table and index, as well as the number of disk blocks occupied by each table and index. This information is kept in the table `pg_class`, in the columns `reltuples` and `relpages`. We can look at it with queries similar to this one:

```
SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';

       relname        | relkind | reltuples | relpages
----------------------+---------+-----------+----------
 tenk1                | r       |     10000 |      358
 tenk1_hundred        | i       |     10000 |       30
 tenk1_thous_tenthous | i       |     10000 |       30
 tenk1_unique1        | i       |     10000 |       30
 tenk1_unique2        | i       |     10000 |       30
(5 rows)
```

Here we can see that `tenk1` contains 10000 rows, as do its indexes, but the indexes are (unsurprisingly) much smaller than the table.

For efficiency reasons, `reltuples` and `relpages` are not updated on-the-fly, and so they usually contain somewhat out-of-date values. They are updated by `VACUUM`, `ANALYZE`, and a few DDL commands such as `CREATE INDEX`. A `VACUUM` or `ANALYZE` operation that does not scan the entire table (which is commonly the case) will incrementally update the `reltuples` count on the basis of the part of the table it did scan, resulting in an approximate value. In any case, the planner will scale the values it finds in `pg_class` to match the current physical table size, thus obtaining a closer approximation.

Most queries retrieve only a fraction of the rows in a table, due to `WHERE` clauses that restrict the rows to be examined. The planner thus needs to make an estimate of the *selectivity* of `WHERE` clauses, that is, the fraction of rows that match each condition in the `WHERE` clause. The information used for this task is stored in the `pg_statistic` system catalog. Entries in `pg_statistic` are updated by the `ANALYZE` and `VACUUM ANALYZE` commands, and are always approximate even when freshly updated.

Rather than look at `pg_statistic` directly, it's better to look at its view `pg_stats` when examining the statistics manually. `pg_stats` is designed to be more easily readable. Furthermore, `pg_stats` is readable by all, whereas `pg_statistic` is only readable by a superuser. (This prevents unprivileged users from learning something about the contents of other people's tables from the statistics. The `pg_stats` view is restricted to show only rows about tables that the current user can read.) For example, we might do:

```
SELECT attname, inherited, n_distinct,
       array_to_string(most_common_vals, E'\n') as most_common_vals
FROM pg_stats
WHERE tablename = 'road';

 attname | inherited | n_distinct |          most_common_vals
---------+-----------+------------+------------------------------------
 name    | f         |  -0.363388 | I- 580                          Ramp+
         |           |            | I- 880                          Ramp+
         |           |            | Sp Railroad                         +
         |           |            | I- 580                              +
         |           |            | I- 680                          Ramp
 name    | t         |  -0.284859 | I- 880                          Ramp+
         |           |            | I- 580                          Ramp+
         |           |            | I- 680                          Ramp+
         |           |            | I- 580                              +
         |           |            | State Hwy 13                    Ramp
(2 rows)
```

Note that two rows are displayed for the same column, one corresponding to the complete inheritance hierarchy starting at the `road` table (`inherited=t`), and another one including only the `road` table itself (`inherited=f`).

The amount of information stored in `pg_statistic` by `ANALYZE`, in particular the maximum number of entries in the `most_common_vals` and `histogram_bounds` arrays for each column, can be set on a column-by-column basis using the `ALTER TABLE SET STATISTICS` command, or globally by setting the default_statistics_target configuration variable. The default limit is presently 100 entries. Raising the limit might allow more accurate planner estimates to be made, particularly for columns with irregular data distributions, at the price of consuming more space in `pg_statistic` and slightly more time to compute the estimates. Conversely, a lower limit might be sufficient for columns with simple data distributions.

Further details about the planner's use of statistics can be found in Chapter 65.

14.3. Controlling the Planner with Explicit JOIN Clauses

It is possible to control the query planner to some extent by using the explicit JOIN syntax. To see why this matters, we first need some background.

In a simple join query, such as:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
```

the planner is free to join the given tables in any order. For example, it could generate a query plan that joins A to B, using the WHERE condition a.id = b.id, and then joins C to this joined table, using the other WHERE condition. Or it could join B to C and then join A to that result. Or it could join A to C and then join them with B — but that would be inefficient, since the full Cartesian product of A and C would have to be formed, there being no applicable condition in the WHERE clause to allow optimization of the join. (All joins in the PostgreSQL executor happen between two input tables, so it's necessary to build up the result in one or another of these fashions.) The important point is that these different join possibilities give semantically equivalent results but might have hugely different execution costs. Therefore, the planner will explore all of them to try to find the most efficient query plan.

When a query only involves two or three tables, there aren't many join orders to worry about. But the number of possible join orders grows exponentially as the number of tables expands. Beyond ten or so input tables it's no longer practical to do an exhaustive search of all the possibilities, and even for six or seven tables planning might take an annoyingly long time. When there are too many input tables, the PostgreSQL planner will switch from exhaustive search to a *genetic* probabilistic search through a limited number of possibilities. (The switch-over threshold is set by the geqo_threshold run-time parameter.) The genetic search takes less time, but it won't necessarily find the best possible plan.

When the query involves outer joins, the planner has less freedom than it does for plain (inner) joins. For example, consider:

```
SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

Although this query's restrictions are superficially similar to the previous example, the semantics are different because a row must be emitted for each row of A that has no matching row in the join of B and C. Therefore the planner has no choice of join order here: it must join B to C and then join A to that result. Accordingly, this query takes less time to plan than the previous query. In other cases, the planner might be able to determine that more than one join order is safe. For example, given:

```
SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);
```

it is valid to join A to either B or C first. Currently, only FULL JOIN completely constrains the join order. Most practical cases involving LEFT JOIN or RIGHT JOIN can be rearranged to some extent.

Explicit inner join syntax (INNER JOIN, CROSS JOIN, or unadorned JOIN) is semantically the same as listing the input relations in FROM, so it does not constrain the join order.

Even though most kinds of JOIN don't completely constrain the join order, it is possible to instruct the PostgreSQL query planner to treat all JOIN clauses as constraining the join order anyway. For example, these three queries are logically equivalent:

```
SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);
```

But if we tell the planner to honor the JOIN order, the second and third take less time to plan than the first. This effect is not worth worrying about for only three tables, but it can be a lifesaver with many tables.

To force the planner to follow the join order laid out by explicit JOINs, set the join_collapse_limit run-time parameter to 1. (Other possible values are discussed below.)

You do not need to constrain the join order completely in order to cut search time, because it's OK to use JOIN operators within items of a plain FROM list. For example, consider:

```
SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;
```

With join_collapse_limit = 1, this forces the planner to join A to B before joining them to other tables, but doesn't constrain its choices otherwise. In this example, the number of possible join orders is reduced by a factor of 5.

Constraining the planner's search in this way is a useful technique both for reducing planning time and for directing the planner to a good query plan. If the planner chooses a bad join order by default, you can force it to choose a better order via JOIN syntax — assuming that you know of a better order, that is. Experimentation is recommended.

A closely related issue that affects planning time is collapsing of subqueries into their parent query. For example, consider:

```
SELECT *
FROM x, y,
    (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

This situation might arise from use of a view that contains a join; the view's SELECT rule will be inserted in place of the view reference, yielding a query much like the above. Normally, the planner will try to collapse the subquery into the parent, yielding:

```
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

This usually results in a better plan than planning the subquery separately. (For example, the outer WHERE conditions might be such that joining X to A first eliminates many rows of A, thus avoiding the need to form the full logical output of the subquery.) But at the same time, we have increased the planning time; here, we have a five-way join problem replacing two separate three-way join problems. Because of the exponential growth of the number of possibilities, this makes a big difference. The planner tries to avoid getting stuck in huge join search problems by not collapsing a subquery if more than from_collapse_limit FROM items would result in the parent query. You can trade off planning time against quality of plan by adjusting this run-time parameter up or down.

from_collapse_limit and join_collapse_limit are similarly named because they do almost the same thing: one controls when the planner will "flatten out" subqueries, and the other controls when it will flatten out explicit joins. Typically you would either set join_collapse_limit equal to from_collapse_limit (so that explicit joins and subqueries act similarly) or set join_collapse_limit to 1 (if you want to control join order with explicit joins). But you might set them differently if you are trying to fine-tune the trade-off between planning time and run time.

14.4. Populating a Database

One might need to insert a large amount of data when first populating a database. This section contains some suggestions on how to make this process as efficient as possible.

14.4.1. Disable Autocommit

When using multiple `INSERT`s, turn off autocommit and just do one commit at the end. (In plain SQL, this means issuing `BEGIN` at the start and `COMMIT` at the end. Some client libraries might do this behind your back, in which case you need to make sure the library does it when you want it done.) If you allow each insertion to be committed separately, PostgreSQL is doing a lot of work for each row that is added. An additional benefit of doing all insertions in one transaction is that if the insertion of one row were to fail then the insertion of all rows inserted up to that point would be rolled back, so you won't be stuck with partially loaded data.

14.4.2. Use `COPY`

Use COPY to load all the rows in one command, instead of using a series of `INSERT` commands. The `COPY` command is optimized for loading large numbers of rows; it is less flexible than `INSERT`, but incurs significantly less overhead for large data loads. Since `COPY` is a single command, there is no need to disable autocommit if you use this method to populate a table.

If you cannot use `COPY`, it might help to use PREPARE to create a prepared `INSERT` statement, and then use `EXECUTE` as many times as required. This avoids some of the overhead of repeatedly parsing and planning `INSERT`. Different interfaces provide this facility in different ways; look for "prepared statements" in the interface documentation.

Note that loading a large number of rows using `COPY` is almost always faster than using `INSERT`, even if `PREPARE` is used and multiple insertions are batched into a single transaction.

`COPY` is fastest when used within the same transaction as an earlier `CREATE TABLE` or `TRUNCATE` command. In such cases no WAL needs to be written, because in case of an error, the files containing the newly loaded data will be removed anyway. However, this consideration only applies when wal_level is `minimal` as all commands must write WAL otherwise.

14.4.3. Remove Indexes

If you are loading a freshly created table, the fastest method is to create the table, bulk load the table's data using `COPY`, then create any indexes needed for the table. Creating an index on pre-existing data is quicker than updating it incrementally as each row is loaded.

If you are adding large amounts of data to an existing table, it might be a win to drop the indexes, load the table, and then recreate the indexes. Of course, the database performance for other users might suffer during the time the indexes are missing. One should also think twice before dropping a unique index, since the error checking afforded by the unique constraint will be lost while the index is missing.

14.4.4. Remove Foreign Key Constraints

Just as with indexes, a foreign key constraint can be checked "in bulk" more efficiently than row-by-row. So it might be useful to drop foreign key constraints, load data, and re-create the constraints. Again, there is a trade-off between data load speed and loss of error checking while the constraint is missing.

What's more, when you load data into a table with existing foreign key constraints, each new row requires an entry in the server's list of pending trigger events (since it is the firing of a trigger that checks the row's foreign key constraint). Loading many millions of rows can cause the trigger event queue to overflow available memory, leading to intolerable swapping or even outright failure of the command. Therefore it may be *necessary*, not just desirable, to drop and re-apply foreign keys when loading large amounts of data. If temporarily removing the constraint isn't acceptable, the only other recourse may be to split up the load operation into smaller transactions.

14.4.5. Increase `maintenance_work_mem`

Temporarily increasing the maintenance_work_mem configuration variable when loading large amounts of data can lead to improved performance. This will help to speed up `CREATE INDEX` commands and `ALTER TABLE ADD FOREIGN KEY` commands. It won't do much for `COPY` itself, so this advice is only useful when you are using one or both of the above techniques.

14.4.6. Increase `max_wal_size`

Temporarily increasing the max_wal_size configuration variable can also make large data loads faster. This is because loading a large amount of data into PostgreSQL will cause checkpoints to occur more often than the normal checkpoint frequency (specified by the `checkpoint_timeout` configuration variable). Whenever a checkpoint occurs, all dirty pages must be flushed to disk. By increasing `max_wal_size` temporarily during bulk data loads, the number of checkpoints that are required can be reduced.

14.4.7. Disable WAL Archival and Streaming Replication

When loading large amounts of data into an installation that uses WAL archiving or streaming replication, it might be faster to take a new base backup after the load has completed than to process a large amount of incremental WAL data. To prevent incremental WAL logging while loading, disable archiving and streaming replication, by setting wal_level to `minimal`, archive_mode to `off`, and max_wal_senders to zero. But note that changing these settings requires a server restart.

Aside from avoiding the time for the archiver or WAL sender to process the WAL data, doing this will actually make certain commands faster, because they are designed not to write WAL at all if `wal_level` is `minimal`. (They can guarantee crash safety more cheaply by doing an `fsync` at the end than by writing WAL.) This applies to the following commands:

- `CREATE TABLE AS SELECT`
- `CREATE INDEX` (and variants such as `ALTER TABLE ADD PRIMARY KEY`)
- `ALTER TABLE SET TABLESPACE`

- CLUSTER
- COPY FROM, when the target table has been created or truncated earlier in the same transaction

14.4.8. Run ANALYZE Afterwards

Whenever you have significantly altered the distribution of data within a table, running ANALYZE is strongly recommended. This includes bulk loading large amounts of data into the table. Running ANALYZE (or VACUUM ANALYZE) ensures that the planner has up-to-date statistics about the table. With no statistics or obsolete statistics, the planner might make poor decisions during query planning, leading to poor performance on any tables with inaccurate or nonexistent statistics. Note that if the autovacuum daemon is enabled, it might run ANALYZE automatically; see Section 23.1.3 and Section 23.1.6 for more information.

14.4.9. Some Notes About pg_dump

Dump scripts generated by pg_dump automatically apply several, but not all, of the above guidelines. To reload a pg_dump dump as quickly as possible, you need to do a few extra things manually. (Note that these points apply while *restoring* a dump, not while *creating* it. The same points apply whether loading a text dump with psql or using pg_restore to load from a pg_dump archive file.)

By default, pg_dump uses COPY, and when it is generating a complete schema-and-data dump, it is careful to load data before creating indexes and foreign keys. So in this case several guidelines are handled automatically. What is left for you to do is to:

- Set appropriate (i.e., larger than normal) values for maintenance_work_mem and max_wal_size.
- If using WAL archiving or streaming replication, consider disabling them during the restore. To do that, set archive_mode to off, wal_level to minimal, and max_wal_senders to zero before loading the dump. Afterwards, set them back to the right values and take a fresh base backup.
- Experiment with the parallel dump and restore modes of both pg_dump and pg_restore and find the optimal number of concurrent jobs to use. Dumping and restoring in parallel by means of the -j option should give you a significantly higher performance over the serial mode.
- Consider whether the whole dump should be restored as a single transaction. To do that, pass the -1 or --single-transaction command-line option to psql or pg_restore. When using this mode, even the smallest of errors will rollback the entire restore, possibly discarding many hours of processing. Depending on how interrelated the data is, that might seem preferable to manual cleanup, or not. COPY commands will run fastest if you use a single transaction and have WAL archiving turned off.
- If multiple CPUs are available in the database server, consider using pg_restore's --jobs option. This allows concurrent data loading and index creation.
- Run ANALYZE afterwards.

A data-only dump will still use COPY, but it does not drop or recreate indexes, and it does not normally touch foreign keys. [1] So when loading a data-only dump, it is up to you to drop and recreate indexes and foreign keys if you wish to use those techniques. It's still useful to increase max_wal_size while loading the data, but don't bother increasing maintenance_work_mem; rather, you'd do that while manually recreating indexes and foreign keys afterwards. And don't forget to ANALYZE when you're done; see Section 23.1.3 and Section 23.1.6 for more information.

14.5. Non-Durable Settings

Durability is a database feature that guarantees the recording of committed transactions even if the server crashes or loses power. However, durability adds significant database overhead, so if your site does not require such a guarantee, PostgreSQL can be configured to run much faster. The following are configuration changes you can make to improve performance in such cases. Except as noted below, durability is still guaranteed in case of a crash of the database software; only abrupt operating system stoppage creates a risk of data loss or corruption when these settings are used.

- Place the database cluster's data directory in a memory-backed file system (i.e., RAM disk). This eliminates all database disk I/O, but limits data storage to the amount of available memory (and perhaps swap).
- Turn off fsync; there is no need to flush data to disk.
- Turn off synchronous_commit; there might be no need to force WAL writes to disk on every commit. This setting does risk transaction loss (though not data corruption) in case of a crash of the *database*.
- Turn off full_page_writes; there is no need to guard against partial page writes.
- Increase max_wal_size and checkpoint_timeout; this reduces the frequency of checkpoints, but increases the storage requirements of /pg_xlog.
- Create unlogged tables to avoid WAL writes, though it makes the tables non-crash-safe.

1. You can get the effect of disabling foreign keys by using the --disable-triggers option — but realize that that eliminates, rather than just postpones, foreign key validation, and so it is possible to insert bad data if you use it.

III. Server Administration

This part covers topics that are of interest to a PostgreSQL database administrator. This includes installation of the software, set up and configuration of the server, management of users and databases, and maintenance tasks. Anyone who runs a PostgreSQL server, even for personal use, but especially in production, should be familiar with the topics covered in this part.

The information in this part is arranged approximately in the order in which a new user should read it. But the chapters are self-contained and can be read individually as desired. The information in this part is presented in a narrative fashion in topical units. Readers looking for a complete description of a particular command should see Part VI.

The first few chapters are written so they can be understood without prerequisite knowledge, so new users who need to set up their own server can begin their exploration with this part. The rest of this part is about tuning and management; that material assumes that the reader is familiar with the general use of the PostgreSQL database system. Readers are encouraged to look at Part I and Part II for additional information.

Chapter 15. Installation from Source Code

This chapter describes the installation of PostgreSQL using the source code distribution. (If you are installing a pre-packaged distribution, such as an RPM or Debian package, ignore this chapter and read the packager's instructions instead.)

15.1. Short Version

```
./configure
make
su
make install
adduser postgres
mkdir /usr/local/pgsql/data
chown postgres /usr/local/pgsql/data
su - postgres
/usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
/usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data >logfile 2>&1 &
/usr/local/pgsql/bin/createdb test
/usr/local/pgsql/bin/psql test
```

The long version is the rest of this chapter.

15.2. Requirements

In general, a modern Unix-compatible platform should be able to run PostgreSQL. The platforms that had received specific testing at the time of release are listed in Section 15.6 below. In the `doc` subdirectory of the distribution there are several platform-specific FAQ documents you might wish to consult if you are having trouble.

The following software packages are required for building PostgreSQL:

- GNU make version 3.80 or newer is required; other make programs or older GNU make versions will *not* work. (GNU make is sometimes installed under the name `gmake`.) To test for GNU make enter:

  ```
  make --version
  ```

- You need an ISO/ANSI C compiler (at least C89-compliant). Recent versions of GCC are recommended, but PostgreSQL is known to build using a wide variety of compilers from different vendors.
- tar is required to unpack the source distribution, in addition to either gzip or bzip2.
- The GNU Readline library is used by default. It allows psql (the PostgreSQL command line SQL interpreter) to remember each command you type, and allows you to use arrow keys to recall and edit previous commands. This is very helpful and is strongly recommended. If you don't want to use it then you must specify the `--without-readline` option to `configure`. As an alternative, you can often use the BSD-licensed `libedit` library, originally developed on NetBSD. The `libedit` library is GNU Readline-compatible and is used if `libreadline` is not found, or if `--with-libedit-preferred` is

used as an option to `configure`. If you are using a package-based Linux distribution, be aware that you need both the `readline` and `readline-devel` packages, if those are separate in your distribution.

- The zlib compression library is used by default. If you don't want to use it then you must specify the `--without-zlib` option to `configure`. Using this option disables support for compressed archives in pg_dump and pg_restore.

The following packages are optional. They are not required in the default configuration, but they are needed when certain build options are enabled, as explained below:

- To build the server programming language PL/Perl you need a full Perl installation, including the `libperl` library and the header files. The minimum required version is Perl 5.8.3. Since PL/Perl will be a shared library, the `libperl` library must be a shared library also on most platforms. This appears to be the default in recent Perl versions, but it was not in earlier versions, and in any case it is the choice of whomever installed Perl at your site. `configure` will fail if building PL/Perl is selected but it cannot find a shared `libperl`. In that case, you will have to rebuild and install Perl manually to be able to build PL/Perl. During the configuration process for Perl, request a shared library.

  If you intend to make more than incidental use of PL/Perl, you should ensure that the Perl installation was built with the `usemultiplicity` option enabled (`perl -V` will show whether this is the case).

- To build the PL/Python server programming language, you need a Python installation with the header files and the distutils module. The minimum required version is Python 2.3. (To work with function arguments of type `numeric`, a 2.3.x installation must include the separately-available `cdecimal` module; note the PL/Python regression tests will not pass if that is missing.) Python 3 is supported if it's version 3.1 or later; but see Section 43.1 when using Python 3.

  Since PL/Python will be a shared library, the `libpython` library must be a shared library also on most platforms. This is not the case in a default Python installation built from source, but a shared library is available in many operating system distributions. `configure` will fail if building PL/Python is selected but it cannot find a shared `libpython`. That might mean that you either have to install additional packages or rebuild (part of) your Python installation to provide this shared library. When building from source, run Python's configure with the `--enable-shared` flag.

- To build the PL/Tcl procedural language, you of course need a Tcl installation. If you are using a pre-8.4 release of Tcl, ensure that it was built without multithreading support.

- To enable Native Language Support (NLS), that is, the ability to display a program's messages in a language other than English, you need an implementation of the Gettext API. Some operating systems have this built-in (e.g., Linux, NetBSD, Solaris), for other systems you can download an add-on package from http://www.gnu.org/software/gettext/. If you are using the Gettext implementation in the GNU C library then you will additionally need the GNU Gettext package for some utility programs. For any of the other implementations you will not need it.

- You need Kerberos, OpenSSL, OpenLDAP, and/or PAM, if you want to support authentication or encryption using those services.

- To build the PostgreSQL documentation, there is a separate set of requirements; see Section J.2.

If you are building from a Git tree instead of using a released source package, or if you want to do server development, you also need the following packages:

- GNU Flex and Bison are needed to build from a Git checkout, or if you changed the actual scanner and parser definition files. If you need them, be sure to get Flex 2.5.31 or later and Bison 1.875 or later. Other lex and yacc programs cannot be used.
- Perl 5.8.3 or later is needed to build from a Git checkout, or if you changed the input files for any of the build steps that use Perl scripts. If building on Windows you will need Perl in any case. Perl is also required to run some test suites.

If you need to get a GNU package, you can find it at your local GNU mirror site (see http://www.gnu.org/order/ftp.html for a list) or at ftp://ftp.gnu.org/gnu/.

Also check that you have sufficient disk space. You will need about 100 MB for the source tree during compilation and about 20 MB for the installation directory. An empty database cluster takes about 35 MB; databases take about five times the amount of space that a flat text file with the same data would take. If you are going to run the regression tests you will temporarily need up to an extra 150 MB. Use the `df` command to check free disk space.

15.3. Getting The Source

The PostgreSQL 9.5.24 sources can be obtained from the download section of our website: https://www.postgresql.org/download/. You should get a file named `postgresql-9.5.24.tar.gz` or `postgresql-9.5.24.tar.bz2`. After you have obtained the file, unpack it:

```
gunzip postgresql-9.5.24.tar.gz
tar xf postgresql-9.5.24.tar
```

(Use `bunzip2` instead of `gunzip` if you have the `.bz2` file.) This will create a directory `postgresql-9.5.24` under the current directory with the PostgreSQL sources. Change into that directory for the rest of the installation procedure.

You can also get the source directly from the version control repository, see Appendix I.

15.4. Installation Procedure

1. Configuration

   The first step of the installation procedure is to configure the source tree for your system and choose the options you would like. This is done by running the `configure` script. For a default installation simply enter:

   ```
   ./configure
   ```

   This script will run a number of tests to determine values for various system dependent variables and detect any quirks of your operating system, and finally will create several files in the build tree to record what it found. You can also run `configure` in a directory outside the source tree, if you want to keep the build directory separate. This procedure is also called a *VPATH* build. Here's how:

   ```
   mkdir build_dir
   cd build_dir
   ```

```
/path/to/source/tree/configure [options go here]
make
```

The default configuration will build the server and utilities, as well as all client applications and interfaces that require only a C compiler. All files will be installed under `/usr/local/pgsql` by default.

You can customize the build and installation process by supplying one or more of the following command line options to `configure`:

`--prefix=`*`PREFIX`*

Install all files under the directory *`PREFIX`* instead of `/usr/local/pgsql`. The actual files will be installed into various subdirectories; no files will ever be installed directly into the *`PREFIX`* directory.

If you have special needs, you can also customize the individual subdirectories with the following options. However, if you leave these with their defaults, the installation will be relocatable, meaning you can move the directory after installation. (The `man` and `doc` locations are not affected by this.)

For relocatable installs, you might want to use `configure`'s `--disable-rpath` option. Also, you will need to tell the operating system how to find the shared libraries.

`--exec-prefix=`*`EXEC-PREFIX`*

You can install architecture-dependent files under a different prefix, *`EXEC-PREFIX`*, than what *`PREFIX`* was set to. This can be useful to share architecture-independent files between hosts. If you omit this, then *`EXEC-PREFIX`* is set equal to *`PREFIX`* and both architecture-dependent and independent files will be installed under the same tree, which is probably what you want.

`--bindir=`*`DIRECTORY`*

Specifies the directory for executable programs. The default is *`EXEC-PREFIX`*`/bin`, which normally means `/usr/local/pgsql/bin`.

`--sysconfdir=`*`DIRECTORY`*

Sets the directory for various configuration files, *`PREFIX`*`/etc` by default.

`--libdir=`*`DIRECTORY`*

Sets the location to install libraries and dynamically loadable modules. The default is *`EXEC-PREFIX`*`/lib`.

`--includedir=`*`DIRECTORY`*

Sets the directory for installing C and C++ header files. The default is *`PREFIX`*`/include`.

`--datarootdir=`*`DIRECTORY`*

Sets the root directory for various types of read-only data files. This only sets the default for some of the following options. The default is *`PREFIX`*`/share`.

`--datadir=`*`DIRECTORY`*

Sets the directory for read-only data files used by the installed programs. The default is *`DATAROOTDIR`*. Note that this has nothing to do with where your database files will be placed.

`--localedir=`*DIRECTORY*

Sets the directory for installing locale data, in particular message translation catalog files. The default is *DATAROOTDIR*`/locale`.

`--mandir=`*DIRECTORY*

The man pages that come with PostgreSQL will be installed under this directory, in their respective `man`*x* subdirectories. The default is *DATAROOTDIR*`/man`.

`--docdir=`*DIRECTORY*

Sets the root directory for installing documentation files, except "man" pages. This only sets the default for the following options. The default value for this option is *DATAROOTDIR*`/doc/postgresql`.

`--htmldir=`*DIRECTORY*

The HTML-formatted documentation for PostgreSQL will be installed under this directory. The default is *DATAROOTDIR*.

Note: Care has been taken to make it possible to install PostgreSQL into shared installation locations (such as `/usr/local/include`) without interfering with the namespace of the rest of the system. First, the string "`/postgresql`" is automatically appended to `datadir`, `sysconfdir`, and `docdir`, unless the fully expanded directory name already contains the string "`postgres`" or "`pgsql`". For example, if you choose `/usr/local` as prefix, the documentation will be installed in `/usr/local/doc/postgresql`, but if the prefix is `/opt/postgres`, then it will be in `/opt/postgres/doc`. The public C header files of the client interfaces are installed into `includedir` and are namespace-clean. The internal header files and the server header files are installed into private directories under `includedir`. See the documentation of each interface for information about how to access its header files. Finally, a private subdirectory will also be created, if appropriate, under `libdir` for dynamically loadable modules.

`--with-extra-version=`*STRING*

Append *STRING* to the PostgreSQL version number. You can use this, for example, to mark binaries built from unreleased Git snapshots or containing custom patches with an extra version string such as a `git describe` identifier or a distribution package release number.

`--with-includes=`*DIRECTORIES*

DIRECTORIES is a colon-separated list of directories that will be added to the list the compiler searches for header files. If you have optional packages (such as GNU Readline) installed in a non-standard location, you have to use this option and probably also the corresponding `--with-libraries` option.

Example: `--with-includes=/opt/gnu/include:/usr/sup/include`.

`--with-libraries=`*DIRECTORIES*

DIRECTORIES is a colon-separated list of directories to search for libraries. You will probably have to use this option (and the corresponding `--with-includes` option) if you have packages installed in non-standard locations.

Example: `--with-libraries=/opt/gnu/lib:/usr/sup/lib`.

`--enable-nls[=`*`LANGUAGES`*`]`

Enables Native Language Support (NLS), that is, the ability to display a program's messages in a language other than English. *`LANGUAGES`* is an optional space-separated list of codes of the languages that you want supported, for example `--enable-nls='de fr'`. (The intersection between your list and the set of actually provided translations will be computed automatically.) If you do not specify a list, then all available translations are installed.

To use this option, you will need an implementation of the Gettext API; see above.

`--with-pgport=`*`NUMBER`*

Set *`NUMBER`* as the default port number for server and clients. The default is 5432. The port can always be changed later on, but if you specify it here then both server and clients will have the same default compiled in, which can be very convenient. Usually the only good reason to select a non-default value is if you intend to run multiple PostgreSQL servers on the same machine.

`--with-perl`

Build the PL/Perl server-side language.

`--with-python`

Build the PL/Python server-side language.

`--with-tcl`

Build the PL/Tcl server-side language.

`--with-tclconfig=`*`DIRECTORY`*

Tcl installs the file `tclConfig.sh`, which contains configuration information needed to build modules interfacing to Tcl. This file is normally found automatically at a well-known location, but if you want to use a different version of Tcl you can specify the directory in which to look for it.

`--with-gssapi`

Build with support for GSSAPI authentication. On many systems, the GSSAPI (usually a part of the Kerberos installation) system is not installed in a location that is searched by default (e.g., `/usr/include`, `/usr/lib`), so you must use the options `--with-includes` and `--with-libraries` in addition to this option. `configure` will check for the required header files and libraries to make sure that your GSSAPI installation is sufficient before proceeding.

`--with-krb-srvnam=`*`NAME`*

The default name of the Kerberos service principal used by GSSAPI. `postgres` is the default. There's usually no reason to change this unless you have a Windows environment, in which case it must be set to upper case `POSTGRES`.

`--with-openssl`

Build with support for SSL (encrypted) connections. This requires the OpenSSL package to be installed. `configure` will check for the required header files and libraries to make sure that your OpenSSL installation is sufficient before proceeding.

`--with-pam`

Build with PAM (Pluggable Authentication Modules) support.

`--with-ldap`

Build with LDAP support for authentication and connection parameter lookup (see Section 31.17 and Section 19.3.7 for more information). On Unix, this requires the OpenLDAP package to be installed. On Windows, the default WinLDAP library is used. `configure` will check for the required header files and libraries to make sure that your OpenLDAP installation is sufficient before proceeding.

`--without-readline`

Prevents use of the Readline library (and libedit as well). This option disables command-line editing and history in psql, so it is not recommended.

`--with-libedit-preferred`

Favors the use of the BSD-licensed libedit library rather than GPL-licensed Readline. This option is significant only if you have both libraries installed; the default in that case is to use Readline.

`--with-bonjour`

Build with Bonjour support. This requires Bonjour support in your operating system. Recommended on OS X.

`--with-uuid=`*`LIBRARY`*

Build the uuid-ossp module (which provides functions to generate UUIDs), using the specified UUID library. *`LIBRARY`* must be one of:

- `bsd` to use the UUID functions found in FreeBSD, NetBSD, and some other BSD-derived systems
- `e2fs` to use the UUID library created by the `e2fsprogs` project; this library is present in most Linux systems and in OS X, and can be obtained for other platforms as well
- `ossp` to use the OSSP UUID library[1]

`--with-ossp-uuid`

Obsolete equivalent of `--with-uuid=ossp`.

`--with-libxml`

Build with libxml (enables SQL/XML support). Libxml version 2.6.23 or later is required for this feature.

Libxml installs a program `xml2-config` that can be used to detect the required compiler and linker options. PostgreSQL will use it automatically if found. To specify a libxml installation at an unusual location, you can either set the environment variable `XML2_CONFIG` to point to the `xml2-config` program belonging to the installation, or use the options `--with-includes` and `--with-libraries`.

`--with-libxslt`

Use libxslt when building the xml2 module. xml2 relies on this library to perform XSL transformations of XML.

1. http://www.ossp.org/pkg/lib/uuid/

`--disable-integer-datetimes`

Disable support for 64-bit integer storage for timestamps and intervals, and store datetime values as floating-point numbers instead. Floating-point datetime storage was the default in PostgreSQL releases prior to 8.4, but it is now deprecated, because it does not support microsecond precision for the full range of `timestamp` values. However, integer-based datetime storage requires a 64-bit integer type. Therefore, this option can be used when no such type is available, or for compatibility with applications written for prior versions of PostgreSQL. See Section 8.5 for more information.

`--disable-float4-byval`

Disable passing float4 values "by value", causing them to be passed "by reference" instead. This option costs performance, but may be needed for compatibility with old user-defined functions that are written in C and use the "version 0" calling convention. A better long-term solution is to update any such functions to use the "version 1" calling convention.

`--disable-float8-byval`

Disable passing float8 values "by value", causing them to be passed "by reference" instead. This option costs performance, but may be needed for compatibility with old user-defined functions that are written in C and use the "version 0" calling convention. A better long-term solution is to update any such functions to use the "version 1" calling convention. Note that this option affects not only float8, but also int8 and some related types such as timestamp. On 32-bit platforms, `--disable-float8-byval` is the default and it is not allowed to select `--enable-float8-byval`.

`--with-segsize=`*`SEGSIZE`*

Set the *segment size*, in gigabytes. Large tables are divided into multiple operating-system files, each of size equal to the segment size. This avoids problems with file size limits that exist on many platforms. The default segment size, 1 gigabyte, is safe on all supported platforms. If your operating system has "largefile" support (which most do, nowadays), you can use a larger segment size. This can be helpful to reduce the number of file descriptors consumed when working with very large tables. But be careful not to select a value larger than is supported by your platform and the file systems you intend to use. Other tools you might wish to use, such as tar, could also set limits on the usable file size. It is recommended, though not absolutely required, that this value be a power of 2. Note that changing this value requires an initdb.

`--with-blocksize=`*`BLOCKSIZE`*

Set the *block size*, in kilobytes. This is the unit of storage and I/O within tables. The default, 8 kilobytes, is suitable for most situations; but other values may be useful in special cases. The value must be a power of 2 between 1 and 32 (kilobytes). Note that changing this value requires an initdb.

`--with-wal-segsize=`*`SEGSIZE`*

Set the *WAL segment size*, in megabytes. This is the size of each individual file in the WAL log. It may be useful to adjust this size to control the granularity of WAL log shipping. The default size is 16 megabytes. The value must be a power of 2 between 1 and 64 (megabytes). Note that changing this value requires an initdb.

`--with-wal-blocksize=`*`BLOCKSIZE`*

Set the *WAL block size*, in kilobytes. This is the unit of storage and I/O within the WAL log. The default, 8 kilobytes, is suitable for most situations; but other values may be useful in special cases. The value must be a power of 2 between 1 and 64 (kilobytes). Note that changing this value requires an initdb.

`--disable-spinlocks`

Allow the build to succeed even if PostgreSQL has no CPU spinlock support for the platform. The lack of spinlock support will result in poor performance; therefore, this option should only be used if the build aborts and informs you that the platform lacks spinlock support. If this option is required to build PostgreSQL on your platform, please report the problem to the PostgreSQL developers.

`--disable-thread-safety`

Disable the thread-safety of client libraries. This prevents concurrent threads in libpq and ECPG programs from safely controlling their private connection handles.

`--with-system-tzdata=`*`DIRECTORY`*

PostgreSQL includes its own time zone database, which it requires for date and time operations. This time zone database is in fact compatible with the IANA time zone database provided by many operating systems such as FreeBSD, Linux, and Solaris, so it would be redundant to install it again. When this option is used, the system-supplied time zone database in *`DIRECTORY`* is used instead of the one included in the PostgreSQL source distribution. *`DIRECTORY`* must be specified as an absolute path. `/usr/share/zoneinfo` is a likely directory on some operating systems. Note that the installation routine will not detect mismatching or erroneous time zone data. If you use this option, you are advised to run the regression tests to verify that the time zone data you have pointed to works correctly with PostgreSQL.

This option is mainly aimed at binary package distributors who know their target operating system well. The main advantage of using this option is that the PostgreSQL package won't need to be upgraded whenever any of the many local daylight-saving time rules change. Another advantage is that PostgreSQL can be cross-compiled more straightforwardly if the time zone database files do not need to be built during the installation.

`--without-zlib`

Prevents use of the Zlib library. This disables support for compressed archives in pg_dump and pg_restore. This option is only intended for those rare systems where this library is not available.

`--enable-debug`

Compiles all programs and libraries with debugging symbols. This means that you can run the programs in a debugger to analyze problems. This enlarges the size of the installed executables considerably, and on non-GCC compilers it usually also disables compiler optimization, causing slowdowns. However, having the symbols available is extremely helpful for dealing with any problems that might arise. Currently, this option is recommended for production installations only if you use GCC. But you should always have it on if you are doing development work or running a beta version.

`--enable-coverage`

If using GCC, all programs and libraries are compiled with code coverage testing instrumenta-

tion. When run, they generate files in the build directory with code coverage metrics. See Section 30.5 for more information. This option is for use only with GCC and when doing development work.

`--enable-profiling`

If using GCC, all programs and libraries are compiled so they can be profiled. On backend exit, a subdirectory will be created that contains the `gmon.out` file for use in profiling. This option is for use only with GCC and when doing development work.

`--enable-cassert`

Enables *assertion* checks in the server, which test for many "cannot happen" conditions. This is invaluable for code development purposes, but the tests can slow down the server significantly. Also, having the tests turned on won't necessarily enhance the stability of your server! The assertion checks are not categorized for severity, and so what might be a relatively harmless bug will still lead to server restarts if it triggers an assertion failure. This option is not recommended for production use, but you should have it on for development work or when running a beta version.

`--enable-depend`

Enables automatic dependency tracking. With this option, the makefiles are set up so that all affected object files will be rebuilt when any header file is changed. This is useful if you are doing development work, but is just wasted overhead if you intend only to compile once and install. At present, this option only works with GCC.

`--enable-dtrace`

Compiles PostgreSQL with support for the dynamic tracing tool DTrace. See Section 27.4 for more information.

To point to the `dtrace` program, the environment variable `DTRACE` can be set. This will often be necessary because `dtrace` is typically installed under `/usr/sbin`, which might not be in the path.

Extra command-line options for the `dtrace` program can be specified in the environment variable `DTRACEFLAGS`. On Solaris, to include DTrace support in a 64-bit binary, you must specify `DTRACEFLAGS="-64"` to configure. For example, using the GCC compiler:

```
./configure CC='gcc -m64' --enable-dtrace DTRACEFLAGS='-64' ...
```

Using Sun's compiler:

```
./configure CC='/opt/SUNWspro/bin/cc -xtarget=native64' --enable-dtrace DTRACEFLAGS=
```

`--enable-tap-tests`

Enable tests using the Perl TAP tools. This requires a Perl installation and the Perl module `IPC::Run`. See Section 30.4 for more information.

If you prefer a C compiler different from the one `configure` picks, you can set the environment variable `CC` to the program of your choice. By default, `configure` will pick `gcc` if available, else the platform's default (usually `cc`). Similarly, you can override the default compiler flags if needed with the `CFLAGS` variable.

You can specify environment variables on the `configure` command line, for example:

```
./configure CC=/opt/bin/gcc CFLAGS='-O2 -pipe'
```

Here is a list of the significant variables that can be set in this manner:

`BISON`

Bison program

`CC`

C compiler

`CFLAGS`

options to pass to the C compiler

`CPP`

C preprocessor

`CPPFLAGS`

options to pass to the C preprocessor

`DTRACE`

location of the `dtrace` program

`DTRACEFLAGS`

options to pass to the `dtrace` program

`FLEX`

Flex program

`LDFLAGS`

options to use when linking either executables or shared libraries

`LDFLAGS_EX`

additional options for linking executables only

`LDFLAGS_SL`

additional options for linking shared libraries only

`MSGFMT`

`msgfmt` program for native language support

`PERL`

Full path name of the Perl interpreter. This will be used to determine the dependencies for building PL/Perl.

`PYTHON`

Full path name of the Python interpreter. This will be used to determine the dependencies for building PL/Python. Also, whether Python 2 or 3 is specified here (or otherwise implicitly chosen) determines which variant of the PL/Python language becomes available. See Section 43.1 for more information. If this is not set, the following are probed in this order: `python python3 python2`.

TCLSH

Full path name of the Tcl interpreter. This will be used to determine the dependencies for building PL/Tcl, and it will be substituted into Tcl scripts.

XML2_CONFIG

xml2-config program used to locate the libxml installation.

Sometimes it is useful to add compiler flags after-the-fact to the set that were chosen by configure. An important example is that gcc's -Werror option cannot be included in the CFLAGS passed to configure, because it will break many of configure's built-in tests. To add such flags, include them in the COPT environment variable while running make. The contents of COPT are added to both the CFLAGS and LDFLAGS options set up by configure. For example, you could do

```
make COPT='-Werror'
```

or

```
export COPT='-Werror'
make
```

Note: When developing code inside the server, it is recommended to use the configure options --enable-cassert (which turns on many run-time error checks) and --enable-debug (which improves the usefulness of debugging tools).

If using GCC, it is best to build with an optimization level of at least -O1, because using no optimization (-O0) disables some important compiler warnings (such as the use of uninitialized variables). However, non-zero optimization levels can complicate debugging because stepping through compiled code will usually not match up one-to-one with source code lines. If you get confused while trying to debug optimized code, recompile the specific files of interest with -O0. An easy way to do this is by passing an option to make: make PROFILE=-O0 file.o.

The COPT and PROFILE environment variables are actually handled identically by the PostgreSQL makefiles. Which to use is a matter of preference, but a common habit among developers is to use PROFILE for one-time flag adjustments, while COPT might be kept set all the time.

2. Build

   To start the build, type:

   ```
   make
   ```

   (Remember to use GNU make.) The build will take a few minutes depending on your hardware. The last line displayed should be:

   ```
   All of PostgreSQL successfully made. Ready to install.
   ```

   If you want to build everything that can be built, including the documentation (HTML and man pages), and the additional modules (contrib), type instead:

   ```
   make world
   ```

   The last line displayed should be:

   ```
   PostgreSQL, contrib, and documentation successfully made. Ready to install.
   ```

3. Regression Tests

If you want to test the newly built server before you install it, you can run the regression tests at this point. The regression tests are a test suite to verify that PostgreSQL runs on your machine in the way the developers expected it to. Type:

```
make check
```

(This won't work as root; do it as an unprivileged user.) Chapter 30 contains detailed information about interpreting the test results. You can repeat this test at any later time by issuing the same command.

4. Installing the Files

> **Note:** If you are upgrading an existing system be sure to read Section 17.6 which has instructions about upgrading a cluster.

To install PostgreSQL enter:

```
make install
```

This will install files into the directories that were specified in step 1. Make sure that you have appropriate permissions to write into that area. Normally you need to do this step as root. Alternatively, you can create the target directories in advance and arrange for appropriate permissions to be granted.

To install the documentation (HTML and man pages), enter:

```
make install-docs
```

If you built the world above, type instead:

```
make install-world
```

This also installs the documentation.

You can use `make install-strip` instead of `make install` to strip the executable files and libraries as they are installed. This will save some space. If you built with debugging support, stripping will effectively remove the debugging support, so it should only be done if debugging is no longer needed. `install-strip` tries to do a reasonable job saving space, but it does not have perfect knowledge of how to strip every unneeded byte from an executable file, so if you want to save all the disk space you possibly can, you will have to do manual work.

The standard installation provides all the header files needed for client application development as well as for server-side program development, such as custom functions or data types written in C. (Prior to PostgreSQL 8.0, a separate `make install-all-headers` command was needed for the latter, but this step has been folded into the standard install.)

Client-only installation: If you want to install only the client applications and interface libraries, then you can use these commands:

```
make -C src/bin install
make -C src/include install
make -C src/interfaces install
make -C doc install
```

`src/bin` has a few binaries for server-only use, but they are small.

Uninstallation: To undo the installation use the command `make uninstall`. However, this will not remove any created directories.

Cleaning: After the installation you can free disk space by removing the built files from the source tree with the command `make clean`. This will preserve the files made by the `configure` program, so that you can rebuild everything with `make` later on. To reset the source tree to the state in which it was distributed, use `make distclean`. If you are going to build for several platforms within the same source tree you must do this and re-configure for each platform. (Alternatively, use a separate build tree for each platform, so that the source tree remains unmodified.)

If you perform a build and then discover that your `configure` options were wrong, or if you change anything that `configure` investigates (for example, software upgrades), then it's a good idea to do `make distclean` before reconfiguring and rebuilding. Without this, your changes in configuration choices might not propagate everywhere they need to.

15.5. Post-Installation Setup

15.5.1. Shared Libraries

On some systems with shared libraries you need to tell the system how to find the newly installed shared libraries. The systems on which this is *not* necessary include FreeBSD, HP-UX, Linux, NetBSD, OpenBSD, and Solaris.

The method to set the shared library search path varies between platforms, but the most widely-used method is to set the environment variable `LD_LIBRARY_PATH` like so: In Bourne shells (`sh`, `ksh`, `bash`, `zsh`):

```
LD_LIBRARY_PATH=/usr/local/pgsql/lib
export LD_LIBRARY_PATH
```

or in `csh` or `tcsh`:

```
setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
```

Replace `/usr/local/pgsql/lib` with whatever you set `--libdir` to in step 1. You should put these commands into a shell start-up file such as `/etc/profile` or `~/.bash_profile`. Some good information about the caveats associated with this method can be found at http://xahlee.org/UnixResource_dir/_/ldpath.html.

On some systems it might be preferable to set the environment variable `LD_RUN_PATH` *before* building.

On Cygwin, put the library directory in the `PATH` or move the `.dll` files into the `bin` directory.

If in doubt, refer to the manual pages of your system (perhaps `ld.so` or `rld`). If you later get a message like:

```
psql: error in loading shared libraries
libpq.so.2.1: cannot open shared object file: No such file or directory
```

then this step was necessary. Simply take care of it then.

If you are on Linux and you have root access, you can run:

```
/sbin/ldconfig /usr/local/pgsql/lib
```

(or equivalent directory) after installation to enable the run-time linker to find the shared libraries faster. Refer to the manual page of `ldconfig` for more information. On FreeBSD, NetBSD, and OpenBSD the command is:

```
/sbin/ldconfig -m /usr/local/pgsql/lib
```

instead. Other systems are not known to have an equivalent command.

15.5.2. Environment Variables

If you installed into `/usr/local/pgsql` or some other location that is not searched for programs by default, you should add `/usr/local/pgsql/bin` (or whatever you set `--bindir` to in step 1) into your `PATH`. Strictly speaking, this is not necessary, but it will make the use of PostgreSQL much more convenient.

To do this, add the following to your shell start-up file, such as `~/.bash_profile` (or `/etc/profile`, if you want it to affect all users):

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

If you are using `csh` or `tcsh`, then use this command:

```
set path = ( /usr/local/pgsql/bin $path )
```

To enable your system to find the man documentation, you need to add lines like the following to a shell start-up file unless you installed into a location that is searched by default:

```
MANPATH=/usr/local/pgsql/share/man:$MANPATH
export MANPATH
```

The environment variables `PGHOST` and `PGPORT` specify to client applications the host and port of the database server, overriding the compiled-in defaults. If you are going to run client applications remotely then it is convenient if every user that plans to use the database sets `PGHOST`. This is not required, however; the settings can be communicated via command line options to most client programs.

15.6. Supported Platforms

A platform (that is, a CPU architecture and operating system combination) is considered supported by the PostgreSQL development community if the code contains provisions to work on that platform and it has recently been verified to build and pass its regression tests on that platform. Currently, most testing of platform compatibility is done automatically by test machines in the PostgreSQL Build Farm[2]. If you are interested in using PostgreSQL on a platform that is not represented in the build farm, but on which the

2. https://buildfarm.postgresql.org/

code works or can be made to work, you are strongly encouraged to set up a build farm member machine so that continued compatibility can be assured.

In general, PostgreSQL can be expected to work on these CPU architectures: x86, x86_64, IA64, PowerPC, PowerPC 64, S/390, S/390x, Sparc, Sparc 64, ARM, MIPS, MIPSEL, M68K, and PA-RISC. Code support exists for M32R and VAX, but these architectures are not known to have been tested recently. It is often possible to build on an unsupported CPU type by configuring with `--disable-spinlocks`, but performance will be poor.

PostgreSQL can be expected to work on these operating systems: Linux (all recent distributions), Windows (Win2000 SP4 and later), FreeBSD, OpenBSD, NetBSD, OS X, AIX, HP/UX, Solaris, and UnixWare. Other Unix-like systems may also work but are not currently being tested. In most cases, all CPU architectures supported by a given operating system will work. Look in the Section 15.7 below to see if there is information specific to your operating system, particularly if using an older system.

If you have installation problems on a platform that is known to be supported according to recent build farm results, please report it to <`pgsql-bugs@lists.postgresql.org`>. If you are interested in porting PostgreSQL to a new platform, <`pgsql-hackers@lists.postgresql.org`> is the appropriate place to discuss that.

15.7. Platform-specific Notes

This section documents additional platform-specific issues regarding the installation and setup of PostgreSQL. Be sure to read the installation instructions, and in particular Section 15.2 as well. Also, check Chapter 30 regarding the interpretation of regression test results.

Platforms that are not covered here have no known platform-specific installation issues.

15.7.1. AIX

PostgreSQL works on AIX, but getting it installed properly can be challenging. AIX versions from 4.3.3 to 6.1 are considered supported. You can use GCC or the native IBM compiler `xlc`. In general, using recent versions of AIX and PostgreSQL helps. Check the build farm for up to date information about which versions of AIX are known to work.

The minimum recommended fix levels for supported AIX versions are:

AIX 4.3.3

Maintenance Level 11 + post ML11 bundle

AIX 5.1

Maintenance Level 9 + post ML9 bundle

AIX 5.2

Technology Level 10 Service Pack 3

AIX 5.3

Technology Level 7

AIX 6.1

Base Level

To check your current fix level, use `oslevel -r` in AIX 4.3.3 to AIX 5.2 ML 7, or `oslevel -s` in later versions.

Use the following `configure` flags in addition to your own if you have installed Readline or libz in `/usr/local`: `--with-includes=/usr/local/include --with-libraries=/usr/local/lib`.

15.7.1.1. GCC Issues

On AIX 5.3, there have been some problems getting PostgreSQL to compile and run using GCC.

You will want to use a version of GCC subsequent to 3.3.2, particularly if you use a prepackaged version. We had good success with 4.0.1. Problems with earlier versions seem to have more to do with the way IBM packaged GCC than with actual issues with GCC, so that if you compile GCC yourself, you might well have success with an earlier version of GCC.

15.7.1.2. Unix-Domain Sockets Broken

AIX 5.3 has a problem where `sockaddr_storage` is not defined to be large enough. In version 5.3, IBM increased the size of `sockaddr_un`, the address structure for Unix-domain sockets, but did not correspondingly increase the size of `sockaddr_storage`. The result of this is that attempts to use Unix-domain sockets with PostgreSQL lead to libpq overflowing the data structure. TCP/IP connections work OK, but not Unix-domain sockets, which prevents the regression tests from working.

The problem was reported to IBM, and is recorded as bug report PMR29657. If you upgrade to maintenance level 5300-03 or later, that will include this fix. A quick workaround is to alter `_SS_MAXSIZE` to 1025 in `/usr/include/sys/socket.h`. In either case, recompile PostgreSQL once you have the corrected header file.

15.7.1.3. Internet Address Issues

PostgreSQL relies on the system's `getaddrinfo` function to parse IP addresses in `listen_addresses`, `pg_hba.conf`, etc. Older versions of AIX have assorted bugs in this function. If you have problems related to these settings, updating to the appropriate AIX fix level shown above should take care of it.

One user reports:

When implementing PostgreSQL version 8.1 on AIX 5.3, we periodically ran into problems where the statistics collector would "mysteriously" not come up successfully. This appears to be the result of unexpected behavior in the IPv6 implementation. It looks like PostgreSQL and IPv6 do not play very well together on AIX 5.3.

Any of the following actions "fix" the problem.

- Delete the IPv6 address for localhost:

```
(as root)
# ifconfig lo0 inet6 ::1/0 delete
```

- Remove IPv6 from net services. The file `/etc/netsvc.conf` on AIX is roughly equivalent to `/etc/nsswitch.conf` on Solaris/Linux. The default, on AIX, is thus:

  ```
  hosts=local,bind
  ```

  Replace this with:

  ```
  hosts=local4,bind4
  ```

  to deactivate searching for IPv6 addresses.

> **Warning**
>
> This is really a workaround for problems relating to immaturity of IPv6 support, which improved visibly during the course of AIX 5.3 releases. It has worked with AIX version 5.3, but does not represent an elegant solution to the problem. It has been reported that this workaround is not only unnecessary, but causes problems on AIX 6.1, where IPv6 support has become more mature.

15.7.1.4. Memory Management

AIX can be somewhat peculiar with regards to the way it does memory management. You can have a server with many multiples of gigabytes of RAM free, but still get out of memory or address space errors when running applications. One example is `createlang` failing with unusual errors. For example, running as the owner of the PostgreSQL installation:

```
-bash-3.00$ createlang plperl template1
createlang: language installation failed: ERROR:  could not load library "/opt/dbs/pgsql748
```

Running as a non-owner in the group possessing the PostgreSQL installation:

```
-bash-3.00$ createlang plperl template1
createlang: language installation failed: ERROR:  could not load library "/opt/dbs/pgsql748
```

Another example is out of memory errors in the PostgreSQL server logs, with every memory allocation near or greater than 256 MB failing.

The overall cause of all these problems is the default bittedness and memory model used by the server process. By default, all binaries built on AIX are 32-bit. This does not depend upon hardware type or kernel in use. These 32-bit processes are limited to 4 GB of memory laid out in 256 MB segments using one of a few models. The default allows for less than 256 MB in the heap as it shares a single segment with the stack.

In the case of the `createlang` example, above, check your umask and the permissions of the binaries in your PostgreSQL installation. The binaries involved in that example were 32-bit and installed as mode 750 instead of 755. Due to the permissions being set in this fashion, only the owner or a member of the possessing group can load the library. Since it isn't world-readable, the loader places the object into the process' heap instead of the shared library segments where it would otherwise be placed.

The "ideal" solution for this is to use a 64-bit build of PostgreSQL, but that is not always practical, because systems with 32-bit processors can build, but not run, 64-bit binaries.

If a 32-bit binary is desired, set LDR_CNTRL to MAXDATA=0x*n*0000000, where 1 <= n <= 8, before starting the PostgreSQL server, and try different values and postgresql.conf settings to find a configuration that works satisfactorily. This use of LDR_CNTRL tells AIX that you want the server to have MAXDATA bytes set aside for the heap, allocated in 256 MB segments. When you find a workable configuration, ldedit can be used to modify the binaries so that they default to using the desired heap size. PostgreSQL can also be rebuilt, passing configure LDFLAGS="-Wl,-bmaxdata:0x*n*0000000" to achieve the same effect.

For a 64-bit build, set OBJECT_MODE to 64 and pass CC="gcc -maix64" and LDFLAGS="-Wl,-bbigtoc" to configure. (Options for xlc might differ.) If you omit the export of OBJECT_MODE, your build may fail with linker errors. When OBJECT_MODE is set, it tells AIX's build utilities such as ar, as, and ld what type of objects to default to handling.

By default, overcommit of paging space can happen. While we have not seen this occur, AIX will kill processes when it runs out of memory and the overcommit is accessed. The closest to this that we have seen is fork failing because the system decided that there was not enough memory for another process. Like many other parts of AIX, the paging space allocation method and out-of-memory kill is configurable on a system- or process-wide basis if this becomes a problem.

15.7.2. Cygwin

PostgreSQL can be built using Cygwin, a Linux-like environment for Windows, but that method is inferior to the native Windows build (see Chapter 16) and running a server under Cygwin is no longer recommended.

When building from source, proceed according to the normal installation procedure (i.e., ./configure; make; etc.), noting the following-Cygwin specific differences:

- Set your path to use the Cygwin bin directory before the Windows utilities. This will help prevent problems with compilation.
- The adduser command is not supported; use the appropriate user management application on Windows NT, 2000, or XP. Otherwise, skip this step.
- The su command is not supported; use ssh to simulate su on Windows NT, 2000, or XP. Otherwise, skip this step.
- OpenSSL is not supported.
- Start cygserver for shared memory support. To do this, enter the command /usr/sbin/cygserver &. This program needs to be running anytime you start the PostgreSQL server or initialize a database cluster (initdb). The default cygserver configuration may need to be changed (e.g., increase SEMMNS) to prevent PostgreSQL from failing due to a lack of system resources.
- Building might fail on some systems where a locale other than C is in use. To fix this, set the locale to C by doing export LANG=C.utf8 before building, and then setting it back to the previous setting, after you have installed PostgreSQL.
- The parallel regression tests (make check) can generate spurious regression test failures due to overflowing the listen() backlog queue which causes connection refused errors or hangs. You can limit the number of connections using the make variable MAX_CONNECTIONS thus:

  ```
  make MAX_CONNECTIONS=5 check
  ```

(On some systems you can have up to about 10 simultaneous connections).

It is possible to install `cygserver` and the PostgreSQL server as Windows NT services. For information on how to do this, please refer to the `README` document included with the PostgreSQL binary package on Cygwin. It is installed in the directory `/usr/share/doc/Cygwin`.

15.7.3. HP-UX

PostgreSQL 7.3+ should work on Series 700/800 PA-RISC machines running HP-UX 10.X or 11.X, given appropriate system patch levels and build tools. At least one developer routinely tests on HP-UX 10.20, and we have reports of successful installations on HP-UX 11.00 and 11.11.

Aside from the PostgreSQL source distribution, you will need GNU make (HP's make will not do), and either GCC or HP's full ANSI C compiler. If you intend to build from Git sources rather than a distribution tarball, you will also need Flex (GNU lex) and Bison (GNU yacc). We also recommend making sure you are fairly up-to-date on HP patches. At a minimum, if you are building 64 bit binaries on HP-UX 11.11 you may need PHSS_30966 (11.11) or a successor patch otherwise `initdb` may hang:

PHSS_30966 s700_800 ld(1) and linker tools cumulative patch

On general principles you should be current on libc and ld/dld patches, as well as compiler patches if you are using HP's C compiler. See HP's support sites such as http://itrc.hp.com and ftp://us-ffs.external.hp.com/ for free copies of their latest patches.

If you are building on a PA-RISC 2.0 machine and want to have 64-bit binaries using GCC, you must use GCC 64-bit version. GCC binaries for HP-UX PA-RISC and Itanium are available from http://www.hp.com/go/gcc. Don't forget to get and install binutils at the same time.

If you are building on a PA-RISC 2.0 machine and want the compiled binaries to run on PA-RISC 1.1 machines you will need to specify `+DAportable` in `CFLAGS`.

If you are building on a HP-UX Itanium machine, you will need the latest HP ANSI C compiler with its dependent patch or successor patches:

PHSS_30848 s700_800 HP C Compiler (A.05.57)
PHSS_30849 s700_800 u2comp/be/plugin library Patch

If you have both HP's C compiler and GCC's, then you might want to explicitly select the compiler to use when you run `configure`:

```
./configure CC=cc
```

for HP's C compiler, or

```
./configure CC=gcc
```

for GCC. If you omit this setting, then configure will pick `gcc` if it has a choice.

The default install target location is `/usr/local/pgsql`, which you might want to change to something under `/opt`. If so, use the `--prefix` switch to `configure`.

In the regression tests, there might be some low-order-digit differences in the geometry tests, which vary depending on which compiler and math library versions you use. Any other error is cause for suspicion.

15.7.4. macOS

On recent macOS releases, it's necessary to embed the "sysroot" path in the include switches used to find some system header files. This results in the outputs of the configure script varying depending on which SDK version was used during configure. That shouldn't pose any problem in simple scenarios, but if you are trying to do something like building an extension on a different machine than the server code was built on, you may need to force use of a different sysroot path. To do that, set `PG_SYSROOT`, for example

```
make PG_SYSROOT=/desired/path all
```

To find out the appropriate path on your machine, run

```
xcodebuild -version -sdk macosx Path
```

Note that building an extension using a different sysroot version than was used to build the core server is not really recommended; in the worst case it could result in hard-to-debug ABI inconsistencies.

You can also select a non-default sysroot path when configuring, by specifying `PG_SYSROOT` to configure:

```
./configure ... PG_SYSROOT=/desired/path
```

macOS's "System Integrity Protection" (SIP) feature breaks `make check`, because it prevents passing the needed setting of `DYLD_LIBRARY_PATH` down to the executables being tested. You can work around that by doing `make install` before `make check`. Most Postgres developers just turn off SIP, though.

15.7.5. MinGW/Native Windows

PostgreSQL for Windows can be built using MinGW, a Unix-like build environment for Microsoft operating systems, or using Microsoft's Visual C++ compiler suite. The MinGW build variant uses the normal build system described in this chapter; the Visual C++ build works completely differently and is described in Chapter 16. It is a fully native build and uses no additional software like MinGW. A ready-made installer is available on the main PostgreSQL web site.

The native Windows port requires a 32 or 64-bit version of Windows 2000 or later. Earlier operating systems do not have sufficient infrastructure (but Cygwin may be used on those). MinGW, the Unix-like build tools, and MSYS, a collection of Unix tools required to run shell scripts like `configure`, can be downloaded from http://www.mingw.org/. Neither is required to run the resulting binaries; they are needed only for creating the binaries.

To build 64 bit binaries using MinGW, install the 64 bit tool set from http://mingw-w64.sourceforge.net/, put its bin directory in the `PATH`, and run `configure` with the `--host=x86_64-w64-mingw32` option.

After you have everything installed, it is suggested that you run psql under `CMD.EXE`, as the MSYS console has buffering issues.

15.7.5.1. Collecting Crash Dumps on Windows

If PostgreSQL on Windows crashes, it has the ability to generate minidumps that can be used to track down the cause for the crash, similar to core dumps on Unix. These dumps can be read using the Windows Debugger Tools or using Visual Studio. To enable the generation of dumps on Windows, create a subdirectory named `crashdumps` inside the cluster data directory. The dumps will then be written into this directory with a unique name based on the identifier of the crashing process and the current time of the crash.

15.7.6. SCO OpenServer and SCO UnixWare

PostgreSQL can be built on SCO UnixWare 7 and SCO OpenServer 5. On OpenServer, you can use either the OpenServer Development Kit or the Universal Development Kit. However, some tweaking may be needed, as described below.

15.7.6.1. Skunkware

You should locate your copy of the SCO Skunkware CD. The Skunkware CD is included with UnixWare 7 and current versions of OpenServer 5. Skunkware includes ready-to-install versions of many popular programs that are available on the Internet. For example, gzip, gunzip, GNU Make, Flex, and Bison are all included. For UnixWare 7.1, this CD is now labeled "Open License Software Supplement". If you do not have this CD, the software on it is available from http://www.sco.com/skunkware/.

Skunkware has different versions for UnixWare and OpenServer. Make sure you install the correct version for your operating system, except as noted below.

On UnixWare 7.1.3 and beyond, the GCC compiler is included on the UDK CD as is GNU Make.

15.7.6.2. GNU Make

You need to use the GNU Make program, which is on the Skunkware CD. By default, it installs as `/usr/local/bin/make`.

As of UnixWare 7.1.3 and above, the GNU Make program is the OSTK portion of the UDK CD, and is in `/usr/gnu/bin/gmake`.

15.7.6.3. Readline

The Readline library is on the Skunkware CD. But it is not included on the UnixWare 7.1 Skunkware CD. If you have the UnixWare 7.0.0 or 7.0.1 Skunkware CDs, you can install it from there. Otherwise, try http://www.sco.com/skunkware/.

By default, Readline installs into `/usr/local/lib` and `/usr/local/include`. However, the PostgreSQL `configure` program will not find it there without help. If you installed Readline, then use the following options to `configure`:

```
./configure --with-libraries=/usr/local/lib --with-includes=/usr/local/include
```

15.7.6.4. Using the UDK on OpenServer

If you are using the new Universal Development Kit (UDK) compiler on OpenServer, you need to specify the locations of the UDK libraries:

```
./configure --with-libraries=/udk/usr/lib --with-includes=/udk/usr/include
```

Putting these together with the Readline options from above:

```
./configure --with-libraries="/udk/usr/lib /usr/local/lib" --with-includes="/udk/usr/includ
```

15.7.6.5. Reading the PostgreSQL Man Pages

By default, the PostgreSQL man pages are installed into `/usr/local/pgsql/share/man`. By default, UnixWare does not look there for man pages. To be able to read them you need to modify the `MANPATH` variable in `/etc/default/man`, for example:

```
MANPATH=/usr/lib/scohelp/%L/man:/usr/dt/man:/usr/man:/usr/share/man:scohelp:/usr/local/man:
```

On OpenServer, some extra research needs to be invested to make the man pages usable, because the man system is a bit different from other platforms. Currently, PostgreSQL will not install them at all.

15.7.6.6. C99 Issues with the 7.1.1b Feature Supplement

For compilers earlier than the one released with OpenUNIX 8.0.0 (UnixWare 7.1.2), including the 7.1.1b Feature Supplement, you may need to specify `-Xb` in `CFLAGS` or the `CC` environment variable. The indication of this is an error in compiling `tuplesort.c` referencing inline functions. Apparently there was a change in the 7.1.2(8.0.0) compiler and beyond.

15.7.6.7. Threading on UnixWare

For threading, you*must* use `-Kpthread` on *all* libpq-using programs. libpq uses `pthread_*` calls, which are only available with the `-Kpthread`/`-Kthread` flag.

15.7.7. Solaris

PostgreSQL is well-supported on Solaris. The more up to date your operating system, the fewer issues you will experience; details below.

15.7.7.1. Required Tools

You can build with either GCC or Sun's compiler suite. For better code optimization, Sun's compiler is strongly recommended on the SPARC architecture. We have heard reports of problems when using GCC 2.95.1; GCC 2.95.3 or later is recommended. If you are using Sun's compiler, be careful not to select `/usr/ucb/cc`; use `/opt/SUNWspro/bin/cc`.

You can download Sun Studio from http://www.oracle.com/technetwork/server-storage/solarisstudio/downloads/. Many of GNU tools are integrated into Solaris 10, or they are present on the Solaris companion CD. If you like packages for older version of Solaris, you can find these tools at http://www.sunfreeware.com. If you prefer sources, look at http://www.gnu.org/order/ftp.html.

15.7.7.2. Problems with OpenSSL

When you build PostgreSQL with OpenSSL support you might get compilation errors in the following files:

- `src/backend/libpq/crypt.c`
- `src/backend/libpq/password.c`
- `src/interfaces/libpq/fe-auth.c`
- `src/interfaces/libpq/fe-connect.c`

This is because of a namespace conflict between the standard `/usr/include/crypt.h` header and the header files provided by OpenSSL.

Upgrading your OpenSSL installation to version 0.9.6a fixes this problem. Solaris 9 and above has a newer version of OpenSSL.

15.7.7.3. configure Complains About a Failed Test Program

If `configure` complains about a failed test program, this is probably a case of the run-time linker being unable to find some library, probably libz, libreadline or some other non-standard library such as libssl. To point it to the right location, set the `LDFLAGS` environment variable on the `configure` command line, e.g.,

```
configure ... LDFLAGS="-R /usr/sfw/lib:/opt/sfw/lib:/usr/local/lib"
```

See the ld man page for more information.

15.7.7.4. 64-bit Build Sometimes Crashes

On Solaris 7 and older, the 64-bit version of libc has a buggy `vsnprintf` routine, which leads to erratic core dumps in PostgreSQL. The simplest known workaround is to force PostgreSQL to use its own version of `vsnprintf` rather than the library copy. To do this, after you run `configure` edit a file produced by `configure`: In `src/Makefile.global`, change the line

```
LIBOBJS =
```

to read

```
LIBOBJS = snprintf.o
```

(There might be other files already listed in this variable. Order does not matter.) Then build as usual.

15.7.7.5. Compiling for Optimal Performance

On the SPARC architecture, Sun Studio is strongly recommended for compilation. Try using the `-xO5` optimization flag to generate significantly faster binaries. Do not use any flags that modify behavior of floating-point operations and `errno` processing (e.g., `-fast`). These flags could raise some nonstandard PostgreSQL behavior for example in the date/time computing.

If you do not have a reason to use 64-bit binaries on SPARC, prefer the 32-bit version. The 64-bit operations are slower and 64-bit binaries are slower than the 32-bit variants. And on other hand, 32-bit code on the AMD64 CPU family is not native, and that is why 32-bit code is significant slower on this CPU family.

15.7.7.6. Using DTrace for Tracing PostgreSQL

Yes, using DTrace is possible. See Section 27.4 for further information. You can also find more information in this article: https://blogs.oracle.com/robertlor/entry/user_level_dtrace_probes_in.

If you see the linking of the `postgres` executable abort with an error message like:

```
Undefined                       first referenced
 symbol                             in file
AbortTransaction                    utils/probes.o
CommitTransaction                   utils/probes.o
ld: fatal: Symbol referencing errors. No output written to postgres
collect2: ld returned 1 exit status
make: *** [postgres] Error 1
```

your DTrace installation is too old to handle probes in static functions. You need Solaris 10u4 or newer.

Chapter 16. Installation from Source Code on Windows

It is recommended that most users download the binary distribution for Windows, available as a graphical installer package from the PostgreSQL website. Building from source is only intended for people developing PostgreSQL or extensions.

There are several different ways of building PostgreSQL on Windows. The simplest way to build with Microsoft tools is to install Visual Studio 2019 and use the included compiler. It is also possible to build with the full Microsoft Visual C++ 2005 to 2019. In some cases that requires the installation of the Windows SDK in addition to the compiler.

It is also possible to build PostgreSQL using the GNU compiler tools provided by MinGW, or using Cygwin for older versions of Windows.

Finally, the client access library (libpq) can be built using Visual C++ 7.1 or Borland C++ for compatibility with statically linked applications built using these tools.

Building using MinGW or Cygwin uses the normal build system, see Chapter 15 and the specific notes in Section 15.7.5 and Section 15.7.2. To produce native 64 bit binaries in these environments, use the tools from MinGW-w64. These tools can also be used to cross-compile for 32 bit and 64 bit Windows targets on other hosts, such as Linux and Darwin. Cygwin is not recommended for running a production server, and it should only be used for running on older versions of Windows where the native build does not work, such as Windows 98. The official binaries are built using Visual Studio.

Native builds of psql don't support command line editing. The Cygwin build does support command line editing, so it should be used where psql is needed for interactive use on Windows.

16.1. Building with Visual C++ or the Microsoft Windows SDK

PostgreSQL can be built using the Visual C++ compiler suite from Microsoft. These compilers can be either from Visual Studio, Visual Studio Express or some versions of the Microsoft Windows SDK. If you do not already have a Visual Studio environment set up, the easiest ways are to use the compilers from Visual Studio 2019 or those in the Windows SDK 10, which are both free downloads from Microsoft.

Both 32-bit and 64-bit builds are possible with the Microsoft Compiler suite. 32-bit PostgreSQL builds are possible with Visual Studio 2005 to Visual Studio 2019, as well as standalone Windows SDK releases 6.0 to 10. 64-bit PostgreSQL builds are supported with Microsoft Windows SDK version 6.0a to 10 or Visual Studio 2008 and above. Compilation is supported down to Windows XP and Windows Server 2003 when building with Visual Studio 2005 to Visual Studio 2013. Building with Visual Studio 2015 is supported down to Windows Vista and Windows Server 2008. Building with Visual Studio 2017 and Visual Studio 2019 is supported down to Windows 7 SP1 and Windows Server 2008 R2 SP1.

The tools for building using Visual C++ or Platform SDK are in the `src/tools/msvc` directory. When building, make sure there are no tools from MinGW or Cygwin present in your system PATH. Also, make sure you have all the required Visual C++ tools available in the PATH. In Visual Studio, start the Visual Studio Command Prompt. If you wish to build a 64-bit version, you must use the 64-bit version of the

command, and vice versa. In the Microsoft Windows SDK, start the CMD shell listed under the SDK on the Start Menu. In recent SDK versions you can change the targeted CPU architecture, build type, and target OS by using the `setenv` command, e.g., `setenv /x86 /release /xp` to target Windows XP or later with a 32-bit release build. See `/?` for other options to `setenv`. All commands should be run from the `src\tools\msvc` directory.

Before you build, you may need to edit the file `config.pl` to reflect any configuration options you want to change, or the paths to any third party libraries to use. The complete configuration is determined by first reading and parsing the file `config_default.pl`, and then apply any changes from `config.pl`. For example, to specify the location of your Python installation, put the following in `config.pl`:

```
$config->{python} = 'c:\python26';
```

You only need to specify those parameters that are different from what's in `config_default.pl`.

If you need to set any other environment variables, create a file called `buildenv.pl` and put the required commands there. For example, to add the path for bison when it's not in the PATH, create a file containing:

```
$ENV{PATH}=$ENV{PATH} . ';c:\some\where\bison\bin';
```

16.1.1. Requirements

The following additional products are required to build PostgreSQL. Use the `config.pl` file to specify which directories the libraries are available in.

Microsoft Windows SDK

If your build environment doesn't ship with a supported version of the Microsoft Windows SDK it is recommended that you upgrade to the latest version (currently version 10), available for download from https://www.microsoft.com/download.

You must always include the Windows Headers and Libraries part of the SDK. If you install a Windows SDK including the Visual C++ Compilers, you don't need Visual Studio to build. Note that as of Version 8.0a the Windows SDK no longer ships with a complete command-line build environment.

ActiveState Perl

ActiveState Perl is required to run the build generation scripts. MinGW or Cygwin Perl will not work. It must also be present in the PATH. Binaries can be downloaded from http://www.activestate.com (Note: version 5.8.3 or later is required, the free Standard Distribution is sufficient).

The following additional products are not required to get started, but are required to build the complete package. Use the `config.pl` file to specify which directories the libraries are available in.

ActiveState TCL

Required for building PL/TCL (Note: version 8.4 is required, the free Standard Distribution is sufficient).

Bison and Flex

Bison and Flex are required to build from Git, but not required when building from a release file. Only Bison 1.875 or versions 2.2 and later will work. Flex must be version 2.5.31 or later.

Both Bison and Flex are included in the msys tool suite, available from http://www.mingw.org/wiki/MSYS as part of the MinGW compiler suite.

You will need to add the directory containing `flex.exe` and `bison.exe` to the PATH environment variable in `buildenv.pl` unless they are already in PATH. In the case of MinGW, the directory is the `\msys\1.0\bin` subdirectory of your MinGW installation directory.

Note: The Bison distribution from GnuWin32 appears to have a bug that causes Bison to malfunction when installed in a directory with spaces in the name, such as the default location on English installations `C:\Program Files\GnuWin32`. Consider installing into `C:\GnuWin32` or use the NTFS short name path to GnuWin32 in your PATH environment setting (e.g., `C:\PROGRA~1\GnuWin32`).

Note: The obsolete "winflex" binaries distributed on the PostgreSQL FTP site and referenced in older documentation will fail with "flex: fatal internal error, exec failed" on 64-bit Windows hosts. Use flex from msys instead.

Diff

Diff is required to run the regression tests, and can be downloaded from http://gnuwin32.sourceforge.net.

Gettext

Gettext is required to build with NLS support, and can be downloaded from http://gnuwin32.sourceforge.net. Note that binaries, dependencies and developer files are all needed.

MIT Kerberos

Required for GSSAPI authentication support. MIT Kerberos can be downloaded from https://web.mit.edu/Kerberos/dist/index.html.

libxml2 and libxslt

Required for XML support. Binaries can be downloaded from https://zlatkovic.com/pub/libxml or source from http://xmlsoft.org. Note that libxml2 requires iconv, which is available from the same download location.

openssl

Required for SSL support. Binaries can be downloaded from http://www.slproweb.com/products/Win32OpenSSL.html or source from http://www.openssl.org.

ossp-uuid

Required for UUID-OSSP support (contrib only). Source can be downloaded from http://www.ossp.org/pkg/lib/uuid/.

Python

Required for building PL/Python. Binaries can be downloaded from http://www.python.org.

zlib

Required for compression support in pg_dump and pg_restore. Binaries can be downloaded from https://www.zlib.net.

16.1.2. Special Considerations for 64-bit Windows

PostgreSQL will only build for the x64 architecture on 64-bit Windows, there is no support for Itanium processors.

Mixing 32- and 64-bit versions in the same build tree is not supported. The build system will automatically detect if it's running in a 32- or 64-bit environment, and build PostgreSQL accordingly. For this reason, it is important to start the correct command prompt before building.

To use a server-side third party library such as python or openssl, this library *must* also be 64-bit. There is no support for loading a 32-bit library in a 64-bit server. Several of the third party libraries that PostgreSQL supports may only be available in 32-bit versions, in which case they cannot be used with 64-bit PostgreSQL.

16.1.3. Building

To build all of PostgreSQL in release configuration (the default), run the command:

```
build
```

To build all of PostgreSQL in debug configuration, run the command:

```
build DEBUG
```

To build just a single project, for example psql, run the commands:

```
build psql
build DEBUG psql
```

To change the default build configuration to debug, put the following in the `buildenv.pl` file:

```
$ENV{CONFIG}="Debug";
```

It is also possible to build from inside the Visual Studio GUI. In this case, you need to run:

```
perl mkvcbuild.pl
```

from the command prompt, and then open the generated `pgsql.sln` (in the root directory of the source tree) in Visual Studio.

16.1.4. Cleaning and Installing

Most of the time, the automatic dependency tracking in Visual Studio will handle changed files. But if there have been large changes, you may need to clean the installation. To do this, simply run the `clean.bat` command, which will automatically clean out all generated files. You can also run it with the `dist` parameter, in which case it will behave like **`make distclean`** and remove the flex/bison output files as well.

By default, all files are written into a subdirectory of the `debug` or `release` directories. To install these files using the standard layout, and also generate the files required to initialize and use the database, run the command:

```
install c:\destination\directory
```

If you want to install only the client applications and interface libraries, then you can use these commands:

```
install c:\destination\directory client
```

16.1.5. Running the Regression Tests

To run the regression tests, make sure you have completed the build of all required parts first. Also, make sure that the DLLs required to load all parts of the system (such as the Perl and Python DLLs for the procedural languages) are present in the system path. If they are not, set it through the `buildenv.pl` file. To run the tests, run one of the following commands from the `src\tools\msvc` directory:

```
vcregress check
vcregress installcheck
vcregress plcheck
vcregress contribcheck
vcregress modulescheck
vcregress ecpgcheck
vcregress isolationcheck
vcregress bincheck
vcregress upgradecheck
```

To change the schedule used (default is parallel), append it to the command line like:

```
vcregress check serial
```

For more information about the regression tests, see Chapter 30.

Running the regression tests on client programs, with "vcregress bincheck", requires an additional Perl module to be installed:

IPC::Run

> As of this writing, `IPC::Run` is not included in the ActiveState Perl installation, nor in the ActiveState Perl Package Manager (PPM) library. To install, download the `IPC-Run-<version>.tar.gz` source archive from CPAN, at https://metacpan.org/release/IPC-Run/, and uncompress. Edit the

`buildenv.pl` file, and add a PERL5LIB variable to point to the `lib` subdirectory from the extracted archive. For example:

```
$ENV{PERL5LIB}=$ENV{PERL5LIB} . ';c:\IPC-Run-0.94\lib';
```

16.1.6. Building the Documentation

Building the PostgreSQL documentation in HTML format requires several tools and files. Create a root directory for all these files, and store them in the subdirectories in the list below.

OpenJade 1.3.1-2

Download from http://sourceforge.net/projects/openjade/files/openjade/1.3.1/openjade-1_3_1-2-bin.zip/download and uncompress in the subdirectory `openjade-1.3.1`.

DocBook DTD 4.2

Download from http://www.oasis-open.org/docbook/sgml/4.2/docbook-4.2.zip and uncompress in the subdirectory `docbook`.

DocBook DSSSL 1.79

Download from http://sourceforge.net/projects/docbook/files/docbook-dsssl/1.79/docbook-dsssl-1.79.zip/download and uncompress in the subdirectory `docbook-dsssl-1.79`.

ISO character entities

Download from http://www.oasis-open.org/cover/ISOEnts.zip and uncompress in the subdirectory `docbook`.

Edit the `buildenv.pl` file, and add a variable for the location of the root directory, for example:

```
$ENV{DOCROOT}='c:\docbook';
```

To build the documentation, run the command `builddoc.bat`. Note that this will actually run the build twice, in order to generate the indexes. The generated HTML files will be in `doc\src\sgml`.

16.2. Building libpq with Visual C++ or Borland C++

Using Visual C++ 7.1-9.0 or Borland C++ to build libpq is only recommended if you need a version with different debug/release flags, or if you need a static library to link into an application. For normal use the MinGW or Visual Studio or Windows SDK method is recommended.

To build the libpq client library using Visual Studio 7.1 or later, change into the `src` directory and type the command:

```
nmake /f win32.mak
```

To build a 64-bit version of the libpq client library using Visual Studio 8.0 or later, change into the `src` directory and type in the command:

```
nmake /f win32.mak CPU=AMD64
```

See the `win32.mak` file for further details about supported variables.

To build the libpq client library using Borland C++, change into the `src` directory and type the command:

```
make -N -DCFG=Release /f bcc32.mak
```

16.2.1. Generated Files

The following files will be built:

`interfaces\libpq\Release\libpq.dll`

The dynamically linkable frontend library

`interfaces\libpq\Release\libpqdll.lib`

Import library to link your programs to `libpq.dll`

`interfaces\libpq\Release\libpq.lib`

Static version of the frontend library

Normally you do not need to install any of the client files. You should place the `libpq.dll` file in the same directory as your applications executable file. Do not install `libpq.dll` into your `Windows`, `System` or `System32` directory unless absolutely necessary. If this file is installed using a setup program, then it should be installed with version checking using the `VERSIONINFO` resource included in the file, to ensure that a newer version of the library is not overwritten.

If you are planning to do development using libpq on this machine, you will have to add the `src\include` and `src\interfaces\libpq` subdirectories of the source tree to the include path in your compiler's settings.

To use the library, you must add the `libpqdll.lib` file to your project. (In Visual C++, just right-click on the project and choose to add it.)

Chapter 17. Server Setup and Operation

This chapter discusses how to set up and run the database server and its interactions with the operating system.

17.1. The PostgreSQL User Account

As with any server daemon that is accessible to the outside world, it is advisable to run PostgreSQL under a separate user account. This user account should only own the data that is managed by the server, and should not be shared with other daemons. (For example, using the user `nobody` is a bad idea.) It is not advisable to install executables owned by this user because compromised systems could then modify their own binaries.

To add a Unix user account to your system, look for a command `useradd` or `adduser`. The user name postgres is often used, and is assumed throughout this book, but you can use another name if you like.

17.2. Creating a Database Cluster

Before you can do anything, you must initialize a database storage area on disk. We call this a *database cluster*. (The SQL standard uses the term catalog cluster.) A database cluster is a collection of databases that is managed by a single instance of a running database server. After initialization, a database cluster will contain a database named `postgres`, which is meant as a default database for use by utilities, users and third party applications. The database server itself does not require the `postgres` database to exist, but many external utility programs assume it exists. Another database created within each cluster during initialization is called `template1`. As the name suggests, this will be used as a template for subsequently created databases; it should not be used for actual work. (See Chapter 21 for information about creating new databases within a cluster.)

In file system terms, a database cluster is a single directory under which all data will be stored. We call this the *data directory* or *data area*. It is completely up to you where you choose to store your data. There is no default, although locations such as `/usr/local/pgsql/data` or `/var/lib/pgsql/data` are popular. To initialize a database cluster, use the command initdb, which is installed with PostgreSQL. The desired file system location of your database cluster is indicated by the `-D` option, for example:

```
$ initdb -D /usr/local/pgsql/data
```

Note that you must execute this command while logged into the PostgreSQL user account, which is described in the previous section.

> **Tip:** As an alternative to the `-D` option, you can set the environment variable `PGDATA`.

Alternatively, you can run `initdb` via the pg_ctl program like so:

```
$ pg_ctl -D /usr/local/pgsql/data initdb
```

This may be more intuitive if you are using pg_ctl for starting and stopping the server (see Section 17.3), so that pg_ctl would be the sole command you use for managing the database server instance.

initdb will attempt to create the directory you specify if it does not already exist. Of course, this will fail if initdb does not have permissions to write in the parent directory. It's generally recommendable that the PostgreSQL user own not just the data directory but its parent directory as well, so that this should not be a problem. If the desired parent directory doesn't exist either, you will need to create it first, using root privileges if the grandparent directory isn't writable. So the process might look like this:

```
root# mkdir /usr/local/pgsql
root# chown postgres /usr/local/pgsql
root# su postgres
postgres$ initdb -D /usr/local/pgsql/data
```

initdb will refuse to run if the data directory exists and already contains files; this is to prevent accidentally overwriting an existing installation.

Because the data directory contains all the data stored in the database, it is essential that it be secured from unauthorized access. initdb therefore revokes access permissions from everyone but the PostgreSQL user.

However, while the directory contents are secure, the default client authentication setup allows any local user to connect to the database and even become the database superuser. If you do not trust other local users, we recommend you use one of initdb's -W, --pwprompt or --pwfile options to assign a password to the database superuser. Also, specify -A md5 or -A password so that the default trust authentication mode is not used; or modify the generated pg_hba.conf file after running initdb, but *before* you start the server for the first time. (Other reasonable approaches include using peer authentication or file system permissions to restrict connections. See Chapter 19 for more information.)

initdb also initializes the default locale for the database cluster. Normally, it will just take the locale settings in the environment and apply them to the initialized database. It is possible to specify a different locale for the database; more information about that can be found in Section 22.1. The default sort order used within the particular database cluster is set by initdb, and while you can create new databases using different sort order, the order used in the template databases that initdb creates cannot be changed without dropping and recreating them. There is also a performance impact for using locales other than C or POSIX. Therefore, it is important to make this choice correctly the first time.

initdb also sets the default character set encoding for the database cluster. Normally this should be chosen to match the locale setting. For details see Section 22.3.

Non-C and and non-POSIX locales rely on the operating system's collation library for character set ordering. This controls the ordering of keys stored in indexes. For this reason, a cluster cannot switch to an incompatible collation library version, either through snapshot restore, binary streaming replication, a different operating system, or an operating system upgrade.

17.2.1. Use of Secondary File Systems

Many installations create their database clusters on file systems (volumes) other than the machine's "root" volume. If you choose to do this, it is not advisable to try to use the secondary volume's topmost directory (mount point) as the data directory. Best practice is to create a directory within the mount-point directory

that is owned by the PostgreSQL user, and then create the data directory within that. This avoids permissions problems, particularly for operations such as pg_upgrade, and it also ensures clean failures if the secondary volume is taken offline.

17.2.2. Use of Network File Systems

Many installations create their database clusters on network file systems. Sometimes this is done via NFS, or by using a Network Attached Storage (NAS) device that uses NFS internally. PostgreSQL does nothing special for NFS file systems, meaning it assumes NFS behaves exactly like locally-connected drives. If the client or server NFS implementation does not provide standard file system semantics, this can cause reliability problems (see http://www.time-travellers.org/shane/papers/NFS_considered_harmful.html). Specifically, delayed (asynchronous) writes to the NFS server can cause data corruption problems. If possible, mount the NFS file system synchronously (without caching) to avoid this hazard. Also, soft-mounting the NFS file system is not recommended.

Storage Area Networks (SAN) typically use communication protocols other than NFS, and may or may not be subject to hazards of this sort. It's advisable to consult the vendor's documentation concerning data consistency guarantees. PostgreSQL cannot be more reliable than the file system it's using.

17.3. Starting the Database Server

Before anyone can access the database, you must start the database server. The database server program is called `postgres`. The `postgres` program must know where to find the data it is supposed to use. This is done with the `-D` option. Thus, the simplest way to start the server is:

```
$ postgres -D /usr/local/pgsql/data
```

which will leave the server running in the foreground. This must be done while logged into the PostgreSQL user account. Without `-D`, the server will try to use the data directory named by the environment variable `PGDATA`. If that variable is not provided either, it will fail.

Normally it is better to start `postgres` in the background. For this, use the usual Unix shell syntax:

```
$ postgres -D /usr/local/pgsql/data >logfile 2>&1 &
```

It is important to store the server's stdout and stderr output somewhere, as shown above. It will help for auditing purposes and to diagnose problems. (See Section 23.3 for a more thorough discussion of log file handling.)

The `postgres` program also takes a number of other command-line options. For more information, see the postgres reference page and Chapter 18 below.

This shell syntax can get tedious quickly. Therefore the wrapper program pg_ctl is provided to simplify some tasks. For example:

```
pg_ctl start -l logfile
```

will start the server in the background and put the output into the named log file. The `-D` option has the same meaning here as for `postgres`. `pg_ctl` is also capable of stopping the server.

Normally, you will want to start the database server when the computer boots. Autostart scripts are operating-system-specific. There are a few distributed with PostgreSQL in the `contrib/start-scripts` directory. Installing one will require root privileges.

Different systems have different conventions for starting up daemons at boot time. Many systems have a file `/etc/rc.local` or `/etc/rc.d/rc.local`. Others use `init.d` or `rc.d` directories. Whatever you do, the server must be run by the PostgreSQL user account *and not by root* or any other user. Therefore you probably should form your commands using `su postgres -c '...'`. For example:

```
su postgres -c 'pg_ctl start -D /usr/local/pgsql/data -l serverlog'
```

Here are a few more operating-system-specific suggestions. (In each case be sure to use the proper installation directory and user name where we show generic values.)

- For FreeBSD, look at the file `contrib/start-scripts/freebsd` in the PostgreSQL source distribution.
- On OpenBSD, add the following lines to the file `/etc/rc.local`:

  ```
  if [ -x /usr/local/pgsql/bin/pg_ctl -a -x /usr/local/pgsql/bin/postgres ]; then
      su -l postgres -c '/usr/local/pgsql/bin/pg_ctl start -s -l /var/postgresql/log -D /usr
      echo -n ' postgresql'
  fi
  ```

- On Linux systems either add

  ```
  /usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data
  ```

  to `/etc/rc.d/rc.local` or `/etc/rc.local` or look at the file `contrib/start-scripts/linux` in the PostgreSQL source distribution.
- On NetBSD, use either the FreeBSD or Linux start scripts, depending on preference.
- On Solaris, create a file called `/etc/init.d/postgresql` that contains the following line:

  ```
  su - postgres -c "/usr/local/pgsql/bin/pg_ctl start -l logfile -D /usr/local/pgsql/data"
  ```

  Then, create a symbolic link to it in `/etc/rc3.d` as `S99postgresql`.

While the server is running, its PID is stored in the file `postmaster.pid` in the data directory. This is used to prevent multiple server instances from running in the same data directory and can also be used for shutting down the server.

17.3.1. Server Start-up Failures

There are several common reasons the server might fail to start. Check the server's log file, or start it by hand (without redirecting standard output or standard error) and see what error messages appear. Below we explain some of the most common error messages in more detail.

```
LOG:  could not bind IPv4 socket: Address already in use
HINT:  Is another postmaster already running on port 5432? If not, wait a few seconds and r
FATAL:  could not create TCP/IP listen socket
```

This usually means just what it suggests: you tried to start another server on the same port where one is already running. However, if the kernel error message is not `Address already in use` or some variant of that, there might be a different problem. For example, trying to start a server on a reserved port number might draw something like:

```
$ postgres -p 666
LOG:  could not bind IPv4 socket: Permission denied
HINT:  Is another postmaster already running on port 666? If not, wait a few seconds and re
FATAL:  could not create TCP/IP listen socket
```

A message like:

```
FATAL:  could not create shared memory segment: Invalid argument
DETAIL:  Failed system call was shmget(key=5440001, size=4011376640, 03600).
```

probably means your kernel's limit on the size of shared memory is smaller than the work area PostgreSQL is trying to create (4011376640 bytes in this example). Or it could mean that you do not have System-V-style shared memory support configured into your kernel at all. As a temporary workaround, you can try starting the server with a smaller-than-normal number of buffers (shared_buffers). You will eventually want to reconfigure your kernel to increase the allowed shared memory size. You might also see this message when trying to start multiple servers on the same machine, if their total space requested exceeds the kernel limit.

An error like:

```
FATAL:  could not create semaphores: No space left on device
DETAIL:  Failed system call was semget(5440126, 17, 03600).
```

does *not* mean you've run out of disk space. It means your kernel's limit on the number of System V semaphores is smaller than the number PostgreSQL wants to create. As above, you might be able to work around the problem by starting the server with a reduced number of allowed connections (max_connections), but you'll eventually want to increase the kernel limit.

If you get an "illegal system call" error, it is likely that shared memory or semaphores are not supported in your kernel at all. In that case your only option is to reconfigure the kernel to enable these features.

Details about configuring System V IPC facilities are given in Section 17.4.1.

17.3.2. Client Connection Problems

Although the error conditions possible on the client side are quite varied and application-dependent, a few of them might be directly related to how the server was started. Conditions other than those shown below should be documented with the respective client application.

```
psql: could not connect to server: Connection refused
        Is the server running on host "server.joe.com" and accepting
        TCP/IP connections on port 5432?
```

This is the generic "I couldn't find a server to talk to" failure. It looks like the above when TCP/IP communication is attempted. A common mistake is to forget to configure the server to allow TCP/IP connections.

Alternatively, you'll get this when attempting Unix-domain socket communication to a local server:

```
psql: could not connect to server: No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

The last line is useful in verifying that the client is trying to connect to the right place. If there is in fact no server running there, the kernel error message will typically be either `Connection refused` or `No such file or directory`, as illustrated. (It is important to realize that `Connection refused` in this context does *not* mean that the server got your connection request and rejected it. That case will produce a different message, as shown in Section 19.4.) Other error messages such as `Connection timed out` might indicate more fundamental problems, like lack of network connectivity.

17.4. Managing Kernel Resources

PostgreSQL can sometimes exhaust various operating system resource limits, especially when multiple copies of the server are running on the same system, or in very large installations. This section explains the kernel resources used by PostgreSQL and the steps you can take to resolve problems related to kernel resource consumption.

17.4.1. Shared Memory and Semaphores

Shared memory and semaphores are collectively referred to as "System V IPC" (together with message queues, which are not relevant for PostgreSQL). Except on Windows, where PostgreSQL provides its own replacement implementation of these facilities, these facilities are required in order to run PostgreSQL.

The complete lack of these facilities is usually manifested by an Illegal system call error upon server start. In that case there is no alternative but to reconfigure your kernel. PostgreSQL won't work without them. This situation is rare, however, among modern operating systems.

When PostgreSQL exceeds one of the various hard IPC limits, the server will refuse to start and should leave an instructive error message describing the problem and what to do about it. (See also Section 17.3.1.) The relevant kernel parameters are named consistently across different systems; Table 17-1 gives an overview. The methods to set them, however, vary. Suggestions for some platforms are given below.

> **Note:** Prior to PostgreSQL 9.3, the amount of System V shared memory required to start the server was much larger. If you are running an older version of the server, please consult the documentation for your server version.

Table 17-1. System V IPC Parameters

Name	Description	Reasonable values
SHMMAX	Maximum size of shared memory segment (bytes)	at least 1kB (more if running many copies of the server)
SHMMIN	Minimum size of shared memory segment (bytes)	1
SHMALL	Total amount of shared memory available (bytes or pages)	if bytes, same as SHMMAX; if pages, ceil(SHMMAX/PAGE_SIZE)
SHMSEG	Maximum number of shared memory segments per process	only 1 segment is needed, but the default is much higher
SHMMNI	Maximum number of shared memory segments system-wide	like SHMSEG plus room for other applications
SEMMNI	Maximum number of semaphore identifiers (i.e., sets)	at least ceil((max_connections + autovacuum_max_workers + max_worker_processes + 5) / 16)
SEMMNS	Maximum number of semaphores system-wide	ceil((max_connections + autovacuum_max_workers + max_worker_processes + 5) / 16) * 17 plus room for other applications
SEMMSL	Maximum number of semaphores per set	at least 17
SEMMAP	Number of entries in semaphore map	see text
SEMVMX	Maximum value of semaphore	at least 1000 (The default is often 32767; do not change unless necessary)

PostgreSQL requires a few bytes of System V shared memory (typically 48 bytes, on 64-bit platforms) for each copy of the server. On most modern operating systems, this amount can easily be allocated. However, if you are running many copies of the server, or if other applications are also using System V shared memory, it may be necessary to increase SHMMAX, the maximum size in bytes of a shared memory segment, or SHMALL, the total amount of System V shared memory system-wide. Note that SHMALL is measured in pages rather than bytes on many systems.

Less likely to cause problems is the minimum size for shared memory segments (SHMMIN), which should be at most approximately 32 bytes for PostgreSQL (it is usually just 1). The maximum number of segments system-wide (SHMMNI) or per-process (SHMSEG) are unlikely to cause a problem unless your system has them set to zero.

PostgreSQL uses one semaphore per allowed connection (max_connections), allowed autovacuum worker process (autovacuum_max_workers) and allowed background process (max_worker_processes), in sets of 16. Each such set will also contain a 17th semaphore which contains a “magic number”, to detect

collision with semaphore sets used by other applications. The maximum number of semaphores in the system is set by `SEMMNS`, which consequently must be at least as high as `max_connections` plus `autovacuum_max_workers` plus `max_worker_processes`, plus one extra for each 16 allowed connections plus workers (see the formula in Table 17-1). The parameter `SEMMNI` determines the limit on the number of semaphore sets that can exist on the system at one time. Hence this parameter must be at least `ceil((max_connections + autovacuum_max_workers + max_worker_processes + 5) / 16)`. Lowering the number of allowed connections is a temporary workaround for failures, which are usually confusingly worded "No space left on device", from the function `semget`.

In some cases it might also be necessary to increase `SEMMAP` to be at least on the order of `SEMMNS`. If the system has this parameter (many do not), it defines the size of the semaphore resource map, in which each contiguous block of available semaphores needs an entry. When a semaphore set is freed it is either added to an existing entry that is adjacent to the freed block or it is registered under a new map entry. If the map is full, the freed semaphores get lost (until reboot). Fragmentation of the semaphore space could over time lead to fewer available semaphores than there should be.

The `SEMMSL` parameter, which determines how many semaphores can be in a set, must be at least 17 for PostgreSQL.

Various other settings related to "semaphore undo", such as `SEMMNU` and `SEMUME`, do not affect PostgreSQL.

AIX

At least as of version 5.1, it should not be necessary to do any special configuration for such parameters as `SHMMAX`, as it appears this is configured to allow all memory to be used as shared memory. That is the sort of configuration commonly used for other databases such as DB/2.

It might, however, be necessary to modify the global `ulimit` information in `/etc/security/limits`, as the default hard limits for file sizes (`fsize`) and numbers of files (`nofiles`) might be too low.

FreeBSD

The default IPC settings can be changed using the `sysctl` or `loader` interfaces. The following parameters can be set using `sysctl`:

```
# sysctl kern.ipc.shmall=32768
# sysctl kern.ipc.shmmax=134217728
```

To make these settings persist over reboots, modify `/etc/sysctl.conf`.

These semaphore-related settings are read-only as far as `sysctl` is concerned, but can be set in `/boot/loader.conf`:

```
kern.ipc.semmni=256
kern.ipc.semmns=512
```

After modifying that file, a reboot is required for the new settings to take effect.

You might also want to configure your kernel to lock shared memory into RAM and prevent it from being paged out to swap. This can be accomplished using the `sysctl` setting `kern.ipc.shm_use_phys`.

If running in FreeBSD jails by enabling sysctl's `security.jail.sysvipc_allowed`, postmasters running in different jails should be run by different operating system users. This improves security because it prevents non-root users from interfering with shared memory or semaphores in different jails, and it allows the PostgreSQL IPC cleanup code to function properly. (In FreeBSD 6.0 and later

the IPC cleanup code does not properly detect processes in other jails, preventing the running of postmasters on the same port in different jails.)

FreeBSD versions before 4.0 work like old OpenBSD (see below).

NetBSD

In NetBSD 5.0 and later, IPC parameters can be adjusted using `sysctl`, for example:

```
# sysctl -w kern.ipc.semmni=100
```

To make these settings persist over reboots, modify `/etc/sysctl.conf`.

You will usually want to increase `kern.ipc.semmni` and `kern.ipc.semmns`, as NetBSD's default settings for these are uncomfortably small.

You might also want to configure your kernel to lock shared memory into RAM and prevent it from being paged out to swap. This can be accomplished using the `sysctl` setting `kern.ipc.shm_use_phys`.

NetBSD versions before 5.0 work like old OpenBSD (see below), except that kernel parameters should be set with the keyword `options` not `option`.

OpenBSD

In OpenBSD 3.3 and later, IPC parameters can be adjusted using `sysctl`, for example:

```
# sysctl kern.seminfo.semmni=100
```

To make these settings persist over reboots, modify `/etc/sysctl.conf`.

You will usually want to increase `kern.seminfo.semmni` and `kern.seminfo.semmns`, as OpenBSD's default settings for these are uncomfortably small.

In older OpenBSD versions, you will need to build a custom kernel to change the IPC parameters. Make sure that the options `SYSVSHM` and `SYSVSEM` are enabled, too. (They are by default.) The following shows an example of how to set the various parameters in the kernel configuration file:

```
option        SYSVSHM
option        SHMMAXPGS=4096
option        SHMSEG=256

option        SYSVSEM
option        SEMMNI=256
option        SEMMNS=512
option        SEMMNU=256
```

HP-UX

The default settings tend to suffice for normal installations. On HP-UX 10, the factory default for `SEMMNS` is 128, which might be too low for larger database sites.

IPC parameters can be set in the System Administration Manager (SAM) under Kernel Configuration⟶Configurable Parameters. Choose Create A New Kernel when you're done.

Linux

The default maximum segment size is 32 MB, and the default maximum total size is 2097152 pages. A page is almost always 4096 bytes except in unusual kernel configurations with "huge pages" (use `getconf PAGE_SIZE` to verify).

The shared memory size settings can be changed via the `sysctl` interface. For example, to allow 16 GB:

```
$ sysctl -w kernel.shmmax=17179869184
$ sysctl -w kernel.shmall=4194304
```

In addition these settings can be preserved between reboots in the file `/etc/sysctl.conf`. Doing that is highly recommended.

Ancient distributions might not have the `sysctl` program, but equivalent changes can be made by manipulating the `/proc` file system:

```
$ echo 17179869184 >/proc/sys/kernel/shmmax
$ echo 4194304 >/proc/sys/kernel/shmall
```

The remaining defaults are quite generously sized, and usually do not require changes.

OS X

The recommended method for configuring shared memory in OS X is to create a file named `/etc/sysctl.conf`, containing variable assignments such as:

```
kern.sysv.shmmax=4194304
kern.sysv.shmmin=1
kern.sysv.shmmni=32
kern.sysv.shmseg=8
kern.sysv.shmall=1024
```

Note that in some OS X versions, *all five* shared-memory parameters must be set in `/etc/sysctl.conf`, else the values will be ignored.

Beware that recent releases of OS X ignore attempts to set `SHMMAX` to a value that isn't an exact multiple of 4096.

`SHMALL` is measured in 4 kB pages on this platform.

In older OS X versions, you will need to reboot to have changes in the shared memory parameters take effect. As of 10.5 it is possible to change all but `SHMMNI` on the fly, using sysctl. But it's still best to set up your preferred values via `/etc/sysctl.conf`, so that the values will be kept across reboots.

The file `/etc/sysctl.conf` is only honored in OS X 10.3.9 and later. If you are running a previous 10.3.x release, you must edit the file `/etc/rc` and change the values in the following commands:

```
sysctl -w kern.sysv.shmmax
sysctl -w kern.sysv.shmmin
sysctl -w kern.sysv.shmmni
sysctl -w kern.sysv.shmseg
sysctl -w kern.sysv.shmall
```

Note that `/etc/rc` is usually overwritten by OS X system updates, so you should expect to have to redo these edits after each update.

In OS X 10.2 and earlier, instead edit these commands in the file `/System/Library/StartupItems/SystemTuning/SystemTuning`.

SCO OpenServer

In the default configuration, only 512 kB of shared memory per segment is allowed. To increase the setting, first change to the directory `/etc/conf/cf.d`. To display the current value of `SHMMAX`, run:

```
./configure -y SHMMAX
```

To set a new value for SHMMAX, run:

```
./configure SHMMAX=value
```

where *value* is the new value you want to use (in bytes). After setting SHMMAX, rebuild the kernel:

```
./link_unix
```

and reboot.

Solaris 2.6 to 2.9 (Solaris 6 to Solaris 9)

The relevant settings can be changed in /etc/system, for example:

```
set shmsys:shminfo_shmmax=0x2000000
set shmsys:shminfo_shmmin=1
set shmsys:shminfo_shmmni=256
set shmsys:shminfo_shmseg=256

set semsys:seminfo_semmap=256
set semsys:seminfo_semmni=512
set semsys:seminfo_semmns=512
set semsys:seminfo_semmsl=32
```

You need to reboot for the changes to take effect. See also http://sunsite.uakom.sk/sunworldonline/swol-09-1997/swol-09-insidesolaris.html for information on shared memory under older versions of Solaris.

Solaris 2.10 (Solaris 10) and later
OpenSolaris

In Solaris 10 and later, and OpenSolaris, the default shared memory and semaphore settings are good enough for most PostgreSQL applications. Solaris now defaults to a SHMMAX of one-quarter of system RAM. To further adjust this setting, use a project setting associated with the postgres user. For example, run the following as root:

```
projadd -c "PostgreSQL DB User" -K "project.max-shm-memory=(privileged,8GB,deny)" -U pos
```

This command adds the user.postgres project and sets the shared memory maximum for the postgres user to 8GB, and takes effect the next time that user logs in, or when you restart PostgreSQL (not reload). The above assumes that PostgreSQL is run by the postgres user in the postgres group. No server reboot is required.

Other recommended kernel setting changes for database servers which will have a large number of connections are:

```
project.max-shm-ids=(priv,32768,deny)
project.max-sem-ids=(priv,4096,deny)
project.max-msg-ids=(priv,4096,deny)
```

Additionally, if you are running PostgreSQL inside a zone, you may need to raise the zone resource usage limits as well. See "Chapter2: Projects and Tasks" in the *System Administrator's Guide* for more information on projects and prctl.

UnixWare

On UnixWare 7, the maximum size for shared memory segments is 512 kB in the default configuration. To display the current value of SHMMAX, run:

```
/etc/conf/bin/idtune -g SHMMAX
```

which displays the current, default, minimum, and maximum values. To set a new value for SHMMAX, run:

```
/etc/conf/bin/idtune SHMMAX value
```
where *value* is the new value you want to use (in bytes). After setting SHMMAX, rebuild the kernel:

```
/etc/conf/bin/idbuild -B
```
and reboot.

17.4.2. systemd RemoveIPC

If systemd is in use, some care must be taken that IPC resources (shared memory and semaphores) are not prematurely removed by the operating system. This is especially of concern when installing PostgreSQL from source. Users of distribution packages of PostgreSQL are less likely to be affected, as the postgres user is then normally created as a system user.

The setting RemoveIPC in logind.conf controls whether IPC objects are removed when a user fully logs out. System users are exempt. This setting defaults to on in stock systemd, but some operating system distributions default it to off.

A typical observed effect when this setting is on is that the semaphore objects used by a PostgreSQL server are removed at apparently random times, leading to the server crashing with log messages like

```
LOG: semctl(1234567890, 0, IPC_RMID, ...) failed: Invalid argument
```

Different types of IPC objects (shared memory vs. semaphores, System V vs. POSIX) are treated slightly differently by systemd, so one might observe that some IPC resources are not removed in the same way as others. But it is not advisable to rely on these subtle differences.

A "user logging out" might happen as part of a maintenance job or manually when an administrator logs in as the postgres user or something similar, so it is hard to prevent in general.

What is a "system user" is determined at systemd compile time from the SYS_UID_MAX setting in /etc/login.defs.

Packaging and deployment scripts should be careful to create the postgres user as a system user by using useradd -r, adduser --system, or equivalent.

Alternatively, if the user account was created incorrectly or cannot be changed, it is recommended to set

```
RemoveIPC=no
```

in /etc/systemd/logind.conf or another appropriate configuration file.

> **Caution**
>
> At least one of these two things has to be ensured, or the PostgreSQL server will be very unreliable.

17.4.3. Resource Limits

Unix-like operating systems enforce various kinds of resource limits that might interfere with the operation of your PostgreSQL server. Of particular importance are limits on the number of processes per user, the number of open files per process, and the amount of memory available to each process. Each

of these have a "hard" and a "soft" limit. The soft limit is what actually counts but it can be changed by the user up to the hard limit. The hard limit can only be changed by the root user. The system call `setrlimit` is responsible for setting these parameters. The shell's built-in command `ulimit` (Bourne shells) or `limit` (csh) is used to control the resource limits from the command line. On BSD-derived systems the file `/etc/login.conf` controls the various resource limits set during login. See the operating system documentation for details. The relevant parameters are `maxproc`, `openfiles`, and `datasize`. For example:

```
default:\
...
        :datasize-cur=256M:\
        :maxproc-cur=256:\
        :openfiles-cur=256:\
...
```

(`-cur` is the soft limit. Append `-max` to set the hard limit.)

Kernels can also have system-wide limits on some resources.

- On Linux `/proc/sys/fs/file-max` determines the maximum number of open files that the kernel will support. It can be changed by writing a different number into the file or by adding an assignment in `/etc/sysctl.conf`. The maximum limit of files per process is fixed at the time the kernel is compiled; see `/usr/src/linux/Documentation/proc.txt` for more information.

The PostgreSQL server uses one process per connection so you should provide for at least as many processes as allowed connections, in addition to what you need for the rest of your system. This is usually not a problem but if you run several servers on one machine things might get tight.

The factory default limit on open files is often set to "socially friendly" values that allow many users to coexist on a machine without using an inappropriate fraction of the system resources. If you run many servers on a machine this is perhaps what you want, but on dedicated servers you might want to raise this limit.

On the other side of the coin, some systems allow individual processes to open large numbers of files; if more than a few processes do so then the system-wide limit can easily be exceeded. If you find this happening, and you do not want to alter the system-wide limit, you can set PostgreSQL's max_files_per_process configuration parameter to limit the consumption of open files.

17.4.4. Linux Memory Overcommit

In Linux 2.4 and later, the default virtual memory behavior is not optimal for PostgreSQL. Because of the way that the kernel implements memory overcommit, the kernel might terminate the PostgreSQL postmaster (the master server process) if the memory demands of either PostgreSQL or another process cause the system to run out of virtual memory.

If this happens, you will see a kernel message that looks like this (consult your system documentation and configuration on where to look for such a message):

```
Out of Memory: Killed process 12345 (postgres).
```

This indicates that the postgres process has been terminated due to memory pressure. Although existing database connections will continue to function normally, no new connections will be accepted. To recover, PostgreSQL will need to be restarted.

One way to avoid this problem is to run PostgreSQL on a machine where you can be sure that other processes will not run the machine out of memory. If memory is tight, increasing the swap space of the operating system can help avoid the problem, because the out-of-memory (OOM) killer is invoked only when physical memory and swap space are exhausted.

If PostgreSQL itself is the cause of the system running out of memory, you can avoid the problem by changing your configuration. In some cases, it may help to lower memory-related configuration parameters, particularly shared_buffers and work_mem. In other cases, the problem may be caused by allowing too many connections to the database server itself. In many cases, it may be better to reduce max_connections and instead make use of external connection-pooling software.

On Linux 2.6 and later, it is possible to modify the kernel's behavior so that it will not "overcommit" memory. Although this setting will not prevent the OOM killer[1] from being invoked altogether, it will lower the chances significantly and will therefore lead to more robust system behavior. This is done by selecting strict overcommit mode via sysctl:

```
sysctl -w vm.overcommit_memory=2
```

or placing an equivalent entry in /etc/sysctl.conf. You might also wish to modify the related setting vm.overcommit_ratio. For details see the kernel documentation file Documentation/vm/overcommit-accounting.

Another approach, which can be used with or without altering vm.overcommit_memory, is to set the process-specific *OOM score adjustment* value for the postmaster process to -1000, thereby guaranteeing it will not be targeted by the OOM killer. The simplest way to do this is to execute

```
echo -1000 > /proc/self/oom_score_adj
```

in the postmaster's startup script just before invoking the postmaster. Note that this action must be done as root, or it will have no effect; so a root-owned startup script is the easiest place to do it. If you do this, you should also set these environment variables in the startup script before invoking the postmaster:

```
export PG_OOM_ADJUST_FILE=/proc/self/oom_score_adj
export PG_OOM_ADJUST_VALUE=0
```

These settings will cause postmaster child processes to run with the normal OOM score adjustment of zero, so that the OOM killer can still target them at need. You could use some other value for PG_OOM_ADJUST_VALUE if you want the child processes to run with some other OOM score adjustment. (PG_OOM_ADJUST_VALUE can also be omitted, in which case it defaults to zero.) If you do not set PG_OOM_ADJUST_FILE, the child processes will run with the same OOM score adjustment as the postmaster, which is unwise since the whole point is to ensure that the postmaster has a preferential setting.

Older Linux kernels do not offer /proc/self/oom_score_adj, but may have a previous version of the same functionality called /proc/self/oom_adj. This works the same except the disable value is -17 not -1000.

1. http://lwn.net/Articles/104179/

Note: Some vendors' Linux 2.4 kernels are reported to have early versions of the 2.6 overcommit `sysctl` parameter. However, setting `vm.overcommit_memory` to 2 on a 2.4 kernel that does not have the relevant code will make things worse, not better. It is recommended that you inspect the actual kernel source code (see the function `vm_enough_memory` in the file `mm/mmap.c`) to verify what is supported in your kernel before you try this in a 2.4 installation. The presence of the `overcommit-accounting` documentation file should *not* be taken as evidence that the feature is there. If in any doubt, consult a kernel expert or your kernel vendor.

17.4.5. Linux Huge Pages

Using huge pages reduces overhead when using large contiguous chunks of memory, as PostgreSQL does, particularly when using large values of shared_buffers. To use this feature in PostgreSQL you need a kernel with `CONFIG_HUGETLBFS=y` and `CONFIG_HUGETLB_PAGE=y`. You will also have to adjust the kernel setting `vm.nr_hugepages`. To estimate the number of huge pages needed, start PostgreSQL without huge pages enabled and check the postmaster's `VmPeak` value, as well as the system's huge page size, using the `/proc` file system. This might look like:

```
$ head -1 $PGDATA/postmaster.pid
4170
$ grep ^VmPeak /proc/4170/status
VmPeak:  6490428 kB
$ grep ^Hugepagesize /proc/meminfo
Hugepagesize:       2048 kB
```

`6490428` / `2048` gives approximately `3169.154`, so in this example we need at least `3170` huge pages, which we can set with:

```
$ sysctl -w vm.nr_hugepages=3170
```

A larger setting would be appropriate if other programs on the machine also need huge pages. Don't forget to add this setting to `/etc/sysctl.conf` so that it will be reapplied after reboots.

Sometimes the kernel is not able to allocate the desired number of huge pages immediately, so it might be necessary to repeat the command or to reboot. (Immediately after a reboot, most of the machine's memory should be available to convert into huge pages.) To verify the huge page allocation situation, use:

```
$ grep Huge /proc/meminfo
```

It may also be necessary to give the database server's operating system user permission to use huge pages by setting `vm.hugetlb_shm_group` via sysctl, and/or give permission to lock memory with `ulimit -l`.

The default behavior for huge pages in PostgreSQL is to use them when possible and to fall back to normal pages when failing. To enforce the use of huge pages, you can set huge_pages to `on` in `postgresql.conf`. Note that with this setting PostgreSQL will fail to start if not enough huge pages are available.

For a detailed description of the Linux huge pages feature have a look at https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.

17.5. Shutting Down the Server

There are several ways to shut down the database server. You control the type of shutdown by sending different signals to the master `postgres` process.

SIGTERM

This is the *Smart Shutdown* mode. After receiving SIGTERM, the server disallows new connections, but lets existing sessions end their work normally. It shuts down only after all of the sessions terminate. If the server is in online backup mode, it additionally waits until online backup mode is no longer active. While backup mode is active, new connections will still be allowed, but only to superusers (this exception allows a superuser to connect to terminate online backup mode). If the server is in recovery when a smart shutdown is requested, recovery and streaming replication will be stopped only after all regular sessions have terminated.

SIGINT

This is the *Fast Shutdown* mode. The server disallows new connections and sends all existing server processes SIGTERM, which will cause them to abort their current transactions and exit promptly. It then waits for all server processes to exit and finally shuts down. If the server is in online backup mode, backup mode will be terminated, rendering the backup useless.

SIGQUIT

This is the *Immediate Shutdown* mode. The server will send SIGQUIT to all child processes and wait for them to terminate. If any do not terminate within 5 seconds, they will be sent SIGKILL. The master server process exits as soon as all child processes have exited, without doing normal database shutdown processing. This will lead to recovery (by replaying the WAL log) upon next start-up. This is recommended only in emergencies.

The pg_ctl program provides a convenient interface for sending these signals to shut down the server. Alternatively, you can send the signal directly using `kill` on non-Windows systems. The PID of the `postgres` process can be found using the `ps` program, or from the file `postmaster.pid` in the data directory. For example, to do a fast shutdown:

```
$ kill -INT `head -1 /usr/local/pgsql/data/postmaster.pid`
```

Important: It is best not to use SIGKILL to shut down the server. Doing so will prevent the server from releasing shared memory and semaphores, which might then have to be done manually before a new server can be started. Furthermore, SIGKILL kills the `postgres` process without letting it relay the signal to its subprocesses, so it will be necessary to kill the individual subprocesses by hand as well.

To terminate an individual session while allowing other sessions to continue, use `pg_terminate_backend()` (see Table 9-67) or send a SIGTERM signal to the child process associated with the session.

17.6. Upgrading a PostgreSQL Cluster

This section discusses how to upgrade your database data from one PostgreSQL release to a newer one.

PostgreSQL major versions are represented by the first two digit groups of the version number, e.g., 8.4. PostgreSQL minor versions are represented by the third group of version digits, e.g., 8.4.2 is the second minor release of 8.4. Minor releases never change the internal storage format and are always compatible with earlier and later minor releases of the same major version number, e.g., 8.4.2 is compatible with 8.4, 8.4.1 and 8.4.6. To update between compatible versions, you simply replace the executables while the server is down and restart the server. The data directory remains unchanged — minor upgrades are that simple.

For *major* releases of PostgreSQL, the internal data storage format is subject to change, thus complicating upgrades. The traditional method for moving data to a new major version is to dump and reload the database, though this can be slow. A faster method is pg_upgrade. Replication methods are also available, as discussed below.

New major versions also typically introduce some user-visible incompatibilities, so application programming changes might be required. All user-visible changes are listed in the release notes (Appendix E); pay particular attention to the section labeled "Migration". If you are upgrading across several major versions, be sure to read the release notes for each intervening version.

Cautious users will want to test their client applications on the new version before switching over fully; therefore, it's often a good idea to set up concurrent installations of old and new versions. When testing a PostgreSQL major upgrade, consider the following categories of possible changes:

Administration

: The capabilities available for administrators to monitor and control the server often change and improve in each major release.

SQL

: Typically this includes new SQL command capabilities and not changes in behavior, unless specifically mentioned in the release notes.

Library API

: Typically libraries like libpq only add new functionality, again unless mentioned in the release notes.

System Catalogs

: System catalog changes usually only affect database management tools.

Server C-language API

: This involves changes in the backend function API, which is written in the C programming language. Such changes affect code that references backend functions deep inside the server.

17.6.1. Upgrading Data via pg_dumpall

One upgrade method is to dump data from one major version of PostgreSQL and reload it in another — to do this, you must use a *logical* backup tool like pg_dumpall; file system level backup methods will not work. (There are checks in place that prevent you from using a data directory with an incompatible version of PostgreSQL, so no great harm can be done by trying to start the wrong server version on a data directory.)

It is recommended that you use the pg_dump and pg_dumpall programs from the *newer* version of PostgreSQL, to take advantage of enhancements that might have been made in these programs. Current releases of the dump programs can read data from any server version back to 7.0.

These instructions assume that your existing installation is under the `/usr/local/pgsql` directory, and that the data area is in `/usr/local/pgsql/data`. Substitute your paths appropriately.

1. If making a backup, make sure that your database is not being updated. This does not affect the integrity of the backup, but the changed data would of course not be included. If necessary, edit the permissions in the file `/usr/local/pgsql/data/pg_hba.conf` (or equivalent) to disallow access from everyone except you. See Chapter 19 for additional information on access control.

   To back up your database installation, type:

   ```
   pg_dumpall > outputfile
   ```

   To make the backup, you can use the pg_dumpall command from the version you are currently running; see Section 24.1.2 for more details. For best results, however, try to use the pg_dumpall command from PostgreSQL 9.5.24, since this version contains bug fixes and improvements over older versions. While this advice might seem idiosyncratic since you haven't installed the new version yet, it is advisable to follow it if you plan to install the new version in parallel with the old version. In that case you can complete the installation normally and transfer the data later. This will also decrease the downtime.

2. Shut down the old server:

   ```
   pg_ctl stop
   ```

   On systems that have PostgreSQL started at boot time, there is probably a start-up file that will accomplish the same thing. For example, on a Red Hat Linux system one might find that this works:

   ```
   /etc/rc.d/init.d/postgresql stop
   ```

   See Chapter 17 for details about starting and stopping the server.

3. If restoring from backup, rename or delete the old installation directory if it is not version-specific. It is a good idea to rename the directory, rather than delete it, in case you have trouble and need to revert to it. Keep in mind the directory might consume significant disk space. To rename the directory, use a command like this:

   ```
   mv /usr/local/pgsql /usr/local/pgsql.old
   ```

   (Be sure to move the directory as a single unit so relative paths remain unchanged.)

4. Install the new version of PostgreSQL as outlined in Section 15.4.

5. Create a new database cluster if needed. Remember that you must execute these commands while logged in to the special database user account (which you already have if you are upgrading).

   ```
   /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
   ```

6. Restore your previous `pg_hba.conf` and any `postgresql.conf` modifications.

7. Start the database server, again using the special database user account:

   /usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data

8. Finally, restore your data from backup with:

   /usr/local/pgsql/bin/psql -d postgres -f *outputfile*
   using the *new* psql.

The least downtime can be achieved by installing the new server in a different directory and running both the old and the new servers in parallel, on different ports. Then you can use something like:

```
pg_dumpall -p 5432 | psql -d postgres -p 5433
```

to transfer your data.

17.6.2. Upgrading Data via pg_upgrade

The pg_upgrade module allows an installation to be migrated in-place from one major PostgreSQL version to another. Upgrades can be performed in minutes, particularly with `--link` mode. It requires steps similar to pg_dumpall above, e.g., starting/stopping the server, running initdb. The pg_upgrade documentation outlines the necessary steps.

17.6.3. Upgrading Data via Replication

It is also possible to use certain replication methods, such as Slony, to create a standby server with the updated version of PostgreSQL. This is possible because Slony supports replication between different major versions of PostgreSQL. The standby can be on the same computer or a different computer. Once it has synced up with the master server (running the older version of PostgreSQL), you can switch masters and make the standby the master and shut down the older database instance. Such a switch-over results in only several seconds of downtime for an upgrade.

17.7. Preventing Server Spoofing

While the server is running, it is not possible for a malicious user to take the place of the normal database server. However, when the server is down, it is possible for a local user to spoof the normal server by starting their own server. The spoof server could read passwords and queries sent by clients, but could not return any data because the `PGDATA` directory would still be secure because of directory permissions. Spoofing is possible because any user can start a database server; a client cannot identify an invalid server unless it is specially configured.

The simplest way to prevent spoofing for `local` connections is to use a Unix domain socket directory (unix_socket_directories) that has write permission only for a trusted local user. This prevents a malicious user from creating their own socket file in that directory. If you are concerned that some applications might still reference `/tmp` for the socket file and hence be vulnerable to spoofing, during operating system startup create a symbolic link `/tmp/.s.PGSQL.5432` that points to the relocated socket file. You also might need to modify your `/tmp` cleanup script to prevent removal of the symbolic link.

To prevent spoofing on TCP connections, the best solution is to use SSL certificates and make sure that clients check the server's certificate. To do that, the server must be configured to accept only `hostssl` connections (Section 19.1) and have SSL key and certificate files (Section 17.9). The TCP client must connect using `sslmode=verify-ca` or `verify-full` and have the appropriate root certificate file installed (Section 31.18.1).

17.8. Encryption Options

PostgreSQL offers encryption at several levels, and provides flexibility in protecting data from disclosure due to database server theft, unscrupulous administrators, and insecure networks. Encryption might also be required to secure sensitive data such as medical records or financial transactions.

Password Storage Encryption

By default, database user passwords are stored as MD5 hashes, so the administrator cannot determine the actual password assigned to the user. If MD5 encryption is used for client authentication, the unencrypted password is never even temporarily present on the server because the client MD5-encrypts it before being sent across the network.

Encryption For Specific Columns

The pgcrypto module allows certain fields to be stored encrypted. This is useful if only some of the data is sensitive. The client supplies the decryption key and the data is decrypted on the server and then sent to the client.

The decrypted data and the decryption key are present on the server for a brief time while it is being decrypted and communicated between the client and server. This presents a brief moment where the data and keys can be intercepted by someone with complete access to the database server, such as the system administrator.

Data Partition Encryption

Storage encryption can be performed at the file system level or the block level. Linux file system encryption options include eCryptfs and EncFS, while FreeBSD uses PEFS. Block level or full disk encryption options include dm-crypt + LUKS on Linux and GEOM modules geli and gbde on FreeBSD. Many other operating systems support this functionality, including Windows.

This mechanism prevents unencrypted data from being read from the drives if the drives or the entire computer is stolen. This does not protect against attacks while the file system is mounted, because when mounted, the operating system provides an unencrypted view of the data. However, to mount the file system, you need some way for the encryption key to be passed to the operating system, and sometimes the key is stored somewhere on the host that mounts the disk.

Encrypting Passwords Across A Network

The `MD5` authentication method double-encrypts the password on the client before sending it to the server. It first MD5-encrypts it based on the user name, and then encrypts it based on a random salt sent by the server when the database connection was made. It is this double-encrypted value that is sent over the network to the server. Double-encryption not only prevents the password from being discovered, it also prevents another connection from using the same encrypted password to connect to the database server at a later time.

Encrypting Data Across A Network

SSL connections encrypt all data sent across the network: the password, the queries, and the data returned. The `pg_hba.conf` file allows administrators to specify which hosts can use non-encrypted connections (`host`) and which require SSL-encrypted connections (`hostssl`). Also, clients can specify that they connect to servers only via SSL. Stunnel or SSH can also be used to encrypt transmissions.

SSL Host Authentication

It is possible for both the client and server to provide SSL certificates to each other. It takes some extra configuration on each side, but this provides stronger verification of identity than the mere use of passwords. It prevents a computer from pretending to be the server just long enough to read the password sent by the client. It also helps prevent "man in the middle" attacks where a computer between the client and server pretends to be the server and reads and passes all data between the client and server.

Client-Side Encryption

If the system administrator for the server's machine cannot be trusted, it is necessary for the client to encrypt the data; this way, unencrypted data never appears on the database server. Data is encrypted on the client before being sent to the server, and database results have to be decrypted on the client before being used.

17.9. Secure TCP/IP Connections with SSL

PostgreSQL has native support for using SSL connections to encrypt client/server communications for increased security. This requires that OpenSSL is installed on both client and server systems and that support in PostgreSQL is enabled at build time (see Chapter 15).

With SSL support compiled in, the PostgreSQL server can be started with SSL enabled by setting the parameter ssl to `on` in `postgresql.conf`. The server will listen for both normal and SSL connections on the same TCP port, and will negotiate with any connecting client on whether to use SSL. By default, this is at the client's option; see Section 19.1 about how to set up the server to require use of SSL for some or all connections.

PostgreSQL reads the system-wide OpenSSL configuration file. By default, this file is named `openssl.cnf` and is located in the directory reported by `openssl version -d`. This default can be overridden by setting environment variable `OPENSSL_CONF` to the name of the desired configuration file.

OpenSSL supports a wide range of ciphers and authentication algorithms, of varying strength. While a list of ciphers can be specified in the OpenSSL configuration file, you can specify ciphers specifically for use by the database server by modifying ssl_ciphers in `postgresql.conf`.

Note: It is possible to have authentication without encryption overhead by using `NULL-SHA` or `NULL-MD5` ciphers. However, a man-in-the-middle could read and pass communications between client and server. Also, encryption overhead is minimal compared to the overhead of authentication. For these reasons NULL ciphers are not recommended.

To start in SSL mode, files containing the server certificate and private key must exist. By default, these files are expected to be named `server.crt` and `server.key`, respectively, in the server's data directory, but other names and locations can be specified using the configuration parameters ssl_cert_file and ssl_key_file. On Unix systems, the permissions on `server.key` must disallow any access to world or group; achieve this by the command `chmod 0600 server.key`. If the private key is protected with a passphrase, the server will prompt for the passphrase and will not start until it has been entered.

The first certificate in `server.crt` must be the server's certificate because it must match the server's private key. The certificates of "intermediate" certificate authorities can also be appended to the file. Doing this avoids the necessity of storing intermediate certificates on clients, assuming the root and intermediate certificates were created with `v3_ca` extensions. (This sets the certificate's basic constraint of `CA` to `true`.) This allows easier expiration of intermediate certificates.

It is not necessary to add the root certificate to `server.crt`. Instead, clients must have the root certificate of the server's certificate chain.

17.9.1. Using Client Certificates

To require the client to supply a trusted certificate, place certificates of the root certificate authorities (CAs) you trust in a file in the data directory, set the parameter ssl_ca_file in `postgresql.conf` to the new file name, and add the authentication option `clientcert=1` to the appropriate `hostssl` line(s) in `pg_hba.conf`. A certificate will then be requested from the client during SSL connection startup. (See Section 31.18 for a description of how to set up certificates on the client.) The server will verify that the client's certificate is signed by one of the trusted certificate authorities.

Intermediate certificates that chain up to existing root certificates can also appear in the file `root.crt` if you wish to avoid storing them on clients (assuming the root and intermediate certificates were created with `v3_ca` extensions). Certificate Revocation List (CRL) entries are also checked if the parameter ssl_crl_file is set.

The `clientcert` option in `pg_hba.conf` is available for all authentication methods, but only for rows specified as `hostssl`. When `clientcert` is not specified or is set to 0, the server will still verify presented client certificates against its CA list, if one is configured, — but it will not insist that a client certificate be presented.

If you are setting up client certificates, you may wish to use the `cert` authentication method, so that the certificates control user authentication as well as providing connection security. See Section 19.3.9 for details.

17.9.2. SSL Server File Usage

Table 17-2 summarizes the files that are relevant to the SSL setup on the server. (The shown file names are default or typical names. The locally configured names could be different.)

Table 17-2. SSL Server File Usage

File	Contents	Effect
ssl_cert_file (`$PGDATA/server.crt`)	server certificate	sent to client to indicate server's identity

File	Contents	Effect
ssl_key_file (`$PGDATA/server.key`)	server private key	proves server certificate was sent by the owner; does not indicate certificate owner is trustworthy
ssl_ca_file (`$PGDATA/root.crt`)	trusted certificate authorities	checks that client certificate is signed by a trusted certificate authority
ssl_crl_file (`$PGDATA/root.crl`)	certificates revoked by certificate authorities	client certificate must not be on this list

The files `server.key`, `server.crt`, `root.crt`, and `root.crl` (or their configured alternative names) are only examined during server start; so you must restart the server for changes in them to take effect.

17.9.3. Creating Certificates

To create a simple self-signed certificate for the server, valid for 365 days, use the following OpenSSL command, replacing *`dbhost.yourdomain.com`* with the server's host name:

```
openssl req -new -x509 -days 365 -nodes -text -out server.crt \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
```

Then do:

```
chmod og-rwx server.key
```

because the server will reject the file if its permissions are more liberal than this. For more details on how to create your server private key and certificate, refer to the OpenSSL documentation.

While a self-signed certificate can be used for testing, a certificate signed by a certificate authority (CA) (usually an enterprise-wide root CA) should be used in production.

To create a server certificate whose identity can be validated by clients, first create a certificate signing request (CSR) and a public/private key file:

```
openssl req -new -nodes -text -out root.csr \
  -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
```

Then, sign the request with the key to create a root certificate authority (using the default OpenSSL configuration file location on Linux):

```
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt
```

Finally, create a server certificate signed by the new root certificate authority:

```
openssl req -new -nodes -text -out server.csr \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key
```

```
openssl x509 -req -in server.csr -text -days 365 \
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out server.crt
```

`server.crt` and `server.key` should be stored on the server, and `root.crt` should be stored on the client so the client can verify that the server's leaf certificate was signed by its trusted root certificate. `root.key` should be stored offline for use in creating future certificates.

It is also possible to create a chain of trust that includes intermediate certificates:

```
# root
openssl req -new -nodes -text -out root.csr \
  -keyout root.key -subj "/CN=root.yourdomain.com"
chmod og-rwx root.key
openssl x509 -req -in root.csr -text -days 3650 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -signkey root.key -out root.crt

# intermediate
openssl req -new -nodes -text -out intermediate.csr \
  -keyout intermediate.key -subj "/CN=intermediate.yourdomain.com"
chmod og-rwx intermediate.key
openssl x509 -req -in intermediate.csr -text -days 1825 \
  -extfile /etc/ssl/openssl.cnf -extensions v3_ca \
  -CA root.crt -CAkey root.key -CAcreateserial \
  -out intermediate.crt

# leaf
openssl req -new -nodes -text -out server.csr \
  -keyout server.key -subj "/CN=dbhost.yourdomain.com"
chmod og-rwx server.key
openssl x509 -req -in server.csr -text -days 365 \
  -CA intermediate.crt -CAkey intermediate.key -CAcreateserial \
  -out server.crt
```

`server.crt` and `intermediate.crt` should be concatenated into a certificate file bundle and stored on the server. `server.key` should also be stored on the server. `root.crt` should be stored on the client so the client can verify that the server's leaf certificate was signed by a chain of certificates linked to its trusted root certificate. `root.key` and `intermediate.key` should be stored offline for use in creating future certificates.

17.10. Secure TCP/IP Connections with SSH Tunnels

It is possible to use SSH to encrypt the network connection between clients and a PostgreSQL server. Done properly, this provides an adequately secure network connection, even for non-SSL-capable clients.

First make sure that an SSH server is running properly on the same machine as the PostgreSQL server and that you can log in using `ssh` as some user; you then can establish a secure tunnel to the remote server. A secure tunnel listens on a local port and forwards all traffic to a port on the remote machine. Traffic sent to the remote port can arrive on its `localhost` address, or different bind address if desired; it does not

appear as coming from your local machine. This command creates a secure tunnel from the client machine to the remote machine `foo.com`:

```
ssh -L 63333:localhost:5432 joe@foo.com
```

The first number in the `-L` argument, 63333, is the local port number of the tunnel; it can be any unused port. (IANA reserves ports 49152 through 65535 for private use.) The name or IP address after this is the remote bind address you are connecting to, i.e., `localhost`, which is the default. The second number, 5432, is the remote end of the tunnel, e.g., the port number your database server is using. In order to connect to the database server using this tunnel, you connect to port 63333 on the local machine:

```
psql -h localhost -p 63333 postgres
```

To the database server it will then look as though you are user `joe` on host `foo.com` connecting to the `localhost` bind address, and it will use whatever authentication procedure was configured for connections by that user to that bind address. Note that the server will not think the connection is SSL-encrypted, since in fact it is not encrypted between the SSH server and the PostgreSQL server. This should not pose any extra security risk because they are on the same machine.

In order for the tunnel setup to succeed you must be allowed to connect via `ssh` as `joe@foo.com`, just as if you had attempted to use `ssh` to create a terminal session.

You could also have set up port forwarding as

```
ssh -L 63333:foo.com:5432 joe@foo.com
```

but then the database server will see the connection as coming in on its `foo.com` bind address, which is not opened by the default setting `listen_addresses = 'localhost'`. This is usually not what you want.

If you have to "hop" to the database server via some login host, one possible setup could look like this:

```
ssh -L 63333:db.foo.com:5432 joe@shell.foo.com
```

Note that this way the connection from `shell.foo.com` to `db.foo.com` will not be encrypted by the SSH tunnel. SSH offers quite a few configuration possibilities when the network is restricted in various ways. Please refer to the SSH documentation for details.

> **Tip:** Several other applications exist that can provide secure tunnels using a procedure similar in concept to the one just described.

17.11. Registering Event Log on Windows

To register a Windows event log library with the operating system, issue this command:

```
regsvr32 pgsql_library_directory/pgevent.dll
```

This creates registry entries used by the event viewer, under the default event source named `PostgreSQL`.

To specify a different event source name (see event_source), use the `/n` and `/i` options:

```
regsvr32 /n /i:event_source_name pgsql_library_directory/pgevent.dll
```

To unregister the event log library from the operating system, issue this command:

```
regsvr32 /u [/i:event_source_name] pgsql_library_directory/pgevent.dll
```

> **Note:** To enable event logging in the database server, modify log_destination to include `eventlog` in `postgresql.conf`.

Chapter 18. Server Configuration

There are many configuration parameters that affect the behavior of the database system. In the first section of this chapter we describe how to interact with configuration parameters. The subsequent sections discuss each parameter in detail.

18.1. Setting Parameters

18.1.1. Parameter Names and Values

All parameter names are case-insensitive. Every parameter takes a value of one of five types: boolean, string, integer, floating point, or enumerated (enum). The type determines the syntax for setting the parameter:

- *Boolean:* Values can be written as `on`, `off`, `true`, `false`, `yes`, `no`, `1`, `0` (all case-insensitive) or any unambiguous prefix of one of these.
- *String:* In general, enclose the value in single quotes, doubling any single quotes within the value. Quotes can usually be omitted if the value is a simple number or identifier, however.
- *Numeric (integer and floating point):* A decimal point is permitted only for floating-point parameters. Do not use thousands separators. Quotes are not required.
- *Numeric with Unit:* Some numeric parameters have an implicit unit, because they describe quantities of memory or time. The unit might be kilobytes, blocks (typically eight kilobytes), milliseconds, seconds, or minutes. An unadorned numeric value for one of these settings will use the setting's default unit, which can be learned from `pg_settings.unit`. For convenience, settings can be given with a unit specified explicitly, for example `'120 ms'` for a time value, and they will be converted to whatever the parameter's actual unit is. Note that the value must be written as a string (with quotes) to use this feature. The unit name is case-sensitive, and there can be whitespace between the numeric value and the unit.
  - Valid memory units are `kB` (kilobytes), `MB` (megabytes), `GB` (gigabytes), and `TB` (terabytes). The multiplier for memory units is 1024, not 1000.
  - Valid time units are `ms` (milliseconds), `s` (seconds), `min` (minutes), `h` (hours), and `d` (days).
- *Enumerated:* Enumerated-type parameters are written in the same way as string parameters, but are restricted to have one of a limited set of values. The values allowable for such a parameter can be found from `pg_settings.enumvals`. Enum parameter values are case-insensitive.

18.1.2. Parameter Interaction via the Configuration File

The most fundamental way to set these parameters is to edit the file `postgresql.conf`, which is normally kept in the data directory. A default copy is installed when the database cluster directory is initial-

ized. An example of what this file might look like is:

```
# This is a comment
log_connections = yes
log_destination = 'syslog'
search_path = '"$user", public'
shared_buffers = 128MB
```

One parameter is specified per line. The equal sign between name and value is optional. Whitespace is insignificant (except within a quoted parameter value) and blank lines are ignored. Hash marks (#) designate the remainder of the line as a comment. Parameter values that are not simple identifiers or numbers must be single-quoted. To embed a single quote in a parameter value, write either two quotes (preferred) or backslash-quote. If the file contains multiple entries for the same parameter, all but the last one are ignored.

Parameters set in this way provide default values for the cluster. The settings seen by active sessions will be these values unless they are overridden. The following sections describe ways in which the administrator or user can override these defaults.

The configuration file is reread whenever the main server process receives a SIGHUP signal; this signal is most easily sent by running `pg_ctl reload` from the command line or by calling the SQL function `pg_reload_conf()`. The main server process also propagates this signal to all currently running server processes, so that existing sessions also adopt the new values (this will happen after they complete any currently-executing client command). Alternatively, you can send the signal to a single server process directly. Some parameters can only be set at server start; any changes to their entries in the configuration file will be ignored until the server is restarted. Invalid parameter settings in the configuration file are likewise ignored (but logged) during SIGHUP processing.

In addition to `postgresql.conf`, a PostgreSQL data directory contains a file `postgresql.auto.conf`, which has the same format as `postgresql.conf` but is intended to be edited automatically not manually. This file holds settings provided through the ALTER SYSTEM command. This file is read whenever `postgresql.conf` is, and its settings take effect in the same way. Settings in `postgresql.auto.conf` override those in `postgresql.conf`.

External tools may also modify `postgresql.auto.conf`. It is not recommended to do this while the server is running, since a concurrent `ALTER SYSTEM` command could overwrite such changes. Such tools might simply append new settings to the end, or they might choose to remove duplicate settings and/or comments (as `ALTER SYSTEM` will).

The system view `pg_file_settings` can be helpful for pre-testing changes to the configuration files, or for diagnosing problems if a SIGHUP signal did not have the desired effects.

18.1.3. Parameter Interaction via SQL

PostgreSQL provides three SQL commands to establish configuration defaults. The already-mentioned ALTER SYSTEM command provides a SQL-accessible means of changing global defaults; it is functionally equivalent to editing `postgresql.conf`. In addition, there are two commands that allow setting of defaults on a per-database or per-role basis:

- The ALTER DATABASE command allows global settings to be overridden on a per-database basis.

- The ALTER ROLE command allows both global and per-database settings to be overridden with user-specific values.

Values set with `ALTER DATABASE` and `ALTER ROLE` are applied only when starting a fresh database session. They override values obtained from the configuration files or server command line, and constitute defaults for the rest of the session. Note that some settings cannot be changed after server start, and so cannot be set with these commands (or the ones listed below).

Once a client is connected to the database, PostgreSQL provides two additional SQL commands (and equivalent functions) to interact with session-local configuration settings:

- The SHOW command allows inspection of the current value of all parameters. The corresponding function is `current_setting(setting_name text)`.
- The SET command allows modification of the current value of those parameters that can be set locally to a session; it has no effect on other sessions. The corresponding function is `set_config(setting_name, new_value, is_local)`.

In addition, the system view `pg_settings` can be used to view and change session-local values:

- Querying this view is similar to using `SHOW ALL` but provides more detail. It is also more flexible, since it's possible to specify filter conditions or join against other relations.
- Using UPDATE on this view, specifically updating the `setting` column, is the equivalent of issuing `SET` commands. For example, the equivalent of

  ```
  SET configuration_parameter TO DEFAULT;
  ```
  is:
  ```
  UPDATE pg_settings SET setting = reset_val WHERE name = 'configuration_parameter';
  ```

18.1.4. Parameter Interaction via the Shell

In addition to setting global defaults or attaching overrides at the database or role level, you can pass settings to PostgreSQL via shell facilities. Both the server and libpq client library accept parameter values via the shell.

- During server startup, parameter settings can be passed to the `postgres` command via the `-c` command-line parameter. For example,

  ```
  postgres -c log_connections=yes -c log_destination='syslog'
  ```
  Settings provided in this way override those set via `postgresql.conf` or `ALTER SYSTEM`, so they cannot be changed globally without restarting the server.
- When starting a client session via libpq, parameter settings can be specified using the `PGOPTIONS` environment variable. Settings established in this way constitute defaults for the life of the session, but do not affect other sessions. For historical reasons, the format of `PGOPTIONS` is similar to that used when launching the `postgres` command; specifically, the `-c` flag must be specified. For example,

  ```
  env PGOPTIONS="-c geqo=off -c statement_timeout=5min" psql
  ```

  Other clients and libraries might provide their own mechanisms, via the shell or otherwise, that allow the user to alter session settings without direct use of SQL commands.

18.1.5. Managing Configuration File Contents

PostgreSQL provides several features for breaking down complex `postgresql.conf` files into sub-files. These features are especially useful when managing multiple servers with related, but not identical, configurations.

In addition to individual parameter settings, the `postgresql.conf` file can contain *include directives*, which specify another file to read and process as if it were inserted into the configuration file at this point. This feature allows a configuration file to be divided into physically separate parts. Include directives simply look like:

```
include 'filename'
```

If the file name is not an absolute path, it is taken as relative to the directory containing the referencing configuration file. Inclusions can be nested.

There is also an `include_if_exists` directive, which acts the same as the `include` directive, except when the referenced file does not exist or cannot be read. A regular `include` will consider this an error condition, but `include_if_exists` merely logs a message and continues processing the referencing configuration file.

The `postgresql.conf` file can also contain `include_dir` directives, which specify an entire directory of configuration files to include. These look like

```
include_dir 'directory'
```

Non-absolute directory names are taken as relative to the directory containing the referencing configuration file. Within the specified directory, only non-directory files whose names end with the suffix `.conf` will be included. File names that start with the `.` character are also ignored, to prevent mistakes since such files are hidden on some platforms. Multiple files within an include directory are processed in file name order (according to C locale rules, i.e., numbers before letters, and uppercase letters before lowercase ones).

Include files or directories can be used to logically separate portions of the database configuration, rather than having a single large `postgresql.conf` file. Consider a company that has two database servers, each with a different amount of memory. There are likely elements of the configuration both will share, for things such as logging. But memory-related parameters on the server will vary between the two. And there might be server specific customizations, too. One way to manage this situation is to break the custom configuration changes for your site into three files. You could add this to the end of your `postgresql.conf` file to include them:

```
include 'shared.conf'
include 'memory.conf'
include 'server.conf'
```

All systems would have the same `shared.conf`. Each server with a particular amount of memory could share the same `memory.conf`; you might have one for all servers with 8GB of RAM, another for those having 16GB. And finally `server.conf` could have truly server-specific configuration information in it.

Another possibility is to create a configuration file directory and put this information into files there. For example, a `conf.d` directory could be referenced at the end of `postgresql.conf`:

```
include_dir 'conf.d'
```

Then you could name the files in the conf.d directory like this:

```
00shared.conf
01memory.conf
02server.conf
```

This naming convention establishes a clear order in which these files will be loaded. This is important because only the last setting encountered for a particular parameter while the server is reading configuration files will be used. In this example, something set in conf.d/02server.conf would override a value set in conf.d/01memory.conf.

You might instead use this approach to naming the files descriptively:

```
00shared.conf
01memory-8GB.conf
02server-foo.conf
```

This sort of arrangement gives a unique name for each configuration file variation. This can help eliminate ambiguity when several servers have their configurations all stored in one place, such as in a version control repository. (Storing database configuration files under version control is another good practice to consider.)

18.2. File Locations

In addition to the postgresql.conf file already mentioned, PostgreSQL uses two other manually-edited configuration files, which control client authentication (their use is discussed in Chapter 19). By default, all three configuration files are stored in the database cluster's data directory. The parameters described in this section allow the configuration files to be placed elsewhere. (Doing so can ease administration. In particular it is often easier to ensure that the configuration files are properly backed-up when they are kept separate.)

data_directory (string)

Specifies the directory to use for data storage. This parameter can only be set at server start.

config_file (string)

Specifies the main server configuration file (customarily called postgresql.conf). This parameter can only be set on the postgres command line.

hba_file (string)

Specifies the configuration file for host-based authentication (customarily called pg_hba.conf). This parameter can only be set at server start.

ident_file (string)

Specifies the configuration file for user name mapping (customarily called pg_ident.conf). This parameter can only be set at server start. See also Section 19.2.

external_pid_file (string)

Specifies the name of an additional process-ID (PID) file that the server should create for use by server administration programs. This parameter can only be set at server start.

In a default installation, none of the above parameters are set explicitly. Instead, the data directory is specified by the -D command-line option or the PGDATA environment variable, and the configuration files are all found within the data directory.

If you wish to keep the configuration files elsewhere than the data directory, the postgres -D command-line option or PGDATA environment variable must point to the directory containing the configuration files, and the data_directory parameter must be set in postgresql.conf (or on the command line) to show where the data directory is actually located. Notice that data_directory overrides -D and PGDATA for the location of the data directory, but not for the location of the configuration files.

If you wish, you can specify the configuration file names and locations individually using the parameters config_file, hba_file and/or ident_file. config_file can only be specified on the postgres command line, but the others can be set within the main configuration file. If all three parameters plus data_directory are explicitly set, then it is not necessary to specify -D or PGDATA.

When setting any of these parameters, a relative path will be interpreted with respect to the directory in which postgres is started.

18.3. Connections and Authentication

18.3.1. Connection Settings

listen_addresses (string)

Specifies the TCP/IP address(es) on which the server is to listen for connections from client applications. The value takes the form of a comma-separated list of host names and/or numeric IP addresses. The special entry * corresponds to all available IP interfaces. The entry 0.0.0.0 allows listening for all IPv4 addresses and :: allows listening for all IPv6 addresses. If the list is empty, the server does not listen on any IP interface at all, in which case only Unix-domain sockets can be used to connect to it. The default value is localhost, which allows only local TCP/IP "loopback" connections to be made. While client authentication (Chapter 19) allows fine-grained control over who can access the server, listen_addresses controls which interfaces accept connection attempts, which can help prevent repeated malicious connection requests on insecure network interfaces. This parameter can only be set at server start.

port (integer)

The TCP port the server listens on; 5432 by default. Note that the same port number is used for all IP addresses the server listens on. This parameter can only be set at server start.

max_connections (integer)

Determines the maximum number of concurrent connections to the database server. The default is typically 100 connections, but might be less if your kernel settings will not support it (as determined during initdb). This parameter can only be set at server start.

When running a standby server, you must set this parameter to the same or higher value than on the master server. Otherwise, queries will not be allowed in the standby server.

`superuser_reserved_connections` (`integer`)

Determines the number of connection "slots" that are reserved for connections by PostgreSQL superusers. At most max_connections connections can ever be active simultaneously. Whenever the number of active concurrent connections is at least `max_connections` minus `superuser_reserved_connections`, new connections will be accepted only for superusers, and no new replication connections will be accepted.

The default value is three connections. The value must be less than the value of `max_connections`. This parameter can only be set at server start.

`unix_socket_directories` (`string`)

Specifies the directory of the Unix-domain socket(s) on which the server is to listen for connections from client applications. Multiple sockets can be created by listing multiple directories separated by commas. Whitespace between entries is ignored; surround a directory name with double quotes if you need to include whitespace or commas in the name. An empty value specifies not listening on any Unix-domain sockets, in which case only TCP/IP sockets can be used to connect to the server. The default value is normally `/tmp`, but that can be changed at build time. This parameter can only be set at server start.

In addition to the socket file itself, which is named `.s.PGSQL.`*`nnnn`* where *`nnnn`* is the server's port number, an ordinary file named `.s.PGSQL.`*`nnnn`*`.lock` will be created in each of the `unix_socket_directories` directories. Neither file should ever be removed manually.

This parameter is irrelevant on Windows, which does not have Unix-domain sockets.

`unix_socket_group` (`string`)

Sets the owning group of the Unix-domain socket(s). (The owning user of the sockets is always the user that starts the server.) In combination with the parameter `unix_socket_permissions` this can be used as an additional access control mechanism for Unix-domain connections. By default this is the empty string, which uses the default group of the server user. This parameter can only be set at server start.

This parameter is irrelevant on Windows, which does not have Unix-domain sockets.

`unix_socket_permissions` (`integer`)

Sets the access permissions of the Unix-domain socket(s). Unix-domain sockets use the usual Unix file system permission set. The parameter value is expected to be a numeric mode specified in the format accepted by the `chmod` and `umask` system calls. (To use the customary octal format the number must start with a `0` (zero).)

The default permissions are `0777`, meaning anyone can connect. Reasonable alternatives are `0770` (only user and group, see also `unix_socket_group`) and `0700` (only user). (Note that for a Unix-domain socket, only write permission matters, so there is no point in setting or revoking read or execute permissions.)

This access control mechanism is independent of the one described in Chapter 19.

This parameter can only be set at server start.

This parameter is irrelevant on systems, notably Solaris as of Solaris 10, that ignore socket permissions entirely. There, one can achieve a similar effect by pointing `unix_socket_directories` to a

directory having search permission limited to the desired audience. This parameter is also irrelevant on Windows, which does not have Unix-domain sockets.

`bonjour` (`boolean`)

Enables advertising the server's existence via Bonjour. The default is off. This parameter can only be set at server start.

`bonjour_name` (`string`)

Specifies the Bonjour service name. The computer name is used if this parameter is set to the empty string `''` (which is the default). This parameter is ignored if the server was not compiled with Bonjour support. This parameter can only be set at server start.

`tcp_keepalives_idle` (`integer`)

Specifies the number of seconds of inactivity after which TCP should send a keepalive message to the client. A value of 0 uses the system default. This parameter is supported only on systems that support `TCP_KEEPIDLE` or an equivalent socket option, and on Windows; on other systems, it must be zero. In sessions connected via a Unix-domain socket, this parameter is ignored and always reads as zero.

> **Note:** On Windows, a value of 0 will set this parameter to 2 hours, since Windows does not provide a way to read the system default value.

`tcp_keepalives_interval` (`integer`)

Specifies the number of seconds after which a TCP keepalive message that is not acknowledged by the client should be retransmitted. A value of 0 uses the system default. This parameter is supported only on systems that support `TCP_KEEPINTVL` or an equivalent socket option, and on Windows; on other systems, it must be zero. In sessions connected via a Unix-domain socket, this parameter is ignored and always reads as zero.

> **Note:** On Windows, a value of 0 will set this parameter to 1 second, since Windows does not provide a way to read the system default value.

`tcp_keepalives_count` (`integer`)

Specifies the number of TCP keepalives that can be lost before the server's connection to the client is considered dead. A value of 0 uses the system default. This parameter is supported only on systems that support `TCP_KEEPCNT` or an equivalent socket option; on other systems, it must be zero. In sessions connected via a Unix-domain socket, this parameter is ignored and always reads as zero.

> **Note:** This parameter is not supported on Windows, and must be zero.

18.3.2. Security and Authentication

`authentication_timeout` (`integer`)

Maximum time to complete client authentication, in seconds. If a would-be client has not completed the authentication protocol in this much time, the server closes the connection. This prevents hung clients from occupying a connection indefinitely. The default is one minute (`1m`). This parameter can only be set in the `postgresql.conf` file or on the server command line.

`ssl` (`boolean`)

Enables SSL connections. Please read Section 17.9 before using this. The default is `off`. This parameter can only be set at server start. SSL communication is only possible with TCP/IP connections.

`ssl_ca_file` (`string`)

Specifies the name of the file containing the SSL server certificate authority (CA). The default is empty, meaning no CA file is loaded, and client certificate verification is not performed. (In previous releases of PostgreSQL, the name of this file was hard-coded as `root.crt`.) Relative paths are relative to the data directory. This parameter can only be set at server start.

`ssl_cert_file` (`string`)

Specifies the name of the file containing the SSL server certificate. The default is `server.crt`. Relative paths are relative to the data directory. This parameter can only be set at server start.

`ssl_crl_file` (`string`)

Specifies the name of the file containing the SSL server certificate revocation list (CRL). The default is empty, meaning no CRL file is loaded. (In previous releases of PostgreSQL, the name of this file was hard-coded as `root.crl`.) Relative paths are relative to the data directory. This parameter can only be set at server start.

`ssl_key_file` (`string`)

Specifies the name of the file containing the SSL server private key. The default is `server.key`. Relative paths are relative to the data directory. This parameter can only be set at server start.

`ssl_ciphers` (`string`)

Specifies a list of SSL cipher suites that are allowed to be used by SSL connections. See the ciphers manual page in the OpenSSL package for the syntax of this setting and a list of supported values. Only connections using TLS version 1.2 and lower are affected. There is currently no setting that controls the cipher choices used by TLS version 1.3 connections. The default value is `HIGH:MEDIUM:+3DES:!aNULL`. It is usually reasonable, unless you have specific security requirements. This parameter can only be set at server start.

Explanation of the default value:

`HIGH`

Cipher suites that use ciphers from `HIGH` group (e.g., AES, Camellia, 3DES)

`MEDIUM`

Cipher suites that use ciphers from `MEDIUM` group (e.g., RC4, SEED)

`+3DES`

The OpenSSL default order for `HIGH` is problematic because it orders 3DES higher than AES128. This is wrong because 3DES offers less security than AES128, and it is also much slower. `+3DES` reorders it after all other `HIGH` and `MEDIUM` ciphers.

`!aNULL`

Disables anonymous cipher suites that do no authentication. Such cipher suites are vulnerable to man-in-the-middle attacks and therefore should not be used.

Available cipher suite details will vary across OpenSSL versions. Use the command `openssl ciphers -v 'HIGH:MEDIUM:+3DES:!aNULL'` to see actual details for the currently installed OpenSSL version. Note that this list is filtered at run time based on the server key type.

`ssl_prefer_server_ciphers` (`bool`)

Specifies whether to use the server's SSL cipher preferences, rather than the client's. The default is true. This parameter can only be set at server start.

Older PostgreSQL versions do not have this setting and always use the client's preferences. This setting is mainly for backward compatibility with those versions. Using the server's preferences is usually better because it is more likely that the server is appropriately configured.

`ssl_ecdh_curve` (`string`)

Specifies the name of the curve to use in ECDH key exchange. It needs to be supported by all clients that connect. It does not need to be same curve as used by server's Elliptic Curve key. The default is `prime256v1`. This parameter can only be set at server start.

OpenSSL names for most common curves: `prime256v1` (NIST P-256), `secp384r1` (NIST P-384), `secp521r1` (NIST P-521).

The full list of available curves can be shown with the command `openssl ecparam -list_curves`. Not all of them are usable in TLS though.

`password_encryption` (`boolean`)

When a password is specified in CREATE USER or ALTER ROLE without writing either `ENCRYPTED` or `UNENCRYPTED`, this parameter determines whether the password is to be encrypted. The default is `on` (encrypt the password).

`krb_server_keyfile` (`string`)

Sets the location of the Kerberos server key file. See Section 19.3.3 for details. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`krb_caseins_users` (`boolean`)

Sets whether GSSAPI user names should be treated case-insensitively. The default is `off` (case sensitive). This parameter can only be set in the `postgresql.conf` file or on the server command line.

`db_user_namespace` (`boolean`)

This parameter enables per-database user names. It is off by default. This parameter can only be set in the `postgresql.conf` file or on the server command line.

If this is on, you should create users as `username@dbname`. When `username` is passed by a connecting client, `@` and the database name are appended to the user name and that database-specific user name is looked up by the server. Note that when you create users with names containing `@` within the SQL environment, you will need to quote the user name.

With this parameter enabled, you can still create ordinary global users. Simply append `@` when specifying the user name in the client, e.g., `joe@`. The `@` will be stripped off before the user name is looked up by the server.

`db_user_namespace` causes the client's and server's user name representation to differ. Authentication checks are always done with the server's user name so authentication methods must be configured for the server's user name, not the client's. Because `md5` uses the user name as salt on both the client and server, `md5` cannot be used with `db_user_namespace`.

> **Note:** This feature is intended as a temporary measure until a complete solution is found. At that time, this option will be removed.

18.4. Resource Consumption

18.4.1. Memory

`shared_buffers` (`integer`)

Sets the amount of memory the database server uses for shared memory buffers. The default is typically 128 megabytes (`128MB`), but might be less if your kernel settings will not support it (as determined during initdb). This setting must be at least 128 kilobytes. (Non-default values of `BLCKSZ` change the minimum.) However, settings significantly higher than the minimum are usually needed for good performance. This parameter can only be set at server start.

If you have a dedicated database server with 1GB or more of RAM, a reasonable starting value for `shared_buffers` is 25% of the memory in your system. There are some workloads where even large settings for `shared_buffers` are effective, but because PostgreSQL also relies on the operating system cache, it is unlikely that an allocation of more than 40% of RAM to `shared_buffers` will work better than a smaller amount. Larger settings for `shared_buffers` usually require a corresponding increase in `max_wal_size`, in order to spread out the process of writing large quantities of new or changed data over a longer period of time.

On systems with less than 1GB of RAM, a smaller percentage of RAM is appropriate, so as to leave adequate space for the operating system. Also, on Windows, large values for `shared_buffers` aren't as effective. You may find better results keeping the setting relatively low and using the operating system cache more instead. The useful range for `shared_buffers` on Windows systems is generally from 64MB to 512MB.

huge_pages (enum)

Enables/disables the use of huge memory pages. Valid values are try (the default), on, and off.

At present, this feature is supported only on Linux. The setting is ignored on other systems when set to try.

The use of huge pages results in smaller page tables and less CPU time spent on memory management, increasing performance. For more details, see Section 17.4.5.

With huge_pages set to try, the server will try to use huge pages, but fall back to using normal allocation if that fails. With on, failure to use huge pages will prevent the server from starting up. With off, huge pages will not be used.

temp_buffers (integer)

Sets the maximum number of temporary buffers used by each database session. These are session-local buffers used only for access to temporary tables. The default is eight megabytes (8MB). The setting can be changed within individual sessions, but only before the first use of temporary tables within the session; subsequent attempts to change the value will have no effect on that session.

A session will allocate temporary buffers as needed up to the limit given by temp_buffers. The cost of setting a large value in sessions that do not actually need many temporary buffers is only a buffer descriptor, or about 64 bytes, per increment in temp_buffers. However if a buffer is actually used an additional 8192 bytes will be consumed for it (or in general, BLCKSZ bytes).

max_prepared_transactions (integer)

Sets the maximum number of transactions that can be in the "prepared" state simultaneously (see PREPARE TRANSACTION). Setting this parameter to zero (which is the default) disables the prepared-transaction feature. This parameter can only be set at server start.

If you are not planning to use prepared transactions, this parameter should be set to zero to prevent accidental creation of prepared transactions. If you are using prepared transactions, you will probably want max_prepared_transactions to be at least as large as max_connections, so that every session can have a prepared transaction pending.

When running a standby server, you must set this parameter to the same or higher value than on the master server. Otherwise, queries will not be allowed in the standby server.

work_mem (integer)

Specifies the amount of memory to be used by internal sort operations and hash tables before writing to temporary disk files. The value defaults to four megabytes (4MB). Note that for a complex query, several sort or hash operations might be running in parallel; each operation will be allowed to use as much memory as this value specifies before it starts to write data into temporary files. Also, several running sessions could be doing such operations concurrently. Therefore, the total memory used could be many times the value of work_mem; it is necessary to keep this fact in mind when choosing the value. Sort operations are used for ORDER BY, DISTINCT, and merge joins. Hash tables are used in hash joins, hash-based aggregation, and hash-based processing of IN subqueries.

maintenance_work_mem (integer)

Specifies the maximum amount of memory to be used by maintenance operations, such as VACUUM, CREATE INDEX, and ALTER TABLE ADD FOREIGN KEY. It defaults to 64 megabytes (64MB). Since only one of these operations can be executed at a time by a database session, and an installation normally doesn't have many of them running concurrently, it's safe to set this value significantly

larger than work_mem. Larger settings might improve performance for vacuuming and for restoring database dumps.

Note that when autovacuum runs, up to autovacuum_max_workers times this memory may be allocated, so be careful not to set the default value too high. It may be useful to control for this by separately setting autovacuum_work_mem.

autovacuum_work_mem (integer)

Specifies the maximum amount of memory to be used by each autovacuum worker process. It defaults to -1, indicating that the value of maintenance_work_mem should be used instead. The setting has no effect on the behavior of VACUUM when run in other contexts.

max_stack_depth (integer)

Specifies the maximum safe depth of the server's execution stack. The ideal setting for this parameter is the actual stack size limit enforced by the kernel (as set by ulimit -s or local equivalent), less a safety margin of a megabyte or so. The safety margin is needed because the stack depth is not checked in every routine in the server, but only in key potentially-recursive routines such as expression evaluation. The default setting is two megabytes (2MB), which is conservatively small and unlikely to risk crashes. However, it might be too small to allow execution of complex functions. Only superusers can change this setting.

Setting max_stack_depth higher than the actual kernel limit will mean that a runaway recursive function can crash an individual backend process. On platforms where PostgreSQL can determine the kernel limit, the server will not allow this variable to be set to an unsafe value. However, not all platforms provide the information, so caution is recommended in selecting a value.

dynamic_shared_memory_type (enum)

Specifies the dynamic shared memory implementation that the server should use. Possible values are posix (for POSIX shared memory allocated using shm_open), sysv (for System V shared memory allocated via shmget), windows (for Windows shared memory), mmap (to simulate shared memory using memory-mapped files stored in the data directory), and none (to disable this feature). Not all values are supported on all platforms; the first supported option is the default for that platform. The use of the mmap option, which is not the default on any platform, is generally discouraged because the operating system may write modified pages back to disk repeatedly, increasing system I/O load; however, it may be useful for debugging, when the pg_dynshmem directory is stored on a RAM disk, or when other shared memory facilities are not available.

18.4.2. Disk

temp_file_limit (integer)

Specifies the maximum amount of disk space that a session can use for temporary files, such as sort and hash temporary files, or the storage file for a held cursor. A transaction attempting to exceed this limit will be canceled. The value is specified in kilobytes, and -1 (the default) means no limit. Only superusers can change this setting.

This setting constrains the total space used at any instant by all temporary files used by a given PostgreSQL session. It should be noted that disk space used for explicit temporary tables, as opposed to temporary files used behind-the-scenes in query execution, does *not* count against this limit.

18.4.3. Kernel Resource Usage

`max_files_per_process` (`integer`)

Sets the maximum number of simultaneously open files allowed to each server subprocess. The default is one thousand files. If the kernel is enforcing a safe per-process limit, you don't need to worry about this setting. But on some platforms (notably, most BSD systems), the kernel will allow individual processes to open many more files than the system can actually support if many processes all try to open that many files. If you find yourself seeing "Too many open files" failures, try reducing this setting. This parameter can only be set at server start.

18.4.4. Cost-based Vacuum Delay

During the execution of VACUUM and ANALYZE commands, the system maintains an internal counter that keeps track of the estimated cost of the various I/O operations that are performed. When the accumulated cost reaches a limit (specified by `vacuum_cost_limit`), the process performing the operation will sleep for a short period of time, as specified by `vacuum_cost_delay`. Then it will reset the counter and continue execution.

The intent of this feature is to allow administrators to reduce the I/O impact of these commands on concurrent database activity. There are many situations where it is not important that maintenance commands like `VACUUM` and `ANALYZE` finish quickly; however, it is usually very important that these commands do not significantly interfere with the ability of the system to perform other database operations. Cost-based vacuum delay provides a way for administrators to achieve this.

This feature is disabled by default for manually issued `VACUUM` commands. To enable it, set the `vacuum_cost_delay` variable to a nonzero value.

`vacuum_cost_delay` (`integer`)

The length of time, in milliseconds, that the process will sleep when the cost limit has been exceeded. The default value is zero, which disables the cost-based vacuum delay feature. Positive values enable cost-based vacuuming. Note that on many systems, the effective resolution of sleep delays is 10 milliseconds; setting `vacuum_cost_delay` to a value that is not a multiple of 10 might have the same results as setting it to the next higher multiple of 10.

When using cost-based vacuuming, appropriate values for `vacuum_cost_delay` are usually quite small, perhaps 10 or 20 milliseconds. Adjusting vacuum's resource consumption is best done by changing the other vacuum cost parameters.

`vacuum_cost_page_hit` (`integer`)

The estimated cost for vacuuming a buffer found in the shared buffer cache. It represents the cost to lock the buffer pool, lookup the shared hash table and scan the content of the page. The default value is one.

`vacuum_cost_page_miss` (`integer`)

The estimated cost for vacuuming a buffer that has to be read from disk. This represents the effort to lock the buffer pool, lookup the shared hash table, read the desired block in from the disk and scan its content. The default value is 10.

vacuum_cost_page_dirty (integer)

The estimated cost charged when vacuum modifies a block that was previously clean. It represents the extra I/O required to flush the dirty block out to disk again. The default value is 20.

vacuum_cost_limit (integer)

The accumulated cost that will cause the vacuuming process to sleep. The default value is 200.

Note: There are certain operations that hold critical locks and should therefore complete as quickly as possible. Cost-based vacuum delays do not occur during such operations. Therefore it is possible that the cost accumulates far higher than the specified limit. To avoid uselessly long delays in such cases, the actual delay is calculated as vacuum_cost_delay * accumulated_balance / vacuum_cost_limit with a maximum of vacuum_cost_delay * 4.

18.4.5. Background Writer

There is a separate server process called the *background writer*, whose function is to issue writes of "dirty" (new or modified) shared buffers. It writes shared buffers so server processes handling user queries seldom or never need to wait for a write to occur. However, the background writer does cause a net overall increase in I/O load, because while a repeatedly-dirtied page might otherwise be written only once per checkpoint interval, the background writer might write it several times as it is dirtied in the same interval. The parameters discussed in this subsection can be used to tune the behavior for local needs.

bgwriter_delay (integer)

Specifies the delay between activity rounds for the background writer. In each round the writer issues writes for some number of dirty buffers (controllable by the following parameters). It then sleeps for bgwriter_delay milliseconds, and repeats. When there are no dirty buffers in the buffer pool, though, it goes into a longer sleep regardless of bgwriter_delay. The default value is 200 milliseconds (200ms). Note that on many systems, the effective resolution of sleep delays is 10 milliseconds; setting bgwriter_delay to a value that is not a multiple of 10 might have the same results as setting it to the next higher multiple of 10. This parameter can only be set in the postgresql.conf file or on the server command line.

bgwriter_lru_maxpages (integer)

In each round, no more than this many buffers will be written by the background writer. Setting this to zero disables background writing. (Note that checkpoints, which are managed by a separate, dedicated auxiliary process, are unaffected.) The default value is 100 buffers. This parameter can only be set in the postgresql.conf file or on the server command line.

bgwriter_lru_multiplier (floating point)

The number of dirty buffers written in each round is based on the number of new buffers that have been needed by server processes during recent rounds. The average recent need is multiplied by bgwriter_lru_multiplier to arrive at an estimate of the number of buffers that will be needed during the next round. Dirty buffers are written until there are that many clean, reusable buffers available. (However, no more than bgwriter_lru_maxpages buffers will be written per round.) Thus, a setting of 1.0 represents a "just in time" policy of writing exactly the number of buffers

predicted to be needed. Larger values provide some cushion against spikes in demand, while smaller values intentionally leave writes to be done by server processes. The default is 2.0. This parameter can only be set in the `postgresql.conf` file or on the server command line.

Smaller values of `bgwriter_lru_maxpages` and `bgwriter_lru_multiplier` reduce the extra I/O load caused by the background writer, but make it more likely that server processes will have to issue writes for themselves, delaying interactive queries.

18.4.6. Asynchronous Behavior

`effective_io_concurrency` (`integer`)

Sets the number of concurrent disk I/O operations that PostgreSQL expects can be executed simultaneously. Raising this value will increase the number of I/O operations that any individual PostgreSQL session attempts to initiate in parallel. The allowed range is 1 to 1000, or zero to disable issuance of asynchronous I/O requests. Currently, this setting only affects bitmap heap scans.

A good starting point for this setting is the number of separate drives comprising a RAID 0 stripe or RAID 1 mirror being used for the database. (For RAID 5 the parity drive should not be counted.) However, if the database is often busy with multiple queries issued in concurrent sessions, lower values may be sufficient to keep the disk array busy. A value higher than needed to keep the disks busy will only result in extra CPU overhead.

For more exotic systems, such as memory-based storage or a RAID array that is limited by bus bandwidth, the correct value might be the number of I/O paths available. Some experimentation may be needed to find the best value.

Asynchronous I/O depends on an effective `posix_fadvise` function, which some operating systems lack. If the function is not present then setting this parameter to anything but zero will result in an error. On some operating systems (e.g., Solaris), the function is present but does not actually do anything.

The default is 1 on supported systems, otherwise 0.

`max_worker_processes` (`integer`)

Sets the maximum number of background processes that the system can support. This parameter can only be set at server start.

When running a standby server, you must set this parameter to the same or higher value than on the master server. Otherwise, queries will not be allowed in the standby server.

18.5. Write Ahead Log

For additional information on tuning these settings, see Section 29.4.

18.5.1. Settings

wal_level (enum)

wal_level determines how much information is written to the WAL. The default value is minimal, which writes only the information needed to recover from a crash or immediate shutdown. archive adds logging required for WAL archiving; hot_standby further adds information required to run read-only queries on a standby server; and, finally logical adds information necessary to support logical decoding. Each level includes the information logged at all lower levels. This parameter can only be set at server start.

In minimal level, WAL-logging of some bulk operations can be safely skipped, which can make those operations much faster (see Section 14.4.7). Operations in which this optimization can be applied include:

CREATE TABLE AS
CREATE INDEX
CLUSTER
COPY into tables that were created or truncated in the same transaction

But minimal WAL does not contain enough information to reconstruct the data from a base backup and the WAL logs, so archive or higher must be used to enable WAL archiving (archive_mode) and streaming replication.

In hot_standby level, the same information is logged as with archive, plus information needed to reconstruct the status of running transactions from the WAL. To enable read-only queries on a standby server, wal_level must be set to hot_standby or higher on the primary, and hot_standby must be enabled in the standby. It is thought that there is little measurable difference in performance between using hot_standby and archive levels, so feedback is welcome if any production impacts are noticeable.

In logical level, the same information is logged as with hot_standby, plus information needed to allow extracting logical change sets from the WAL. Using a level of logical will increase the WAL volume, particularly if many tables are configured for REPLICA IDENTITY FULL and many UPDATE and DELETE statements are executed.

fsync (boolean)

If this parameter is on, the PostgreSQL server will try to make sure that updates are physically written to disk, by issuing fsync() system calls or various equivalent methods (see wal_sync_method). This ensures that the database cluster can recover to a consistent state after an operating system or hardware crash.

While turning off fsync is often a performance benefit, this can result in unrecoverable data corruption in the event of a power failure or system crash. Thus it is only advisable to turn off fsync if you can easily recreate your entire database from external data.

Examples of safe circumstances for turning off fsync include the initial loading of a new database cluster from a backup file, using a database cluster for processing a batch of data after which the database will be thrown away and recreated, or for a read-only database clone which gets recreated frequently and is not used for failover. High quality hardware alone is not a sufficient justification for turning off fsync.

For reliable recovery when changing fsync off to on, it is necessary to force all modified buffers in the kernel to durable storage. This can be done while the cluster is shutdown or while fsync is on by running initdb --sync-only, running sync, unmounting the file system, or rebooting the server.

In many situations, turning off synchronous_commit for noncritical transactions can provide much of the potential performance benefit of turning off fsync, without the attendant risks of data corruption.

fsync can only be set in the postgresql.conf file or on the server command line. If you turn this parameter off, also consider turning off full_page_writes.

synchronous_commit (enum)

Specifies whether transaction commit will wait for WAL records to be written to disk before the command returns a "success" indication to the client. Valid values are on, remote_write, local, and off. The default, and safe, setting is on. When off, there can be a delay between when success is reported to the client and when the transaction is really guaranteed to be safe against a server crash. (The maximum delay is three times wal_writer_delay.) Unlike fsync, setting this parameter to off does not create any risk of database inconsistency: an operating system or database crash might result in some recent allegedly-committed transactions being lost, but the database state will be just the same as if those transactions had been aborted cleanly. So, turning synchronous_commit off can be a useful alternative when performance is more important than exact certainty about the durability of a transaction. For more discussion see Section 29.3.

If synchronous_standby_names is set, this parameter also controls whether or not transaction commits will wait for the transaction's WAL records to be replicated to the standby server. When set to on, commits will wait until a reply from the current synchronous standby indicates it has received the commit record of the transaction and flushed it to disk. This ensures the transaction will not be lost unless both primary and standby suffer corruption of their database storage. When set to remote_write, commits will wait until a reply from the current synchronous standby indicates it has received the commit record of the transaction and written it out to the standby's operating system, but the data has not necessarily reached durable storage on the standby. This setting is sufficient to ensure data preservation even if the standby instance of PostgreSQL were to crash, but not if the standby suffers an operating-system-level crash.

When synchronous replication is in use, it will normally be sensible either to wait for both local flush to disk and replication of WAL records, or to allow the transaction to commit asynchronously. However, the setting local is available for transactions that wish to wait for local flush to disk, but not synchronous replication. If synchronous_standby_names is not set, the settings on, remote_write and local all provide the same synchronization level: transaction commits only wait for local flush to disk.

This parameter can be changed at any time; the behavior for any one transaction is determined by the setting in effect when it commits. It is therefore possible, and useful, to have some transactions commit synchronously and others asynchronously. For example, to make a single multistatement transaction commit asynchronously when the default is the opposite, issue SET LOCAL synchronous_commit TO OFF within the transaction.

wal_sync_method (enum)

Method used for forcing WAL updates out to disk. If fsync is off then this setting is irrelevant, since WAL file updates will not be forced out at all. Possible values are:

- open_datasync (write WAL files with open() option O_DSYNC)
- fdatasync (call fdatasync() at each commit)
- fsync (call fsync() at each commit)

- fsync_writethrough (call fsync() at each commit, forcing write-through of any disk write cache)
- open_sync (write WAL files with open() option O_SYNC)

The open_* options also use O_DIRECT if available. Not all of these choices are available on all platforms. The default is the first method in the above list that is supported by the platform, except that fdatasync is the default on Linux. The default is not necessarily ideal; it might be necessary to change this setting or other aspects of your system configuration in order to create a crash-safe configuration or achieve optimal performance. These aspects are discussed in Section 29.1. This parameter can only be set in the postgresql.conf file or on the server command line.

full_page_writes (boolean)

When this parameter is on, the PostgreSQL server writes the entire content of each disk page to WAL during the first modification of that page after a checkpoint. This is needed because a page write that is in process during an operating system crash might be only partially completed, leading to an on-disk page that contains a mix of old and new data. The row-level change data normally stored in WAL will not be enough to completely restore such a page during post-crash recovery. Storing the full page image guarantees that the page can be correctly restored, but at the price of increasing the amount of data that must be written to WAL. (Because WAL replay always starts from a checkpoint, it is sufficient to do this during the first change of each page after a checkpoint. Therefore, one way to reduce the cost of full-page writes is to increase the checkpoint interval parameters.)

Turning this parameter off speeds normal operation, but might lead to either unrecoverable data corruption, or silent data corruption, after a system failure. The risks are similar to turning off fsync, though smaller, and it should be turned off only based on the same circumstances recommended for that parameter.

Turning off this parameter does not affect use of WAL archiving for point-in-time recovery (PITR) (see Section 24.3).

This parameter can only be set in the postgresql.conf file or on the server command line. The default is on.

wal_log_hints (boolean)

When this parameter is on, the PostgreSQL server writes the entire content of each disk page to WAL during the first modification of that page after a checkpoint, even for non-critical modifications of so-called hint bits.

If data checksums are enabled, hint bit updates are always WAL-logged and this setting is ignored. You can use this setting to test how much extra WAL-logging would occur if your database had data checksums enabled.

This parameter can only be set at server start. The default value is off.

wal_compression (boolean)

When this parameter is on, the PostgreSQL server compresses a full page image written to WAL when full_page_writes is on or during a base backup. A compressed page image will be decompressed during WAL replay. The default value is off. Only superusers can change this setting.

Turning this parameter on can reduce the WAL volume without increasing the risk of unrecoverable data corruption, but at the cost of some extra CPU spent on the compression during WAL logging and on the decompression during WAL replay.

wal_buffers (integer)

The amount of shared memory used for WAL data that has not yet been written to disk. The default setting of -1 selects a size equal to 1/32nd (about 3%) of shared_buffers, but not less than 64kB nor more than the size of one WAL segment, typically 16MB. This value can be set manually if the automatic choice is too large or too small, but any positive value less than 32kB will be treated as 32kB. This parameter can only be set at server start.

The contents of the WAL buffers are written out to disk at every transaction commit, so extremely large values are unlikely to provide a significant benefit. However, setting this value to at least a few megabytes can improve write performance on a busy server where many clients are committing at once. The auto-tuning selected by the default setting of -1 should give reasonable results in most cases.

wal_writer_delay (integer)

Specifies the delay between activity rounds for the WAL writer. In each round the writer will flush WAL to disk. It then sleeps for wal_writer_delay milliseconds, and repeats. The default value is 200 milliseconds (200ms). Note that on many systems, the effective resolution of sleep delays is 10 milliseconds; setting wal_writer_delay to a value that is not a multiple of 10 might have the same results as setting it to the next higher multiple of 10. This parameter can only be set in the postgresql.conf file or on the server command line.

commit_delay (integer)

commit_delay adds a time delay, measured in microseconds, before a WAL flush is initiated. This can improve group commit throughput by allowing a larger number of transactions to commit via a single WAL flush, if system load is high enough that additional transactions become ready to commit within the given interval. However, it also increases latency by up to commit_delay microseconds for each WAL flush. Because the delay is just wasted if no other transactions become ready to commit, a delay is only performed if at least commit_siblings other transactions are active when a flush is about to be initiated. Also, no delays are performed if fsync is disabled. The default commit_delay is zero (no delay). Only superusers can change this setting.

In PostgreSQL releases prior to 9.3, commit_delay behaved differently and was much less effective: it affected only commits, rather than all WAL flushes, and waited for the entire configured delay even if the WAL flush was completed sooner. Beginning in PostgreSQL 9.3, the first process that becomes ready to flush waits for the configured interval, while subsequent processes wait only until the leader completes the flush operation.

commit_siblings (integer)

Minimum number of concurrent open transactions to require before performing the commit_delay delay. A larger value makes it more probable that at least one other transaction will become ready to commit during the delay interval. The default is five transactions.

18.5.2. Checkpoints

checkpoint_timeout (integer)

Maximum time between automatic WAL checkpoints, in seconds. The valid range is between 30 seconds and one hour. The default is five minutes (5min). Increasing this parameter can increase the

amount of time needed for crash recovery. This parameter can only be set in the postgresql.conf file or on the server command line.

checkpoint_completion_target (floating point)

Specifies the target of checkpoint completion, as a fraction of total time between checkpoints. The default is 0.5. This parameter can only be set in the postgresql.conf file or on the server command line.

checkpoint_warning (integer)

Write a message to the server log if checkpoints caused by the filling of checkpoint segment files happen closer together than this many seconds (which suggests that max_wal_size ought to be raised). The default is 30 seconds (30s). Zero disables the warning. No warnings will be generated if checkpoint_timeout is less than checkpoint_warning. This parameter can only be set in the postgresql.conf file or on the server command line.

max_wal_size (integer)

Maximum size to let the WAL grow during automatic checkpoints. This is a soft limit; WAL size can exceed max_wal_size under special circumstances, like under heavy load, a failing archive_command, or a high wal_keep_segments setting. The default is 1 GB. Increasing this parameter can increase the amount of time needed for crash recovery. This parameter can only be set in the postgresql.conf file or on the server command line.

min_wal_size (integer)

As long as WAL disk usage stays below this setting, old WAL files are always recycled for future use at a checkpoint, rather than removed. This can be used to ensure that enough WAL space is reserved to handle spikes in WAL usage, for example when running large batch jobs. The default is 80 MB. This parameter can only be set in the postgresql.conf file or on the server command line.

18.5.3. Archiving

archive_mode (enum)

When archive_mode is enabled, completed WAL segments are sent to archive storage by setting archive_command. In addition to off, to disable, there are two modes: on, and always. During normal operation, there is no difference between the two modes, but when set to always the WAL archiver is enabled also during archive recovery or standby mode. In always mode, all files restored from the archive or streamed with streaming replication will be archived (again). See Section 25.2.9 for details.

archive_mode and archive_command are separate variables so that archive_command can be changed without leaving archiving mode. This parameter can only be set at server start. archive_mode cannot be enabled when wal_level is set to minimal.

archive_command (string)

The local shell command to execute to archive a completed WAL file segment. Any %p in the string is replaced by the path name of the file to archive, and any %f is replaced by only the file name. (The path name is relative to the working directory of the server, i.e., the cluster's data directory.) Use %% to embed an actual % character in the command. It is important for the command to return a zero exit status only if it succeeds. For more information see Section 24.3.1.

This parameter can only be set in the postgresql.conf file or on the server command line. It is ignored unless archive_mode was enabled at server start. If archive_command is an empty string (the default) while archive_mode is enabled, WAL archiving is temporarily disabled, but the server continues to accumulate WAL segment files in the expectation that a command will soon be provided. Setting archive_command to a command that does nothing but return true, e.g., /bin/true (REM on Windows), effectively disables archiving, but also breaks the chain of WAL files needed for archive recovery, so it should only be used in unusual circumstances.

archive_timeout (integer)

The archive_command is only invoked for completed WAL segments. Hence, if your server generates little WAL traffic (or has slack periods where it does so), there could be a long delay between the completion of a transaction and its safe recording in archive storage. To limit how old unarchived data can be, you can set archive_timeout to force the server to switch to a new WAL segment file periodically. When this parameter is greater than zero, the server will switch to a new segment file whenever this many seconds have elapsed since the last segment file switch, and there has been any database activity, including a single checkpoint. (Increasing checkpoint_timeout will reduce unnecessary checkpoints on an idle system.) Note that archived files that are closed early due to a forced switch are still the same length as completely full files. Therefore, it is unwise to use a very short archive_timeout — it will bloat your archive storage. archive_timeout settings of a minute or so are usually reasonable. You should consider using streaming replication, instead of archiving, if you want data to be copied off the master server more quickly than that. This parameter can only be set in the postgresql.conf file or on the server command line.

18.6. Replication

These settings control the behavior of the built-in *streaming replication* feature (see Section 25.2.5). Servers will be either a Master or a Standby server. Masters can send data, while Standby(s) are always receivers of replicated data. When cascading replication (see Section 25.2.7) is used, Standby server(s) can also be senders, as well as receivers. Parameters are mainly for Sending and Standby servers, though some parameters have meaning only on the Master server. Settings may vary across the cluster without problems if that is required.

18.6.1. Sending Server(s)

These parameters can be set on any server that is to send replication data to one or more standby servers. The master is always a sending server, so these parameters must always be set on the master. The role and meaning of these parameters does not change after a standby becomes the master.

max_wal_senders (integer)

Specifies the maximum number of concurrent connections from standby servers or streaming base backup clients (i.e., the maximum number of simultaneously running WAL sender processes). The default is zero, meaning replication is disabled. WAL sender processes count towards the total number of connections, so the parameter cannot be set higher than max_connections. Abrupt streaming client disconnection might cause an orphaned connection slot until a timeout is reached, so this parameter should be set slightly higher than the maximum number of expected clients so disconnected

clients can immediately reconnect. This parameter can only be set at server start. `wal_level` must be set to `archive` or higher to allow connections from standby servers.

`max_replication_slots` (`integer`)

Specifies the maximum number of replication slots (see Section 25.2.6) that the server can support. The default is zero. This parameter can only be set at server start. `wal_level` must be set to `archive` or higher to allow replication slots to be used. Setting it to a lower value than the number of currently existing replication slots will prevent the server from starting.

`wal_keep_segments` (`integer`)

Specifies the minimum number of past log file segments kept in the `pg_xlog` directory, in case a standby server needs to fetch them for streaming replication. Each segment is normally 16 megabytes. If a standby server connected to the sending server falls behind by more than `wal_keep_segments` segments, the sending server might remove a WAL segment still needed by the standby, in which case the replication connection will be terminated. Downstream connections will also eventually fail as a result. (However, the standby server can recover by fetching the segment from archive, if WAL archiving is in use.)

This sets only the minimum number of segments retained in `pg_xlog`; the system might need to retain more segments for WAL archival or to recover from a checkpoint. If `wal_keep_segments` is zero (the default), the system doesn't keep any extra segments for standby purposes, so the number of old WAL segments available to standby servers is a function of the location of the previous checkpoint and status of WAL archiving. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`wal_sender_timeout` (`integer`)

Terminate replication connections that are inactive longer than the specified number of milliseconds. This is useful for the sending server to detect a standby crash or network outage. A value of zero disables the timeout mechanism. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default value is 60 seconds.

`track_commit_timestamp` (`bool`)

Record commit time of transactions. This parameter can only be set in `postgresql.conf` file or on the server command line. The default value is `off`.

18.6.2. Master Server

These parameters can be set on the master/primary server that is to send replication data to one or more standby servers. Note that in addition to these parameters, wal_level must be set appropriately on the master server, and optionally WAL archiving can be enabled as well (see Section 18.5.3). The values of these parameters on standby servers are irrelevant, although you may wish to set them there in preparation for the possibility of a standby becoming the master.

`synchronous_standby_names` (`string`)

Specifies a comma-separated list of standby names that can support *synchronous replication*, as described in Section 25.2.8. At any one time there will be at most one active synchronous standby; transactions waiting for commit will be allowed to proceed after this standby server confirms receipt of their data. The synchronous standby will be the first standby named in this list that is both

currently connected and streaming data in real-time (as shown by a state of `streaming` in the `pg_stat_replication` view). Other standby servers appearing later in this list represent potential synchronous standbys. If the current synchronous standby disconnects for whatever reason, it will be replaced immediately with the next-highest-priority standby. Specifying more than one standby name can allow very high availability.

The name of a standby server for this purpose is the `application_name` setting of the standby, as set in the `primary_conninfo` of the standby's WAL receiver. There is no mechanism to enforce uniqueness. In case of duplicates one of the matching standbys will be chosen to be the synchronous standby, though exactly which one is indeterminate. The special entry `*` matches any `application_name`, including the default application name of `walreceiver`.

If no synchronous standby names are specified here, then synchronous replication is not enabled and transaction commits will not wait for replication. This is the default configuration. Even when synchronous replication is enabled, individual transactions can be configured not to wait for replication by setting the synchronous_commit parameter to `local` or `off`.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

`vacuum_defer_cleanup_age` (`integer`)

Specifies the number of transactions by which `VACUUM` and HOT updates will defer cleanup of dead row versions. The default is zero transactions, meaning that dead row versions can be removed as soon as possible, that is, as soon as they are no longer visible to any open transaction. You may wish to set this to a non-zero value on a primary server that is supporting hot standby servers, as described in Section 25.5. This allows more time for queries on the standby to complete without incurring conflicts due to early cleanup of rows. However, since the value is measured in terms of number of write transactions occurring on the primary server, it is difficult to predict just how much additional grace time will be made available to standby queries. This parameter can only be set in the `postgresql.conf` file or on the server command line.

You should also consider setting `hot_standby_feedback` on standby server(s) as an alternative to using this parameter.

18.6.3. Standby Servers

These settings control the behavior of a standby server that is to receive replication data. Their values on the master server are irrelevant.

`hot_standby` (`boolean`)

Specifies whether or not you can connect and run queries during recovery, as described in Section 25.5. The default value is `off`. This parameter can only be set at server start. It only has effect during archive recovery or in standby mode.

`max_standby_archive_delay` (`integer`)

When Hot Standby is active, this parameter determines how long the standby server should wait before canceling standby queries that conflict with about-to-be-applied WAL entries, as described in Section 25.5.2. `max_standby_archive_delay` applies when WAL data is being read from WAL archive (and is therefore not current). The default is 30 seconds. Units are milliseconds if not specified. A value of -1 allows the standby to wait forever for conflicting queries to complete. This parameter can only be set in the `postgresql.conf` file or on the server command line.

Note that `max_standby_archive_delay` is not the same as the maximum length of time a query can run before cancellation; rather it is the maximum total time allowed to apply any one WAL segment's data. Thus, if one query has resulted in significant delay earlier in the WAL segment, subsequent conflicting queries will have much less grace time.

`max_standby_streaming_delay` (`integer`)

When Hot Standby is active, this parameter determines how long the standby server should wait before canceling standby queries that conflict with about-to-be-applied WAL entries, as described in Section 25.5.2. `max_standby_streaming_delay` applies when WAL data is being received via streaming replication. The default is 30 seconds. Units are milliseconds if not specified. A value of -1 allows the standby to wait forever for conflicting queries to complete. This parameter can only be set in the `postgresql.conf` file or on the server command line.

Note that `max_standby_streaming_delay` is not the same as the maximum length of time a query can run before cancellation; rather it is the maximum total time allowed to apply WAL data once it has been received from the primary server. Thus, if one query has resulted in significant delay, subsequent conflicting queries will have much less grace time until the standby server has caught up again.

`wal_receiver_status_interval` (`integer`)

Specifies the minimum frequency for the WAL receiver process on the standby to send information about replication progress to the primary or upstream standby, where it can be seen using the `pg_stat_replication` view. The standby will report the last transaction log position it has written, the last position it has flushed to disk, and the last position it has applied. This parameter's value is the maximum interval, in seconds, between reports. Updates are sent each time the write or flush positions change, or at least as often as specified by this parameter. Thus, the apply position may lag slightly behind the true position. Setting this parameter to zero disables status updates completely. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default value is 10 seconds.

`hot_standby_feedback` (`boolean`)

Specifies whether or not a hot standby will send feedback to the primary or upstream standby about queries currently executing on the standby. This parameter can be used to eliminate query cancels caused by cleanup records, but can cause database bloat on the primary for some workloads. Feedback messages will not be sent more frequently than once per `wal_receiver_status_interval`. The default value is `off`. This parameter can only be set in the `postgresql.conf` file or on the server command line.

If cascaded replication is in use the feedback is passed upstream until it eventually reaches the primary. Standbys make no other use of feedback they receive other than to pass upstream.

`wal_receiver_timeout` (`integer`)

Terminate replication connections that are inactive longer than the specified number of milliseconds. This is useful for the receiving standby server to detect a primary node crash or network outage. A value of zero disables the timeout mechanism. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default value is 60 seconds.

`wal_retrieve_retry_interval` (`integer`)

Specify how long the standby server should wait when WAL data is not available from any sources (streaming replication, local `pg_xlog` or WAL archive) before retrying to retrieve WAL data. This

parameter can only be set in the `postgresql.conf` file or on the server command line. The default value is 5 seconds. Units are milliseconds if not specified.

This parameter is useful in configurations where a node in recovery needs to control the amount of time to wait for new WAL data to be available. For example, in archive recovery, it is possible to make the recovery more responsive in the detection of a new WAL log file by reducing the value of this parameter. On a system with low WAL activity, increasing it reduces the amount of requests necessary to access WAL archives, something useful for example in cloud environments where the amount of times an infrastructure is accessed is taken into account.

18.7. Query Planning

18.7.1. Planner Method Configuration

These configuration parameters provide a crude method of influencing the query plans chosen by the query optimizer. If the default plan chosen by the optimizer for a particular query is not optimal, a *temporary* solution is to use one of these configuration parameters to force the optimizer to choose a different plan. Better ways to improve the quality of the plans chosen by the optimizer include adjusting the planer cost constants (see Section 18.7.2), running ANALYZE manually, increasing the value of the default_statistics_target configuration parameter, and increasing the amount of statistics collected for specific columns using `ALTER TABLE SET STATISTICS`.

`enable_bitmapscan` (`boolean`)

Enables or disables the query planner's use of bitmap-scan plan types. The default is `on`.

`enable_hashagg` (`boolean`)

Enables or disables the query planner's use of hashed aggregation plan types. The default is `on`.

`enable_hashjoin` (`boolean`)

Enables or disables the query planner's use of hash-join plan types. The default is `on`.

`enable_indexscan` (`boolean`)

Enables or disables the query planner's use of index-scan plan types. The default is `on`.

`enable_indexonlyscan` (`boolean`)

Enables or disables the query planner's use of index-only-scan plan types. The default is `on`.

`enable_material` (`boolean`)

Enables or disables the query planner's use of materialization. It is impossible to suppress materialization entirely, but turning this variable off prevents the planner from inserting materialize nodes except in cases where it is required for correctness. The default is `on`.

`enable_mergejoin` (`boolean`)

Enables or disables the query planner's use of merge-join plan types. The default is `on`.

`enable_nestloop` (`boolean`)

Enables or disables the query planner's use of nested-loop join plans. It is impossible to suppress nested-loop joins entirely, but turning this variable off discourages the planner from using one if there are other methods available. The default is `on`.

`enable_seqscan` (`boolean`)

Enables or disables the query planner's use of sequential scan plan types. It is impossible to suppress sequential scans entirely, but turning this variable off discourages the planner from using one if there are other methods available. The default is `on`.

`enable_sort` (`boolean`)

Enables or disables the query planner's use of explicit sort steps. It is impossible to suppress explicit sorts entirely, but turning this variable off discourages the planner from using one if there are other methods available. The default is `on`.

`enable_tidscan` (`boolean`)

Enables or disables the query planner's use of TID scan plan types. The default is `on`.

18.7.2. Planner Cost Constants

The *cost* variables described in this section are measured on an arbitrary scale. Only their relative values matter, hence scaling them all up or down by the same factor will result in no change in the planner's choices. By default, these cost variables are based on the cost of sequential page fetches; that is, `seq_page_cost` is conventionally set to `1.0` and the other cost variables are set with reference to that. But you can use a different scale if you prefer, such as actual execution times in milliseconds on a particular machine.

Note: Unfortunately, there is no well-defined method for determining ideal values for the cost variables. They are best treated as averages over the entire mix of queries that a particular installation will receive. This means that changing them on the basis of just a few experiments is very risky.

`seq_page_cost` (`floating point`)

Sets the planner's estimate of the cost of a disk page fetch that is part of a series of sequential fetches. The default is 1.0. This value can be overridden for tables and indexes in a particular tablespace by setting the tablespace parameter of the same name (see ALTER TABLESPACE).

`random_page_cost` (`floating point`)

Sets the planner's estimate of the cost of a non-sequentially-fetched disk page. The default is 4.0. This value can be overridden for tables and indexes in a particular tablespace by setting the tablespace parameter of the same name (see ALTER TABLESPACE).

Reducing this value relative to `seq_page_cost` will cause the system to prefer index scans; raising it will make index scans look relatively more expensive. You can raise or lower both values together to change the importance of disk I/O costs relative to CPU costs, which are described by the following parameters.

Random access to mechanical disk storage is normally much more expensive than four times sequential access. However, a lower default is used (4.0) because the majority of random accesses to disk, such as indexed reads, are assumed to be in cache. The default value can be thought of as modeling random access as 40 times slower than sequential, while expecting 90% of random reads to be cached.

If you believe a 90% cache rate is an incorrect assumption for your workload, you can increase random_page_cost to better reflect the true cost of random storage reads. Correspondingly, if your data is likely to be completely in cache, such as when the database is smaller than the total server memory, decreasing random_page_cost can be appropriate. Storage that has a low random read cost relative to sequential, e.g., solid-state drives, might also be better modeled with a lower value for random_page_cost, e.g., `1.1`.

> **Tip:** Although the system will let you set `random_page_cost` to less than `seq_page_cost`, it is not physically sensible to do so. However, setting them equal makes sense if the database is entirely cached in RAM, since in that case there is no penalty for touching pages out of sequence. Also, in a heavily-cached database you should lower both values relative to the CPU parameters, since the cost of fetching a page already in RAM is much smaller than it would normally be.

`cpu_tuple_cost` (`floating point`)

Sets the planner's estimate of the cost of processing each row during a query. The default is 0.01.

`cpu_index_tuple_cost` (`floating point`)

Sets the planner's estimate of the cost of processing each index entry during an index scan. The default is 0.005.

`cpu_operator_cost` (`floating point`)

Sets the planner's estimate of the cost of processing each operator or function executed during a query. The default is 0.0025.

`effective_cache_size` (`integer`)

Sets the planner's assumption about the effective size of the disk cache that is available to a single query. This is factored into estimates of the cost of using an index; a higher value makes it more likely index scans will be used, a lower value makes it more likely sequential scans will be used. When setting this parameter you should consider both PostgreSQL's shared buffers and the portion of the kernel's disk cache that will be used for PostgreSQL data files, though some data might exist in both places. Also, take into account the expected number of concurrent queries on different tables, since they will have to share the available space. This parameter has no effect on the size of shared memory allocated by PostgreSQL, nor does it reserve kernel disk cache; it is used only for estimation purposes. The system also does not assume data remains in the disk cache between queries. The default is 4 gigabytes (`4GB`).

18.7.3. Genetic Query Optimizer

The genetic query optimizer (GEQO) is an algorithm that does query planning using heuristic searching. This reduces planning time for complex queries (those joining many relations), at the cost of producing

plans that are sometimes inferior to those found by the normal exhaustive-search algorithm. For more information see Chapter 57.

`geqo` (`boolean`)

Enables or disables genetic query optimization. This is on by default. It is usually best not to turn it off in production; the `geqo_threshold` variable provides more granular control of GEQO.

`geqo_threshold` (`integer`)

Use genetic query optimization to plan queries with at least this many `FROM` items involved. (Note that a `FULL OUTER JOIN` construct counts as only one `FROM` item.) The default is 12. For simpler queries it is usually best to use the regular, exhaustive-search planner, but for queries with many tables the exhaustive search takes too long, often longer than the penalty of executing a suboptimal plan. Thus, a threshold on the size of the query is a convenient way to manage use of GEQO.

`geqo_effort` (`integer`)

Controls the trade-off between planning time and query plan quality in GEQO. This variable must be an integer in the range from 1 to 10. The default value is five. Larger values increase the time spent doing query planning, but also increase the likelihood that an efficient query plan will be chosen.

`geqo_effort` doesn't actually do anything directly; it is only used to compute the default values for the other variables that influence GEQO behavior (described below). If you prefer, you can set the other parameters by hand instead.

`geqo_pool_size` (`integer`)

Controls the pool size used by GEQO, that is the number of individuals in the genetic population. It must be at least two, and useful values are typically 100 to 1000. If it is set to zero (the default setting) then a suitable value is chosen based on `geqo_effort` and the number of tables in the query.

`geqo_generations` (`integer`)

Controls the number of generations used by GEQO, that is the number of iterations of the algorithm. It must be at least one, and useful values are in the same range as the pool size. If it is set to zero (the default setting) then a suitable value is chosen based on `geqo_pool_size`.

`geqo_selection_bias` (`floating point`)

Controls the selection bias used by GEQO. The selection bias is the selective pressure within the population. Values can be from 1.50 to 2.00; the latter is the default.

`geqo_seed` (`floating point`)

Controls the initial value of the random number generator used by GEQO to select random paths through the join order search space. The value can range from zero (the default) to one. Varying the value changes the set of join paths explored, and may result in a better or worse best path being found.

18.7.4. Other Planner Options

`default_statistics_target` (`integer`)

Sets the default statistics target for table columns without a column-specific target set via `ALTER TABLE SET STATISTICS`. Larger values increase the time needed to do `ANALYZE`, but might im-

prove the quality of the planner's estimates. The default is 100. For more information on the use of statistics by the PostgreSQL query planner, refer to Section 14.2.

`constraint_exclusion` (`enum`)

Controls the query planner's use of table constraints to optimize queries. The allowed values of `constraint_exclusion` are `on` (examine constraints for all tables), `off` (never examine constraints), and `partition` (examine constraints only for inheritance child tables and `UNION ALL` subqueries). `partition` is the default setting. It is often used with inheritance and partitioned tables to improve performance.

When this parameter allows it for a particular table, the planner compares query conditions with the table's `CHECK` constraints, and omits scanning tables for which the conditions contradict the constraints. For example:

```
CREATE TABLE parent(key integer, ...);
CREATE TABLE child1000(check (key between 1000 and 1999)) INHERITS(parent);
CREATE TABLE child2000(check (key between 2000 and 2999)) INHERITS(parent);
...
SELECT * FROM parent WHERE key = 2400;
```

With constraint exclusion enabled, this `SELECT` will not scan `child1000` at all, improving performance.

Currently, constraint exclusion is enabled by default only for cases that are often used to implement table partitioning. Turning it on for all tables imposes extra planning overhead that is quite noticeable on simple queries, and most often will yield no benefit for simple queries. If you have no partitioned tables you might prefer to turn it off entirely.

Refer to Section 5.10.4 for more information on using constraint exclusion and partitioning.

`cursor_tuple_fraction` (`floating point`)

Sets the planner's estimate of the fraction of a cursor's rows that will be retrieved. The default is 0.1. Smaller values of this setting bias the planner towards using "fast start" plans for cursors, which will retrieve the first few rows quickly while perhaps taking a long time to fetch all rows. Larger values put more emphasis on the total estimated time. At the maximum setting of 1.0, cursors are planned exactly like regular queries, considering only the total estimated time and not how soon the first rows might be delivered.

`from_collapse_limit` (`integer`)

The planner will merge sub-queries into upper queries if the resulting `FROM` list would have no more than this many items. Smaller values reduce planning time but might yield inferior query plans. The default is eight. For more information see Section 14.3.

Setting this value to geqo_threshold or more may trigger use of the GEQO planner, resulting in non-optimal plans. See Section 18.7.3.

`join_collapse_limit` (`integer`)

The planner will rewrite explicit `JOIN` constructs (except `FULL JOIN`s) into lists of `FROM` items whenever a list of no more than this many items would result. Smaller values reduce planning time but might yield inferior query plans.

By default, this variable is set the same as `from_collapse_limit`, which is appropriate for most uses. Setting it to 1 prevents any reordering of explicit `JOIN`s. Thus, the explicit join order specified in the query will be the actual order in which the relations are joined. Because the query planner does

not always choose the optimal join order, advanced users can elect to temporarily set this variable to 1, and then specify the join order they desire explicitly. For more information see Section 14.3.

Setting this value to geqo_threshold or more may trigger use of the GEQO planner, resulting in non-optimal plans. See Section 18.7.3.

18.8. Error Reporting and Logging

18.8.1. Where To Log

`log_destination` (`string`)

PostgreSQL supports several methods for logging server messages, including stderr, csvlog and syslog. On Windows, eventlog is also supported. Set this parameter to a list of desired log destinations separated by commas. The default is to log to stderr only. This parameter can only be set in the `postgresql.conf` file or on the server command line.

If csvlog is included in `log_destination`, log entries are output in "comma separated value" (CSV) format, which is convenient for loading logs into programs. See Section 18.8.4 for details. logging_collector must be enabled to generate CSV-format log output.

> **Note:** On most Unix systems, you will need to alter the configuration of your system's syslog daemon in order to make use of the syslog option for `log_destination`. PostgreSQL can log to syslog facilities `LOCAL0` through `LOCAL7` (see syslog_facility), but the default syslog configuration on most platforms will discard all such messages. You will need to add something like:
>
> ```
> local0.*    /var/log/postgresql
> ```
>
> to the syslog daemon's configuration file to make it work.
>
> On Windows, when you use the `eventlog` option for `log_destination`, you should register an event source and its library with the operating system so that the Windows Event Viewer can display event log messages cleanly. See Section 17.11 for details.

`logging_collector` (`boolean`)

This parameter enables the *logging collector*, which is a background process that captures log messages sent to stderr and redirects them into log files. This approach is often more useful than logging to syslog, since some types of messages might not appear in syslog output. (One common example is dynamic-linker failure messages; another is error messages produced by scripts such as `archive_command`.) This parameter can only be set at server start.

> **Note:** It is possible to log to stderr without using the logging collector; the log messages will just go to wherever the server's stderr is directed. However, that method is only suitable for low log volumes, since it provides no convenient way to rotate log files. Also, on some platforms not using the logging collector can result in lost or garbled log output, because multiple processes writing concurrently to the same log file can overwrite each other's output.

Note: The logging collector is designed to never lose messages. This means that in case of extremely high load, server processes could be blocked while trying to send additional log messages when the collector has fallen behind. In contrast, syslog prefers to drop messages if it cannot write them, which means it may fail to log some messages in such cases but it will not block the rest of the system.

`log_directory` (`string`)

When `logging_collector` is enabled, this parameter determines the directory in which log files will be created. It can be specified as an absolute path, or relative to the cluster data directory. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default is `pg_log`.

`log_filename` (`string`)

When `logging_collector` is enabled, this parameter sets the file names of the created log files. The value is treated as a `strftime` pattern, so `%`-escapes can be used to specify time-varying file names. (Note that if there are any time-zone-dependent `%`-escapes, the computation is done in the zone specified by log_timezone.) The supported `%`-escapes are similar to those listed in the Open Group's strftime [1] specification. Note that the system's `strftime` is not used directly, so platform-specific (nonstandard) extensions do not work. The default is `postgresql-%Y-%m-%d_%H%M%S.log`.

If you specify a file name without escapes, you should plan to use a log rotation utility to avoid eventually filling the entire disk. In releases prior to 8.4, if no `%` escapes were present, PostgreSQL would append the epoch of the new log file's creation time, but this is no longer the case.

If CSV-format output is enabled in `log_destination`, `.csv` will be appended to the timestamped log file name to create the file name for CSV-format output. (If `log_filename` ends in `.log`, the suffix is replaced instead.)

This parameter can only be set in the `postgresql.conf` file or on the server command line.

`log_file_mode` (`integer`)

On Unix systems this parameter sets the permissions for log files when `logging_collector` is enabled. (On Microsoft Windows this parameter is ignored.) The parameter value is expected to be a numeric mode specified in the format accepted by the `chmod` and `umask` system calls. (To use the customary octal format the number must start with a `0` (zero).)

The default permissions are `0600`, meaning only the server owner can read or write the log files. The other commonly useful setting is `0640`, allowing members of the owner's group to read the files. Note however that to make use of such a setting, you'll need to alter log_directory to store the files somewhere outside the cluster data directory. In any case, it's unwise to make the log files world-readable, since they might contain sensitive data.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

`log_rotation_age` (`integer`)

When `logging_collector` is enabled, this parameter determines the maximum lifetime of an individual log file. After this many minutes have elapsed, a new log file will be created. Set to zero to dis-

1. https://pubs.opengroup.org/onlinepubs/009695399/functions/strftime.html

able time-based creation of new log files. This parameter can only be set in the postgresql.conf file or on the server command line.

log_rotation_size (integer)

When logging_collector is enabled, this parameter determines the maximum size of an individual log file. After this many kilobytes have been emitted into a log file, a new log file will be created. Set to zero to disable size-based creation of new log files. This parameter can only be set in the postgresql.conf file or on the server command line.

log_truncate_on_rotation (boolean)

When logging_collector is enabled, this parameter will cause PostgreSQL to truncate (overwrite), rather than append to, any existing log file of the same name. However, truncation will occur only when a new file is being opened due to time-based rotation, not during server startup or size-based rotation. When off, pre-existing files will be appended to in all cases. For example, using this setting in combination with a log_filename like postgresql-%H.log would result in generating twenty-four hourly log files and then cyclically overwriting them. This parameter can only be set in the postgresql.conf file or on the server command line.

Example: To keep 7 days of logs, one log file per day named server_log.Mon, server_log.Tue, etc, and automatically overwrite last week's log with this week's log, set log_filename to server_log.%a, log_truncate_on_rotation to on, and log_rotation_age to 1440.

Example: To keep 24 hours of logs, one log file per hour, but also rotate sooner if the log file size exceeds 1GB, set log_filename to server_log.%H%M, log_truncate_on_rotation to on, log_rotation_age to 60, and log_rotation_size to 1000000. Including %M in log_filename allows any size-driven rotations that might occur to select a file name different from the hour's initial file name.

syslog_facility (enum)

When logging to syslog is enabled, this parameter determines the syslog "facility" to be used. You can choose from LOCAL0, LOCAL1, LOCAL2, LOCAL3, LOCAL4, LOCAL5, LOCAL6, LOCAL7; the default is LOCAL0. See also the documentation of your system's syslog daemon. This parameter can only be set in the postgresql.conf file or on the server command line.

syslog_ident (string)

When logging to syslog is enabled, this parameter determines the program name used to identify PostgreSQL messages in syslog logs. The default is postgres. This parameter can only be set in the postgresql.conf file or on the server command line.

event_source (string)

When logging to event log is enabled, this parameter determines the program name used to identify PostgreSQL messages in the log. The default is PostgreSQL. This parameter can only be set in the postgresql.conf file or on the server command line.

18.8.2. When To Log

`log_min_messages` (`enum`)

Controls which message levels are written to the server log. Valid values are `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL`, and `PANIC`. Each level includes all the levels that follow it. The later the level, the fewer messages are sent to the log. The default is `WARNING`. Note that `LOG` has a different rank here than in client_min_messages. Only superusers can change this setting.

`log_min_error_statement` (`enum`)

Controls which SQL statements that cause an error condition are recorded in the server log. The current SQL statement is included in the log entry for any message of the specified severity or higher. Valid values are `DEBUG5`, `DEBUG4`, `DEBUG3`, `DEBUG2`, `DEBUG1`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, `LOG`, `FATAL`, and `PANIC`. The default is `ERROR`, which means statements causing errors, log messages, fatal errors, or panics will be logged. To effectively turn off logging of failing statements, set this parameter to `PANIC`. Only superusers can change this setting.

`log_min_duration_statement` (`integer`)

Causes the duration of each completed statement to be logged if the statement ran for at least the specified number of milliseconds. Setting this to zero prints all statement durations. Minus-one (the default) disables logging statement durations. For example, if you set it to `250ms` then all SQL statements that run 250ms or longer will be logged. Enabling this parameter can be helpful in tracking down unoptimized queries in your applications. Only superusers can change this setting.

For clients using extended query protocol, durations of the Parse, Bind, and Execute steps are logged independently.

> **Note:** When using this option together with log_statement, the text of statements that are logged because of `log_statement` will not be repeated in the duration log message. If you are not using syslog, it is recommended that you log the PID or session ID using log_line_prefix so that you can link the statement message to the later duration message using the process ID or session ID.

Table 18-1 explains the message severity levels used by PostgreSQL. If logging output is sent to syslog or Windows' eventlog, the severity levels are translated as shown in the table.

Table 18-1. Message Severity Levels

Severity	Usage	syslog	eventlog
`DEBUG1..DEBUG5`	Provides successively-more-detailed information for use by developers.	`DEBUG`	`INFORMATION`
`INFO`	Provides information implicitly requested by the user, e.g., output from `VACUUM VERBOSE`.	`INFO`	`INFORMATION`

Severity	Usage	syslog	eventlog
NOTICE	Provides information that might be helpful to users, e.g., notice of truncation of long identifiers.	NOTICE	INFORMATION
WARNING	Provides warnings of likely problems, e.g., COMMIT outside a transaction block.	NOTICE	WARNING
ERROR	Reports an error that caused the current command to abort.	WARNING	ERROR
LOG	Reports information of interest to administrators, e.g., checkpoint activity.	INFO	INFORMATION
FATAL	Reports an error that caused the current session to abort.	ERR	ERROR
PANIC	Reports an error that caused all database sessions to abort.	CRIT	ERROR

18.8.3. What To Log

application_name (string)

The application_name can be any string of less than NAMEDATALEN characters (64 characters in a standard build). It is typically set by an application upon connection to the server. The name will be displayed in the pg_stat_activity view and included in CSV log entries. It can also be included in regular log entries via the log_line_prefix parameter. Only printable ASCII characters may be used in the application_name value. Other characters will be replaced with question marks (?).

debug_print_parse (boolean)
debug_print_rewritten (boolean)
debug_print_plan (boolean)

These parameters enable various debugging output to be emitted. When set, they print the resulting parse tree, the query rewriter output, or the execution plan for each executed query. These messages are emitted at LOG message level, so by default they will appear in the server log but will not be sent to the client. You can change that by adjusting client_min_messages and/or log_min_messages. These parameters are off by default.

debug_pretty_print (boolean)

When set, debug_pretty_print indents the messages produced by debug_print_parse,

`debug_print_rewritten`, or `debug_print_plan`. This results in more readable but much longer output than the "compact" format used when it is off. It is on by default.

`log_checkpoints` (`boolean`)

Causes checkpoints and restartpoints to be logged in the server log. Some statistics are included in the log messages, including the number of buffers written and the time spent writing them. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default is off.

`log_connections` (`boolean`)

Causes each attempted connection to the server to be logged, as well as successful completion of client authentication. Only superusers can change this parameter at session start, and it cannot be changed at all within a session. The default is `off`.

> **Note:** Some client programs, like psql, attempt to connect twice while determining if a password is required, so duplicate "connection received" messages do not necessarily indicate a problem.

`log_disconnections` (`boolean`)

Causes session terminations to be logged. The log output provides information similar to `log_connections`, plus the duration of the session. Only superusers can change this parameter at session start, and it cannot be changed at all within a session. The default is `off`.

`log_duration` (`boolean`)

Causes the duration of every completed statement to be logged. The default is `off`. Only superusers can change this setting.

For clients using extended query protocol, durations of the Parse, Bind, and Execute steps are logged independently.

> **Note:** The difference between setting this option and setting log_min_duration_statement to zero is that exceeding `log_min_duration_statement` forces the text of the query to be logged, but this option doesn't. Thus, if `log_duration` is `on` and `log_min_duration_statement` has a positive value, all durations are logged but the query text is included only for statements exceeding the threshold. This behavior can be useful for gathering statistics in high-load installations.

`log_error_verbosity` (`enum`)

Controls the amount of detail written in the server log for each message that is logged. Valid values are `TERSE`, `DEFAULT`, and `VERBOSE`, each adding more fields to displayed messages. `TERSE` excludes the logging of `DETAIL`, `HINT`, `QUERY`, and `CONTEXT` error information. `VERBOSE` output includes the `SQLSTATE` error code (see also Appendix A) and the source code file name, function name, and line number that generated the error. Only superusers can change this setting.

`log_hostname` (`boolean`)

By default, connection log messages only show the IP address of the connecting host. Turning this parameter on causes logging of the host name as well. Note that depending on your host name resolution setup this might impose a non-negligible performance penalty. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`log_line_prefix` (`string`)

This is a `printf`-style string that is output at the beginning of each log line. `%` characters begin "escape sequences" that are replaced with status information as outlined below. Unrecognized escapes are ignored. Other characters are copied straight to the log line. Some escapes are only recognized by session processes, and will be treated as empty by background processes such as the main server process. Status information may be aligned either left or right by specifying a numeric literal after the % and before the option. A negative value will cause the status information to be padded on the right with spaces to give it a minimum width, whereas a positive value will pad on the left. Padding can be useful to aid human readability in log files. This parameter can only be set in the `postgresql.conf` file or on the server command line. The default is an empty string.

Escape	Effect	Session only
`%a`	Application name	yes
`%u`	User name	yes
`%d`	Database name	yes
`%r`	Remote host name or IP address, and remote port	yes
`%h`	Remote host name or IP address	yes
`%p`	Process ID	no
`%t`	Time stamp without milliseconds	no
`%m`	Time stamp with milliseconds	no
`%i`	Command tag: type of session's current command	yes
`%e`	SQLSTATE error code	no
`%c`	Session ID: see below	no
`%l`	Number of the log line for each session or process, starting at 1	no
`%s`	Process start time stamp	no
`%v`	Virtual transaction ID (backendID/localXID)	no
`%x`	Transaction ID (0 if none is assigned)	no
`%q`	Produces no output, but tells non-session processes to stop at this point in the string; ignored by session processes	no
`%%`	Literal `%`	no

The `%c` escape prints a quasi-unique session identifier, consisting of two 4-byte hexadecimal numbers (without leading zeros) separated by a dot. The numbers are the process start time and the process ID, so `%c` can also be used as a space saving way of printing those items. For example, to generate the session identifier from `pg_stat_activity`, use this query:

```
SELECT to_hex(trunc(EXTRACT(EPOCH FROM backend_start))::integer) || '.' ||
       to_hex(pid)
FROM pg_stat_activity;
```

Tip: If you set a nonempty value for `log_line_prefix`, you should usually make its last character be a space, to provide visual separation from the rest of the log line. A punctuation character can be used too.

Tip: Syslog produces its own time stamp and process ID information, so you probably do not want to include those escapes if you are logging to syslog.

`log_lock_waits` (`boolean`)

Controls whether a log message is produced when a session waits longer than deadlock_timeout to acquire a lock. This is useful in determining if lock waits are causing poor performance. The default is `off`. Only superusers can change this setting.

`log_statement` (`enum`)

Controls which SQL statements are logged. Valid values are `none` (off), `ddl`, `mod`, and `all` (all statements). `ddl` logs all data definition statements, such as `CREATE`, `ALTER`, and `DROP` statements. `mod` logs all `ddl` statements, plus data-modifying statements such as `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, and `COPY FROM`. `PREPARE`, `EXECUTE`, and `EXPLAIN ANALYZE` statements are also logged if their contained command is of an appropriate type. For clients using extended query protocol, logging occurs when an Execute message is received, and values of the Bind parameters are included (with any embedded single-quote marks doubled).

The default is `none`. Only superusers can change this setting.

Note: Statements that contain simple syntax errors are not logged even by the `log_statement = all` setting, because the log message is emitted only after basic parsing has been done to determine the statement type. In the case of extended query protocol, this setting likewise does not log statements that fail before the Execute phase (i.e., during parse analysis or planning). Set `log_min_error_statement` to `ERROR` (or lower) to log such statements.

`log_replication_commands` (`boolean`)

Causes each replication command to be logged in the server log. See Section 50.3 for more information about replication command. The default value is `off`. Only superusers can change this setting.

`log_temp_files` (`integer`)

Controls logging of temporary file names and sizes. Temporary files can be created for sorts, hashes, and temporary query results. A log entry is made for each temporary file when it is deleted. A value of zero logs all temporary file information, while positive values log only files whose size is greater than or equal to the specified number of kilobytes. The default setting is -1, which disables such logging. Only superusers can change this setting.

log_timezone (string)

Sets the time zone used for timestamps written in the server log. Unlike TimeZone, this value is cluster-wide, so that all sessions will report timestamps consistently. The built-in default is GMT, but that is typically overridden in postgresql.conf; initdb will install a setting there corresponding to its system environment. See Section 8.5.3 for more information. This parameter can only be set in the postgresql.conf file or on the server command line.

18.8.4. Using CSV-Format Log Output

Including csvlog in the log_destination list provides a convenient way to import log files into a database table. This option emits log lines in comma-separated-values (CSV) format, with these columns: time stamp with milliseconds, user name, database name, process ID, client host:port number, session ID, per-session line number, command tag, session start time, virtual transaction ID, regular transaction ID, error severity, SQLSTATE code, error message, error message detail, hint, internal query that led to the error (if any), character count of the error position therein, error context, user query that led to the error (if any and enabled by log_min_error_statement), character count of the error position therein, location of the error in the PostgreSQL source code (if log_error_verbosity is set to verbose), and application name. Here is a sample table definition for storing CSV-format log output:

```
CREATE TABLE postgres_log
(
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text,
  PRIMARY KEY (session_id, session_line_num)
);
```

To import a log file into this table, use the COPY FROM command:

```
COPY postgres_log FROM '/full/path/to/logfile.csv' WITH csv;
```

It is also possible to access the file as a foreign table, using the supplied file_fdw module.

There are a few things you need to do to simplify importing CSV log files:

1. Set `log_filename` and `log_rotation_age` to provide a consistent, predictable naming scheme for your log files. This lets you predict what the file name will be and know when an individual log file is complete and therefore ready to be imported.
2. Set `log_rotation_size` to 0 to disable size-based log rotation, as it makes the log file name difficult to predict.
3. Set `log_truncate_on_rotation` to `on` so that old log data isn't mixed with the new in the same file.
4. The table definition above includes a primary key specification. This is useful to protect against accidentally importing the same information twice. The `COPY` command commits all of the data it imports at one time, so any error will cause the entire import to fail. If you import a partial log file and later import the file again when it is complete, the primary key violation will cause the import to fail. Wait until the log is complete and closed before importing. This procedure will also protect against accidentally importing a partial line that hasn't been completely written, which would also cause `COPY` to fail.

18.8.5. Process Title

These settings control how the process title as seen by `ps` is modified. See Section 27.1 for details.

`cluster_name` (`string`)

Sets the cluster name that appears in the process title for all processes in this cluster. The name can be any string of less than `NAMEDATALEN` characters (64 characters in a standard build). Only printable ASCII characters may be used in the `cluster_name` value. Other characters will be replaced with question marks (`?`). No name is shown if this parameter is set to the empty string `''` (which is the default). This parameter can only be set at server start.

The process title is typically viewed using programs like ps or, on Windows, Process Explorer.

`update_process_title` (`boolean`)

Enables updating of the process title every time a new SQL command is received by the server. The process title is typically viewed by the `ps` command, or in Windows by using the Process Explorer. Only superusers can change this setting.

18.9. Run-time Statistics

18.9.1. Query and Index Statistics Collector

These parameters control server-wide statistics collection features. When statistics collection is enabled, the data that is produced can be accessed via the `pg_stat` and `pg_statio` family of system views. Refer to Chapter 27 for more information.

`track_activities` (`boolean`)

Enables the collection of information on the currently executing command of each session, along with the time when that command began execution. This parameter is on by default. Note that even when enabled, this information is not visible to all users, only to superusers and the user owning the session being reported on, so it should not represent a security risk. Only superusers can change this setting.

`track_activity_query_size` (`integer`)

Specifies the number of bytes reserved to track the currently executing command for each active session, for the `pg_stat_activity.query` field. The default value is 1024. This parameter can only be set at server start.

`track_counts` (`boolean`)

Enables collection of statistics on database activity. This parameter is on by default, because the autovacuum daemon needs the collected information. Only superusers can change this setting.

`track_io_timing` (`boolean`)

Enables timing of database I/O calls. This parameter is off by default, because it will repeatedly query the operating system for the current time, which may cause significant overhead on some platforms. You can use the pg_test_timing tool to measure the overhead of timing on your system. I/O timing information is displayed in pg_stat_database, in the output of EXPLAIN when the `BUFFERS` option is used, and by pg_stat_statements. Only superusers can change this setting.

`track_functions` (`enum`)

Enables tracking of function call counts and time used. Specify `pl` to track only procedural-language functions, `all` to also track SQL and C language functions. The default is `none`, which disables function statistics tracking. Only superusers can change this setting.

> **Note:** SQL-language functions that are simple enough to be “inlined” into the calling query will not be tracked, regardless of this setting.

`stats_temp_directory` (`string`)

Sets the directory to store temporary statistics data in. This can be a path relative to the data directory or an absolute path. The default is `pg_stat_tmp`. Pointing this at a RAM-based file system will decrease physical I/O requirements and can lead to improved performance. This parameter can only be set in the `postgresql.conf` file or on the server command line.

18.9.2. Statistics Monitoring

`log_statement_stats` (`boolean`)
`log_parser_stats` (`boolean`)
`log_planner_stats` (`boolean`)
`log_executor_stats` (`boolean`)

For each query, output performance statistics of the respective module to the server log. This is a crude profiling instrument, similar to the Unix `getrusage()` operating system facility. `log_statement_stats` reports total statement statistics, while the others report per-module statistics. `log_statement_stats` cannot be enabled together with any of the per-module options. All of these options are disabled by default. Only superusers can change these settings.

18.10. Automatic Vacuuming

These settings control the behavior of the *autovacuum* feature. Refer to Section 23.1.6 for more information. Note that many of these settings can be overridden on a per-table basis; see *Storage Parameters*.

`autovacuum` (`boolean`)

Controls whether the server should run the autovacuum launcher daemon. This is on by default; however, track_counts must also be enabled for autovacuum to work. This parameter can only be set in the `postgresql.conf` file or on the server command line; however, autovacuuming can be disabled for individual tables by changing table storage parameters.

Note that even when this parameter is disabled, the system will launch autovacuum processes if necessary to prevent transaction ID wraparound. See Section 23.1.5 for more information.

`log_autovacuum_min_duration` (`integer`)

Causes each action executed by autovacuum to be logged if it ran for at least the specified number of milliseconds. Setting this to zero logs all autovacuum actions. Minus-one (the default) disables logging autovacuum actions. For example, if you set this to `250ms` then all automatic vacuums and analyzes that run 250ms or longer will be logged. In addition, when this parameter is set to any value other than `-1`, a message will be logged if an autovacuum action is skipped due to the existence of a conflicting lock. Enabling this parameter can be helpful in tracking autovacuum activity. This parameter can only be set in the `postgresql.conf` file or on the server command line; but the setting can be overridden for individual tables by changing table storage parameters.

`autovacuum_max_workers` (`integer`)

Specifies the maximum number of autovacuum processes (other than the autovacuum launcher) that may be running at any one time. The default is three. This parameter can only be set at server start.

`autovacuum_naptime` (`integer`)

Specifies the minimum delay between autovacuum runs on any given database. In each round the daemon examines the database and issues `VACUUM` and `ANALYZE` commands as needed for tables in that database. The delay is measured in seconds, and the default is one minute (`1min`). This parameter can only be set in the `postgresql.conf` file or on the server command line.

autovacuum_vacuum_threshold (integer)

Specifies the minimum number of updated or deleted tuples needed to trigger a VACUUM in any one table. The default is 50 tuples. This parameter can only be set in the postgresql.conf file or on the server command line; but the setting can be overridden for individual tables by changing table storage parameters.

autovacuum_analyze_threshold (integer)

Specifies the minimum number of inserted, updated or deleted tuples needed to trigger an ANALYZE in any one table. The default is 50 tuples. This parameter can only be set in the postgresql.conf file or on the server command line; but the setting can be overridden for individual tables by changing table storage parameters.

autovacuum_vacuum_scale_factor (floating point)

Specifies a fraction of the table size to add to autovacuum_vacuum_threshold when deciding whether to trigger a VACUUM. The default is 0.2 (20% of table size). This parameter can only be set in the postgresql.conf file or on the server command line; but the setting can be overridden for individual tables by changing table storage parameters.

autovacuum_analyze_scale_factor (floating point)

Specifies a fraction of the table size to add to autovacuum_analyze_threshold when deciding whether to trigger an ANALYZE. The default is 0.1 (10% of table size). This parameter can only be set in the postgresql.conf file or on the server command line; but the setting can be overridden for individual tables by changing table storage parameters.

autovacuum_freeze_max_age (integer)

Specifies the maximum age (in transactions) that a table's pg_class.relfrozenxid field can attain before a VACUUM operation is forced to prevent transaction ID wraparound within the table. Note that the system will launch autovacuum processes to prevent wraparound even when autovacuum is otherwise disabled.

Vacuum also allows removal of old files from the pg_clog subdirectory, which is why the default is a relatively low 200 million transactions. This parameter can only be set at server start, but the setting can be reduced for individual tables by changing table storage parameters. For more information see Section 23.1.5.

autovacuum_multixact_freeze_max_age (integer)

Specifies the maximum age (in multixacts) that a table's pg_class.relminmxid field can attain before a VACUUM operation is forced to prevent multixact ID wraparound within the table. Note that the system will launch autovacuum processes to prevent wraparound even when autovacuum is otherwise disabled.

Vacuuming multixacts also allows removal of old files from the pg_multixact/members and pg_multixact/offsets subdirectories, which is why the default is a relatively low 400 million multixacts. This parameter can only be set at server start, but the setting can be reduced for individual tables by changing table storage parameters. For more information see Section 23.1.5.1.

autovacuum_vacuum_cost_delay (integer)

Specifies the cost delay value that will be used in automatic VACUUM operations. If -1 is specified, the regular vacuum_cost_delay value will be used. The default value is 20 milliseconds. This parameter

can only be set in the postgresql.conf file or on the server command line; but the setting can be overridden for individual tables by changing table storage parameters.

autovacuum_vacuum_cost_limit (integer)

Specifies the cost limit value that will be used in automatic VACUUM operations. If -1 is specified (which is the default), the regular vacuum_cost_limit value will be used. Note that the value is distributed proportionally among the running autovacuum workers, if there is more than one, so that the sum of the limits for each worker does not exceed the value of this variable. This parameter can only be set in the postgresql.conf file or on the server command line; but the setting can be overridden for individual tables by changing table storage parameters.

18.11. Client Connection Defaults

18.11.1. Statement Behavior

client_min_messages (enum)

Controls which message levels are sent to the client. Valid values are DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, LOG, NOTICE, WARNING, and ERROR. Each level includes all the levels that follow it. The later the level, the fewer messages are sent. The default is NOTICE. Note that LOG has a different rank here than in log_min_messages.

INFO level messages are always sent to the client.

search_path (string)

This variable specifies the order in which schemas are searched when an object (table, data type, function, etc.) is referenced by a simple name with no schema specified. When there are objects of identical names in different schemas, the one found first in the search path is used. An object that is not in any of the schemas in the search path can only be referenced by specifying its containing schema with a qualified (dotted) name.

The value for search_path must be a comma-separated list of schema names. Any name that is not an existing schema, or is a schema for which the user does not have USAGE permission, is silently ignored.

If one of the list items is the special name $user, then the schema having the name returned by CURRENT_USER is substituted, if there is such a schema and the user has USAGE permission for it. (If not, $user is ignored.)

The system catalog schema, pg_catalog, is always searched, whether it is mentioned in the path or not. If it is mentioned in the path then it will be searched in the specified order. If pg_catalog is not in the path then it will be searched *before* searching any of the path items.

Likewise, the current session's temporary-table schema, pg_temp_*nnn*, is always searched if it exists. It can be explicitly listed in the path by using the alias pg_temp. If it is not listed in the path then it is searched first (even before pg_catalog). However, the temporary schema is only searched for relation (table, view, sequence, etc) and data type names. It is never searched for function or operator names.

When objects are created without specifying a particular target schema, they will be placed in the first valid schema named in search_path. An error is reported if the search path is empty.

The default value for this parameter is "$user", public. This setting supports shared use of a database (where no users have private schemas, and all share use of public), private per-user schemas, and combinations of these. Other effects can be obtained by altering the default search path setting, either globally or per-user.

For more information on schema handling, see Section 5.8. In particular, the default configuration is suitable only when the database has a single user or a few mutually-trusting users.

The current effective value of the search path can be examined via the SQL function current_schemas (see Section 9.25). This is not quite the same as examining the value of search_path, since current_schemas shows how the items appearing in search_path were resolved.

row_security (boolean)

This variable controls whether to raise an error in lieu of applying a row security policy. When set to on, policies apply normally. When set to off, queries fail which would otherwise apply at least one policy. The default is on. Change to off where limited row visibility could cause incorrect results; for example, pg_dump makes that change by default. This variable has no effect on roles which bypass every row security policy, to wit, superusers and roles with the BYPASSRLS attribute.

For more information on row security policies, see CREATE POLICY.

default_tablespace (string)

This variable specifies the default tablespace in which to create objects (tables and indexes) when a CREATE command does not explicitly specify a tablespace.

The value is either the name of a tablespace, or an empty string to specify using the default tablespace of the current database. If the value does not match the name of any existing tablespace, PostgreSQL will automatically use the default tablespace of the current database. If a nondefault tablespace is specified, the user must have CREATE privilege for it, or creation attempts will fail.

This variable is not used for temporary tables; for them, temp_tablespaces is consulted instead.

This variable is also not used when creating databases. By default, a new database inherits its tablespace setting from the template database it is copied from.

For more information on tablespaces, see Section 21.6.

temp_tablespaces (string)

This variable specifies tablespaces in which to create temporary objects (temp tables and indexes on temp tables) when a CREATE command does not explicitly specify a tablespace. Temporary files for purposes such as sorting large data sets are also created in these tablespaces.

The value is a list of names of tablespaces. When there is more than one name in the list, PostgreSQL chooses a random member of the list each time a temporary object is to be created; except that within a transaction, successively created temporary objects are placed in successive tablespaces from the list. If the selected element of the list is an empty string, PostgreSQL will automatically use the default tablespace of the current database instead.

When temp_tablespaces is set interactively, specifying a nonexistent tablespace is an error, as is specifying a tablespace for which the user does not have CREATE privilege. However, when using a

previously set value, nonexistent tablespaces are ignored, as are tablespaces for which the user lacks CREATE privilege. In particular, this rule applies when using a value set in postgresql.conf.

The default value is an empty string, which results in all temporary objects being created in the default tablespace of the current database.

See also default_tablespace.

check_function_bodies (boolean)

This parameter is normally on. When set to off, it disables validation of the function body string during CREATE FUNCTION. Disabling validation avoids side effects of the validation process and avoids false positives due to problems such as forward references. Set this parameter to off before loading functions on behalf of other users; pg_dump does so automatically.

default_transaction_isolation (enum)

Each SQL transaction has an isolation level, which can be either "read uncommitted", "read committed", "repeatable read", or "serializable". This parameter controls the default isolation level of each new transaction. The default is "read committed".

Consult Chapter 13 and SET TRANSACTION for more information.

default_transaction_read_only (boolean)

A read-only SQL transaction cannot alter non-temporary tables. This parameter controls the default read-only status of each new transaction. The default is off (read/write).

Consult SET TRANSACTION for more information.

default_transaction_deferrable (boolean)

When running at the serializable isolation level, a deferrable read-only SQL transaction may be delayed before it is allowed to proceed. However, once it begins executing it does not incur any of the overhead required to ensure serializability; so serialization code will have no reason to force it to abort because of concurrent updates, making this option suitable for long-running read-only transactions.

This parameter controls the default deferrable status of each new transaction. It currently has no effect on read-write transactions or those operating at isolation levels lower than serializable. The default is off.

Consult SET TRANSACTION for more information.

session_replication_role (enum)

Controls firing of replication-related triggers and rules for the current session. Setting this variable requires superuser privilege and results in discarding any previously cached query plans. Possible values are origin (the default), replica and local. See ALTER TABLE for more information.

statement_timeout (integer)

Abort any statement that takes more than the specified number of milliseconds, starting from the time the command arrives at the server from the client. If log_min_error_statement is set to ERROR or lower, the statement that timed out will also be logged. A value of zero (the default) turns this off.

Setting statement_timeout in postgresql.conf is not recommended because it would affect all sessions.

lock_timeout (integer)

Abort any statement that waits longer than the specified number of milliseconds while attempting to acquire a lock on a table, index, row, or other database object. The time limit applies separately to each lock acquisition attempt. The limit applies both to explicit locking requests (such as LOCK TABLE, or SELECT FOR UPDATE without NOWAIT) and to implicitly-acquired locks. A value of zero (the default) turns this off.

Unlike statement_timeout, this timeout can only occur while waiting for locks. Note that if statement_timeout is nonzero, it is rather pointless to set lock_timeout to the same or larger value, since the statement timeout would always trigger first. If log_min_error_statement is set to ERROR or lower, the statement that timed out will be logged.

Setting lock_timeout in postgresql.conf is not recommended because it would affect all sessions.

vacuum_freeze_table_age (integer)

VACUUM performs a whole-table scan if the table's pg_class.relfrozenxid field has reached the age specified by this setting. The default is 150 million transactions. Although users can set this value anywhere from zero to two billions, VACUUM will silently limit the effective value to 95% of autovacuum_freeze_max_age, so that a periodical manual VACUUM has a chance to run before an anti-wraparound autovacuum is launched for the table. For more information see Section 23.1.5.

vacuum_freeze_min_age (integer)

Specifies the cutoff age (in transactions) that VACUUM should use to decide whether to freeze row versions while scanning a table. The default is 50 million transactions. Although users can set this value anywhere from zero to one billion, VACUUM will silently limit the effective value to half the value of autovacuum_freeze_max_age, so that there is not an unreasonably short time between forced autovacuums. For more information see Section 23.1.5.

vacuum_multixact_freeze_table_age (integer)

VACUUM performs a whole-table scan if the table's pg_class.relminmxid field has reached the age specified by this setting. The default is 150 million multixacts. Although users can set this value anywhere from zero to two billions, VACUUM will silently limit the effective value to 95% of autovacuum_multixact_freeze_max_age, so that a periodical manual VACUUM has a chance to run before an anti-wraparound is launched for the table. For more information see Section 23.1.5.1.

vacuum_multixact_freeze_min_age (integer)

Specifies the cutoff age (in multixacts) that VACUUM should use to decide whether to replace multixact IDs with a newer transaction ID or multixact ID while scanning a table. The default is 5 million multixacts. Although users can set this value anywhere from zero to one billion, VACUUM will silently limit the effective value to half the value of autovacuum_multixact_freeze_max_age, so that there is not an unreasonably short time between forced autovacuums. For more information see Section 23.1.5.1.

bytea_output (enum)

Sets the output format for values of type bytea. Valid values are hex (the default) and escape (the traditional PostgreSQL format). See Section 8.4 for more information. The bytea type always accepts both formats on input, regardless of this setting.

xmlbinary (enum)

Sets how binary values are to be encoded in XML. This applies for example when bytea values are converted to XML by the functions xmlelement or xmlforest. Possible values are base64 and hex, which are both defined in the XML Schema standard. The default is base64. For further information about XML-related functions, see Section 9.14.

The actual choice here is mostly a matter of taste, constrained only by possible restrictions in client applications. Both methods support all possible values, although the hex encoding will be somewhat larger than the base64 encoding.

xmloption (enum)

Sets whether DOCUMENT or CONTENT is implicit when converting between XML and character string values. See Section 8.13 for a description of this. Valid values are DOCUMENT and CONTENT. The default is CONTENT.

According to the SQL standard, the command to set this option is

```
SET XML OPTION { DOCUMENT | CONTENT };
```

This syntax is also available in PostgreSQL.

gin_pending_list_limit (integer)

Sets the maximum size of the GIN pending list which is used when fastupdate is enabled. If the list grows larger than this maximum size, it is cleaned up by moving the entries in it to the main GIN data structure in bulk. The default is four megabytes (4MB). This setting can be overridden for individual GIN indexes by changing index storage parameters. See Section 61.4.1 and Section 61.5 for more information.

18.11.2. Locale and Formatting

DateStyle (string)

Sets the display format for date and time values, as well as the rules for interpreting ambiguous date input values. For historical reasons, this variable contains two independent components: the output format specification (ISO, Postgres, SQL, or German) and the input/output specification for year/month/day ordering (DMY, MDY, or YMD). These can be set separately or together. The keywords Euro and European are synonyms for DMY; the keywords US, NonEuro, and NonEuropean are synonyms for MDY. See Section 8.5 for more information. The built-in default is ISO, MDY, but initdb will initialize the configuration file with a setting that corresponds to the behavior of the chosen lc_time locale.

IntervalStyle (enum)

Sets the display format for interval values. The value sql_standard will produce output matching SQL standard interval literals. The value postgres (which is the default) will produce output matching PostgreSQL releases prior to 8.4 when the DateStyle parameter was set to ISO. The value postgres_verbose will produce output matching PostgreSQL releases prior to 8.4 when the DateStyle parameter was set to non-ISO output. The value iso_8601 will produce output matching the time interval "format with designators" defined in section 4.4.3.2 of ISO 8601.

The IntervalStyle parameter also affects the interpretation of ambiguous interval input. See Section 8.5.4 for more information.

`TimeZone` (`string`)

Sets the time zone for displaying and interpreting time stamps. The built-in default is `GMT`, but that is typically overridden in `postgresql.conf`; initdb will install a setting there corresponding to its system environment. See Section 8.5.3 for more information.

`timezone_abbreviations` (`string`)

Sets the collection of time zone abbreviations that will be accepted by the server for datetime input. The default is `'Default'`, which is a collection that works in most of the world; there are also `'Australia'` and `'India'`, and other collections can be defined for a particular installation. See Section B.4 for more information.

`extra_float_digits` (`integer`)

This parameter adjusts the number of digits displayed for floating-point values, including `float4`, `float8`, and geometric data types. The parameter value is added to the standard number of digits (`FLT_DIG` or `DBL_DIG` as appropriate). The value can be set as high as 3, to include partially-significant digits; this is especially useful for dumping float data that needs to be restored exactly. Or it can be set negative to suppress unwanted digits. See also Section 8.1.3.

`client_encoding` (`string`)

Sets the client-side encoding (character set). The default is to use the database encoding. The character sets supported by the PostgreSQL server are described in Section 22.3.1.

`lc_messages` (`string`)

Sets the language in which messages are displayed. Acceptable values are system-dependent; see Section 22.1 for more information. If this variable is set to the empty string (which is the default) then the value is inherited from the execution environment of the server in a system-dependent way.

On some systems, this locale category does not exist. Setting this variable will still work, but there will be no effect. Also, there is a chance that no translated messages for the desired language exist. In that case you will continue to see the English messages.

Only superusers can change this setting, because it affects the messages sent to the server log as well as to the client, and an improper value might obscure the readability of the server logs.

`lc_monetary` (`string`)

Sets the locale to use for formatting monetary amounts, for example with the `to_char` family of functions. Acceptable values are system-dependent; see Section 22.1 for more information. If this variable is set to the empty string (which is the default) then the value is inherited from the execution environment of the server in a system-dependent way.

`lc_numeric` (`string`)

Sets the locale to use for formatting numbers, for example with the `to_char` family of functions. Acceptable values are system-dependent; see Section 22.1 for more information. If this variable is set to the empty string (which is the default) then the value is inherited from the execution environment of the server in a system-dependent way.

`lc_time` (`string`)

Sets the locale to use for formatting dates and times, for example with the `to_char` family of functions. Acceptable values are system-dependent; see Section 22.1 for more information. If this vari-

able is set to the empty string (which is the default) then the value is inherited from the execution environment of the server in a system-dependent way.

`default_text_search_config` (`string`)

Selects the text search configuration that is used by those variants of the text search functions that do not have an explicit argument specifying the configuration. See Chapter 12 for further information. The built-in default is `pg_catalog.simple`, but initdb will initialize the configuration file with a setting that corresponds to the chosen `lc_ctype` locale, if a configuration matching that locale can be identified.

18.11.3. Shared Library Preloading

Several settings are available for preloading shared libraries into the server, in order to load additional functionality or achieve performance benefits. For example, a setting of `'$libdir/mylib'` would cause `mylib.so` (or on some platforms, `mylib.sl`) to be preloaded from the installation's standard library directory. The differences between the settings are when they take effect and what privileges are required to change them.

PostgreSQL procedural language libraries can be preloaded in this way, typically by using the syntax `'$libdir/plXXX'` where `XXX` is `pgsql`, `perl`, `tcl`, or `python`.

For each parameter, if more than one library is to be loaded, separate their names with commas. All library names are converted to lower case unless double-quoted.

Only shared libraries specifically intended to be used with PostgreSQL can be loaded this way. Every PostgreSQL-supported library has a "magic block" that is checked to guarantee compatibility. For this reason, non-PostgreSQL libraries cannot be loaded in this way. You might be able to use operating-system facilities such as `LD_PRELOAD` for that.

In general, refer to the documentation of a specific module for the recommended way to load that module.

`local_preload_libraries` (`string`)

This variable specifies one or more shared libraries that are to be preloaded at connection start. The parameter value only takes effect at the start of the connection. Subsequent changes have no effect. If a specified library is not found, the connection attempt will fail.

This option can be set by any user. Because of that, the libraries that can be loaded are restricted to those appearing in the `plugins` subdirectory of the installation's standard library directory. (It is the database administrator's responsibility to ensure that only "safe" libraries are installed there.) Entries in `local_preload_libraries` can specify this directory explicitly, for example `$libdir/plugins/mylib`, or just specify the library name — `mylib` would have the same effect as `$libdir/plugins/mylib`.

The intent of this feature is to allow unprivileged users to load debugging or performance-measurement libraries into specific sessions without requiring an explicit `LOAD` command. To that end, it would be typical to set this parameter using the `PGOPTIONS` environment variable on the client or by using `ALTER ROLE SET`.

However, unless a module is specifically designed to be used in this way by non-superusers, this is usually not the right setting to use. Look at session_preload_libraries instead.

session_preload_libraries (string)

This variable specifies one or more shared libraries that are to be preloaded at connection start. Only superusers can change this setting. The parameter value only takes effect at the start of the connection. Subsequent changes have no effect. If a specified library is not found, the connection attempt will fail.

The intent of this feature is to allow debugging or performance-measurement libraries to be loaded into specific sessions without an explicit LOAD command being given. For example, auto_explain could be enabled for all sessions under a given user name by setting this parameter with ALTER ROLE SET. Also, this parameter can be changed without restarting the server (but changes only take effect when a new session is started), so it is easier to add new modules this way, even if they should apply to all sessions.

Unlike shared_preload_libraries, there is no large performance advantage to loading a library at session start rather than when it is first used. There is some advantage, however, when connection pooling is used.

shared_preload_libraries (string)

This variable specifies one or more shared libraries to be preloaded at server start. This parameter can only be set at server start. If a specified library is not found, the server will fail to start.

Some libraries need to perform certain operations that can only take place at postmaster start, such as allocating shared memory, reserving light-weight locks, or starting background workers. Those libraries must be loaded at server start through this parameter. See the documentation of each library for details.

Other libraries can also be preloaded. By preloading a shared library, the library startup time is avoided when the library is first used. However, the time to start each new server process might increase slightly, even if that process never uses the library. So this parameter is recommended only for libraries that will be used in most sessions. Also, changing this parameter requires a server restart, so this is not the right setting to use for short-term debugging tasks, say. Use session_preload_libraries for that instead.

Note: On Windows hosts, preloading a library at server start will not reduce the time required to start each new server process; each server process will re-load all preload libraries. However, shared_preload_libraries is still useful on Windows hosts for libraries that need to perform operations at postmaster start time.

18.11.4. Other Defaults

dynamic_library_path (string)

If a dynamically loadable module needs to be opened and the file name specified in the CREATE FUNCTION or LOAD command does not have a directory component (i.e., the name does not contain a slash), the system will search this path for the required file.

The value for dynamic_library_path must be a list of absolute directory paths separated by colons (or semi-colons on Windows). If a list element starts with the special string $libdir, the compiled-in PostgreSQL package library directory is substituted for $libdir; this is where the modules provided by the standard PostgreSQL distribution are installed. (Use pg_config --pkglibdir to find out the name of this directory.) For example:

```
dynamic_library_path = '/usr/local/lib/postgresql:/home/my_project/lib:$libdir'
```

or, in a Windows environment:

```
dynamic_library_path = 'C:\tools\postgresql;H:\my_project\lib;$libdir'
```

The default value for this parameter is '$libdir'. If the value is set to an empty string, the automatic path search is turned off.

This parameter can be changed at run time by superusers, but a setting done that way will only persist until the end of the client connection, so this method should be reserved for development purposes. The recommended way to set this parameter is in the postgresql.conf configuration file.

gin_fuzzy_search_limit (integer)

Soft upper limit of the size of the set returned by GIN index scans. For more information see Section 61.5.

18.12. Lock Management

deadlock_timeout (integer)

This is the amount of time, in milliseconds, to wait on a lock before checking to see if there is a deadlock condition. The check for deadlock is relatively expensive, so the server doesn't run it every time it waits for a lock. We optimistically assume that deadlocks are not common in production applications and just wait on the lock for a while before checking for a deadlock. Increasing this value reduces the amount of time wasted in needless deadlock checks, but slows down reporting of real deadlock errors. The default is one second (1s), which is probably about the smallest value you would want in practice. On a heavily loaded server you might want to raise it. Ideally the setting should exceed your typical transaction time, so as to improve the odds that a lock will be released before the waiter decides to check for deadlock. Only superusers can change this setting.

When log_lock_waits is set, this parameter also determines the length of time to wait before a log message is issued about the lock wait. If you are trying to investigate locking delays you might want to set a shorter than normal deadlock_timeout.

max_locks_per_transaction (integer)

The shared lock table tracks locks on max_locks_per_transaction * (max_connections + max_prepared_transactions) objects (e.g., tables); hence, no more than this many distinct objects can be locked at any one time. This parameter controls the average number of object locks allocated for each transaction; individual transactions can lock more objects as long as the locks of all transactions fit in the lock table. This is *not* the number of rows that can be locked; that value is unlimited. The default, 64, has historically proven sufficient, but you might need to raise this value if you have queries that touch many different tables in a single transaction, e.g., query of a parent table with many children. This parameter can only be set at server start.

When running a standby server, you must set this parameter to the same or higher value than on the master server. Otherwise, queries will not be allowed in the standby server.

`max_pred_locks_per_transaction` (`integer`)

The shared predicate lock table tracks locks on `max_pred_locks_per_transaction` * (max_connections + max_prepared_transactions) objects (e.g., tables); hence, no more than this many distinct objects can be locked at any one time. This parameter controls the average number of object locks allocated for each transaction; individual transactions can lock more objects as long as the locks of all transactions fit in the lock table. This is *not* the number of rows that can be locked; that value is unlimited. The default, 64, has generally been sufficient in testing, but you might need to raise this value if you have clients that touch many different tables in a single serializable transaction. This parameter can only be set at server start.

18.13. Version and Platform Compatibility

18.13.1. Previous PostgreSQL Versions

`array_nulls` (`boolean`)

This controls whether the array input parser recognizes unquoted `NULL` as specifying a null array element. By default, this is `on`, allowing array values containing null values to be entered. However, PostgreSQL versions before 8.2 did not support null values in arrays, and therefore would treat `NULL` as specifying a normal array element with the string value "NULL". For backward compatibility with applications that require the old behavior, this variable can be turned `off`.

Note that it is possible to create array values containing null values even when this variable is `off`.

`backslash_quote` (`enum`)

This controls whether a quote mark can be represented by `\'` in a string literal. The preferred, SQL-standard way to represent a quote mark is by doubling it (`''`) but PostgreSQL has historically also accepted `\'`. However, use of `\'` creates security risks because in some client character set encodings, there are multibyte characters in which the last byte is numerically equivalent to ASCII `\`. If client-side code does escaping incorrectly then a SQL-injection attack is possible. This risk can be prevented by making the server reject queries in which a quote mark appears to be escaped by a backslash. The allowed values of `backslash_quote` are `on` (allow `\'` always), `off` (reject always), and `safe_encoding` (allow only if client encoding does not allow ASCII `\` within a multibyte character). `safe_encoding` is the default setting.

Note that in a standard-conforming string literal, `\` just means `\` anyway. This parameter only affects the handling of non-standard-conforming literals, including escape string syntax (`E'...'`).

`default_with_oids` (`boolean`)

This controls whether `CREATE TABLE` and `CREATE TABLE AS` include an OID column in newly-created tables, if neither `WITH OIDS` nor `WITHOUT OIDS` is specified. It also determines whether OIDs will be included in tables created by `SELECT INTO`. The parameter is `off` by default; in PostgreSQL 8.0 and earlier, it was `on` by default.

The use of OIDs in user tables is considered deprecated, so most installations should leave this variable disabled. Applications that require OIDs for a particular table should specify `WITH OIDS` when creating the table. This variable can be enabled for compatibility with old applications that do not follow this behavior.

`escape_string_warning` (`boolean`)

When on, a warning is issued if a backslash (`\`) appears in an ordinary string literal (`'...'` syntax) and `standard_conforming_strings` is off. The default is `on`.

Applications that wish to use backslash as escape should be modified to use escape string syntax (`E'...'`), because the default behavior of ordinary strings is now to treat backslash as an ordinary character, per SQL standard. This variable can be enabled to help locate code that needs to be changed.

`lo_compat_privileges` (`boolean`)

In PostgreSQL releases prior to 9.0, large objects did not have access privileges and were, therefore, always readable and writable by all users. Setting this variable to `on` disables the new privilege checks, for compatibility with prior releases. The default is `off`. Only superusers can change this setting.

Setting this variable does not disable all security checks related to large objects — only those for which the default behavior has changed in PostgreSQL 9.0. For example, `lo_import()` and `lo_export()` need superuser privileges regardless of this setting.

`operator_precedence_warning` (`boolean`)

When on, the parser will emit a warning for any construct that might have changed meanings since PostgreSQL 9.4 as a result of changes in operator precedence. This is useful for auditing applications to see if precedence changes have broken anything; but it is not meant to be kept turned on in production, since it will warn about some perfectly valid, standard-compliant SQL code. The default is `off`.

See Section 4.1.6 for more information.

`quote_all_identifiers` (`boolean`)

When the database generates SQL, force all identifiers to be quoted, even if they are not (currently) keywords. This will affect the output of `EXPLAIN` as well as the results of functions like `pg_get_viewdef`. See also the `--quote-all-identifiers` option of pg_dump and pg_dumpall.

`sql_inheritance` (`boolean`)

This setting controls whether undecorated table references are considered to include inheritance child tables. The default is `on`, which means child tables are included (thus, a `*` suffix is assumed by default). If turned `off`, child tables are not included (thus, an `ONLY` prefix is assumed). The SQL standard requires child tables to be included, so the `off` setting is not spec-compliant, but it is provided for compatibility with PostgreSQL releases prior to 7.1. See Section 5.9 for more information.

Turning `sql_inheritance` off is deprecated, because that behavior has been found to be error-prone as well as contrary to SQL standard. Discussions of inheritance behavior elsewhere in this manual generally assume that it is `on`.

`standard_conforming_strings` (`boolean`)

This controls whether ordinary string literals (`'...'`) treat backslashes literally, as specified in the SQL standard. Beginning in PostgreSQL 9.1, the default is `on` (prior releases defaulted to `off`). Ap-

plications can check this parameter to determine how string literals will be processed. The presence of this parameter can also be taken as an indication that the escape string syntax (`E'...'`) is supported. Escape string syntax (Section 4.1.2.2) should be used if an application desires backslashes to be treated as escape characters.

`synchronize_seqscans` (`boolean`)

This allows sequential scans of large tables to synchronize with each other, so that concurrent scans read the same block at about the same time and hence share the I/O workload. When this is enabled, a scan might start in the middle of the table and then "wrap around" the end to cover all rows, so as to synchronize with the activity of scans already in progress. This can result in unpredictable changes in the row ordering returned by queries that have no `ORDER BY` clause. Setting this parameter to `off` ensures the pre-8.3 behavior in which a sequential scan always starts from the beginning of the table. The default is `on`.

18.13.2. Platform and Client Compatibility

`transform_null_equals` (`boolean`)

When on, expressions of the form *`expr`* `= NULL` (or `NULL =` *`expr`*) are treated as *`expr`* `IS NULL`, that is, they return true if *`expr`* evaluates to the null value, and false otherwise. The correct SQL-spec-compliant behavior of *`expr`* `= NULL` is to always return null (unknown). Therefore this parameter defaults to `off`.

However, filtered forms in Microsoft Access generate queries that appear to use *`expr`* `= NULL` to test for null values, so if you use that interface to access the database you might want to turn this option on. Since expressions of the form *`expr`* `= NULL` always return the null value (using the SQL standard interpretation), they are not very useful and do not appear often in normal applications so this option does little harm in practice. But new users are frequently confused about the semantics of expressions involving null values, so this option is off by default.

Note that this option only affects the exact form `= NULL`, not other comparison operators or other expressions that are computationally equivalent to some expression involving the equals operator (such as `IN`). Thus, this option is not a general fix for bad programming.

Refer to Section 9.2 for related information.

18.14. Error Handling

`exit_on_error` (`boolean`)

If true, any error will terminate the current session. By default, this is set to false, so that only FATAL errors will terminate the session.

`restart_after_crash` (`boolean`)

When set to true, which is the default, PostgreSQL will automatically reinitialize after a backend crash. Leaving this value set to true is normally the best way to maximize the availability of the

database. However, in some circumstances, such as when PostgreSQL is being invoked by clusterware, it may be useful to disable the restart so that the clusterware can gain control and take any actions it deems appropriate.

`data_sync_retry` (`boolean`)

When set to false, which is the default, PostgreSQL will raise a PANIC-level error on failure to flush modified data files to the filesystem. This causes the database server to crash. This parameter can only be set at server start.

On some operating systems, the status of data in the kernel's page cache is unknown after a write-back failure. In some cases it might have been entirely forgotten, making it unsafe to retry; the second attempt may be reported as successful, when in fact the data has been lost. In these circumstances, the only way to avoid data loss is to recover from the WAL after any failure is reported, preferably after investigating the root cause of the failure and replacing any faulty hardware.

If set to true, PostgreSQL will instead report an error but continue to run so that the data flushing operation can be retried in a later checkpoint. Only set it to true after investigating the operating system's treatment of buffered data in case of write-back failure.

18.15. Preset Options

The following "parameters" are read-only, and are determined when PostgreSQL is compiled or when it is installed. As such, they have been excluded from the sample `postgresql.conf` file. These options report various aspects of PostgreSQL behavior that might be of interest to certain applications, particularly administrative front-ends.

`block_size` (`integer`)

Reports the size of a disk block. It is determined by the value of `BLCKSZ` when building the server. The default value is 8192 bytes. The meaning of some configuration variables (such as shared_buffers) is influenced by `block_size`. See Section 18.4 for information.

`data_checksums` (`boolean`)

Reports whether data checksums are enabled for this cluster. See data checksums for more information.

`debug_assertions` (`boolean`)

Reports whether PostgreSQL has been built with assertions enabled. That is the case if the macro `USE_ASSERT_CHECKING` is defined when PostgreSQL is built (accomplished e.g., by the `configure` option `--enable-cassert`). By default PostgreSQL is built without assertions.

`integer_datetimes` (`boolean`)

Reports whether PostgreSQL was built with support for 64-bit-integer dates and times. This can be disabled by configuring with `--disable-integer-datetimes` when building PostgreSQL. The default value is `on`.

`lc_collate` (`string`)

Reports the locale in which sorting of textual data is done. See Section 22.1 for more information. This value is determined when a database is created.

`lc_ctype` (`string`)

Reports the locale that determines character classifications. See Section 22.1 for more information. This value is determined when a database is created. Ordinarily this will be the same as `lc_collate`, but for special applications it might be set differently.

`max_function_args` (`integer`)

Reports the maximum number of function arguments. It is determined by the value of `FUNC_MAX_ARGS` when building the server. The default value is 100 arguments.

`max_identifier_length` (`integer`)

Reports the maximum identifier length. It is determined as one less than the value of `NAMEDATALEN` when building the server. The default value of `NAMEDATALEN` is 64; therefore the default `max_identifier_length` is 63 bytes, which can be less than 63 characters when using multibyte encodings.

`max_index_keys` (`integer`)

Reports the maximum number of index keys. It is determined by the value of `INDEX_MAX_KEYS` when building the server. The default value is 32 keys.

`segment_size` (`integer`)

Reports the number of blocks (pages) that can be stored within a file segment. It is determined by the value of `RELSEG_SIZE` when building the server. The maximum size of a segment file in bytes is equal to `segment_size` multiplied by `block_size`; by default this is 1GB.

`server_encoding` (`string`)

Reports the database encoding (character set). It is determined when the database is created. Ordinarily, clients need only be concerned with the value of client_encoding.

`server_version` (`string`)

Reports the version number of the server. It is determined by the value of `PG_VERSION` when building the server.

`server_version_num` (`integer`)

Reports the version number of the server as an integer. It is determined by the value of `PG_VERSION_NUM` when building the server.

`wal_block_size` (`integer`)

Reports the size of a WAL disk block. It is determined by the value of `XLOG_BLCKSZ` when building the server. The default value is 8192 bytes.

`wal_segment_size` (`integer`)

Reports the number of blocks (pages) in a WAL segment file. The total size of a WAL segment file in bytes is equal to `wal_segment_size` multiplied by `wal_block_size`; by default this is 16MB. See Section 29.4 for more information.

18.16. Customized Options

This feature was designed to allow parameters not normally known to PostgreSQL to be added by add-on modules (such as procedural languages). This allows extension modules to be configured in the standard ways.

Custom options have two-part names: an extension name, then a dot, then the parameter name proper, much like qualified names in SQL. An example is `plpgsql.variable_conflict`.

Because custom options may need to be set in processes that have not loaded the relevant extension module, PostgreSQL will accept a setting for any two-part parameter name. Such variables are treated as placeholders and have no function until the module that defines them is loaded. When an extension module is loaded, it will add its variable definitions, convert any placeholder values according to those definitions, and issue warnings for any unrecognized placeholders that begin with its extension name.

18.17. Developer Options

The following parameters are intended for work on the PostgreSQL source code, and in some cases to assist with recovery of severely damaged databases. There should be no reason to use them on a production database. As such, they have been excluded from the sample `postgresql.conf` file. Note that many of these parameters require special source compilation flags to work at all.

`allow_system_table_mods` (`boolean`)

Allows modification of the structure of system tables. This is used by `initdb`. This parameter can only be set at server start.

`ignore_system_indexes` (`boolean`)

Ignore system indexes when reading system tables (but still update the indexes when modifying the tables). This is useful when recovering from damaged system indexes. This parameter cannot be changed after session start.

`post_auth_delay` (`integer`)

If nonzero, a delay of this many seconds occurs when a new server process is started, after it conducts the authentication procedure. This is intended to give developers an opportunity to attach to the server process with a debugger. This parameter cannot be changed after session start.

`pre_auth_delay` (`integer`)

If nonzero, a delay of this many seconds occurs just after a new server process is forked, before it conducts the authentication procedure. This is intended to give developers an opportunity to attach to the server process with a debugger to trace down misbehavior in authentication. This parameter can only be set in the `postgresql.conf` file or on the server command line.

`trace_notify` (`boolean`)

Generates a great amount of debugging output for the `LISTEN` and `NOTIFY` commands. client_min_messages or log_min_messages must be `DEBUG1` or lower to send this output to the client or server logs, respectively.

trace_recovery_messages (enum)

Enables logging of recovery-related debugging output that otherwise would not be logged. This parameter allows the user to override the normal setting of log_min_messages, but only for specific messages. This is intended for use in debugging Hot Standby. Valid values are DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, and LOG. The default, LOG, does not affect logging decisions at all. The other values cause recovery-related debug messages of that priority or higher to be logged as though they had LOG priority; for common settings of log_min_messages this results in unconditionally sending them to the server log. This parameter can only be set in the postgresql.conf file or on the server command line.

trace_sort (boolean)

If on, emit information about resource usage during sort operations. This parameter is only available if the TRACE_SORT macro was defined when PostgreSQL was compiled. (However, TRACE_SORT is currently defined by default.)

trace_locks (boolean)

If on, emit information about lock usage. Information dumped includes the type of lock operation, the type of lock and the unique identifier of the object being locked or unlocked. Also included are bit masks for the lock types already granted on this object as well as for the lock types awaited on this object. For each lock type a count of the number of granted locks and waiting locks is also dumped as well as the totals. An example of the log file output is shown here:

```
LOG:  LockAcquire: new: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  GrantLock: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(2) req(1,0,0,0,0,0,0)=1 grant(1,0,0,0,0,0,0)=1
      wait(0) type(AccessShareLock)
LOG:  UnGrantLock: updated: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(AccessShareLock)
LOG:  CleanUpLock: deleting: lock(0xb7acd844) id(24688,24696,0,0,0,1)
      grantMask(0) req(0,0,0,0,0,0,0)=0 grant(0,0,0,0,0,0,0)=0
      wait(0) type(INVALID)
```

Details of the structure being dumped may be found in src/include/storage/lock.h.

This parameter is only available if the LOCK_DEBUG macro was defined when PostgreSQL was compiled.

trace_lwlocks (boolean)

If on, emit information about lightweight lock usage. Lightweight locks are intended primarily to provide mutual exclusion of access to shared-memory data structures.

This parameter is only available if the LOCK_DEBUG macro was defined when PostgreSQL was compiled.

trace_userlocks (boolean)

If on, emit information about user lock usage. Output is the same as for trace_locks, only for advisory locks.

This parameter is only available if the LOCK_DEBUG macro was defined when PostgreSQL was compiled.

`trace_lock_oidmin` (`integer`)

If set, do not trace locks for tables below this OID (used to avoid output on system tables).

This parameter is only available if the `LOCK_DEBUG` macro was defined when PostgreSQL was compiled.

`trace_lock_table` (`integer`)

Unconditionally trace locks on this table (OID).

This parameter is only available if the `LOCK_DEBUG` macro was defined when PostgreSQL was compiled.

`debug_deadlocks` (`boolean`)

If set, dumps information about all current locks when a deadlock timeout occurs.

This parameter is only available if the `LOCK_DEBUG` macro was defined when PostgreSQL was compiled.

`log_btree_build_stats` (`boolean`)

If set, logs system resource usage statistics (memory and CPU) on various B-tree operations.

This parameter is only available if the `BTREE_BUILD_STATS` macro was defined when PostgreSQL was compiled.

`wal_debug` (`boolean`)

If on, emit WAL-related debugging output. This parameter is only available if the `WAL_DEBUG` macro was defined when PostgreSQL was compiled.

`ignore_checksum_failure` (`boolean`)

Only has effect if data checksums are enabled.

Detection of a checksum failure during a read normally causes PostgreSQL to report an error, aborting the current transaction. Setting `ignore_checksum_failure` to on causes the system to ignore the failure (but still report a warning), and continue processing. This behavior may *cause crashes, propagate or hide corruption, or other serious problems*. However, it may allow you to get past the error and retrieve undamaged tuples that might still be present in the table if the block header is still sane. If the header is corrupt an error will be reported even if this option is enabled. The default setting is `off`, and it can only be changed by a superuser.

`zero_damaged_pages` (`boolean`)

Detection of a damaged page header normally causes PostgreSQL to report an error, aborting the current transaction. Setting `zero_damaged_pages` to on causes the system to instead report a warning, zero out the damaged page in memory, and continue processing. This behavior *will destroy data*, namely all the rows on the damaged page. However, it does allow you to get past the error and retrieve rows from any undamaged pages that might be present in the table. It is useful for recovering data if corruption has occurred due to a hardware or software error. You should generally not set this on until you have given up hope of recovering data from the damaged pages of a table. Zeroed-out pages are not forced to disk so it is recommended to recreate the table or the index before turning this parameter off again. The default setting is `off`, and it can only be changed by a superuser.

18.18. Short Options

For convenience there are also single letter command-line option switches available for some parameters. They are described in Table 18-2. Some of these options exist for historical reasons, and their presence as a single-letter option does not necessarily indicate an endorsement to use the option heavily.

Table 18-2. Short Option Key

Short Option	Equivalent
`-B` *x*	`shared_buffers = x`
`-d` *x*	`log_min_messages = DEBUG`*x*
`-e`	`datestyle = euro`
`-fb`, `-fh`, `-fi`, `-fm`, `-fn`, `-fo`, `-fs`, `-ft`	`enable_bitmapscan = off`, `enable_hashjoin = off`, `enable_indexscan = off`, `enable_mergejoin = off`, `enable_nestloop = off`, `enable_indexonlyscan = off`, `enable_seqscan = off`, `enable_tidscan = off`
`-F`	`fsync = off`
`-h` *x*	`listen_addresses = x`
`-i`	`listen_addresses = '*'`
`-k` *x*	`unix_socket_directories = x`
`-l`	`ssl = on`
`-N` *x*	`max_connections = x`
`-O`	`allow_system_table_mods = on`
`-p` *x*	`port = x`
`-P`	`ignore_system_indexes = on`
`-s`	`log_statement_stats = on`
`-S` *x*	`work_mem = x`
`-tpa`, `-tpl`, `-te`	`log_parser_stats = on`, `log_planner_stats = on`, `log_executor_stats = on`
`-W` *x*	`post_auth_delay = x`

Chapter 19. Client Authentication

When a client application connects to the database server, it specifies which PostgreSQL database user name it wants to connect as, much the same way one logs into a Unix computer as a particular user. Within the SQL environment the active database user name determines access privileges to database objects — see Chapter 20 for more information. Therefore, it is essential to restrict which database users can connect.

> **Note:** As explained in Chapter 20, PostgreSQL actually does privilege management in terms of "roles". In this chapter, we consistently use *database user* to mean "role with the `LOGIN` privilege".

Authentication is the process by which the database server establishes the identity of the client, and by extension determines whether the client application (or the user who runs the client application) is permitted to connect with the database user name that was requested.

PostgreSQL offers a number of different client authentication methods. The method used to authenticate a particular client connection can be selected on the basis of (client) host address, database, and user.

PostgreSQL database user names are logically separate from user names of the operating system in which the server runs. If all the users of a particular server also have accounts on the server's machine, it makes sense to assign database user names that match their operating system user names. However, a server that accepts remote connections might have many database users who have no local operating system account, and in such cases there need be no connection between database user names and OS user names.

19.1. The `pg_hba.conf` File

Client authentication is controlled by a configuration file, which traditionally is named `pg_hba.conf` and is stored in the database cluster's data directory. (HBA stands for host-based authentication.) A default `pg_hba.conf` file is installed when the data directory is initialized by `initdb`. It is possible to place the authentication configuration file elsewhere, however; see the hba_file configuration parameter.

The general format of the `pg_hba.conf` file is a set of records, one per line. Blank lines are ignored, as is any text after the `#` comment character. Records cannot be continued across lines. A record is made up of a number of fields which are separated by spaces and/or tabs. Fields can contain white space if the field value is double-quoted. Quoting one of the keywords in a database, user, or address field (e.g., `all` or `replication`) makes the word lose its special meaning, and just match a database, user, or host with that name.

Each record specifies a connection type, a client IP address range (if relevant for the connection type), a database name, a user name, and the authentication method to be used for connections matching these parameters. The first record with a matching connection type, client address, requested database, and user name is used to perform authentication. There is no "fall-through" or "backup": if one record is chosen and the authentication fails, subsequent records are not considered. If no record matches, access is denied.

A record can have one of the seven formats

```
local      database  user  auth-method  [auth-options]
host       database  user  address  auth-method  [auth-options]
hostssl    database  user  address  auth-method  [auth-options]
```

```
hostnossl  database  user  address  auth-method  [auth-options]
host       database  user  IP-address  IP-mask  auth-method  [auth-options]
hostssl    database  user  IP-address  IP-mask  auth-method  [auth-options]
hostnossl  database  user  IP-address  IP-mask  auth-method  [auth-options]
```

The meaning of the fields is as follows:

`local`

This record matches connection attempts using Unix-domain sockets. Without a record of this type, Unix-domain socket connections are disallowed.

`host`

This record matches connection attempts made using TCP/IP. `host` records match either SSL or non-SSL connection attempts.

> **Note:** Remote TCP/IP connections will not be possible unless the server is started with an appropriate value for the listen_addresses configuration parameter, since the default behavior is to listen for TCP/IP connections only on the local loopback address `localhost`.

`hostssl`

This record matches connection attempts made using TCP/IP, but only when the connection is made with SSL encryption.

To make use of this option the server must be built with SSL support. Furthermore, SSL must be enabled at server start time by setting the ssl configuration parameter (see Section 17.9 for more information).

`hostnossl`

This record type has the opposite behavior of `hostssl`; it only matches connection attempts made over TCP/IP that do not use SSL.

`database`

Specifies which database name(s) this record matches. The value `all` specifies that it matches all databases. The value `sameuser` specifies that the record matches if the requested database has the same name as the requested user. The value `samerole` specifies that the requested user must be a member of the role with the same name as the requested database. (`samegroup` is an obsolete but still accepted spelling of `samerole`.) Superusers are not considered to be members of a role for the purposes of `samerole` unless they are explicitly members of the role, directly or indirectly, and not just by virtue of being a superuser. The value `replication` specifies that the record matches if a replication connection is requested (note that replication connections do not specify any particular database). Otherwise, this is the name of a specific PostgreSQL database. Multiple database names can be supplied by separating them with commas. A separate file containing database names can be specified by preceding the file name with `@`.

`user`

Specifies which database user name(s) this record matches. The value `all` specifies that it matches all users. Otherwise, this is either the name of a specific database user, or a group name preceded by

+. (Recall that there is no real distinction between users and groups in PostgreSQL; a + mark really means "match any of the roles that are directly or indirectly members of this role", while a name without a + mark matches only that specific role.) For this purpose, a superuser is only considered to be a member of a role if they are explicitly a member of the role, directly or indirectly, and not just by virtue of being a superuser. Multiple user names can be supplied by separating them with commas. A separate file containing user names can be specified by preceding the file name with `@`.

address

Specifies the client machine address(es) that this record matches. This field can contain either a host name, an IP address range, or one of the special key words mentioned below.

An IP address range is specified using standard numeric notation for the range's starting address, then a slash (`/`) and a CIDR mask length. The mask length indicates the number of high-order bits of the client IP address that must match. Bits to the right of this should be zero in the given IP address. There must not be any white space between the IP address, the `/`, and the CIDR mask length.

Typical examples of an IPv4 address range specified this way are `172.20.143.89/32` for a single host, or `172.20.143.0/24` for a small network, or `10.6.0.0/16` for a larger one. An IPv6 address range might look like `::1/128` for a single host (in this case the IPv6 loopback address) or `fe80::7a31:c1ff:0000:0000/96` for a small network. `0.0.0.0/0` represents all IPv4 addresses, and `::0/0` represents all IPv6 addresses. To specify a single host, use a mask length of 32 for IPv4 or 128 for IPv6. In a network address, do not omit trailing zeroes.

An entry given in IPv4 format will match only IPv4 connections, and an entry given in IPv6 format will match only IPv6 connections, even if the represented address is in the IPv4-in-IPv6 range. Note that entries in IPv6 format will be rejected if the system's C library does not have support for IPv6 addresses.

You can also write `all` to match any IP address, `samehost` to match any of the server's own IP addresses, or `samenet` to match any address in any subnet that the server is directly connected to.

If a host name is specified (anything that is not an IP address range or a special key word is treated as a host name), that name is compared with the result of a reverse name resolution of the client's IP address (e.g., reverse DNS lookup, if DNS is used). Host name comparisons are case insensitive. If there is a match, then a forward name resolution (e.g., forward DNS lookup) is performed on the host name to check whether any of the addresses it resolves to are equal to the client's IP address. If both directions match, then the entry is considered to match. (The host name that is used in `pg_hba.conf` should be the one that address-to-name resolution of the client's IP address returns, otherwise the line won't be matched. Some host name databases allow associating an IP address with multiple host names, but the operating system will only return one host name when asked to resolve an IP address.)

A host name specification that starts with a dot (`.`) matches a suffix of the actual host name. So `.example.com` would match `foo.example.com` (but not just `example.com`).

When host names are specified in `pg_hba.conf`, you should make sure that name resolution is reasonably fast. It can be of advantage to set up a local name resolution cache such as `nscd`. Also, you may wish to enable the configuration parameter `log_hostname` to see the client's host name instead of the IP address in the log.

This field only applies to `host`, `hostssl`, and `hostnossl` records.

> Users sometimes wonder why host names are handled in this seemingly complicated way, with two name resolutions including a reverse lookup of the client's IP address. This complicates use of the feature in case the client's reverse DNS entry is not set up or yields some undesirable host name. It is done primarily for efficiency: this way, a connection attempt requires at most two resolver lookups, one reverse and one forward. If there is a resolver problem with some address, it becomes only that client's problem. A hypothetical alternative implementation that only did forward lookups would have to resolve every host name mentioned in `pg_hba.conf` during every connection attempt. That could be quite slow if many names are listed. And if there is a resolver problem with one of the host names, it becomes everyone's problem.
>
> Also, a reverse lookup is necessary to implement the suffix matching feature, because the actual client host name needs to be known in order to match it against the pattern.
>
> Note that this behavior is consistent with other popular implementations of host name-based access control, such as the Apache HTTP Server and TCP Wrappers.

IP-address
IP-mask

These two fields can be used as an alternative to the *IP-address/mask-length* notation. Instead of specifying the mask length, the actual mask is specified in a separate column. For example, `255.0.0.0` represents an IPv4 CIDR mask length of 8, and `255.255.255.255` represents a CIDR mask length of 32.

These fields only apply to `host`, `hostssl`, and `hostnossl` records.

auth-method

Specifies the authentication method to use when a connection matches this record. The possible choices are summarized here; details are in Section 19.3.

`trust`

Allow the connection unconditionally. This method allows anyone that can connect to the PostgreSQL database server to login as any PostgreSQL user they wish, without the need for a password or any other authentication. See Section 19.3.1 for details.

`reject`

Reject the connection unconditionally. This is useful for "filtering out" certain hosts from a group, for example a `reject` line could block a specific host from connecting, while a later line allows the remaining hosts in a specific network to connect.

`md5`

Require the client to supply a double-MD5-hashed password for authentication. See Section 19.3.2 for details.

`password`

Require the client to supply an unencrypted password for authentication. Since the password is sent in clear text over the network, this should not be used on untrusted networks. See Section

19.3.2 for details.

`gss`

Use GSSAPI to authenticate the user. This is only available for TCP/IP connections. See Section 19.3.3 for details.

`sspi`

Use SSPI to authenticate the user. This is only available on Windows. See Section 19.3.4 for details.

`ident`

Obtain the operating system user name of the client by contacting the ident server on the client and check if it matches the requested database user name. Ident authentication can only be used on TCP/IP connections. When specified for local connections, peer authentication will be used instead. See Section 19.3.5 for details.

`peer`

Obtain the client's operating system user name from the operating system and check if it matches the requested database user name. This is only available for local connections. See Section 19.3.6 for details.

`ldap`

Authenticate using an LDAP server. See Section 19.3.7 for details.

`radius`

Authenticate using a RADIUS server. See Section 19.3.8 for details.

`cert`

Authenticate using SSL client certificates. See Section 19.3.9 for details.

`pam`

Authenticate using the Pluggable Authentication Modules (PAM) service provided by the operating system. See Section 19.3.10 for details.

`auth-options`

After the *`auth-method`* field, there can be field(s) of the form *`name=value`* that specify options for the authentication method. Details about which options are available for which authentication methods appear below.

Files included by `@` constructs are read as lists of names, which can be separated by either whitespace or commas. Comments are introduced by `#`, just as in `pg_hba.conf`, and nested `@` constructs are allowed. Unless the file name following `@` is an absolute path, it is taken to be relative to the directory containing the referencing file.

Since the `pg_hba.conf` records are examined sequentially for each connection attempt, the order of the records is significant. Typically, earlier records will have tight connection match parameters and weaker authentication methods, while later records will have looser match parameters and stronger authentication methods. For example, one might wish to use `trust` authentication for local TCP/IP connections but

require a password for remote TCP/IP connections. In this case a record specifying `trust` authentication for connections from 127.0.0.1 would appear before a record specifying password authentication for a wider range of allowed client IP addresses.

The `pg_hba.conf` file is read on start-up and when the main server process receives a SIGHUP signal. If you edit the file on an active system, you will need to signal the postmaster (using `pg_ctl reload`, calling the SQL function `pg_reload_conf()`, or using `kill -HUP`) to make it re-read the file.

> **Tip:** To connect to a particular database, a user must not only pass the `pg_hba.conf` checks, but must have the `CONNECT` privilege for the database. If you wish to restrict which users can connect to which databases, it's usually easier to control this by granting/revoking `CONNECT` privilege than to put the rules in `pg_hba.conf` entries.

Some examples of `pg_hba.conf` entries are shown in Example 19-1. See the next section for details on the different authentication methods.

Example 19-1. Example `pg_hba.conf` Entries

```
# Allow any user on the local system to connect to any database with
# any database user name using Unix-domain sockets (the default for local
# connections).
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             all                                     trust

# The same using local loopback TCP/IP connections.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             127.0.0.1/32            trust

# The same as the previous line, but using a separate netmask column
#
# TYPE  DATABASE        USER            IP-ADDRESS      IP-MASK             METHOD
host    all             all             127.0.0.1       255.255.255.255     trust

# The same over IPv6.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             ::1/128                 trust

# The same using a host name (would typically cover both IPv4 and IPv6).
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             localhost               trust

# Allow any user from any host with IP address 192.168.93.x to connect
# to database "postgres" as the same user name that ident reports for
# the connection (typically the operating system user name).
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    postgres        all             192.168.93.0/24         ident
```

```
# Allow any user from host 192.168.12.10 to connect to database
# "postgres" if the user's password is correctly supplied.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    postgres        all             192.168.12.10/32        md5

# Allow any user from hosts in the example.com domain to connect to
# any database if the user's password is correctly supplied.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             .example.com            md5

# In the absence of preceding "host" lines, these two lines will
# reject all connections from 192.168.54.1 (since that entry will be
# matched first), but allow GSSAPI connections from anywhere else
# on the Internet.  The zero mask causes no bits of the host IP
# address to be considered, so it matches any host.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             192.168.54.1/32         reject
host    all             all             0.0.0.0/0               gss

# Allow users from 192.168.x.x hosts to connect to any database, if
# they pass the ident check.  If, for example, ident says the user is
# "bryanh" and he requests to connect as PostgreSQL user "guest1", the
# connection is allowed if there is an entry in pg_ident.conf for map
# "omicron" that says "bryanh" is allowed to connect as "guest1".
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             192.168.0.0/16          ident map=omicron

# If these are the only three lines for local connections, they will
# allow local users to connect only to their own databases (databases
# with the same name as their database user name) except for administrators
# and members of role "support", who can connect to all databases.  The file
# $PGDATA/admins contains a list of names of administrators.  Passwords
# are required in all cases.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   sameuser        all                                     md5
local   all             @admins                                 md5
local   all             +support                                md5

# The last two lines above can be combined into a single line:
local   all             @admins,+support                        md5

# The database column can also use lists and file names:
local   db1,db2,@demodbs  all                                   md5
```

19.2. User Name Maps

When using an external authentication system like Ident or GSSAPI, the name of the operating system user that initiated the connection might not be the same as the database user he needs to connect as. In this case, a user name map can be applied to map the operating system user name to a database user. To use user name mapping, specify `map=`*`map-name`* in the options field in `pg_hba.conf`. This option is supported for all authentication methods that receive external user names. Since different mappings might be needed for different connections, the name of the map to be used is specified in the *`map-name`* parameter in `pg_hba.conf` to indicate which map to use for each individual connection.

User name maps are defined in the ident map file, which by default is named `pg_ident.conf` and is stored in the cluster's data directory. (It is possible to place the map file elsewhere, however; see the ident_file configuration parameter.) The ident map file contains lines of the general form:

```
map-name system-username database-username
```

Comments and whitespace are handled in the same way as in `pg_hba.conf`. The *`map-name`* is an arbitrary name that will be used to refer to this mapping in `pg_hba.conf`. The other two fields specify an operating system user name and a matching database user name. The same *`map-name`* can be used repeatedly to specify multiple user-mappings within a single map.

There is no restriction regarding how many database users a given operating system user can correspond to, nor vice versa. Thus, entries in a map should be thought of as meaning "this operating system user is allowed to connect as this database user", rather than implying that they are equivalent. The connection will be allowed if there is any map entry that pairs the user name obtained from the external authentication system with the database user name that the user has requested to connect as.

If the *`system-username`* field starts with a slash (`/`), the remainder of the field is treated as a regular expression. (See Section 9.7.3.1 for details of PostgreSQL's regular expression syntax.) The regular expression can include a single capture, or parenthesized subexpression, which can then be referenced in the *`database-username`* field as `\1` (backslash-one). This allows the mapping of multiple user names in a single line, which is particularly useful for simple syntax substitutions. For example, these entries

```
mymap    /^(.*)@mydomain\.com$       \1
mymap    /^(.*)@otherdomain\.com$    guest
```

will remove the domain part for users with system user names that end with `@mydomain.com`, and allow any user whose system name ends with `@otherdomain.com` to log in as `guest`.

> **Tip:** Keep in mind that by default, a regular expression can match just part of a string. It's usually wise to use `^` and `$`, as shown in the above example, to force the match to be to the entire system user name.

The `pg_ident.conf` file is read on start-up and when the main server process receives a SIGHUP signal. If you edit the file on an active system, you will need to signal the postmaster (using `pg_ctl reload`, calling the SQL function `pg_reload_conf()`, or using `kill -HUP`) to make it re-read the file.

A `pg_ident.conf` file that could be used in conjunction with the `pg_hba.conf` file in Example 19-1 is shown in Example 19-2. In this example, anyone logged in to a machine on the 192.168 network that does not have the operating system user name `bryanh`, `ann`, or `robert` would not be granted access. Unix user `robert` would only be allowed access when he tries to connect as PostgreSQL user `bob`, not as `robert`

or anyone else. ann would only be allowed to connect as ann. User bryanh would be allowed to connect as either bryanh or as guest1.

Example 19-2. An Example pg_ident.conf File

```
# MAPNAME       SYSTEM-USERNAME         PG-USERNAME

omicron         bryanh                  bryanh
omicron         ann                     ann
# bob has user name robert on these machines
omicron         robert                  bob
# bryanh can also connect as guest1
omicron         bryanh                  guest1
```

19.3. Authentication Methods

The following subsections describe the authentication methods in more detail.

19.3.1. Trust Authentication

When trust authentication is specified, PostgreSQL assumes that anyone who can connect to the server is authorized to access the database with whatever database user name they specify (even superuser names). Of course, restrictions made in the database and user columns still apply. This method should only be used when there is adequate operating-system-level protection on connections to the server.

trust authentication is appropriate and very convenient for local connections on a single-user workstation. It is usually *not* appropriate by itself on a multiuser machine. However, you might be able to use trust even on a multiuser machine, if you restrict access to the server's Unix-domain socket file using file-system permissions. To do this, set the unix_socket_permissions (and possibly unix_socket_group) configuration parameters as described in Section 18.3. Or you could set the unix_socket_directories configuration parameter to place the socket file in a suitably restricted directory.

Setting file-system permissions only helps for Unix-socket connections. Local TCP/IP connections are not restricted by file-system permissions. Therefore, if you want to use file-system permissions for local security, remove the host ... 127.0.0.1 ... line from pg_hba.conf, or change it to a non-trust authentication method.

trust authentication is only suitable for TCP/IP connections if you trust every user on every machine that is allowed to connect to the server by the pg_hba.conf lines that specify trust. It is seldom reasonable to use trust for any TCP/IP connections other than those from localhost (127.0.0.1).

19.3.2. Password Authentication

The password-based authentication methods are md5 and password. These methods operate similarly except for the way that the password is sent across the connection, namely MD5-hashed and clear-text respectively.

If you are at all concerned about password "sniffing" attacks then `md5` is preferred. Plain `password` should always be avoided if possible. However, `md5` cannot be used with the db_user_namespace feature. If the connection is protected by SSL encryption then `password` can be used safely (though SSL certificate authentication might be a better choice if one is depending on using SSL).

PostgreSQL database passwords are separate from operating system user passwords. The password for each database user is stored in the `pg_authid` system catalog. Passwords can be managed with the SQL commands CREATE USER and ALTER ROLE, e.g., **`CREATE USER foo WITH PASSWORD 'secret'`**. If no password has been set up for a user, the stored password is null and password authentication will always fail for that user.

19.3.3. GSSAPI Authentication

GSSAPI is an industry-standard protocol for secure authentication defined in RFC 2743. PostgreSQL supports GSSAPI with Kerberos authentication according to RFC 1964. GSSAPI provides automatic authentication (single sign-on) for systems that support it. The authentication itself is secure, but the data sent over the database connection will be sent unencrypted unless SSL is used.

GSSAPI support has to be enabled when PostgreSQL is built; see Chapter 15 for more information.

When GSSAPI uses Kerberos, it uses a standard principal in the format *`servicename/hostname@realm`*. The PostgreSQL server will accept any principal that is included in the keytab used by the server, but care needs to be taken to specify the correct principal details when making the connection from the client using the `krbsrvname` connection parameter. (See also Section 31.1.2.) The installation default can be changed from the default `postgres` at build time using `./configure --with-krb-srvnam=`*`whatever`*. In most environments, this parameter never needs to be changed. Some Kerberos implementations might require a different service name, such as Microsoft Active Directory which requires the service name to be in upper case (`POSTGRES`).

`hostname` is the fully qualified host name of the server machine. The service principal's realm is the preferred realm of the server machine.

Client principals can be mapped to different PostgreSQL database user names with `pg_ident.conf`. For example, `pgusername@realm` could be mapped to just `pgusername`. Alternatively, you can use the full `username@realm` principal as the role name in PostgreSQL without any mapping.

PostgreSQL also supports a parameter to strip the realm from the principal. This method is supported for backwards compatibility and is strongly discouraged as it is then impossible to distinguish different users with the same user name but coming from different realms. To enable this, set `include_realm` to 0. For simple single-realm installations, `include_realm` combined with the `krb_realm` parameter (which checks that the realm provided matches exactly what is in the `krb_realm` parameter) would be a secure but less capable option compared to specifying an explicit mapping in `pg_ident.conf`.

Make sure that your server keytab file is readable (and preferably only readable) by the PostgreSQL server account. (See also Section 17.1.) The location of the key file is specified by the krb_server_keyfile configuration parameter. The default is `/usr/local/pgsql/etc/krb5.keytab` (or whatever directory was specified as `sysconfdir` at build time). For security reasons, it is recommended to use a separate keytab just for the PostgreSQL server rather than opening up permissions on the system keytab file.

The keytab file is generated by the Kerberos software; see the Kerberos documentation for details. The following example is for MIT-compatible Kerberos 5 implementations:

```
kadmin% ank -randkey postgres/server.my.domain.org
kadmin% ktadd -k krb5.keytab postgres/server.my.domain.org
```

When connecting to the database make sure you have a ticket for a principal matching the requested database user name. For example, for database user name `fred`, principal `fred@EXAMPLE.COM` would be able to connect. To also allow principal `fred/users.example.com@EXAMPLE.COM`, use a user name map, as described in Section 19.2.

The following configuration options are supported for GSSAPI:

`include_realm`

If set to 0, the realm name from the authenticated user principal is stripped off before being passed through the user name mapping (Section 19.2). This is discouraged and is primarily available for backwards compatibility as it is not secure in multi-realm environments unless `krb_realm` is also used. Users are recommended to leave include_realm set to the default (1) and to provide an explicit mapping in `pg_ident.conf`.

`map`

Allows for mapping between system and database user names. See Section 19.2 for details. For a GSSAPI/Kerberos principal, such as `username@EXAMPLE.COM` (or, less commonly, `username/hostbased@EXAMPLE.COM`), the user name used for mapping is `username@EXAMPLE.COM` (or `username/hostbased@EXAMPLE.COM`, respectively), unless `include_realm` has been set to 0, in which case `username` (or `username/hostbased`) is what is seen as the system user name when mapping.

`krb_realm`

Sets the realm to match user principal names against. If this parameter is set, only users of that realm will be accepted. If it is not set, users of any realm can connect, subject to whatever user name mapping is done.

19.3.4. SSPI Authentication

SSPI is a Windows technology for secure authentication with single sign-on. PostgreSQL will use SSPI in `negotiate` mode, which will use Kerberos when possible and automatically fall back to NTLM in other cases. SSPI authentication only works when both server and client are running Windows, or, on non-Windows platforms, when GSSAPI is available.

When using Kerberos authentication, SSPI works the same way GSSAPI does; see Section 19.3.3 for details.

The following configuration options are supported for SSPI:

`include_realm`

If set to 0, the realm name from the authenticated user principal is stripped off before being passed through the user name mapping (Section 19.2). This is discouraged and is primarily available for backwards compatibility as it is not secure in multi-realm environments unless `krb_realm` is also

used. Users are recommended to leave include_realm set to the default (1) and to provide an explicit mapping in `pg_ident.conf`.

`map`

Allows for mapping between system and database user names. See Section 19.2 for details. For a SSPI/Kerberos principal, such as `username@EXAMPLE.COM` (or, less commonly, `username/hostbased@EXAMPLE.COM`), the user name used for mapping is `username@EXAMPLE.COM` (or `username/hostbased@EXAMPLE.COM`, respectively), unless `include_realm` has been set to 0, in which case `username` (or `username/hostbased`) is what is seen as the system user name when mapping.

`krb_realm`

Sets the realm to match user principal names against. If this parameter is set, only users of that realm will be accepted. If it is not set, users of any realm can connect, subject to whatever user name mapping is done.

19.3.5. Ident Authentication

The ident authentication method works by obtaining the client's operating system user name from an ident server and using it as the allowed database user name (with an optional user name mapping). This is only supported on TCP/IP connections.

Note: When ident is specified for a local (non-TCP/IP) connection, peer authentication (see Section 19.3.6) will be used instead.

The following configuration options are supported for ident:

`map`

Allows for mapping between system and database user names. See Section 19.2 for details.

The "Identification Protocol" is described in RFC 1413. Virtually every Unix-like operating system ships with an ident server that listens on TCP port 113 by default. The basic functionality of an ident server is to answer questions like "What user initiated the connection that goes out of your port *X* and connects to my port *Y*?". Since PostgreSQL knows both *X* and *Y* when a physical connection is established, it can interrogate the ident server on the host of the connecting client and can theoretically determine the operating system user for any given connection.

The drawback of this procedure is that it depends on the integrity of the client: if the client machine is untrusted or compromised, an attacker could run just about any program on port 113 and return any user name he chooses. This authentication method is therefore only appropriate for closed networks where each client machine is under tight control and where the database and system administrators operate in close contact. In other words, you must trust the machine running the ident server. Heed the warning:

> The Identification Protocol is not intended as an authorization or access control protocol.

—RFC 1413

Some ident servers have a nonstandard option that causes the returned user name to be encrypted, using a key that only the originating machine's administrator knows. This option *must not* be used when using the ident server with PostgreSQL, since PostgreSQL does not have any way to decrypt the returned string to determine the actual user name.

19.3.6. Peer Authentication

The peer authentication method works by obtaining the client's operating system user name from the kernel and using it as the allowed database user name (with optional user name mapping). This method is only supported on local connections.

The following configuration options are supported for peer:

`map`

Allows for mapping between system and database user names. See Section 19.2 for details.

Peer authentication is only available on operating systems providing the `getpeereid()` function, the `SO_PEERCRED` socket parameter, or similar mechanisms. Currently that includes Linux, most flavors of BSD including OS X, and Solaris.

19.3.7. LDAP Authentication

This authentication method operates similarly to `password` except that it uses LDAP as the password verification method. LDAP is used only to validate the user name/password pairs. Therefore the user must already exist in the database before LDAP can be used for authentication.

LDAP authentication can operate in two modes. In the first mode, which we will call the simple bind mode, the server will bind to the distinguished name constructed as *`prefix username suffix`*. Typically, the *`prefix`* parameter is used to specify `cn=`, or *`DOMAIN`*`\` in an Active Directory environment. *`suffix`* is used to specify the remaining part of the DN in a non-Active Directory environment.

In the second mode, which we will call the search+bind mode, the server first binds to the LDAP directory with a fixed user name and password, specified with *`ldapbinddn`* and *`ldapbindpasswd`*, and performs a search for the user trying to log in to the database. If no user and password is configured, an anonymous bind will be attempted to the directory. The search will be performed over the subtree at *`ldapbasedn`*, and will try to do an exact match of the attribute specified in *`ldapsearchattribute`*. Once the user has been found in this search, the server disconnects and re-binds to the directory as this user, using the password specified by the client, to verify that the login is correct. This mode is the same as that used by LDAP authentication schemes in other software, such as Apache `mod_authnz_ldap` and `pam_ldap`. This method allows for significantly more flexibility in where the user objects are located in the directory, but will cause two separate connections to the LDAP server to be made.

The following configuration options are used in both modes:

`ldapserver`

Names or IP addresses of LDAP servers to connect to. Multiple servers may be specified, separated by spaces.

`ldapport`

Port number on LDAP server to connect to. If no port is specified, the LDAP library's default port setting will be used.

`ldaptls`

Set to 1 to make the connection between PostgreSQL and the LDAP server use TLS encryption. Note that this only encrypts the traffic to the LDAP server — the connection to the client will still be unencrypted unless SSL is used.

The following options are used in simple bind mode only:

`ldapprefix`

String to prepend to the user name when forming the DN to bind as, when doing simple bind authentication.

`ldapsuffix`

String to append to the user name when forming the DN to bind as, when doing simple bind authentication.

The following options are used in search+bind mode only:

`ldapbasedn`

Root DN to begin the search for the user in, when doing search+bind authentication.

`ldapbinddn`

DN of user to bind to the directory with to perform the search when doing search+bind authentication.

`ldapbindpasswd`

Password for user to bind to the directory with to perform the search when doing search+bind authentication.

`ldapsearchattribute`

Attribute to match against the user name in the search when doing search+bind authentication. If no attribute is specified, the `uid` attribute will be used.

`ldapurl`

An RFC 4516 LDAP URL. This is an alternative way to write some of the other LDAP options in a more compact and standard form. The format is

`ldap://`*`host`*`[:`*`port`*`]/`*`basedn`*`[?[`*`attribute`*`][?[`*`scope`*`]]]`
`scope` must be one of `base`, `one`, `sub`, typically the latter. Only one attribute is used, and some other components of standard LDAP URLs such as filters and extensions are not supported.

For non-anonymous binds, `ldapbinddn` and `ldapbindpasswd` must be specified as separate options.

To use encrypted LDAP connections, the `ldaptls` option has to be used in addition to `ldapurl`. The `ldaps` URL scheme (direct SSL connection) is not supported.

LDAP URLs are currently only supported with OpenLDAP, not on Windows.

It is an error to mix configuration options for simple bind with options for search+bind.

Here is an example for a simple-bind LDAP configuration:

```
host ... ldap ldapserver=ldap.example.net ldapprefix="cn=" ldapsuffix=", dc=example, dc=net
```

When a connection to the database server as database user `someuser` is requested, PostgreSQL will attempt to bind to the LDAP server using the DN `cn=someuser, dc=example, dc=net` and the password provided by the client. If that connection succeeds, the database access is granted.

Here is an example for a search+bind configuration:

```
host ... ldap ldapserver=ldap.example.net ldapbasedn="dc=example, dc=net" ldapsearchattribu
```

When a connection to the database server as database user `someuser` is requested, PostgreSQL will attempt to bind anonymously (since `ldapbinddn` was not specified) to the LDAP server, perform a search for `(uid=someuser)` under the specified base DN. If an entry is found, it will then attempt to bind using that found information and the password supplied by the client. If that second connection succeeds, the database access is granted.

Here is the same search+bind configuration written as a URL:

```
host ... ldap ldapurl="ldap://ldap.example.net/dc=example,dc=net?uid?sub"
```

Some other software that supports authentication against LDAP uses the same URL format, so it will be easier to share the configuration.

Tip: Since LDAP often uses commas and spaces to separate the different parts of a DN, it is often necessary to use double-quoted parameter values when configuring LDAP options, as shown in the examples.

19.3.8. RADIUS Authentication

This authentication method operates similarly to `password` except that it uses RADIUS as the password verification method. RADIUS is used only to validate the user name/password pairs. Therefore the user must already exist in the database before RADIUS can be used for authentication.

When using RADIUS authentication, an Access Request message will be sent to the configured RADIUS server. This request will be of type `Authenticate Only`, and include parameters for `user name`, `password` (encrypted) and `NAS Identifier`. The request will be encrypted using a secret shared with the server. The RADIUS server will respond to this server with either `Access Accept` or `Access Reject`. There is no support for RADIUS accounting.

The following configuration options are supported for RADIUS:

`radiusserver`

The name or IP address of the RADIUS server to connect to. This parameter is required.

`radiussecret`

The shared secret used when talking securely to the RADIUS server. This must have exactly the same value on the PostgreSQL and RADIUS servers. It is recommended that this be a string of at least 16 characters. This parameter is required.

> **Note:** The encryption vector used will only be cryptographically strong if PostgreSQL is built with support for OpenSSL. In other cases, the transmission to the RADIUS server should only be considered obfuscated, not secured, and external security measures should be applied if necessary.

`radiusport`

The port number on the RADIUS server to connect to. If no port is specified, the default port `1812` will be used.

`radiusidentifier`

The string used as `NAS Identifier` in the RADIUS requests. This parameter can be used as a second parameter identifying for example which database user the user is attempting to authenticate as, which can be used for policy matching on the RADIUS server. If no identifier is specified, the default `postgresql` will be used.

19.3.9. Certificate Authentication

This authentication method uses SSL client certificates to perform authentication. It is therefore only available for SSL connections. When using this authentication method, the server will require that the client provide a valid certificate. No password prompt will be sent to the client. The `cn` (Common Name) attribute of the certificate will be compared to the requested database user name, and if they match the login will be allowed. User name mapping can be used to allow `cn` to be different from the database user name.

The following configuration options are supported for SSL certificate authentication:

`map`

Allows for mapping between system and database user names. See Section 19.2 for details.

19.3.10. PAM Authentication

This authentication method operates similarly to `password` except that it uses PAM (Pluggable Authentication Modules) as the authentication mechanism. The default PAM service name is `postgresql`. PAM is used only to validate user name/password pairs. Therefore the user must already exist in the database

before PAM can be used for authentication. For more information about PAM, please read the Linux-PAM Page[1].

The following configuration options are supported for PAM:

`pamservice`

PAM service name.

Note: If PAM is set up to read `/etc/shadow`, authentication will fail because the PostgreSQL server is started by a non-root user. However, this is not an issue when PAM is configured to use LDAP or other authentication methods.

19.4. Authentication Problems

Authentication failures and related problems generally manifest themselves through error messages like the following:

```
FATAL:  no pg_hba.conf entry for host "123.123.123.123", user "andym", database "testdb"
```

This is what you are most likely to get if you succeed in contacting the server, but it does not want to talk to you. As the message suggests, the server refused the connection request because it found no matching entry in its `pg_hba.conf` configuration file.

```
FATAL:  password authentication failed for user "andym"
```

Messages like this indicate that you contacted the server, and it is willing to talk to you, but not until you pass the authorization method specified in the `pg_hba.conf` file. Check the password you are providing, or check your Kerberos or ident software if the complaint mentions one of those authentication types.

```
FATAL:  user "andym" does not exist
```

The indicated database user name was not found.

```
FATAL:  database "testdb" does not exist
```

The database you are trying to connect to does not exist. Note that if you do not specify a database name, it defaults to the database user name, which might or might not be the right thing.

Tip: The server log might contain more information about an authentication failure than is reported to the client. If you are confused about the reason for a failure, check the server log.

1. http://www.kernel.org/pub/linux/libs/pam/

Chapter 20. Database Roles

PostgreSQL manages database access permissions using the concept of *roles*. A role can be thought of as either a database user, or a group of database users, depending on how the role is set up. Roles can own database objects (for example, tables and functions) and can assign privileges on those objects to other roles to control who has access to which objects. Furthermore, it is possible to grant *membership* in a role to another role, thus allowing the member role to use privileges assigned to another role.

The concept of roles subsumes the concepts of "users" and "groups". In PostgreSQL versions before 8.1, users and groups were distinct kinds of entities, but now there are only roles. Any role can act as a user, a group, or both.

This chapter describes how to create and manage roles. More information about the effects of role privileges on various database objects can be found in Section 5.6.

20.1. Database Roles

Database roles are conceptually completely separate from operating system users. In practice it might be convenient to maintain a correspondence, but this is not required. Database roles are global across a database cluster installation (and not per individual database). To create a role use the CREATE ROLE SQL command:

```
CREATE ROLE name;
```

name follows the rules for SQL identifiers: either unadorned without special characters, or double-quoted. (In practice, you will usually want to add additional options, such as `LOGIN`, to the command. More details appear below.) To remove an existing role, use the analogous DROP ROLE command:

```
DROP ROLE name;
```

For convenience, the programs createuser and dropuser are provided as wrappers around these SQL commands that can be called from the shell command line:

```
createuser name
dropuser name
```

To determine the set of existing roles, examine the `pg_roles` system catalog, for example

```
SELECT rolname FROM pg_roles;
```

The psql program's `\du` meta-command is also useful for listing the existing roles.

In order to bootstrap the database system, a freshly initialized system always contains one predefined role. This role is always a "superuser", and by default (unless altered when running `initdb`) it will have the same name as the operating system user that initialized the database cluster. Customarily, this role will be named `postgres`. In order to create more roles you first have to connect as this initial role.

Every connection to the database server is made using the name of some particular role, and this role determines the initial access privileges for commands issued in that connection. The role name to use for a particular database connection is indicated by the client that is initiating the connection request in an application-specific fashion. For example, the `psql` program uses the `-U` command line option to indicate the role to connect as. Many applications assume the name of the current operating system user by default (including `createuser` and `psql`). Therefore it is often convenient to maintain a naming correspondence between roles and operating system users.

The set of database roles a given client connection can connect as is determined by the client authentication setup, as explained in Chapter 19. (Thus, a client is not limited to connect as the role matching its operating system user, just as a person's login name need not match her real name.) Since the role identity determines the set of privileges available to a connected client, it is important to carefully configure privileges when setting up a multiuser environment.

20.2. Role Attributes

A database role can have a number of attributes that define its privileges and interact with the client authentication system.

login privilege

Only roles that have the `LOGIN` attribute can be used as the initial role name for a database connection. A role with the `LOGIN` attribute can be considered the same as a "database user". To create a role with login privilege, use either:

```
CREATE ROLE name LOGIN;
CREATE USER name;
```

(`CREATE USER` is equivalent to `CREATE ROLE` except that `CREATE USER` assumes `LOGIN` by default, while `CREATE ROLE` does not.)

superuser status

A database superuser bypasses all permission checks, except the right to log in. This is a dangerous privilege and should not be used carelessly; it is best to do most of your work as a role that is not a superuser. To create a new database superuser, use `CREATE ROLE name SUPERUSER`. You must do this as a role that is already a superuser.

database creation

A role must be explicitly given permission to create databases (except for superusers, since those bypass all permission checks). To create such a role, use `CREATE ROLE name CREATEDB`.

role creation

A role must be explicitly given permission to create more roles (except for superusers, since those bypass all permission checks). To create such a role, use `CREATE ROLE name CREATEROLE`. A role with `CREATEROLE` privilege can alter and drop other roles, too, as well as grant or revoke membership in them. However, to create, alter, drop, or change membership of a superuser role, superuser status is required; `CREATEROLE` is insufficient for that.

initiating replication

A role must explicitly be given permission to initiate streaming replication (except for superusers, since those bypass all permission checks). A role used for streaming replication must have LOGIN permission as well. To create such a role, use CREATE ROLE *name* REPLICATION LOGIN.

password

A password is only significant if the client authentication method requires the user to supply a password when connecting to the database. The password and md5 authentication methods make use of passwords. Database passwords are separate from operating system passwords. Specify a password upon role creation with CREATE ROLE *name* PASSWORD '*string*'.

A role's attributes can be modified after creation with ALTER ROLE. See the reference pages for the CREATE ROLE and ALTER ROLE commands for details.

Tip: It is good practice to create a role that has the CREATEDB and CREATEROLE privileges, but is not a superuser, and then use this role for all routine management of databases and roles. This approach avoids the dangers of operating as a superuser for tasks that do not really require it.

A role can also have role-specific defaults for many of the run-time configuration settings described in Chapter 18. For example, if for some reason you want to disable index scans (hint: not a good idea) anytime you connect, you can use:

```
ALTER ROLE myname SET enable_indexscan TO off;
```

This will save the setting (but not set it immediately). In subsequent connections by this role it will appear as though SET enable_indexscan TO off had been executed just before the session started. You can still alter this setting during the session; it will only be the default. To remove a role-specific default setting, use ALTER ROLE *rolename* RESET *varname*. Note that role-specific defaults attached to roles without LOGIN privilege are fairly useless, since they will never be invoked.

20.3. Role Membership

It is frequently convenient to group users together to ease management of privileges: that way, privileges can be granted to, or revoked from, a group as a whole. In PostgreSQL this is done by creating a role that represents the group, and then granting *membership* in the group role to individual user roles.

To set up a group role, first create the role:

```
CREATE ROLE name;
```

Typically a role being used as a group would not have the LOGIN attribute, though you can set it if you wish.

Once the group role exists, you can add and remove members using the GRANT and REVOKE commands:

```
GRANT group_role TO role1, ... ;
REVOKE group_role FROM role1, ... ;
```

You can grant membership to other group roles, too (since there isn't really any distinction between group roles and non-group roles). The database will not let you set up circular membership loops. Also, it is not permitted to grant membership in a role to `PUBLIC`.

The members of a group role can use the privileges of the role in two ways. First, every member of a group can explicitly do SET ROLE to temporarily "become" the group role. In this state, the database session has access to the privileges of the group role rather than the original login role, and any database objects created are considered owned by the group role not the login role. Second, member roles that have the `INHERIT` attribute automatically have use of the privileges of roles of which they are members, including any privileges inherited by those roles. As an example, suppose we have done:

```
CREATE ROLE joe LOGIN INHERIT;
CREATE ROLE admin NOINHERIT;
CREATE ROLE wheel NOINHERIT;
GRANT admin TO joe;
GRANT wheel TO admin;
```

Immediately after connecting as role `joe`, a database session will have use of privileges granted directly to `joe` plus any privileges granted to `admin`, because `joe` "inherits" `admin`'s privileges. However, privileges granted to `wheel` are not available, because even though `joe` is indirectly a member of `wheel`, the membership is via `admin` which has the `NOINHERIT` attribute. After:

```
SET ROLE admin;
```

the session would have use of only those privileges granted to `admin`, and not those granted to `joe`. After:

```
SET ROLE wheel;
```

the session would have use of only those privileges granted to `wheel`, and not those granted to either `joe` or `admin`. The original privilege state can be restored with any of:

```
SET ROLE joe;
SET ROLE NONE;
RESET ROLE;
```

Note: The `SET ROLE` command always allows selecting any role that the original login role is directly or indirectly a member of. Thus, in the above example, it is not necessary to become `admin` before becoming `wheel`.

Note: In the SQL standard, there is a clear distinction between users and roles, and users do not automatically inherit privileges while roles do. This behavior can be obtained in PostgreSQL by giving roles being used as SQL roles the `INHERIT` attribute, while giving roles being used as SQL users the `NOINHERIT` attribute. However, PostgreSQL defaults to giving all roles the `INHERIT` attribute, for backward compatibility with pre-8.1 releases in which users always had use of permissions granted to groups they were members of.

The role attributes LOGIN, SUPERUSER, CREATEDB, and CREATEROLE can be thought of as special privileges, but they are never inherited as ordinary privileges on database objects are. You must actually SET ROLE to a specific role having one of these attributes in order to make use of the attribute. Continuing the above example, we might choose to grant CREATEDB and CREATEROLE to the admin role. Then a session connecting as role joe would not have these privileges immediately, only after doing SET ROLE admin.

To destroy a group role, use DROP ROLE:

```
DROP ROLE name;
```

Any memberships in the group role are automatically revoked (but the member roles are not otherwise affected).

20.4. Dropping Roles

Because roles can own database objects and can hold privileges to access other objects, dropping a role is often not just a matter of a quick DROP ROLE. Any objects owned by the role must first be dropped or reassigned to other owners; and any permissions granted to the role must be revoked.

Ownership of objects can be transferred one at a time using ALTER commands, for example:

```
ALTER TABLE bobs_table OWNER TO alice;
```

Alternatively, the REASSIGN OWNED command can be used to reassign ownership of all objects owned by the role-to-be-dropped to a single other role. Because REASSIGN OWNED cannot access objects in other databases, it is necessary to run it in each database that contains objects owned by the role. (Note that the first such REASSIGN OWNED will change the ownership of any shared-across-databases objects, that is databases or tablespaces, that are owned by the role-to-be-dropped.)

Once any valuable objects have been transferred to new owners, any remaining objects owned by the role-to-be-dropped can be dropped with the DROP OWNED command. Again, this command cannot access objects in other databases, so it is necessary to run it in each database that contains objects owned by the role. Also, DROP OWNED will not drop entire databases or tablespaces, so it is necessary to do that manually if the role owns any databases or tablespaces that have not been transferred to new owners.

DROP OWNED also takes care of removing any privileges granted to the target role for objects that do not belong to it. Because REASSIGN OWNED does not touch such objects, it's typically necessary to run both REASSIGN OWNED and DROP OWNED (in that order!) to fully remove the dependencies of a role to be dropped.

In short then, the most general recipe for removing a role that has been used to own objects is:

```
REASSIGN OWNED BY doomed_role TO successor_role;
DROP OWNED BY doomed_role;
-- repeat the above commands in each database of the cluster
DROP ROLE doomed_role;
```

When not all owned objects are to be transferred to the same successor owner, it's best to handle the exceptions manually and then perform the above steps to mop up.

If `DROP ROLE` is attempted while dependent objects still remain, it will issue messages identifying which objects need to be reassigned or dropped.

20.5. Function Security

Functions, triggers and row-level security policies allow users to insert code into the backend server that other users might execute unintentionally. Hence, these mechanisms permit users to "Trojan horse" others with relative ease. The strongest protection is tight control over who can define objects. Where that is infeasible, write queries referring only to objects having trusted owners. Remove from `search_path` the public schema and any other schemas that permit untrusted users to create objects.

Functions run inside the backend server process with the operating system permissions of the database server daemon. If the programming language used for the function allows unchecked memory accesses, it is possible to change the server's internal data structures. Hence, among many other things, such functions can circumvent any system access controls. Function languages that allow such access are considered "untrusted", and PostgreSQL allows only superusers to create functions written in those languages.

Chapter 21. Managing Databases

Every instance of a running PostgreSQL server manages one or more databases. Databases are therefore the topmost hierarchical level for organizing SQL objects ("database objects"). This chapter describes the properties of databases, and how to create, manage, and destroy them.

21.1. Overview

A small number of objects, like role, database, and tablespace names, are defined at the cluster level and stored in the `pg_global` tablespace. Inside the cluster are multiple databases, which are isolated from each other but can access cluster-level objects. Inside each database are multiple schemas, which contain objects like tables and functions. So the full hierarchy is: cluster, database, schema, table (or some other kind of object, such as a function).

When connecting to the database server, a client must specify the database name in its connection request. It is not possible to access more than one database per connection. However, clients can open multiple connections to the same database, or different databases. Database-level security has two components: access control (see Section 19.1), managed at the connection level, and authorization control (see Section 5.6), managed via the grant system. Foreign data wrappers (see postgres_fdw) allow for objects within one database to act as proxies for objects in other database or clusters. The older dblink module (see dblink) provides a similar capability. By default, all users can connect to all databases using all connection methods.

If one PostgreSQL server cluster is planned to contain unrelated projects or users that should be, for the most part, unaware of each other, it is recommended to put them into separate databases and adjust authorizations and access controls accordingly. If the projects or users are interrelated, and thus should be able to use each other's resources, they should be put in the same database but probably into separate schemas; this provides a modular structure with namespace isolation and authorization control. More information about managing schemas is in Section 5.8.

While multiple databases can be created within a single cluster, it is advised to consider carefully whether the benefits outweigh the risks and limitations. In particular, the impact that having a shared WAL (see Chapter 29) has on backup and recovery options. While individual databases in the cluster are isolated when considered from the user's perspective, they are closely bound from the database administrator's point-of-view.

Databases are created with the `CREATE DATABASE` command (see Section 21.2) and destroyed with the `DROP DATABASE` command (see Section 21.5). To determine the set of existing databases, examine the `pg_database` system catalog, for example

```
SELECT datname FROM pg_database;
```

The psql program's `\l` meta-command and `-l` command-line option are also useful for listing the existing databases.

> **Note:** The SQL standard calls databases "catalogs", but there is no difference in practice.

21.2. Creating a Database

In order to create a database, the PostgreSQL server must be up and running (see Section 17.3).

Databases are created with the SQL command CREATE DATABASE:

```
CREATE DATABASE name;
```

where *name* follows the usual rules for SQL identifiers. The current role automatically becomes the owner of the new database. It is the privilege of the owner of a database to remove it later (which also removes all the objects in it, even if they have a different owner).

The creation of databases is a restricted operation. See Section 20.2 for how to grant permission.

Since you need to be connected to the database server in order to execute the `CREATE DATABASE` command, the question remains how the *first* database at any given site can be created. The first database is always created by the `initdb` command when the data storage area is initialized. (See Section 17.2.) This database is called `postgres`. So to create the first "ordinary" database you can connect to `postgres`.

A second database, `template1`, is also created during database cluster initialization. Whenever a new database is created within the cluster, `template1` is essentially cloned. This means that any changes you make in `template1` are propagated to all subsequently created databases. Because of this, avoid creating objects in `template1` unless you want them propagated to every newly created database. More details appear in Section 21.3.

As a convenience, there is a program you can execute from the shell to create new databases, `createdb`.

```
createdb dbname
```

`createdb` does no magic. It connects to the `postgres` database and issues the `CREATE DATABASE` command, exactly as described above. The createdb reference page contains the invocation details. Note that `createdb` without any arguments will create a database with the current user name.

> **Note:** Chapter 19 contains information about how to restrict who can connect to a given database.

Sometimes you want to create a database for someone else, and have him become the owner of the new database, so he can configure and manage it himself. To achieve that, use one of the following commands:

```
CREATE DATABASE dbname OWNER rolename;
```

from the SQL environment, or:

```
createdb -O rolename dbname
```

from the shell. Only the superuser is allowed to create a database for someone else (that is, for a role you are not a member of).

21.3. Template Databases

CREATE DATABASE actually works by copying an existing database. By default, it copies the standard system database named template1. Thus that database is the "template" from which new databases are made. If you add objects to template1, these objects will be copied into subsequently created user databases. This behavior allows site-local modifications to the standard set of objects in databases. For example, if you install the procedural language PL/Perl in template1, it will automatically be available in user databases without any extra action being taken when those databases are created.

There is a second standard system database named template0. This database contains the same data as the initial contents of template1, that is, only the standard objects predefined by your version of PostgreSQL. template0 should never be changed after the database cluster has been initialized. By instructing CREATE DATABASE to copy template0 instead of template1, you can create a "virgin" user database that contains none of the site-local additions in template1. This is particularly handy when restoring a pg_dump dump: the dump script should be restored in a virgin database to ensure that one recreates the correct contents of the dumped database, without conflicting with objects that might have been added to template1 later on.

Another common reason for copying template0 instead of template1 is that new encoding and locale settings can be specified when copying template0, whereas a copy of template1 must use the same settings it does. This is because template1 might contain encoding-specific or locale-specific data, while template0 is known not to.

To create a database by copying template0, use:

```
CREATE DATABASE dbname TEMPLATE template0;
```

from the SQL environment, or:

```
createdb -T template0 dbname
```

from the shell.

It is possible to create additional template databases, and indeed one can copy any database in a cluster by specifying its name as the template for CREATE DATABASE. It is important to understand, however, that this is not (yet) intended as a general-purpose "COPY DATABASE" facility. The principal limitation is that no other sessions can be connected to the source database while it is being copied. CREATE DATABASE will fail if any other connection exists when it starts; during the copy operation, new connections to the source database are prevented.

Two useful flags exist in pg_database for each database: the columns datistemplate and datallowconn. datistemplate can be set to indicate that a database is intended as a template for CREATE DATABASE. If this flag is set, the database can be cloned by any user with CREATEDB privileges; if it is not set, only superusers and the owner of the database can clone it. If datallowconn is false, then no new connections to that database will be allowed (but existing sessions are not terminated simply by setting the flag false). The template0 database is normally marked datallowconn = false to prevent its modification. Both template0 and template1 should always be marked with datistemplate = true.

> **Note:** template1 and template0 do not have any special status beyond the fact that the name template1 is the default source database name for CREATE DATABASE. For example, one could drop template1 and recreate it from template0 without any ill effects. This course of action might be ad-

visable if one has carelessly added a bunch of junk in `template1`. (To delete `template1`, it must have `pg_database.datistemplate = false`.)

The `postgres` database is also created when a database cluster is initialized. This database is meant as a default database for users and applications to connect to. It is simply a copy of `template1` and can be dropped and recreated if necessary.

21.4. Database Configuration

Recall from Chapter 18 that the PostgreSQL server provides a large number of run-time configuration variables. You can set database-specific default values for many of these settings.

For example, if for some reason you want to disable the GEQO optimizer for a given database, you'd ordinarily have to either disable it for all databases or make sure that every connecting client is careful to issue `SET geqo TO off`. To make this setting the default within a particular database, you can execute the command:

```
ALTER DATABASE mydb SET geqo TO off;
```

This will save the setting (but not set it immediately). In subsequent connections to this database it will appear as though `SET geqo TO off;` had been executed just before the session started. Note that users can still alter this setting during their sessions; it will only be the default. To undo any such setting, use `ALTER DATABASE` *`dbname`* `RESET` *`varname`*.

21.5. Destroying a Database

Databases are destroyed with the command DROP DATABASE:

```
DROP DATABASE name;
```

Only the owner of the database, or a superuser, can drop a database. Dropping a database removes all objects that were contained within the database. The destruction of a database cannot be undone.

You cannot execute the `DROP DATABASE` command while connected to the victim database. You can, however, be connected to any other database, including the `template1` database. `template1` would be the only option for dropping the last user database of a given cluster.

For convenience, there is also a shell program to drop databases, dropdb:

```
dropdb dbname
```

(Unlike `createdb`, it is not the default action to drop the database with the current user name.)

21.6. Tablespaces

Tablespaces in PostgreSQL allow database administrators to define locations in the file system where the files representing database objects can be stored. Once created, a tablespace can be referred to by name when creating database objects.

By using tablespaces, an administrator can control the disk layout of a PostgreSQL installation. This is useful in at least two ways. First, if the partition or volume on which the cluster was initialized runs out of space and cannot be extended, a tablespace can be created on a different partition and used until the system can be reconfigured.

Second, tablespaces allow an administrator to use knowledge of the usage pattern of database objects to optimize performance. For example, an index which is very heavily used can be placed on a very fast, highly available disk, such as an expensive solid state device. At the same time a table storing archived data which is rarely used or not performance critical could be stored on a less expensive, slower disk system.

> **Warning**
>
> Even though located outside the main PostgreSQL data directory, tablespaces are an integral part of the database cluster and *cannot* be treated as an autonomous collection of data files. They are dependent on metadata contained in the main data directory, and therefore cannot be attached to a different database cluster or backed up individually. Similarly, if you lose a tablespace (file deletion, disk failure, etc), the database cluster might become unreadable or unable to start. Placing a tablespace on a temporary file system like a RAM disk risks the reliability of the entire cluster.

To define a tablespace, use the CREATE TABLESPACE command, for example::

```
CREATE TABLESPACE fastspace LOCATION '/ssd1/postgresql/data';
```

The location must be an existing, empty directory that is owned by the PostgreSQL operating system user. All objects subsequently created within the tablespace will be stored in files underneath this directory. The location must not be on removable or transient storage, as the cluster might fail to function if the tablespace is missing or lost.

> **Note:** There is usually not much point in making more than one tablespace per logical file system, since you cannot control the location of individual files within a logical file system. However, PostgreSQL does not enforce any such limitation, and indeed it is not directly aware of the file system boundaries on your system. It just stores files in the directories you tell it to use.

Creation of the tablespace itself must be done as a database superuser, but after that you can allow ordinary database users to use it. To do that, grant them the `CREATE` privilege on it.

Tables, indexes, and entire databases can be assigned to particular tablespaces. To do so, a user with the `CREATE` privilege on a given tablespace must pass the tablespace name as a parameter to the relevant command. For example, the following creates a table in the tablespace `space1`:

```
CREATE TABLE foo(i int) TABLESPACE space1;
```

Alternatively, use the default_tablespace parameter:

```
SET default_tablespace = space1;
CREATE TABLE foo(i int);
```

When `default_tablespace` is set to anything but an empty string, it supplies an implicit `TABLESPACE` clause for `CREATE TABLE` and `CREATE INDEX` commands that do not have an explicit one.

There is also a temp_tablespaces parameter, which determines the placement of temporary tables and indexes, as well as temporary files that are used for purposes such as sorting large data sets. This can be a list of tablespace names, rather than only one, so that the load associated with temporary objects can be spread over multiple tablespaces. A random member of the list is picked each time a temporary object is to be created.

The tablespace associated with a database is used to store the system catalogs of that database. Furthermore, it is the default tablespace used for tables, indexes, and temporary files created within the database, if no `TABLESPACE` clause is given and no other selection is specified by `default_tablespace` or `temp_tablespaces` (as appropriate). If a database is created without specifying a tablespace for it, it uses the same tablespace as the template database it is copied from.

Two tablespaces are automatically created when the database cluster is initialized. The `pg_global` tablespace is used for shared system catalogs. The `pg_default` tablespace is the default tablespace of the `template1` and `template0` databases (and, therefore, will be the default tablespace for other databases as well, unless overridden by a `TABLESPACE` clause in `CREATE DATABASE`).

Once created, a tablespace can be used from any database, provided the requesting user has sufficient privilege. This means that a tablespace cannot be dropped until all objects in all databases using the tablespace have been removed.

To remove an empty tablespace, use the DROP TABLESPACE command.

To determine the set of existing tablespaces, examine the `pg_tablespace` system catalog, for example

```
SELECT spcname FROM pg_tablespace;
```

The psql program's `\db` meta-command is also useful for listing the existing tablespaces.

PostgreSQL makes use of symbolic links to simplify the implementation of tablespaces. This means that tablespaces can be used *only* on systems that support symbolic links.

The directory `$PGDATA/pg_tblspc` contains symbolic links that point to each of the non-built-in tablespaces defined in the cluster. Although not recommended, it is possible to adjust the tablespace layout by hand by redefining these links. Under no circumstances perform this operation while the server is running. Note that in PostgreSQL 9.1 and earlier you will also need to update the `pg_tablespace` catalog with the new locations. (If you do not, `pg_dump` will continue to output the old tablespace locations.)

Chapter 22. Localization

This chapter describes the available localization features from the point of view of the administrator. PostgreSQL supports two localization facilities:

- Using the locale features of the operating system to provide locale-specific collation order, number formatting, translated messages, and other aspects. This is covered in Section 22.1 and Section 22.2.
- Providing a number of different character sets to support storing text in all kinds of languages, and providing character set translation between client and server. This is covered in Section 22.3.

22.1. Locale Support

Locale support refers to an application respecting cultural preferences regarding alphabets, sorting, number formatting, etc. PostgreSQL uses the standard ISO C and POSIX locale facilities provided by the server operating system. For additional information refer to the documentation of your system.

22.1.1. Overview

Locale support is automatically initialized when a database cluster is created using `initdb`. `initdb` will initialize the database cluster with the locale setting of its execution environment by default, so if your system is already set to use the locale that you want in your database cluster then there is nothing else you need to do. If you want to use a different locale (or you are not sure which locale your system is set to), you can instruct `initdb` exactly which locale to use by specifying the `--locale` option. For example:

```
initdb --locale=sv_SE
```

This example for Unix systems sets the locale to Swedish (`sv`) as spoken in Sweden (`SE`). Other possibilities might include `en_US` (U.S. English) and `fr_CA` (French Canadian). If more than one character set can be used for a locale then the specifications can take the form *`language_territory.codeset`*. For example, `fr_BE.UTF-8` represents the French language (fr) as spoken in Belgium (BE), with a UTF-8 character set encoding.

What locales are available on your system under what names depends on what was provided by the operating system vendor and what was installed. On most Unix systems, the command `locale -a` will provide a list of available locales. Windows uses more verbose locale names, such as `German_Germany` or `Swedish_Sweden.1252`, but the principles are the same.

Occasionally it is useful to mix rules from several locales, e.g., use English collation rules but Spanish messages. To support that, a set of locale subcategories exist that control only certain aspects of the localization rules:

`LC_COLLATE`	String sort order

LC_CTYPE	Character classification (What is a letter? Its upper-case equivalent?)
LC_MESSAGES	Language of messages
LC_MONETARY	Formatting of currency amounts
LC_NUMERIC	Formatting of numbers
LC_TIME	Formatting of dates and times

The category names translate into names of initdb options to override the locale choice for a specific category. For instance, to set the locale to French Canadian, but use U.S. rules for formatting currency, use initdb --locale=fr_CA --lc-monetary=en_US.

If you want the system to behave as if it had no locale support, use the special locale name C, or equivalently POSIX.

Some locale categories must have their values fixed when the database is created. You can use different settings for different databases, but once a database is created, you cannot change them for that database anymore. LC_COLLATE and LC_CTYPE are these categories. They affect the sort order of indexes, so they must be kept fixed, or indexes on text columns would become corrupt. (But you can alleviate this restriction using collations, as discussed in Section 22.2.) The default values for these categories are determined when initdb is run, and those values are used when new databases are created, unless specified otherwise in the CREATE DATABASE command.

The other locale categories can be changed whenever desired by setting the server configuration parameters that have the same name as the locale categories (see Section 18.11.2 for details). The values that are chosen by initdb are actually only written into the configuration file postgresql.conf to serve as defaults when the server is started. If you remove these assignments from postgresql.conf then the server will inherit the settings from its execution environment.

Note that the locale behavior of the server is determined by the environment variables seen by the server, not by the environment of any client. Therefore, be careful to configure the correct locale settings before starting the server. A consequence of this is that if client and server are set up in different locales, messages might appear in different languages depending on where they originated.

> **Note:** When we speak of inheriting the locale from the execution environment, this means the following on most operating systems: For a given locale category, say the collation, the following environment variables are consulted in this order until one is found to be set: LC_ALL, LC_COLLATE (or the variable corresponding to the respective category), LANG. If none of these environment variables are set then the locale defaults to C.
>
> Some message localization libraries also look at the environment variable LANGUAGE which overrides all other locale settings for the purpose of setting the language of messages. If in doubt, please refer to the documentation of your operating system, in particular the documentation about gettext.

To enable messages to be translated to the user's preferred language, NLS must have been selected at build time (configure --enable-nls). All other locale support is built in automatically.

22.1.2. Behavior

The locale settings influence the following SQL features:

- Sort order in queries using `ORDER BY` or the standard comparison operators on textual data
- The `upper`, `lower`, and `initcap` functions
- Pattern matching operators (`LIKE`, `SIMILAR TO`, and POSIX-style regular expressions); locales affect both case insensitive matching and the classification of characters by character-class regular expressions
- The `to_char` family of functions
- The ability to use indexes with `LIKE` clauses

The drawback of using locales other than `C` or `POSIX` in PostgreSQL is its performance impact. It slows character handling and prevents ordinary indexes from being used by `LIKE`. For this reason use locales only if you actually need them.

As a workaround to allow PostgreSQL to use indexes with `LIKE` clauses under a non-C locale, several custom operator classes exist. These allow the creation of an index that performs a strict character-by-character comparison, ignoring locale comparison rules. Refer to Section 11.9 for more information. Another approach is to create indexes using the `C` collation, as discussed in Section 22.2.

22.1.3. Problems

If locale support doesn't work according to the explanation above, check that the locale support in your operating system is correctly configured. To check what locales are installed on your system, you can use the command `locale -a` if your operating system provides it.

Check that PostgreSQL is actually using the locale that you think it is. The `LC_COLLATE` and `LC_CTYPE` settings are determined when a database is created, and cannot be changed except by creating a new database. Other locale settings including `LC_MESSAGES` and `LC_MONETARY` are initially determined by the environment the server is started in, but can be changed on-the-fly. You can check the active locale settings using the `SHOW` command.

The directory `src/test/locale` in the source distribution contains a test suite for PostgreSQL's locale support.

Client applications that handle server-side errors by parsing the text of the error message will obviously have problems when the server's messages are in a different language. Authors of such applications are advised to make use of the error code scheme instead.

Maintaining catalogs of message translations requires the on-going efforts of many volunteers that want to see PostgreSQL speak their preferred language well. If messages in your language are currently not available or not fully translated, your assistance would be appreciated. If you want to help, refer to Chapter 52 or write to the developers' mailing list.

22.2. Collation Support

The collation feature allows specifying the sort order and character classification behavior of data per-column, or even per-operation. This alleviates the restriction that the `LC_COLLATE` and `LC_CTYPE` settings of a database cannot be changed after its creation.

22.2.1. Concepts

Conceptually, every expression of a collatable data type has a collation. (The built-in collatable data types are `text`, `varchar`, and `char`. User-defined base types can also be marked collatable, and of course a domain over a collatable data type is collatable.) If the expression is a column reference, the collation of the expression is the defined collation of the column. If the expression is a constant, the collation is the default collation of the data type of the constant. The collation of a more complex expression is derived from the collations of its inputs, as described below.

The collation of an expression can be the "default" collation, which means the locale settings defined for the database. It is also possible for an expression's collation to be indeterminate. In such cases, ordering operations and other operations that need to know the collation will fail.

When the database system has to perform an ordering or a character classification, it uses the collation of the input expression. This happens, for example, with `ORDER BY` clauses and function or operator calls such as <. The collation to apply for an `ORDER BY` clause is simply the collation of the sort key. The collation to apply for a function or operator call is derived from the arguments, as described below. In addition to comparison operators, collations are taken into account by functions that convert between lower and upper case letters, such as `lower`, `upper`, and `initcap`; by pattern matching operators; and by `to_char` and related functions.

For a function or operator call, the collation that is derived by examining the argument collations is used at run time for performing the specified operation. If the result of the function or operator call is of a collatable data type, the collation is also used at parse time as the defined collation of the function or operator expression, in case there is a surrounding expression that requires knowledge of its collation.

The *collation derivation* of an expression can be implicit or explicit. This distinction affects how collations are combined when multiple different collations appear in an expression. An explicit collation derivation occurs when a `COLLATE` clause is used; all other collation derivations are implicit. When multiple collations need to be combined, for example in a function call, the following rules are used:

1. If any input expression has an explicit collation derivation, then all explicitly derived collations among the input expressions must be the same, otherwise an error is raised. If any explicitly derived collation is present, that is the result of the collation combination.
2. Otherwise, all input expressions must have the same implicit collation derivation or the default collation. If any non-default collation is present, that is the result of the collation combination. Otherwise, the result is the default collation.
3. If there are conflicting non-default implicit collations among the input expressions, then the combination is deemed to have indeterminate collation. This is not an error condition unless the particular function being invoked requires knowledge of the collation it should apply. If it does, an error will be raised at run-time.

For example, consider this table definition:

```
CREATE TABLE test1 (
    a text COLLATE "de_DE",
    b text COLLATE "es_ES",
    ...
);
```

Then in

```
SELECT a < 'foo' FROM test1;
```

the < comparison is performed according to `de_DE` rules, because the expression combines an implicitly derived collation with the default collation. But in

```
SELECT a < ('foo' COLLATE "fr_FR") FROM test1;
```

the comparison is performed using `fr_FR` rules, because the explicit collation derivation overrides the implicit one. Furthermore, given

```
SELECT a < b FROM test1;
```

the parser cannot determine which collation to apply, since the `a` and `b` columns have conflicting implicit collations. Since the < operator does need to know which collation to use, this will result in an error. The error can be resolved by attaching an explicit collation specifier to either input expression, thus:

```
SELECT a < b COLLATE "de_DE" FROM test1;
```

or equivalently

```
SELECT a COLLATE "de_DE" < b FROM test1;
```

On the other hand, the structurally similar case

```
SELECT a || b FROM test1;
```

does not result in an error, because the `||` operator does not care about collations: its result is the same regardless of the collation.

The collation assigned to a function or operator's combined input expressions is also considered to apply to the function or operator's result, if the function or operator delivers a result of a collatable data type. So, in

```
SELECT * FROM test1 ORDER BY a || 'foo';
```

the ordering will be done according to `de_DE` rules. But this query:

```
SELECT * FROM test1 ORDER BY a || b;
```

results in an error, because even though the `||` operator doesn't need to know a collation, the `ORDER BY` clause does. As before, the conflict can be resolved with an explicit collation specifier:

```
SELECT * FROM test1 ORDER BY a || b COLLATE "fr_FR";
```

22.2.2. Managing Collations

A collation is an SQL schema object that maps an SQL name to operating system locales. In particular, it maps to a combination of LC_COLLATE and LC_CTYPE. (As the name would suggest, the main purpose of a collation is to set LC_COLLATE, which controls the sort order. But it is rarely necessary in practice to have an LC_CTYPE setting that is different from LC_COLLATE, so it is more convenient to collect these under one concept than to create another infrastructure for setting LC_CTYPE per expression.) Also, a collation is tied to a character set encoding (see Section 22.3). The same collation name may exist for different encodings.

On all platforms, the collations named default, C, and POSIX are available. Additional collations may be available depending on operating system support. The default collation selects the LC_COLLATE and LC_CTYPE values specified at database creation time. The C and POSIX collations both specify "traditional C" behavior, in which only the ASCII letters "A" through "Z" are treated as letters, and sorting is done strictly by character code byte values.

If the operating system provides support for using multiple locales within a single program (newlocale and related functions), then when a database cluster is initialized, initdb populates the system catalog pg_collation with collations based on all the locales it finds on the operating system at the time. For example, the operating system might provide a locale named de_DE.utf8. initdb would then create a collation named de_DE.utf8 for encoding UTF8 that has both LC_COLLATE and LC_CTYPE set to de_DE.utf8. It will also create a collation with the .utf8 tag stripped off the name. So you could also use the collation under the name de_DE, which is less cumbersome to write and makes the name less encoding-dependent. Note that, nevertheless, the initial set of collation names is platform-dependent.

In case a collation is needed that has different values for LC_COLLATE and LC_CTYPE, a new collation may be created using the CREATE COLLATION command. That command can also be used to create a new collation from an existing collation, which can be useful to be able to use operating-system-independent collation names in applications.

Within any particular database, only collations that use that database's encoding are of interest. Other entries in pg_collation are ignored. Thus, a stripped collation name such as de_DE can be considered unique within a given database even though it would not be unique globally. Use of the stripped collation names is recommended, since it will make one less thing you need to change if you decide to change to another database encoding. Note however that the default, C, and POSIX collations can be used regardless of the database encoding.

PostgreSQL considers distinct collation objects to be incompatible even when they have identical properties. Thus for example,

```
SELECT a COLLATE "C" < b COLLATE "POSIX" FROM test1;
```

will draw an error even though the C and POSIX collations have identical behaviors. Mixing stripped and non-stripped collation names is therefore not recommended.

22.3. Character Set Support

The character set support in PostgreSQL allows you to store text in a variety of character sets (also called encodings), including single-byte character sets such as the ISO 8859 series and multiple-byte character

sets such as EUC (Extended Unix Code), UTF-8, and Mule internal code. All supported character sets can be used transparently by clients, but a few are not supported for use within the server (that is, as a server-side encoding). The default character set is selected while initializing your PostgreSQL database cluster using `initdb`. It can be overridden when you create a database, so you can have multiple databases each with a different character set.

An important restriction, however, is that each database's character set must be compatible with the database's `LC_CTYPE` (character classification) and `LC_COLLATE` (string sort order) locale settings. For `C` or `POSIX` locale, any character set is allowed, but for other locales there is only one character set that will work correctly. (On Windows, however, UTF-8 encoding can be used with any locale.)

22.3.1. Supported Character Sets

Table 22-1 shows the character sets available for use in PostgreSQL.

Table 22-1. PostgreSQL Character Sets

Name	Description	Language	Server?	Bytes/Char	Aliases
`BIG5`	Big Five	Traditional Chinese	No	1-2	`WIN950`, `Windows950`
`EUC_CN`	Extended UNIX Code-CN	Simplified Chinese	Yes	1-3	
`EUC_JP`	Extended UNIX Code-JP	Japanese	Yes	1-3	
`EUC_JIS_2004`	Extended UNIX Code-JP, JIS X 0213	Japanese	Yes	1-3	
`EUC_KR`	Extended UNIX Code-KR	Korean	Yes	1-3	
`EUC_TW`	Extended UNIX Code-TW	Traditional Chinese, Taiwanese	Yes	1-3	
`GB18030`	National Standard	Chinese	No	1-4	
`GBK`	Extended National Standard	Simplified Chinese	No	1-2	`WIN936`, `Windows936`
`ISO_8859_5`	ISO 8859-5, ECMA 113	Latin/Cyrillic	Yes	1	
`ISO_8859_6`	ISO 8859-6, ECMA 114	Latin/Arabic	Yes	1	
`ISO_8859_7`	ISO 8859-7, ECMA 118	Latin/Greek	Yes	1	

Name	Description	Language	Server?	Bytes/Char	Aliases
ISO_8859_8	ISO 8859-8, ECMA 121	Latin/Hebrew	Yes	1	
JOHAB	JOHAB	Korean (Hangul)	No	1-3	
KOI8R	KOI8-R	Cyrillic (Russian)	Yes	1	KOI8
KOI8U	KOI8-U	Cyrillic (Ukrainian)	Yes	1	
LATIN1	ISO 8859-1, ECMA 94	Western European	Yes	1	ISO88591
LATIN2	ISO 8859-2, ECMA 94	Central European	Yes	1	ISO88592
LATIN3	ISO 8859-3, ECMA 94	South European	Yes	1	ISO88593
LATIN4	ISO 8859-4, ECMA 94	North European	Yes	1	ISO88594
LATIN5	ISO 8859-9, ECMA 128	Turkish	Yes	1	ISO88599
LATIN6	ISO 8859-10, ECMA 144	Nordic	Yes	1	ISO885910
LATIN7	ISO 8859-13	Baltic	Yes	1	ISO885913
LATIN8	ISO 8859-14	Celtic	Yes	1	ISO885914
LATIN9	ISO 8859-15	LATIN1 with Euro and accents	Yes	1	ISO885915
LATIN10	ISO 8859-16, ASRO SR 14111	Romanian	Yes	1	ISO885916
MULE_INTERNAL	Mule internal code	Multilingual Emacs	Yes	1-4	
SJIS	Shift JIS	Japanese	No	1-2	Mskanji, ShiftJIS, WIN932, Windows932
SHIFT_JIS_2004	Shift JIS, JIS X 0213	Japanese	No	1-2	
SQL_ASCII	unspecified (see text)	*any*	Yes	1	
UHC	Unified Hangul Code	Korean	No	1-2	WIN949, Windows949
UTF8	Unicode, 8-bit	*all*	Yes	1-4	Unicode

Name	Description	Language	Server?	Bytes/Char	Aliases
WIN866	Windows CP866	Cyrillic	Yes	1	ALT
WIN874	Windows CP874	Thai	Yes	1	
WIN1250	Windows CP1250	Central European	Yes	1	
WIN1251	Windows CP1251	Cyrillic	Yes	1	WIN
WIN1252	Windows CP1252	Western European	Yes	1	
WIN1253	Windows CP1253	Greek	Yes	1	
WIN1254	Windows CP1254	Turkish	Yes	1	
WIN1255	Windows CP1255	Hebrew	Yes	1	
WIN1256	Windows CP1256	Arabic	Yes	1	
WIN1257	Windows CP1257	Baltic	Yes	1	
WIN1258	Windows CP1258	Vietnamese	Yes	1	ABC, TCVN, TCVN5712, VSCII

Not all client APIs support all the listed character sets. For example, the PostgreSQL JDBC driver does not support MULE_INTERNAL, LATIN6, LATIN8, and LATIN10.

The SQL_ASCII setting behaves considerably differently from the other settings. When the server character set is SQL_ASCII, the server interprets byte values 0-127 according to the ASCII standard, while byte values 128-255 are taken as uninterpreted characters. No encoding conversion will be done when the setting is SQL_ASCII. Thus, this setting is not so much a declaration that a specific encoding is in use, as a declaration of ignorance about the encoding. In most cases, if you are working with any non-ASCII data, it is unwise to use the SQL_ASCII setting because PostgreSQL will be unable to help you by converting or validating non-ASCII characters.

22.3.2. Setting the Character Set

initdb defines the default character set (encoding) for a PostgreSQL cluster. For example,

```
initdb -E EUC_JP
```

sets the default character set to EUC_JP (Extended Unix Code for Japanese). You can use --encoding instead of -E if you prefer longer option strings. If no -E or --encoding option is given, initdb attempts to determine the appropriate encoding to use based on the specified or default locale.

You can specify a non-default encoding at database creation time, provided that the encoding is compatible with the selected locale:

```
createdb -E EUC_KR -T template0 --lc-collate=ko_KR.euckr --lc-ctype=ko_KR.euckr korean
```

This will create a database named `korean` that uses the character set `EUC_KR`, and locale `ko_KR`. Another way to accomplish this is to use this SQL command:

```
CREATE DATABASE korean WITH ENCODING 'EUC_KR' LC_COLLATE='ko_KR.euckr' LC_CTYPE='ko_KR.euck
```

Notice that the above commands specify copying the `template0` database. When copying any other database, the encoding and locale settings cannot be changed from those of the source database, because that might result in corrupt data. For more information see Section 21.3.

The encoding for a database is stored in the system catalog `pg_database`. You can see it by using the `psql -l` option or the `\l` command.

```
$ psql -l
                                        List of databases
   Name    |  Owner   | Encoding  |  Collation  |    Ctype    |           Access Privileges
-----------+----------+-----------+-------------+-------------+-------------------------------
 clocaledb | hlinnaka | SQL_ASCII | C           | C           |
 englishdb | hlinnaka | UTF8      | en_GB.UTF8  | en_GB.UTF8  |
 japanese  | hlinnaka | UTF8      | ja_JP.UTF8  | ja_JP.UTF8  |
 korean    | hlinnaka | EUC_KR    | ko_KR.euckr | ko_KR.euckr |
 postgres  | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  |
 template0 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  | {=c/hlinnaka,hlinnaka=CTc/h
 template1 | hlinnaka | UTF8      | fi_FI.UTF8  | fi_FI.UTF8  | {=c/hlinnaka,hlinnaka=CTc/h
(7 rows)
```

> **Important:** On most modern operating systems, PostgreSQL can determine which character set is implied by the `LC_CTYPE` setting, and it will enforce that only the matching database encoding is used. On older systems it is your responsibility to ensure that you use the encoding expected by the locale you have selected. A mistake in this area is likely to lead to strange behavior of locale-dependent operations such as sorting.
>
> PostgreSQL will allow superusers to create databases with `SQL_ASCII` encoding even when `LC_CTYPE` is not `C` or `POSIX`. As noted above, `SQL_ASCII` does not enforce that the data stored in the database has any particular encoding, and so this choice poses risks of locale-dependent misbehavior. Using this combination of settings is deprecated and may someday be forbidden altogether.

22.3.3. Automatic Character Set Conversion Between Server and Client

PostgreSQL supports automatic character set conversion between server and client for certain character set combinations. The conversion information is stored in the `pg_conversion` system catalog. PostgreSQL

comes with some predefined conversions, as shown in Table 22-2. You can create a new conversion using the SQL command `CREATE CONVERSION`.

Table 22-2. Client/Server Character Set Conversions

Server Character Set	Available Client Character Sets
BIG5	*not supported as a server encoding*
EUC_CN	*EUC_CN*, MULE_INTERNAL, UTF8
EUC_JP	*EUC_JP*, MULE_INTERNAL, SJIS, UTF8
EUC_JIS_2004	*EUC_JIS_2004*, SHIFT_JIS_2004, UTF8
EUC_KR	*EUC_KR*, MULE_INTERNAL, UTF8
EUC_TW	*EUC_TW*, BIG5, MULE_INTERNAL, UTF8
GB18030	*not supported as a server encoding*
GBK	*not supported as a server encoding*
ISO_8859_5	*ISO_8859_5*, KOI8R, MULE_INTERNAL, UTF8, WIN866, WIN1251
ISO_8859_6	*ISO_8859_6*, UTF8
ISO_8859_7	*ISO_8859_7*, UTF8
ISO_8859_8	*ISO_8859_8*, UTF8
JOHAB	*not supported as a server encoding*
KOI8R	*KOI8R*, ISO_8859_5, MULE_INTERNAL, UTF8, WIN866, WIN1251
KOI8U	*KOI8U*, UTF8
LATIN1	*LATIN1*, MULE_INTERNAL, UTF8
LATIN2	*LATIN2*, MULE_INTERNAL, UTF8, WIN1250
LATIN3	*LATIN3*, MULE_INTERNAL, UTF8
LATIN4	*LATIN4*, MULE_INTERNAL, UTF8
LATIN5	*LATIN5*, UTF8
LATIN6	*LATIN6*, UTF8
LATIN7	*LATIN7*, UTF8
LATIN8	*LATIN8*, UTF8
LATIN9	*LATIN9*, UTF8
LATIN10	*LATIN10*, UTF8
MULE_INTERNAL	*MULE_INTERNAL*, BIG5, EUC_CN, EUC_JP, EUC_KR, EUC_TW, ISO_8859_5, KOI8R, LATIN1 to LATIN4, SJIS, WIN866, WIN1250, WIN1251
SJIS	*not supported as a server encoding*
SHIFT_JIS_2004	*not supported as a server encoding*
SQL_ASCII	*any (no conversion will be performed)*
UHC	*not supported as a server encoding*
UTF8	*all supported encodings*

Server Character Set	Available Client Character Sets
WIN866	*WIN866*, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN1251
WIN874	*WIN874*, UTF8
WIN1250	*WIN1250*, LATIN2, MULE_INTERNAL, UTF8
WIN1251	*WIN1251*, ISO_8859_5, KOI8R, MULE_INTERNAL, UTF8, WIN866
WIN1252	*WIN1252*, UTF8
WIN1253	*WIN1253*, UTF8
WIN1254	*WIN1254*, UTF8
WIN1255	*WIN1255*, UTF8
WIN1256	*WIN1256*, UTF8
WIN1257	*WIN1257*, UTF8
WIN1258	*WIN1258*, UTF8

To enable automatic character set conversion, you have to tell PostgreSQL the character set (encoding) you would like to use in the client. There are several ways to accomplish this:

- Using the \encoding command in psql. \encoding allows you to change client encoding on the fly. For example, to change the encoding to SJIS, type:

  ```
  \encoding SJIS
  ```

- libpq (Section 31.10) has functions to control the client encoding.
- Using SET client_encoding TO. Setting the client encoding can be done with this SQL command:

  ```
  SET CLIENT_ENCODING TO 'value';
  ```

  Also you can use the standard SQL syntax SET NAMES for this purpose:

  ```
  SET NAMES 'value';
  ```

  To query the current client encoding:

  ```
  SHOW client_encoding;
  ```

  To return to the default encoding:

  ```
  RESET client_encoding;
  ```

- Using PGCLIENTENCODING. If the environment variable PGCLIENTENCODING is defined in the client's environment, that client encoding is automatically selected when a connection to the server is made. (This can subsequently be overridden using any of the other methods mentioned above.)
- Using the configuration variable client_encoding. If the client_encoding variable is set, that client encoding is automatically selected when a connection to the server is made. (This can subsequently be overridden using any of the other methods mentioned above.)

If the conversion of a particular character is not possible — suppose you chose EUC_JP for the server and LATIN1 for the client, and some Japanese characters are returned that do not have a representation in LATIN1 — an error is reported.

If the client character set is defined as `SQL_ASCII`, encoding conversion is disabled, regardless of the server's character set. Just as for the server, use of `SQL_ASCII` is unwise unless you are working with all-ASCII data.

22.3.4. Further Reading

These are good sources to start learning about various kinds of encoding systems.

CJKV Information Processing: Chinese, Japanese, Korean & Vietnamese Computing

Contains detailed explanations of `EUC_JP`, `EUC_CN`, `EUC_KR`, `EUC_TW`.

http://www.unicode.org/

The web site of the Unicode Consortium.

RFC 3629

UTF-8 (8-bit UCS/Unicode Transformation Format) is defined here.

Chapter 23. Routine Database Maintenance Tasks

PostgreSQL, like any database software, requires that certain tasks be performed regularly to achieve optimum performance. The tasks discussed here are *required*, but they are repetitive in nature and can easily be automated using standard tools such as cron scripts or Windows' Task Scheduler. It is the database administrator's responsibility to set up appropriate scripts, and to check that they execute successfully.

One obvious maintenance task is the creation of backup copies of the data on a regular schedule. Without a recent backup, you have no chance of recovery after a catastrophe (disk failure, fire, mistakenly dropping a critical table, etc.). The backup and recovery mechanisms available in PostgreSQL are discussed at length in Chapter 24.

The other main category of maintenance task is periodic "vacuuming" of the database. This activity is discussed in Section 23.1. Closely related to this is updating the statistics that will be used by the query planner, as discussed in Section 23.1.3.

Another task that might need periodic attention is log file management. This is discussed in Section 23.3.

check_postgres[1] is available for monitoring database health and reporting unusual conditions. check_postgres integrates with Nagios and MRTG, but can be run standalone too.

PostgreSQL is low-maintenance compared to some other database management systems. Nonetheless, appropriate attention to these tasks will go far towards ensuring a pleasant and productive experience with the system.

23.1. Routine Vacuuming

PostgreSQL databases require periodic maintenance known as *vacuuming*. For many installations, it is sufficient to let vacuuming be performed by the *autovacuum daemon*, which is described in Section 23.1.6. You might need to adjust the autovacuuming parameters described there to obtain best results for your situation. Some database administrators will want to supplement or replace the daemon's activities with manually-managed `VACUUM` commands, which typically are executed according to a schedule by cron or Task Scheduler scripts. To set up manually-managed vacuuming properly, it is essential to understand the issues discussed in the next few subsections. Administrators who rely on autovacuuming may still wish to skim this material to help them understand and adjust autovacuuming.

23.1.1. Vacuuming Basics

PostgreSQL's VACUUM command has to process each table on a regular basis for several reasons:

1. To recover or reuse disk space occupied by updated or deleted rows.
2. To update data statistics used by the PostgreSQL query planner.
3. To update the visibility map, which speeds up index-only scans.

1. https://bucardo.org/check_postgres/

4. To protect against loss of very old data due to *transaction ID wraparound* or *multixact ID wraparound*.

Each of these reasons dictates performing VACUUM operations of varying frequency and scope, as explained in the following subsections.

There are two variants of VACUUM: standard VACUUM and VACUUM FULL. VACUUM FULL can reclaim more disk space but runs much more slowly. Also, the standard form of VACUUM can run in parallel with production database operations. (Commands such as SELECT, INSERT, UPDATE, and DELETE will continue to function normally, though you will not be able to modify the definition of a table with commands such as ALTER TABLE while it is being vacuumed.) VACUUM FULL requires exclusive lock on the table it is working on, and therefore cannot be done in parallel with other use of the table. Generally, therefore, administrators should strive to use standard VACUUM and avoid VACUUM FULL.

VACUUM creates a substantial amount of I/O traffic, which can cause poor performance for other active sessions. There are configuration parameters that can be adjusted to reduce the performance impact of background vacuuming — see Section 18.4.4.

23.1.2. Recovering Disk Space

In PostgreSQL, an UPDATE or DELETE of a row does not immediately remove the old version of the row. This approach is necessary to gain the benefits of multiversion concurrency control (MVCC, see Chapter 13): the row version must not be deleted while it is still potentially visible to other transactions. But eventually, an outdated or deleted row version is no longer of interest to any transaction. The space it occupies must then be reclaimed for reuse by new rows, to avoid unbounded growth of disk space requirements. This is done by running VACUUM.

The standard form of VACUUM removes dead row versions in tables and indexes and marks the space available for future reuse. However, it will not return the space to the operating system, except in the special case where one or more pages at the end of a table become entirely free and an exclusive table lock can be easily obtained. In contrast, VACUUM FULL actively compacts tables by writing a complete new version of the table file with no dead space. This minimizes the size of the table, but can take a long time. It also requires extra disk space for the new copy of the table, until the operation completes.

The usual goal of routine vacuuming is to do standard VACUUMs often enough to avoid needing VACUUM FULL. The autovacuum daemon attempts to work this way, and in fact will never issue VACUUM FULL. In this approach, the idea is not to keep tables at their minimum size, but to maintain steady-state usage of disk space: each table occupies space equivalent to its minimum size plus however much space gets used up between vacuumings. Although VACUUM FULL can be used to shrink a table back to its minimum size and return the disk space to the operating system, there is not much point in this if the table will just grow again in the future. Thus, moderately-frequent standard VACUUM runs are a better approach than infrequent VACUUM FULL runs for maintaining heavily-updated tables.

Some administrators prefer to schedule vacuuming themselves, for example doing all the work at night when load is low. The difficulty with doing vacuuming according to a fixed schedule is that if a table has an unexpected spike in update activity, it may get bloated to the point that VACUUM FULL is really necessary to reclaim space. Using the autovacuum daemon alleviates this problem, since the daemon schedules vacuuming dynamically in response to update activity. It is unwise to disable the daemon completely unless you have an extremely predictable workload. One possible compromise is to set the daemon's

parameters so that it will only react to unusually heavy update activity, thus keeping things from getting out of hand, while scheduled VACUUMs are expected to do the bulk of the work when the load is typical.

For those not using autovacuum, a typical approach is to schedule a database-wide VACUUM once a day during a low-usage period, supplemented by more frequent vacuuming of heavily-updated tables as necessary. (Some installations with extremely high update rates vacuum their busiest tables as often as once every few minutes.) If you have multiple databases in a cluster, don't forget to VACUUM each one; the program vacuumdb might be helpful.

> **Tip:** Plain VACUUM may not be satisfactory when a table contains large numbers of dead row versions as a result of massive update or delete activity. If you have such a table and you need to reclaim the excess disk space it occupies, you will need to use VACUUM FULL, or alternatively CLUSTER or one of the table-rewriting variants of ALTER TABLE. These commands rewrite an entire new copy of the table and build new indexes for it. All these options require exclusive lock. Note that they also temporarily use extra disk space approximately equal to the size of the table, since the old copies of the table and indexes can't be released until the new ones are complete.

> **Tip:** If you have a table whose entire contents are deleted on a periodic basis, consider doing it with TRUNCATE rather than using DELETE followed by VACUUM. TRUNCATE removes the entire content of the table immediately, without requiring a subsequent VACUUM or VACUUM FULL to reclaim the now-unused disk space. The disadvantage is that strict MVCC semantics are violated.

23.1.3. Updating Planner Statistics

The PostgreSQL query planner relies on statistical information about the contents of tables in order to generate good plans for queries. These statistics are gathered by the ANALYZE command, which can be invoked by itself or as an optional step in VACUUM. It is important to have reasonably accurate statistics, otherwise poor choices of plans might degrade database performance.

The autovacuum daemon, if enabled, will automatically issue ANALYZE commands whenever the content of a table has changed sufficiently. However, administrators might prefer to rely on manually-scheduled ANALYZE operations, particularly if it is known that update activity on a table will not affect the statistics of "interesting" columns. The daemon schedules ANALYZE strictly as a function of the number of rows inserted or updated; it has no knowledge of whether that will lead to meaningful statistical changes.

As with vacuuming for space recovery, frequent updates of statistics are more useful for heavily-updated tables than for seldom-updated ones. But even for a heavily-updated table, there might be no need for statistics updates if the statistical distribution of the data is not changing much. A simple rule of thumb is to think about how much the minimum and maximum values of the columns in the table change. For example, a `timestamp` column that contains the time of row update will have a constantly-increasing maximum value as rows are added and updated; such a column will probably need more frequent statistics updates than, say, a column containing URLs for pages accessed on a website. The URL column might receive changes just as often, but the statistical distribution of its values probably changes relatively slowly.

It is possible to run ANALYZE on specific tables and even just specific columns of a table, so the flexibility exists to update some statistics more frequently than others if your application requires it. In practice,

however, it is usually best to just analyze the entire database, because it is a fast operation. ANALYZE uses a statistically random sampling of the rows of a table rather than reading every single row.

> **Tip:** Although per-column tweaking of ANALYZE frequency might not be very productive, you might find it worthwhile to do per-column adjustment of the level of detail of the statistics collected by ANALYZE. Columns that are heavily used in WHERE clauses and have highly irregular data distributions might require a finer-grain data histogram than other columns. See ALTER TABLE SET STATISTICS, or change the database-wide default using the default_statistics_target configuration parameter.
>
> Also, by default there is limited information available about the selectivity of functions. However, if you create an expression index that uses a function call, useful statistics will be gathered about the function, which can greatly improve query plans that use the expression index.

> **Tip:** The autovacuum daemon does not issue ANALYZE commands for foreign tables, since it has no means of determining how often that might be useful. If your queries require statistics on foreign tables for proper planning, it's a good idea to run manually-managed ANALYZE commands on those tables on a suitable schedule.

23.1.4. Updating The Visibility Map

Vacuum maintains a visibility map for each table to keep track of which pages contain only tuples that are known to be visible to all active transactions (and all future transactions, until the page is again modified). This has two purposes. First, vacuum itself can skip such pages on the next run, since there is nothing to clean up.

Second, it allows PostgreSQL to answer some queries using only the index, without reference to the underlying table. Since PostgreSQL indexes don't contain tuple visibility information, a normal index scan fetches the heap tuple for each matching index entry, to check whether it should be seen by the current transaction. An *index-only scan*, on the other hand, checks the visibility map first. If it's known that all tuples on the page are visible, the heap fetch can be skipped. This is most noticeable on large data sets where the visibility map can prevent disk accesses. The visibility map is vastly smaller than the heap, so it can easily be cached even when the heap is very large.

23.1.5. Preventing Transaction ID Wraparound Failures

PostgreSQL's MVCC transaction semantics depend on being able to compare transaction ID (XID) numbers: a row version with an insertion XID greater than the current transaction's XID is "in the future" and should not be visible to the current transaction. But since transaction IDs have limited size (32 bits) a cluster that runs for a long time (more than 4 billion transactions) would suffer *transaction ID wraparound*: the XID counter wraps around to zero, and all of a sudden transactions that were in the past appear to be in the future — which means their output become invisible. In short, catastrophic data loss. (Actually the data is still there, but that's cold comfort if you cannot get at it.) To avoid this, it is necessary to vacuum every table in every database at least once every two billion transactions.

The reason that periodic vacuuming solves the problem is that `VACUUM` will mark rows as *frozen*, indicating that they were inserted by a transaction that committed sufficiently far in the past that the effects of the inserting transaction are certain to be visible to all current and future transactions. Normal XIDs are compared using modulo-2^{32} arithmetic. This means that for every normal XID, there are two billion XIDs that are "older" and two billion that are "newer"; another way to say it is that the normal XID space is circular with no endpoint. Therefore, once a row version has been created with a particular normal XID, the row version will appear to be "in the past" for the next two billion transactions, no matter which normal XID we are talking about. If the row version still exists after more than two billion transactions, it will suddenly appear to be in the future. To prevent this, PostgreSQL reserves a special XID, `FrozenTransactionId`, which does not follow the normal XID comparison rules and is always considered older than every normal XID. Frozen row versions are treated as if the inserting XID were `FrozenTransactionId`, so that they will appear to be "in the past" to all normal transactions regardless of wraparound issues, and so such row versions will be valid until deleted, no matter how long that is.

> **Note:** In PostgreSQL versions before 9.4, freezing was implemented by actually replacing a row's insertion XID with `FrozenTransactionId`, which was visible in the row's `xmin` system column. Newer versions just set a flag bit, preserving the row's original `xmin` for possible forensic use. However, rows with `xmin` equal to `FrozenTransactionId` (2) may still be found in databases pg_upgrade'd from pre-9.4 versions.
>
> Also, system catalogs may contain rows with `xmin` equal to `BootstrapTransactionId` (1), indicating that they were inserted during the first phase of initdb. Like `FrozenTransactionId`, this special XID is treated as older than every normal XID.

vacuum_freeze_min_age controls how old an XID value has to be before rows bearing that XID will be frozen. Increasing this setting may avoid unnecessary work if the rows that would otherwise be frozen will soon be modified again, but decreasing this setting increases the number of transactions that can elapse before the table must be vacuumed again.

`VACUUM` normally skips pages that don't have any dead row versions, but those pages might still have row versions with old XID values. To ensure all old row versions have been frozen, a scan of the whole table is needed. vacuum_freeze_table_age controls when `VACUUM` does that: a whole table sweep is forced if the table hasn't been fully scanned for `vacuum_freeze_table_age` minus `vacuum_freeze_min_age` transactions. Setting it to 0 forces `VACUUM` to always scan all pages, effectively ignoring the visibility map.

The maximum time that a table can go unvacuumed is two billion transactions minus the `vacuum_freeze_min_age` value at the time `VACUUM` last scanned the whole table. If it were to go unvacuumed for longer than that, data loss could result. To ensure that this does not happen, autovacuum is invoked on any table that might contain unfrozen rows with XIDs older than the age specified by the configuration parameter autovacuum_freeze_max_age. (This will happen even if autovacuum is disabled.)

This implies that if a table is not otherwise vacuumed, autovacuum will be invoked on it approximately once every `autovacuum_freeze_max_age` minus `vacuum_freeze_min_age` transactions. For tables that are regularly vacuumed for space reclamation purposes, this is of little importance. However, for static tables (including tables that receive inserts, but no updates or deletes), there is no need to vacuum for space reclamation, so it can be useful to try to maximize the interval between forced autovacuums on very large static tables. Obviously one can do this either by increasing `autovacuum_freeze_max_age` or decreasing `vacuum_freeze_min_age`.

The effective maximum for vacuum_freeze_table_age is 0.95 * autovacuum_freeze_max_age; a setting higher than that will be capped to the maximum. A value higher than autovacuum_freeze_max_age wouldn't make sense because an anti-wraparound autovacuum would be triggered at that point anyway, and the 0.95 multiplier leaves some breathing room to run a manual VACUUM before that happens. As a rule of thumb, vacuum_freeze_table_age should be set to a value somewhat below autovacuum_freeze_max_age, leaving enough gap so that a regularly scheduled VACUUM or an autovacuum triggered by normal delete and update activity is run in that window. Setting it too close could lead to anti-wraparound autovacuums, even though the table was recently vacuumed to reclaim space, whereas lower values lead to more frequent whole-table scans.

The sole disadvantage of increasing autovacuum_freeze_max_age (and vacuum_freeze_table_age along with it) is that the pg_clog subdirectory of the database cluster will take more space, because it must store the commit status of all transactions back to the autovacuum_freeze_max_age horizon. The commit status uses two bits per transaction, so if autovacuum_freeze_max_age is set to its maximum allowed value of two billion, pg_clog can be expected to grow to about half a gigabyte. If this is trivial compared to your total database size, setting autovacuum_freeze_max_age to its maximum allowed value is recommended. Otherwise, set it depending on what you are willing to allow for pg_clog storage. (The default, 200 million transactions, translates to about 50MB of pg_clog storage.)

One disadvantage of decreasing vacuum_freeze_min_age is that it might cause VACUUM to do useless work: freezing a row version is a waste of time if the row is modified soon thereafter (causing it to acquire a new XID). So the setting should be large enough that rows are not frozen until they are unlikely to change any more.

To track the age of the oldest unfrozen XIDs in a database, VACUUM stores XID statistics in the system tables pg_class and pg_database. In particular, the relfrozenxid column of a table's pg_class row contains the freeze cutoff XID that was used by the last whole-table VACUUM for that table. All rows inserted by transactions with XIDs older than this cutoff XID are guaranteed to have been frozen. Similarly, the datfrozenxid column of a database's pg_database row is a lower bound on the unfrozen XIDs appearing in that database — it is just the minimum of the per-table relfrozenxid values within the database. A convenient way to examine this information is to execute queries such as:

```
SELECT c.oid::regclass as table_name,
       greatest(age(c.relfrozenxid),age(t.relfrozenxid)) as age
FROM pg_class c
LEFT JOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.relkind IN ('r', 'm');

SELECT datname, age(datfrozenxid) FROM pg_database;
```

The age column measures the number of transactions from the cutoff XID to the current transaction's XID.

VACUUM normally only scans pages that have been modified since the last vacuum, but relfrozenxid can only be advanced when the whole table is scanned. The whole table is scanned when relfrozenxid is more than vacuum_freeze_table_age transactions old, when VACUUM's FREEZE option is used, or when all pages happen to require vacuuming to remove dead row versions. When VACUUM scans the whole table, after it's finished age(relfrozenxid) should be a little more than the vacuum_freeze_min_age setting that was used (more by the number of transactions

started since the VACUUM started). If no whole-table-scanning VACUUM is issued on the table until autovacuum_freeze_max_age is reached, an autovacuum will soon be forced for the table.

If for some reason autovacuum fails to clear old XIDs from a table, the system will begin to emit warning messages like this when the database's oldest XIDs reach eleven million transactions from the wraparound point:

```
WARNING:  database "mydb" must be vacuumed within 10985967 transactions
HINT:  To avoid a database shutdown, execute a database-wide VACUUM in that database.
```

(A manual VACUUM should fix the problem, as suggested by the hint; but note that the VACUUM must be performed by a superuser, else it will fail to process system catalogs and thus not be able to advance the database's datfrozenxid.) If these warnings are ignored, the system will shut down and refuse to start any new transactions once there are fewer than 1 million transactions left until wraparound:

```
ERROR:  database is not accepting commands to avoid wraparound data loss in database "mydb"
HINT:  Stop the postmaster and vacuum that database in single-user mode.
```

The 1-million-transaction safety margin exists to let the administrator recover without data loss, by manually executing the required VACUUM commands. However, since the system will not execute commands once it has gone into the safety shutdown mode, the only way to do this is to stop the server and start the server in single-user mode to execute VACUUM. The shutdown mode is not enforced in single-user mode. See the postgres reference page for details about using single-user mode.

23.1.5.1. Multixacts and Wraparound

Multixact IDs are used to support row locking by multiple transactions. Since there is only limited space in a tuple header to store lock information, that information is encoded as a "multiple transaction ID", or multixact ID for short, whenever there is more than one transaction concurrently locking a row. Information about which transaction IDs are included in any particular multixact ID is stored separately in the pg_multixact subdirectory, and only the multixact ID appears in the xmax field in the tuple header. Like transaction IDs, multixact IDs are implemented as a 32-bit counter and corresponding storage, all of which requires careful aging management, storage cleanup, and wraparound handling. There is a separate storage area which holds the list of members in each multixact, which also uses a 32-bit counter and which must also be managed.

During a VACUUM table scan, either partial or of the whole table, any multixact ID older than vacuum_multixact_freeze_min_age is replaced by a different value, which can be the zero value, a single transaction ID, or a newer multixact ID. For each table, pg_class.relminmxid stores the oldest possible multixact ID still appearing in any tuple of that table. If this value is older than vacuum_multixact_freeze_table_age, a whole-table scan is forced. mxid_age() can be used on pg_class.relminmxid to find its age.

Whole-table VACUUM scans, regardless of what causes them, enable advancing the value for that table. Eventually, as all tables in all databases are scanned and their oldest multixact values are advanced, on-disk storage for older multixacts can be removed.

As a safety device, a whole-table vacuum scan will occur for any table whose multixact-age is greater than autovacuum_multixact_freeze_max_age. Whole-table vacuum scans will also occur progressively for all tables, starting with those that have the oldest multixact-age, if the amount of used member storage space

exceeds the amount 50% of the addressable storage space. Both of these kinds of whole-table scans will occur even if autovacuum is nominally disabled.

23.1.6. The Autovacuum Daemon

PostgreSQL has an optional but highly recommended feature called *autovacuum*, whose purpose is to automate the execution of VACUUM and ANALYZE commands. When enabled, autovacuum checks for tables that have had a large number of inserted, updated or deleted tuples. These checks use the statistics collection facility; therefore, autovacuum cannot be used unless track_counts is set to true. In the default configuration, autovacuuming is enabled and the related configuration parameters are appropriately set.

The "autovacuum daemon" actually consists of multiple processes. There is a persistent daemon process, called the *autovacuum launcher*, which is in charge of starting *autovacuum worker* processes for all databases. The launcher will distribute the work across time, attempting to start one worker within each database every autovacuum_naptime seconds. (Therefore, if the installation has *N* databases, a new worker will be launched every autovacuum_naptime/*N* seconds.) A maximum of autovacuum_max_workers worker processes are allowed to run at the same time. If there are more than autovacuum_max_workers databases to be processed, the next database will be processed as soon as the first worker finishes. Each worker process will check each table within its database and execute VACUUM and/or ANALYZE as needed. log_autovacuum_min_duration can be set to monitor autovacuum workers' activity.

If several large tables all become eligible for vacuuming in a short amount of time, all autovacuum workers might become occupied with vacuuming those tables for a long period. This would result in other tables and databases not being vacuumed until a worker becomes available. There is no limit on how many workers might be in a single database, but workers do try to avoid repeating work that has already been done by other workers. Note that the number of running workers does not count towards max_connections or superuser_reserved_connections limits.

Tables whose relfrozenxid value is more than autovacuum_freeze_max_age transactions old are always vacuumed (this also applies to those tables whose freeze max age has been modified via storage parameters; see below). Otherwise, if the number of tuples obsoleted since the last VACUUM exceeds the "vacuum threshold", the table is vacuumed. The vacuum threshold is defined as:

```
vacuum threshold = vacuum base threshold + vacuum scale factor * number of tuples
```

where the vacuum base threshold is autovacuum_vacuum_threshold, the vacuum scale factor is autovacuum_vacuum_scale_factor, and the number of tuples is pg_class.reltuples. The number of obsolete tuples is obtained from the statistics collector; it is a semi-accurate count updated by each UPDATE and DELETE operation. (It is only semi-accurate because some information might be lost under heavy load.) If the relfrozenxid value of the table is more than vacuum_freeze_table_age transactions old, the whole table is scanned to freeze old tuples and advance relfrozenxid, otherwise only pages that have been modified since the last vacuum are scanned.

For analyze, a similar condition is used: the threshold, defined as:

```
analyze threshold = analyze base threshold + analyze scale factor * number of tuples
```

is compared to the total number of tuples inserted, updated, or deleted since the last ANALYZE.

Temporary tables cannot be accessed by autovacuum. Therefore, appropriate vacuum and analyze operations should be performed via session SQL commands.

The default thresholds and scale factors are taken from `postgresql.conf`, but it is possible to override them (and many other autovacuum control parameters) on a per-table basis; see *Storage Parameters* for more information. If a setting has been changed via a table's storage parameters, that value is used when processing that table; otherwise the global settings are used. See Section 18.10 for more details on the global settings.

When multiple workers are running, the autovacuum cost delay parameters (see Section 18.4.4) are "balanced" among all the running workers, so that the total I/O impact on the system is the same regardless of the number of workers actually running. However, any workers processing tables whose per-table `autovacuum_vacuum_cost_delay` or `autovacuum_vacuum_cost_limit` storage parameters have been set are not considered in the balancing algorithm.

Autovacuum workers generally don't block other commands. If a process attempts to acquire a lock that conflicts with the `SHARE UPDATE EXCLUSIVE` lock held by autovacuum, lock acquisition will interrupt the autovacuum. For conflicting lock modes, see Table 13-2. However, if the autovacuum is running to prevent transaction ID wraparound (i.e., the autovacuum query name in the `pg_stat_activity` view ends with `(to prevent wraparound)`), the autovacuum is not automatically interrupted.

> **Warning**
>
> Regularly running commands that acquire locks conflicting with a `SHARE UPDATE EXCLUSIVE` lock (e.g., ANALYZE) can effectively prevent autovacuums from ever completing.

23.2. Routine Reindexing

In some situations it is worthwhile to rebuild indexes periodically with the REINDEX command or a series of individual rebuilding steps.

B-tree index pages that have become completely empty are reclaimed for re-use. However, there is still a possibility of inefficient use of space: if all but a few index keys on a page have been deleted, the page remains allocated. Therefore, a usage pattern in which most, but not all, keys in each range are eventually deleted will see poor use of space. For such usage patterns, periodic reindexing is recommended.

The potential for bloat in non-B-tree indexes has not been well researched. It is a good idea to periodically monitor the index's physical size when using any non-B-tree index type.

Also, for B-tree indexes, a freshly-constructed index is slightly faster to access than one that has been updated many times because logically adjacent pages are usually also physically adjacent in a newly built index. (This consideration does not apply to non-B-tree indexes.) It might be worthwhile to reindex periodically just to improve access speed.

REINDEX can be used safely and easily in all cases. But since the command requires an exclusive table lock, it is often preferable to execute an index rebuild with a sequence of creation and replacement steps. Index types that support CREATE INDEX with the `CONCURRENTLY` option can instead be recreated that way. If that is successful and the resulting index is valid, the original index can then be replaced by the

newly built one using a combination of ALTER INDEX and DROP INDEX. When an index is used to enforce uniqueness or other constraints, ALTER TABLE might be necessary to swap the existing constraint with one enforced by the new index. Review this alternate multistep rebuild approach carefully before using it as there are limitations on which indexes can be reindexed this way, and errors must be handled.

23.3. Log File Maintenance

It is a good idea to save the database server's log output somewhere, rather than just discarding it via `/dev/null`. The log output is invaluable when diagnosing problems. However, the log output tends to be voluminous (especially at higher debug levels) so you won't want to save it indefinitely. You need to *rotate* the log files so that new log files are started and old ones removed after a reasonable period of time.

If you simply direct the stderr of `postgres` into a file, you will have log output, but the only way to truncate the log file is to stop and restart the server. This might be acceptable if you are using PostgreSQL in a development environment, but few production servers would find this behavior acceptable.

A better approach is to send the server's stderr output to some type of log rotation program. There is a built-in log rotation facility, which you can use by setting the configuration parameter `logging_collector` to `true` in `postgresql.conf`. The control parameters for this program are described in Section 18.8.1. You can also use this approach to capture the log data in machine readable CSV (comma-separated values) format.

Alternatively, you might prefer to use an external log rotation program if you have one that you are already using with other server software. For example, the rotatelogs tool included in the Apache distribution can be used with PostgreSQL. To do this, just pipe the server's stderr output to the desired program. If you start the server with `pg_ctl`, then stderr is already redirected to stdout, so you just need a pipe command, for example:

```
pg_ctl start | rotatelogs /var/log/pgsql_log 86400
```

Another production-grade approach to managing log output is to send it to syslog and let syslog deal with file rotation. To do this, set the configuration parameter `log_destination` to `syslog` (to log to syslog only) in `postgresql.conf`. Then you can send a `SIGHUP` signal to the syslog daemon whenever you want to force it to start writing a new log file. If you want to automate log rotation, the logrotate program can be configured to work with log files from syslog.

On many systems, however, syslog is not very reliable, particularly with large log messages; it might truncate or drop messages just when you need them the most. Also, on Linux, syslog will flush each message to disk, yielding poor performance. (You can use a "-" at the start of the file name in the syslog configuration file to disable syncing.)

Note that all the solutions described above take care of starting new log files at configurable intervals, but they do not handle deletion of old, no-longer-useful log files. You will probably want to set up a batch job to periodically delete old log files. Another possibility is to configure the rotation program so that old log files are overwritten cyclically.

pgBadger[2] is an external project that does sophisticated log file analysis. check_postgres[3] provides Nagios

2. https://pgbadger.darold.net/
3. https://bucardo.org/check_postgres/

alerts when important messages appear in the log files, as well as detection of many other extraordinary conditions.

Chapter 24. Backup and Restore

As with everything that contains valuable data, PostgreSQL databases should be backed up regularly. While the procedure is essentially simple, it is important to have a clear understanding of the underlying techniques and assumptions.

There are three fundamentally different approaches to backing up PostgreSQL data:

- SQL dump
- File system level backup
- Continuous archiving

Each has its own strengths and weaknesses; each is discussed in turn in the following sections.

24.1. SQL Dump

The idea behind this dump method is to generate a file with SQL commands that, when fed back to the server, will recreate the database in the same state as it was at the time of the dump. PostgreSQL provides the utility program pg_dump for this purpose. The basic usage of this command is:

```
pg_dump dbname > dumpfile
```

As you see, pg_dump writes its result to the standard output. We will see below how this can be useful. While the above command creates a text file, pg_dump can create files in other formats that allow for parallelism and more fine-grained control of object restoration.

pg_dump is a regular PostgreSQL client application (albeit a particularly clever one). This means that you can perform this backup procedure from any remote host that has access to the database. But remember that pg_dump does not operate with special permissions. In particular, it must have read access to all tables that you want to back up, so in order to back up the entire database you almost always have to run it as a database superuser. (If you do not have sufficient privileges to back up the entire database, you can still back up portions of the database to which you do have access using options such as `-n schema` or `-t table`.)

To specify which database server pg_dump should contact, use the command line options `-h host` and `-p port`. The default host is the local host or whatever your `PGHOST` environment variable specifies. Similarly, the default port is indicated by the `PGPORT` environment variable or, failing that, by the compiled-in default. (Conveniently, the server will normally have the same compiled-in default.)

Like any other PostgreSQL client application, pg_dump will by default connect with the database user name that is equal to the current operating system user name. To override this, either specify the `-U` option or set the environment variable `PGUSER`. Remember that pg_dump connections are subject to the normal client authentication mechanisms (which are described in Chapter 19).

An important advantage of pg_dump over the other backup methods described later is that pg_dump's output can generally be re-loaded into newer versions of PostgreSQL, whereas file-level backups and continuous archiving are both extremely server-version-specific. pg_dump is also the only method that will work when transferring a database to a different machine architecture, such as going from a 32-bit to a 64-bit server.

Dumps created by pg_dump are internally consistent, meaning, the dump represents a snapshot of the database at the time pg_dump began running. pg_dump does not block other operations on the database while it is working. (Exceptions are those operations that need to operate with an exclusive lock, such as most forms of `ALTER TABLE`.)

24.1.1. Restoring the Dump

Text files created by pg_dump are intended to be read in by the psql program. The general command form to restore a dump is

```
psql dbname < dumpfile
```

where *dumpfile* is the file output by the pg_dump command. The database *dbname* will not be created by this command, so you must create it yourself from `template0` before executing psql (e.g., with `createdb -T template0 dbname`). psql supports options similar to pg_dump for specifying the database server to connect to and the user name to use. See the psql reference page for more information. Non-text file dumps are restored using the pg_restore utility.

Before restoring an SQL dump, all the users who own objects or were granted permissions on objects in the dumped database must already exist. If they do not, the restore will fail to recreate the objects with the original ownership and/or permissions. (Sometimes this is what you want, but usually it is not.)

By default, the psql script will continue to execute after an SQL error is encountered. You might wish to run psql with the `ON_ERROR_STOP` variable set to alter that behavior and have psql exit with an exit status of 3 if an SQL error occurs:

```
psql --set ON_ERROR_STOP=on dbname < dumpfile
```

Either way, you will only have a partially restored database. Alternatively, you can specify that the whole dump should be restored as a single transaction, so the restore is either fully completed or fully rolled back. This mode can be specified by passing the `-1` or `--single-transaction` command-line options to psql. When using this mode, be aware that even a minor error can rollback a restore that has already run for many hours. However, that might still be preferable to manually cleaning up a complex database after a partially restored dump.

The ability of pg_dump and psql to write to or read from pipes makes it possible to dump a database directly from one server to another, for example:

```
pg_dump -h host1 dbname | psql -h host2 dbname
```

> **Important:** The dumps produced by pg_dump are relative to `template0`. This means that any languages, procedures, etc. added via `template1` will also be dumped by pg_dump. As a result, when restoring, if you are using a customized `template1`, you must create the empty database from `template0`, as in the example above.

After restoring a backup, it is wise to run ANALYZE on each database so the query optimizer has useful statistics; see Section 23.1.3 and Section 23.1.6 for more information. For more advice on how to load large amounts of data into PostgreSQL efficiently, refer to Section 14.4.

24.1.2. Using pg_dumpall

pg_dump dumps only a single database at a time, and it does not dump information about roles or tablespaces (because those are cluster-wide rather than per-database). To support convenient dumping of the entire contents of a database cluster, the pg_dumpall program is provided. pg_dumpall backs up each database in a given cluster, and also preserves cluster-wide data such as role and tablespace definitions. The basic usage of this command is:

```
pg_dumpall > dumpfile
```

The resulting dump can be restored with psql:

```
psql -f dumpfile postgres
```

(Actually, you can specify any existing database name to start from, but if you are loading into an empty cluster then `postgres` should usually be used.) It is always necessary to have database superuser access when restoring a pg_dumpall dump, as that is required to restore the role and tablespace information. If you use tablespaces, make sure that the tablespace paths in the dump are appropriate for the new installation.

pg_dumpall works by emitting commands to re-create roles, tablespaces, and empty databases, then invoking pg_dump for each database. This means that while each database will be internally consistent, the snapshots of different databases are not synchronized.

Cluster-wide data can be dumped alone using the pg_dumpall `--globals-only` option. This is necessary to fully backup the cluster if running the pg_dump command on individual databases.

24.1.3. Handling Large Databases

Some operating systems have maximum file size limits that cause problems when creating large pg_dump output files. Fortunately, pg_dump can write to the standard output, so you can use standard Unix tools to work around this potential problem. There are several possible methods:

Use compressed dumps. You can use your favorite compression program, for example gzip:

```
pg_dump dbname | gzip > filename.gz
```

Reload with:

```
gunzip -c filename.gz | psql dbname
```

or:

```
cat filename.gz | gunzip | psql dbname
```

Use `split`. The `split` command allows you to split the output into smaller files that are acceptable in size to the underlying file system. For example, to make chunks of 1 megabyte:

```
pg_dump dbname | split -b 1m - filename
```

Reload with:

```
cat filename* | psql dbname
```

Use pg_dump's custom dump format. If PostgreSQL was built on a system with the zlib compression library installed, the custom dump format will compress data as it writes it to the output file. This will produce dump file sizes similar to using `gzip`, but it has the added advantage that tables can be restored selectively. The following command dumps a database using the custom dump format:

```
pg_dump -Fc dbname > filename
```

A custom-format dump is not a script for psql, but instead must be restored with pg_restore, for example:

```
pg_restore -d dbname filename
```

See the pg_dump and pg_restore reference pages for details.

For very large databases, you might need to combine `split` with one of the other two approaches.

Use pg_dump's parallel dump feature. To speed up the dump of a large database, you can use pg_dump's parallel mode. This will dump multiple tables at the same time. You can control the degree of parallelism with the `-j` parameter. Parallel dumps are only supported for the "directory" archive format.

```
pg_dump -j num -F d -f out.dir dbname
```

You can use `pg_restore -j` to restore a dump in parallel. This will work for any archive of either the "custom" or the "directory" archive mode, whether or not it has been created with `pg_dump -j`.

24.2. File System Level Backup

An alternative backup strategy is to directly copy the files that PostgreSQL uses to store the data in the database; Section 17.2 explains where these files are located. You can use whatever method you prefer for doing file system backups; for example:

```
tar -cf backup.tar /usr/local/pgsql/data
```

There are two restrictions, however, which make this method impractical, or at least inferior to the pg_dump method:

1. The database server *must* be shut down in order to get a usable backup. Half-way measures such as disallowing all connections will *not* work (in part because `tar` and similar tools do not take an atomic snapshot of the state of the file system, but also because of internal buffering within the server). Information about stopping the server can be found in Section 17.5. Needless to say, you also need to shut down the server before restoring the data.
2. If you have dug into the details of the file system layout of the database, you might be tempted to try to back up or restore only certain individual tables or databases from their respective files or directories. This will *not* work because the information contained in these files is not usable without the commit log files, `pg_clog/*`, which contain the commit status of all transactions. A table file is only usable with this information. Of course it is also impossible to restore only a table and the

associated pg_clog data because that would render all other tables in the database cluster useless. So file system backups only work for complete backup and restoration of an entire database cluster.

An alternative file-system backup approach is to make a "consistent snapshot" of the data directory, if the file system supports that functionality (and you are willing to trust that it is implemented correctly). The typical procedure is to make a "frozen snapshot" of the volume containing the database, then copy the whole data directory (not just parts, see above) from the snapshot to a backup device, then release the frozen snapshot. This will work even while the database server is running. However, a backup created in this way saves the database files in a state as if the database server was not properly shut down; therefore, when you start the database server on the backed-up data, it will think the previous server instance crashed and will replay the WAL log. This is not a problem; just be aware of it (and be sure to include the WAL files in your backup). You can perform a CHECKPOINT before taking the snapshot to reduce recovery time.

If your database is spread across multiple file systems, there might not be any way to obtain exactly-simultaneous frozen snapshots of all the volumes. For example, if your data files and WAL log are on different disks, or if tablespaces are on different file systems, it might not be possible to use snapshot backup because the snapshots *must* be simultaneous. Read your file system documentation very carefully before trusting the consistent-snapshot technique in such situations.

If simultaneous snapshots are not possible, one option is to shut down the database server long enough to establish all the frozen snapshots. Another option is to perform a continuous archiving base backup (Section 24.3.2) because such backups are immune to file system changes during the backup. This requires enabling continuous archiving just during the backup process; restore is done using continuous archive recovery (Section 24.3.4).

Another option is to use rsync to perform a file system backup. This is done by first running rsync while the database server is running, then shutting down the database server long enough to do an rsync --checksum. (--checksum is necessary because rsync only has file modification-time granularity of one second.) The second rsync will be quicker than the first, because it has relatively little data to transfer, and the end result will be consistent because the server was down. This method allows a file system backup to be performed with minimal downtime.

Note that a file system backup will typically be larger than an SQL dump. (pg_dump does not need to dump the contents of indexes for example, just the commands to recreate them.) However, taking a file system backup might be faster.

24.3. Continuous Archiving and Point-in-Time Recovery (PITR)

At all times, PostgreSQL maintains a *write ahead log* (WAL) in the pg_xlog/ subdirectory of the cluster's data directory. The log records every change made to the database's data files. This log exists primarily for crash-safety purposes: if the system crashes, the database can be restored to consistency by "replaying" the log entries made since the last checkpoint. However, the existence of the log makes it possible to use a third strategy for backing up databases: we can combine a file-system-level backup with backup of the WAL files. If recovery is needed, we restore the file system backup and then replay from the backed-up WAL files to bring the system to a current state. This approach is more complex to administer than either

of the previous approaches, but it has some significant benefits:

- We do not need a perfectly consistent file system backup as the starting point. Any internal inconsistency in the backup will be corrected by log replay (this is not significantly different from what happens during crash recovery). So we do not need a file system snapshot capability, just tar or a similar archiving tool.
- Since we can combine an indefinitely long sequence of WAL files for replay, continuous backup can be achieved simply by continuing to archive the WAL files. This is particularly valuable for large databases, where it might not be convenient to take a full backup frequently.
- It is not necessary to replay the WAL entries all the way to the end. We could stop the replay at any point and have a consistent snapshot of the database as it was at that time. Thus, this technique supports *point-in-time recovery*: it is possible to restore the database to its state at any time since your base backup was taken.
- If we continuously feed the series of WAL files to another machine that has been loaded with the same base backup file, we have a *warm standby* system: at any point we can bring up the second machine and it will have a nearly-current copy of the database.

> **Note:** pg_dump and pg_dumpall do not produce file-system-level backups and cannot be used as part of a continuous-archiving solution. Such dumps are *logical* and do not contain enough information to be used by WAL replay.

As with the plain file-system-backup technique, this method can only support restoration of an entire database cluster, not a subset. Also, it requires a lot of archival storage: the base backup might be bulky, and a busy system will generate many megabytes of WAL traffic that have to be archived. Still, it is the preferred backup technique in many situations where high reliability is needed.

To recover successfully using continuous archiving (also called "online backup" by many database vendors), you need a continuous sequence of archived WAL files that extends back at least as far as the start time of your backup. So to get started, you should set up and test your procedure for archiving WAL files *before* you take your first base backup. Accordingly, we first discuss the mechanics of archiving WAL files.

24.3.1. Setting Up WAL Archiving

In an abstract sense, a running PostgreSQL system produces an indefinitely long sequence of WAL records. The system physically divides this sequence into WAL *segment files*, which are normally 16MB apiece (although the segment size can be altered when building PostgreSQL). The segment files are given numeric names that reflect their position in the abstract WAL sequence. When not using WAL archiving, the system normally creates just a few segment files and then "recycles" them by renaming no-longer-needed segment files to higher segment numbers. It's assumed that segment files whose contents precede the checkpoint-before-last are no longer of interest and can be recycled.

When archiving WAL data, we need to capture the contents of each segment file once it is filled, and save that data somewhere before the segment file is recycled for reuse. Depending on the application and the

available hardware, there could be many different ways of "saving the data somewhere": we could copy the segment files to an NFS-mounted directory on another machine, write them onto a tape drive (ensuring that you have a way of identifying the original name of each file), or batch them together and burn them onto CDs, or something else entirely. To provide the database administrator with flexibility, PostgreSQL tries not to make any assumptions about how the archiving will be done. Instead, PostgreSQL lets the administrator specify a shell command to be executed to copy a completed segment file to wherever it needs to go. The command could be as simple as a `cp`, or it could invoke a complex shell script — it's all up to you.

To enable WAL archiving, set the wal_level configuration parameter to `archive` or higher, archive_mode to `on`, and specify the shell command to use in the archive_command configuration parameter. In practice these settings will always be placed in the `postgresql.conf` file. In `archive_command`, `%p` is replaced by the path name of the file to archive, while `%f` is replaced by only the file name. (The path name is relative to the current working directory, i.e., the cluster's data directory.) Use `%%` if you need to embed an actual `%` character in the command. The simplest useful command is something like:

```
archive_command = 'test ! -f /mnt/server/archivedir/%f && cp %p /mnt/server/archivedir/%f'
archive_command = 'copy "%p" "C:\\server\\archivedir\\%f"'  # Windows
```

which will copy archivable WAL segments to the directory `/mnt/server/archivedir`. (This is an example, not a recommendation, and might not work on all platforms.) After the `%p` and `%f` parameters have been replaced, the actual command executed might look like this:

```
test ! -f /mnt/server/archivedir/000000010000000A900000065 && cp pg_xlog/000000010000000A9000
```

A similar command will be generated for each new file to be archived.

The archive command will be executed under the ownership of the same user that the PostgreSQL server is running as. Since the series of WAL files being archived contains effectively everything in your database, you will want to be sure that the archived data is protected from prying eyes; for example, archive into a directory that does not have group or world read access.

It is important that the archive command return zero exit status if and only if it succeeds. Upon getting a zero result, PostgreSQL will assume that the file has been successfully archived, and will remove or recycle it. However, a nonzero status tells PostgreSQL that the file was not archived; it will try again periodically until it succeeds.

The archive command should generally be designed to refuse to overwrite any pre-existing archive file. This is an important safety feature to preserve the integrity of your archive in case of administrator error (such as sending the output of two different servers to the same archive directory).

It is advisable to test your proposed archive command to ensure that it indeed does not overwrite an existing file, *and that it returns nonzero status in this case*. The example command above for Unix ensures this by including a separate `test` step. On some Unix platforms, `cp` has switches such as `-i` that can be used to do the same thing less verbosely, but you should not rely on these without verifying that the right exit status is returned. (In particular, GNU `cp` will return status zero when `-i` is used and the target file already exists, which is *not* the desired behavior.)

While designing your archiving setup, consider what will happen if the archive command fails repeatedly because some aspect requires operator intervention or the archive runs out of space. For example, this could occur if you write to tape without an autochanger; when the tape fills, nothing further can be archived until the tape is swapped. You should ensure that any error condition or request to a human operator is reported appropriately so that the situation can be resolved reasonably quickly. The `pg_xlog/` directory

will continue to fill with WAL segment files until the situation is resolved. (If the file system containing pg_xlog/ fills up, PostgreSQL will do a PANIC shutdown. No committed transactions will be lost, but the database will remain offline until you free some space.)

The speed of the archiving command is unimportant as long as it can keep up with the average rate at which your server generates WAL data. Normal operation continues even if the archiving process falls a little behind. If archiving falls significantly behind, this will increase the amount of data that would be lost in the event of a disaster. It will also mean that the pg_xlog/ directory will contain large numbers of not-yet-archived segment files, which could eventually exceed available disk space. You are advised to monitor the archiving process to ensure that it is working as you intend.

In writing your archive command, you should assume that the file names to be archived can be up to 64 characters long and can contain any combination of ASCII letters, digits, and dots. It is not necessary to preserve the original relative path (%p) but it is necessary to preserve the file name (%f).

Note that although WAL archiving will allow you to restore any modifications made to the data in your PostgreSQL database, it will not restore changes made to configuration files (that is, postgresql.conf, pg_hba.conf and pg_ident.conf), since those are edited manually rather than through SQL operations. You might wish to keep the configuration files in a location that will be backed up by your regular file system backup procedures. See Section 18.2 for how to relocate the configuration files.

The archive command is only invoked on completed WAL segments. Hence, if your server generates only little WAL traffic (or has slack periods where it does so), there could be a long delay between the completion of a transaction and its safe recording in archive storage. To put a limit on how old unarchived data can be, you can set archive_timeout to force the server to switch to a new WAL segment file at least that often. Note that archived files that are archived early due to a forced switch are still the same length as completely full files. It is therefore unwise to set a very short archive_timeout — it will bloat your archive storage. archive_timeout settings of a minute or so are usually reasonable.

Also, you can force a segment switch manually with pg_switch_xlog if you want to ensure that a just-finished transaction is archived as soon as possible. Other utility functions related to WAL management are listed in Table 9-68.

When wal_level is minimal some SQL commands are optimized to avoid WAL logging, as described in Section 14.4.7. If archiving or streaming replication were turned on during execution of one of these statements, WAL would not contain enough information for archive recovery. (Crash recovery is unaffected.) For this reason, wal_level can only be changed at server start. However, archive_command can be changed with a configuration file reload. If you wish to temporarily stop archiving, one way to do it is to set archive_command to the empty string (''). This will cause WAL files to accumulate in pg_xlog/ until a working archive_command is re-established.

24.3.2. Making a Base Backup

The easiest way to perform a base backup is to use the pg_basebackup tool. It can create a base backup either as regular files or as a tar archive. If more flexibility than pg_basebackup can provide is required, you can also make a base backup using the low level API (see Section 24.3.3).

It is not necessary to be concerned about the amount of time it takes to make a base backup. However, if you normally run the server with full_page_writes disabled, you might notice a drop in performance while the backup runs since full_page_writes is effectively forced on during backup mode.

To make use of the backup, you will need to keep all the WAL segment files generated during and after the file system backup. To aid you in doing this, the base backup process creates a *backup history file* that is immediately stored into the WAL archive area. This file is named after the first WAL segment file that you need for the file system backup. For example, if the starting WAL file is `000000010000123400000055CD` the backup history file will be named something like `000000010000123400000055CD.007C9330.backup`. (The second part of the file name stands for an exact position within the WAL file, and can ordinarily be ignored.) Once you have safely archived the file system backup and the WAL segment files used during the backup (as specified in the backup history file), all archived WAL segments with names numerically less are no longer needed to recover the file system backup and can be deleted. However, you should consider keeping several backup sets to be absolutely certain that you can recover your data.

The backup history file is just a small text file. It contains the label string you gave to pg_basebackup, as well as the starting and ending times and WAL segments of the backup. If you used the label to identify the associated dump file, then the archived history file is enough to tell you which dump file to restore.

Since you have to keep around all the archived WAL files back to your last base backup, the interval between base backups should usually be chosen based on how much storage you want to expend on archived WAL files. You should also consider how long you are prepared to spend recovering, if recovery should be necessary — the system will have to replay all those WAL segments, and that could take awhile if it has been a long time since the last base backup.

24.3.3. Making a Base Backup Using the Low Level API

The procedure for making a base backup using the low level APIs contains a few more steps than the pg_basebackup method, but is relatively simple. It is very important that these steps are executed in sequence, and that the success of a step is verified before proceeding to the next step.

1. Ensure that WAL archiving is enabled and working.
2. Connect to the database as a superuser and issue the command:

   ```
   SELECT pg_start_backup('label');
   ```

   where `label` is any string you want to use to uniquely identify this backup operation. (One good practice is to use the full path where you intend to put the backup dump file.) `pg_start_backup` creates a *backup label* file, called `backup_label`, in the cluster directory with information about your backup, including the start time and label string. The function also creates a *tablespace map* file, called `tablespace_map`, in the cluster directory with information about tablespace symbolic links in `pg_tblspc/` if one or more such link is present. Both files are critical to the integrity of the backup, should you need to restore from it.

   It does not matter which database within the cluster you connect to to issue this command. You can ignore the result returned by the function; but if it reports an error, deal with that before proceeding.

   By default, `pg_start_backup` can take a long time to finish. This is because it performs a checkpoint, and the I/O required for the checkpoint will be spread out over a significant period of time, by default half your inter-checkpoint interval (see the configuration parameter checkpoint_completion_target). This is usually what you want, because it minimizes the impact on query processing. If you want to start the backup as soon as possible, use:

   ```
   SELECT pg_start_backup('label', true);
   ```

This forces the checkpoint to be done as quickly as possible.

3. Perform the backup, using any convenient file-system-backup tool such as tar or cpio (not pg_dump or pg_dumpall). It is neither necessary nor desirable to stop normal operation of the database while you do this.

4. Again connect to the database as a superuser, and issue the command:

   ```
   SELECT pg_stop_backup();
   ```

   This terminates the backup mode and performs an automatic switch to the next WAL segment. The reason for the switch is to arrange for the last WAL segment file written during the backup interval to be ready to archive.

5. Once the WAL segment files active during the backup are archived, you are done. The file identified by `pg_stop_backup`'s result is the last segment that is required to form a complete set of backup files. If `archive_mode` is enabled, `pg_stop_backup` does not return until the last segment has been archived. Archiving of these files happens automatically since you have already configured `archive_command`. In most cases this happens quickly, but you are advised to monitor your archive system to ensure there are no delays. If the archive process has fallen behind because of failures of the archive command, it will keep retrying until the archive succeeds and the backup is complete. If you wish to place a time limit on the execution of `pg_stop_backup`, set an appropriate `statement_timeout` value.

Some file system backup tools emit warnings or errors if the files they are trying to copy change while the copy proceeds. When taking a base backup of an active database, this situation is normal and not an error. However, you need to ensure that you can distinguish complaints of this sort from real errors. For example, some versions of rsync return a separate exit code for "vanished source files", and you can write a driver script to accept this exit code as a non-error case. Also, some versions of GNU tar return an error code indistinguishable from a fatal error if a file was truncated while tar was copying it. Fortunately, GNU tar versions 1.16 and later exit with 1 if a file was changed during the backup, and 2 for other errors. With GNU tar version 1.23 and later, you can use the warning options `--warning=no-file-changed --warning=no-file-removed` to hide the related warning messages.

Be certain that your backup dump includes all of the files under the database cluster directory (e.g., `/usr/local/pgsql/data`). If you are using tablespaces that do not reside underneath this directory, be careful to include them as well (and be sure that your backup dump archives symbolic links as links, otherwise the restore will corrupt your tablespaces).

You can, however, omit from the backup dump the files within the cluster's `pg_xlog/` subdirectory. This slight adjustment is worthwhile because it reduces the risk of mistakes when restoring. This is easy to arrange if `pg_xlog/` is a symbolic link pointing to someplace outside the cluster directory, which is a common setup anyway for performance reasons. You might also want to exclude `postmaster.pid` and `postmaster.opts`, which record information about the running postmaster, not about the postmaster which will eventually use this backup. (These files can confuse pg_ctl.)

It is often a good idea to also omit from the backup dump the files within the cluster's `pg_replslot/` directory, so that replication slots that exist on the master do not become part of the backup. Otherwise, the subsequent use of the backup to create a standby may result in indefinite retention of WAL files on the standby, and possibly bloat on the master if hot standby feedback is enabled, because the clients that are using those replication slots will still be connecting to and updating the slots on the master, not the standby. Even if the backup is only intended for use in creating a new master, copying the replication slots

isn't expected to be particularly useful, since the contents of those slots will likely be badly out of date by the time the new master comes on line.

It's also worth noting that the `pg_start_backup` function makes files named `backup_label` and `tablespace_map` in the database cluster directory, which are removed by `pg_stop_backup`. These files will of course be archived as a part of your backup dump file. The backup label file includes the label string you gave to `pg_start_backup`, as well as the time at which `pg_start_backup` was run, and the name of the starting WAL file. In case of confusion it is therefore possible to look inside a backup dump file and determine exactly which backup session the dump file came from. The tablespace map file includes the symbolic link names as they exist in the directory `pg_tblspc/` and the full path of each symbolic link. These files are not merely for your information; their presence and contents are critical to the proper operation of the system's recovery process.

It is also possible to make a backup dump while the server is stopped. In this case, you obviously cannot use `pg_start_backup` or `pg_stop_backup`, and you will therefore be left to your own devices to keep track of which backup dump is which and how far back the associated WAL files go. It is generally better to follow the continuous archiving procedure above.

24.3.4. Recovering Using a Continuous Archive Backup

Okay, the worst has happened and you need to recover from your backup. Here is the procedure:

1. Stop the server, if it's running.
2. If you have the space to do so, copy the whole cluster data directory and any tablespaces to a temporary location in case you need them later. Note that this precaution will require that you have enough free space on your system to hold two copies of your existing database. If you do not have enough space, you should at least save the contents of the cluster's `pg_xlog` subdirectory, as it might contain logs which were not archived before the system went down.
3. Remove all existing files and subdirectories under the cluster data directory and under the root directories of any tablespaces you are using.
4. Restore the database files from your file system backup. Be sure that they are restored with the right ownership (the database system user, not `root`!) and with the right permissions. If you are using tablespaces, you should verify that the symbolic links in `pg_tblspc/` were correctly restored.
5. Remove any files present in `pg_xlog/`; these came from the file system backup and are therefore probably obsolete rather than current. If you didn't archive `pg_xlog/` at all, then recreate it with proper permissions, being careful to ensure that you re-establish it as a symbolic link if you had it set up that way before.
6. If you have unarchived WAL segment files that you saved in step 2, copy them into `pg_xlog/`. (It is best to copy them, not move them, so you still have the unmodified files if a problem occurs and you have to start over.)
7. Create a recovery command file `recovery.conf` in the cluster data directory (see Chapter 26). You might also want to temporarily modify `pg_hba.conf` to prevent ordinary users from connecting until you are sure the recovery was successful.
8. Start the server. The server will go into recovery mode and proceed to read through the archived WAL files it needs. Should the recovery be terminated because of an external error, the server can simply

be restarted and it will continue recovery. Upon completion of the recovery process, the server will rename `recovery.conf` to `recovery.done` (to prevent accidentally re-entering recovery mode later) and then commence normal database operations.

9. Inspect the contents of the database to ensure you have recovered to the desired state. If not, return to step 1. If all is well, allow your users to connect by restoring `pg_hba.conf` to normal.

The key part of all this is to set up a recovery configuration file that describes how you want to recover and how far the recovery should run. You can use `recovery.conf.sample` (normally located in the installation's `share/` directory) as a prototype. The one thing that you absolutely must specify in `recovery.conf` is the `restore_command`, which tells PostgreSQL how to retrieve archived WAL file segments. Like the `archive_command`, this is a shell command string. It can contain `%f`, which is replaced by the name of the desired log file, and `%p`, which is replaced by the path name to copy the log file to. (The path name is relative to the current working directory, i.e., the cluster's data directory.) Write `%%` if you need to embed an actual `%` character in the command. The simplest useful command is something like:

```
restore_command = 'cp /mnt/server/archivedir/%f %p'
```

which will copy previously archived WAL segments from the directory `/mnt/server/archivedir`. Of course, you can use something much more complicated, perhaps even a shell script that requests the operator to mount an appropriate tape.

It is important that the command return nonzero exit status on failure. The command *will* be called requesting files that are not present in the archive; it must return nonzero when so asked. This is not an error condition. An exception is that if the command was terminated by a signal (other than SIGTERM, which is used as part of a database server shutdown) or an error by the shell (such as command not found), then recovery will abort and the server will not start up.

Not all of the requested files will be WAL segment files; you should also expect requests for files with a suffix of `.history`. Also be aware that the base name of the `%p` path will be different from `%f`; do not expect them to be interchangeable.

WAL segments that cannot be found in the archive will be sought in `pg_xlog/`; this allows use of recent un-archived segments. However, segments that are available from the archive will be used in preference to files in `pg_xlog/`.

Normally, recovery will proceed through all available WAL segments, thereby restoring the database to the current point in time (or as close as possible given the available WAL segments). Therefore, a normal recovery will end with a "file not found" message, the exact text of the error message depending upon your choice of `restore_command`. You may also see an error message at the start of recovery for a file named something like `00000001.history`. This is also normal and does not indicate a problem in simple recovery situations; see Section 24.3.5 for discussion.

If you want to recover to some previous point in time (say, right before the junior DBA dropped your main transaction table), just specify the required stopping point in `recovery.conf`. You can specify the stop point, known as the "recovery target", either by date/time, named restore point or by completion of a specific transaction ID. As of this writing only the date/time and named restore point options are very usable, since there are no tools to help you identify with any accuracy which transaction ID to use.

Note: The stop point must be after the ending time of the base backup, i.e., the end time of `pg_stop_backup`. You cannot use a base backup to recover to a time when that backup was in progress. (To recover to such a time, you must go back to your previous base backup and roll forward from there.)

If recovery finds corrupted WAL data, recovery will halt at that point and the server will not start. In such a case the recovery process could be re-run from the beginning, specifying a "recovery target" before the point of corruption so that recovery can complete normally. If recovery fails for an external reason, such as a system crash or if the WAL archive has become inaccessible, then the recovery can simply be restarted and it will restart almost from where it failed. Recovery restart works much like checkpointing in normal operation: the server periodically forces all its state to disk, and then updates the `pg_control` file to indicate that the already-processed WAL data need not be scanned again.

24.3.5. Timelines

The ability to restore the database to a previous point in time creates some complexities that are akin to science-fiction stories about time travel and parallel universes. For example, in the original history of the database, suppose you dropped a critical table at 5:15PM on Tuesday evening, but didn't realize your mistake until Wednesday noon. Unfazed, you get out your backup, restore to the point-in-time 5:14PM Tuesday evening, and are up and running. In *this* history of the database universe, you never dropped the table. But suppose you later realize this wasn't such a great idea, and would like to return to sometime Wednesday morning in the original history. You won't be able to if, while your database was up-and-running, it overwrote some of the WAL segment files that led up to the time you now wish you could get back to. Thus, to avoid this, you need to distinguish the series of WAL records generated after you've done a point-in-time recovery from those that were generated in the original database history.

To deal with this problem, PostgreSQL has a notion of *timelines*. Whenever an archive recovery completes, a new timeline is created to identify the series of WAL records generated after that recovery. The timeline ID number is part of WAL segment file names so a new timeline does not overwrite the WAL data generated by previous timelines. It is in fact possible to archive many different timelines. While that might seem like a useless feature, it's often a lifesaver. Consider the situation where you aren't quite sure what point-in-time to recover to, and so have to do several point-in-time recoveries by trial and error until you find the best place to branch off from the old history. Without timelines this process would soon generate an unmanageable mess. With timelines, you can recover to *any* prior state, including states in timeline branches that you abandoned earlier.

Every time a new timeline is created, PostgreSQL creates a "timeline history" file that shows which timeline it branched off from and when. These history files are necessary to allow the system to pick the right WAL segment files when recovering from an archive that contains multiple timelines. Therefore, they are archived into the WAL archive area just like WAL segment files. The history files are just small text files, so it's cheap and appropriate to keep them around indefinitely (unlike the segment files which are large). You can, if you like, add comments to a history file to record your own notes about how and why this particular timeline was created. Such comments will be especially valuable when you have a thicket of different timelines as a result of experimentation.

The default behavior of recovery is to recover along the same timeline that was current when the base backup was taken. If you wish to recover into some child timeline (that is, you want to return to some

state that was itself generated after a recovery attempt), you need to specify the target timeline ID in `recovery.conf`. You cannot recover into timelines that branched off earlier than the base backup.

24.3.6. Tips and Examples

Some tips for configuring continuous archiving are given here.

24.3.6.1. Standalone Hot Backups

It is possible to use PostgreSQL's backup facilities to produce standalone hot backups. These are backups that cannot be used for point-in-time recovery, yet are typically much faster to backup and restore than pg_dump dumps. (They are also much larger than pg_dump dumps, so in some cases the speed advantage might be negated.)

As with base backups, the easiest way to produce a standalone hot backup is to use the pg_basebackup tool. If you include the `-X` parameter when calling it, all the transaction log required to use the backup will be included in the backup automatically, and no special action is required to restore the backup.

If more flexibility in copying the backup files is needed, a lower level process can be used for standalone hot backups as well. To prepare for low level standalone hot backups, set `wal_level` to `archive` or higher, `archive_mode` to `on`, and set up an `archive_command` that performs archiving only when a *switch file* exists. For example:

```
archive_command = 'test ! -f /var/lib/pgsql/backup_in_progress || (test ! -f /var/lib/pgsql
```

This command will perform archiving when `/var/lib/pgsql/backup_in_progress` exists, and otherwise silently return zero exit status (allowing PostgreSQL to recycle the unwanted WAL file).

With this preparation, a backup can be taken using a script like the following:

```
touch /var/lib/pgsql/backup_in_progress
psql -c "select pg_start_backup('hot_backup');"
tar -cf /var/lib/pgsql/backup.tar /var/lib/pgsql/data/
psql -c "select pg_stop_backup();"
rm /var/lib/pgsql/backup_in_progress
tar -rf /var/lib/pgsql/backup.tar /var/lib/pgsql/archive/
```

The switch file `/var/lib/pgsql/backup_in_progress` is created first, enabling archiving of completed WAL files to occur. After the backup the switch file is removed. Archived WAL files are then added to the backup so that both base backup and all required WAL files are part of the same tar file. Please remember to add error handling to your backup scripts.

24.3.6.2. Compressed Archive Logs

If archive storage size is a concern, you can use gzip to compress the archive files:

```
archive_command = 'gzip < %p > /var/lib/pgsql/archive/%f'
```

You will then need to use gunzip during recovery:

```
restore_command = 'gunzip < /mnt/server/archivedir/%f > %p'
```

24.3.6.3. `archive_command` Scripts

Many people choose to use scripts to define their `archive_command`, so that their `postgresql.conf` entry looks very simple:

```
archive_command = 'local_backup_script.sh "%p" "%f"'
```

Using a separate script file is advisable any time you want to use more than a single command in the archiving process. This allows all complexity to be managed within the script, which can be written in a popular scripting language such as bash or perl.

Examples of requirements that might be solved within a script include:

- Copying data to secure off-site data storage
- Batching WAL files so that they are transferred every three hours, rather than one at a time
- Interfacing with other backup and recovery software
- Interfacing with monitoring software to report errors

> **Tip:** When using an `archive_command` script, it's desirable to enable logging_collector. Any messages written to stderr from the script will then appear in the database server log, allowing complex configurations to be diagnosed easily if they fail.

24.3.7. Caveats

At this writing, there are several limitations of the continuous archiving technique. These will probably be fixed in future releases:

- Operations on hash indexes are not presently WAL-logged, so replay will not update these indexes. This will mean that any new inserts will be ignored by the index, updated rows will apparently disappear and deleted rows will still retain pointers. In other words, if you modify a table with a hash index on it then you will get incorrect query results on a standby server. When recovery completes it is recommended that you manually REINDEX each such index after completing a recovery operation.
- If a CREATE DATABASE command is executed while a base backup is being taken, and then the template database that the `CREATE DATABASE` copied is modified while the base backup is still in progress, it is possible that recovery will cause those modifications to be propagated into the created database as well. This is of course undesirable. To avoid this risk, it is best not to modify any template databases while taking a base backup.

- CREATE TABLESPACE commands are WAL-logged with the literal absolute path, and will therefore be replayed as tablespace creations with the same absolute path. This might be undesirable if the log is being replayed on a different machine. It can be dangerous even if the log is being replayed on the same machine, but into a new data directory: the replay will still overwrite the contents of the original tablespace. To avoid potential gotchas of this sort, the best practice is to take a new base backup after creating or dropping tablespaces.

It should also be noted that the default WAL format is fairly bulky since it includes many disk page snapshots. These page snapshots are designed to support crash recovery, since we might need to fix partially-written disk pages. Depending on your system hardware and software, the risk of partial writes might be small enough to ignore, in which case you can significantly reduce the total volume of archived logs by turning off page snapshots using the full_page_writes parameter. (Read the notes and warnings in Chapter 29 before you do so.) Turning off page snapshots does not prevent use of the logs for PITR operations. An area for future development is to compress archived WAL data by removing unnecessary page copies even when `full_page_writes` is on. In the meantime, administrators might wish to reduce the number of page snapshots included in WAL by increasing the checkpoint interval parameters as much as feasible.

Chapter 25. High Availability, Load Balancing, and Replication

Database servers can work together to allow a second server to take over quickly if the primary server fails (high availability), or to allow several computers to serve the same data (load balancing). Ideally, database servers could work together seamlessly. Web servers serving static web pages can be combined quite easily by merely load-balancing web requests to multiple machines. In fact, read-only database servers can be combined relatively easily too. Unfortunately, most database servers have a read/write mix of requests, and read/write servers are much harder to combine. This is because though read-only data needs to be placed on each server only once, a write to any server has to be propagated to all servers so that future read requests to those servers return consistent results.

This synchronization problem is the fundamental difficulty for servers working together. Because there is no single solution that eliminates the impact of the sync problem for all use cases, there are multiple solutions. Each solution addresses this problem in a different way, and minimizes its impact for a specific workload.

Some solutions deal with synchronization by allowing only one server to modify the data. Servers that can modify data are called read/write, *master* or *primary* servers. Servers that track changes in the master are called *standby* or *slave* servers. A standby server that cannot be connected to until it is promoted to a master server is called a *warm standby* server, and one that can accept connections and serves read-only queries is called a *hot standby* server.

Some solutions are synchronous, meaning that a data-modifying transaction is not considered committed until all servers have committed the transaction. This guarantees that a failover will not lose any data and that all load-balanced servers will return consistent results no matter which server is queried. In contrast, asynchronous solutions allow some delay between the time of a commit and its propagation to the other servers, opening the possibility that some transactions might be lost in the switch to a backup server, and that load balanced servers might return slightly stale results. Asynchronous communication is used when synchronous would be too slow.

Solutions can also be categorized by their granularity. Some solutions can deal only with an entire database server, while others allow control at the per-table or per-database level.

Performance must be considered in any choice. There is usually a trade-off between functionality and performance. For example, a fully synchronous solution over a slow network might cut performance by more than half, while an asynchronous one might have a minimal performance impact.

The remainder of this section outlines various failover, replication, and load balancing solutions.

25.1. Comparison of Different Solutions

Shared Disk Failover

Shared disk failover avoids synchronization overhead by having only one copy of the database. It uses a single disk array that is shared by multiple servers. If the main database server fails, the standby server is able to mount and start the database as though it were recovering from a database crash. This allows rapid failover with no data loss.

Shared hardware functionality is common in network storage devices. Using a network file system is also possible, though care must be taken that the file system has full POSIX behavior (see Section 17.2.2). One significant limitation of this method is that if the shared disk array fails or becomes corrupt, the primary and standby servers are both nonfunctional. Another issue is that the standby server should never access the shared storage while the primary server is running.

File System (Block-Device) Replication

A modified version of shared hardware functionality is file system replication, where all changes to a file system are mirrored to a file system residing on another computer. The only restriction is that the mirroring must be done in a way that ensures the standby server has a consistent copy of the file system — specifically, writes to the standby must be done in the same order as those on the master. DRBD is a popular file system replication solution for Linux.

Transaction Log Shipping

Warm and hot standby servers can be kept current by reading a stream of write-ahead log (WAL) records. If the main server fails, the standby contains almost all of the data of the main server, and can be quickly made the new master database server. This can be synchronous or asynchronous and can only be done for the entire database server.

A standby server can be implemented using file-based log shipping (Section 25.2) or streaming replication (see Section 25.2.5), or a combination of both. For information on hot standby, see Section 25.5.

Trigger-Based Master-Standby Replication

A master-standby replication setup sends all data modification queries to the master server. The master server asynchronously sends data changes to the standby server. The standby can answer read-only queries while the master server is running. The standby server is ideal for data warehouse queries.

Slony-I is an example of this type of replication, with per-table granularity, and support for multiple standby servers. Because it updates the standby server asynchronously (in batches), there is possible data loss during fail over.

Statement-Based Replication Middleware

With statement-based replication middleware, a program intercepts every SQL query and sends it to one or all servers. Each server operates independently. Read-write queries must be sent to all servers, so that every server receives any changes. But read-only queries can be sent to just one server, allowing the read workload to be distributed among them.

If queries are simply broadcast unmodified, functions like `random()`, `CURRENT_TIMESTAMP`, and sequences can have different values on different servers. This is because each server operates independently, and because SQL queries are broadcast (and not actual modified rows). If this is unacceptable, either the middleware or the application must query such values from a single server and then use those values in write queries. Another option is to use this replication option with a traditional master-standby setup, i.e., data modification queries are sent only to the master and are propagated to the standby servers via master-standby replication, not by the replication middleware. Care must also be taken that all transactions either commit or abort on all servers, perhaps using two-phase commit (PREPARE TRANSACTION and COMMIT PREPARED). Pgpool-II and Continuent Tungsten are examples of this type of replication.

Asynchronous Multimaster Replication

For servers that are not regularly connected or have slow communication links, like laptops or remote servers, keeping data consistent among servers is a challenge. Using asynchronous multimaster replication, each server works independently, and periodically communicates with the other servers to identify conflicting transactions. The conflicts can be resolved by users or conflict resolution rules. Bucardo is an example of this type of replication.

Synchronous Multimaster Replication

In synchronous multimaster replication, each server can accept write requests, and modified data is transmitted from the original server to every other server before each transaction commits. Heavy write activity can cause excessive locking and commit delays, leading to poor performance. Read requests can be sent to any server. Some implementations use shared disk to reduce the communication overhead. Synchronous multimaster replication is best for mostly read workloads, though its big advantage is that any server can accept write requests — there is no need to partition workloads between master and standby servers, and because the data changes are sent from one server to another, there is no problem with non-deterministic functions like `random()`.

PostgreSQL does not offer this type of replication, though PostgreSQL two-phase commit (PREPARE TRANSACTION and COMMIT PREPARED) can be used to implement this in application code or middleware.

Commercial Solutions

Because PostgreSQL is open source and easily extended, a number of companies have taken PostgreSQL and created commercial closed-source solutions with unique failover, replication, and load balancing capabilities.

Table 25-1 summarizes the capabilities of the various solutions listed above.

Table 25-1. High Availability, Load Balancing, and Replication Feature Matrix

Feature	Shared Disk Failover	File System Replication	Transaction Log Shipping	Trigger-Based Master-Standby Replication	Statement-Based Replication Middleware	Asynchronous Multimaster Replication	Synchronous Multimaster Replication
Most Common Implementation	NAS	DRBD	Streaming Repl.	Slony	pgpool-II	Bucardo	
Communication Method	shared disk	disk blocks	WAL	table rows	SQL	table rows	table rows and row locks
No special hardware required		•	•	•	•	•	•

Feature	Shared Disk Failover	File System Replication	Transaction Log Shipping	Trigger-Based Master-Standby Replication	Statement-Based Replication Middleware	Asynchronous Multimaster Replication	Synchronous Multimaster Replication
Allows multiple master servers					•	•	•
No master server overhead	•		•		•		
No waiting for multiple servers	•		with sync off	•		•	
Master failure will never lose data	•	•	with sync on		•		•
Standby accept read-only queries			with hot	•	•	•	•
Per-table granularity				•		•	•
No conflict resolution necessary	•	•	•	•	•		•

There are a few solutions that do not fit into the above categories:

Data Partitioning

Data partitioning splits tables into data sets. Each set can be modified by only one server. For example, data can be partitioned by offices, e.g., London and Paris, with a server in each office. If queries combining London and Paris data are necessary, an application can query both servers, or master/standby replication can be used to keep a read-only copy of the other office's data on each server.

Multiple-Server Parallel Query Execution

Many of the above solutions allow multiple servers to handle multiple queries, but none allow a single query to use multiple servers to complete faster. This solution allows multiple servers to work concurrently on a single query. It is usually accomplished by splitting the data among servers and having each server execute its part of the query and return results to a central server where they are combined and returned to the user. This can be implemented using the PL/Proxy tool set.

25.2. Log-Shipping Standby Servers

Continuous archiving can be used to create a *high availability* (HA) cluster configuration with one or more *standby servers* ready to take over operations if the primary server fails. This capability is widely referred to as *warm standby* or *log shipping*.

The primary and standby server work together to provide this capability, though the servers are only loosely coupled. The primary server operates in continuous archiving mode, while each standby server operates in continuous recovery mode, reading the WAL files from the primary. No changes to the database tables are required to enable this capability, so it offers low administration overhead compared to some other replication solutions. This configuration also has relatively low performance impact on the primary server.

Directly moving WAL records from one database server to another is typically described as log shipping. PostgreSQL implements file-based log shipping by transferring WAL records one file (WAL segment) at a time. WAL files (16MB) can be shipped easily and cheaply over any distance, whether it be to an adjacent system, another system at the same site, or another system on the far side of the globe. The bandwidth required for this technique varies according to the transaction rate of the primary server. Record-based log shipping is more granular and streams WAL changes incrementally over a network connection (see Section 25.2.5).

It should be noted that log shipping is asynchronous, i.e., the WAL records are shipped after transaction commit. As a result, there is a window for data loss should the primary server suffer a catastrophic failure; transactions not yet shipped will be lost. The size of the data loss window in file-based log shipping can be limited by use of the `archive_timeout` parameter, which can be set as low as a few seconds. However such a low setting will substantially increase the bandwidth required for file shipping. Streaming replication (see Section 25.2.5) allows a much smaller window of data loss.

Recovery performance is sufficiently good that the standby will typically be only moments away from full availability once it has been activated. As a result, this is called a warm standby configuration which offers high availability. Restoring a server from an archived base backup and rollforward will take considerably longer, so that technique only offers a solution for disaster recovery, not high availability. A standby server can also be used for read-only queries, in which case it is called a Hot Standby server. See Section 25.5 for more information.

25.2.1. Planning

It is usually wise to create the primary and standby servers so that they are as similar as possible, at least from the perspective of the database server. In particular, the path names associated with tablespaces will be passed across unmodified, so both primary and standby servers must have the same mount paths for tablespaces if that feature is used. Keep in mind that if CREATE TABLESPACE is executed on the primary, any new mount point needed for it must be created on the primary and all standby servers before the command is executed. Hardware need not be exactly the same, but experience shows that maintaining two identical systems is easier than maintaining two dissimilar ones over the lifetime of the application and system. In any case the hardware architecture must be the same — shipping from, say, a 32-bit to a 64-bit system will not work.

In general, log shipping between servers running different major PostgreSQL release levels is not possible. It is the policy of the PostgreSQL Global Development Group not to make changes to disk formats during minor release upgrades, so it is likely that running different minor release levels on primary and standby

servers will work successfully. However, no formal support for that is offered and you are advised to keep primary and standby servers at the same release level as much as possible. When updating to a new minor release, the safest policy is to update the standby servers first — a new minor release is more likely to be able to read WAL files from a previous minor release than vice versa.

25.2.2. Standby Server Operation

In standby mode, the server continuously applies WAL received from the master server. The standby server can read WAL from a WAL archive (see restore_command) or directly from the master over a TCP connection (streaming replication). The standby server will also attempt to restore any WAL found in the standby cluster's `pg_xlog` directory. That typically happens after a server restart, when the standby replays again WAL that was streamed from the master before the restart, but you can also manually copy files to `pg_xlog` at any time to have them replayed.

At startup, the standby begins by restoring all WAL available in the archive location, calling `restore_command`. Once it reaches the end of WAL available there and `restore_command` fails, it tries to restore any WAL available in the `pg_xlog` directory. If that fails, and streaming replication has been configured, the standby tries to connect to the primary server and start streaming WAL from the last valid record found in archive or `pg_xlog`. If that fails or streaming replication is not configured, or if the connection is later disconnected, the standby goes back to step 1 and tries to restore the file from the archive again. This loop of retries from the archive, `pg_xlog`, and via streaming replication goes on until the server is stopped or failover is triggered by a trigger file.

Standby mode is exited and the server switches to normal operation when `pg_ctl promote` is run or a trigger file is found (`trigger_file`). Before failover, any WAL immediately available in the archive or in `pg_xlog` will be restored, but no attempt is made to connect to the master.

25.2.3. Preparing the Master for Standby Servers

Set up continuous archiving on the primary to an archive directory accessible from the standby, as described in Section 24.3. The archive location should be accessible from the standby even when the master is down, i.e., it should reside on the standby server itself or another trusted server, not on the master server.

If you want to use streaming replication, set up authentication on the primary server to allow replication connections from the standby server(s); that is, create a role and provide a suitable entry or entries in `pg_hba.conf` with the database field set to `replication`. Also ensure `max_wal_senders` is set to a sufficiently large value in the configuration file of the primary server. If replication slots will be used, ensure that `max_replication_slots` is set sufficiently high as well.

Take a base backup as described in Section 24.3.2 to bootstrap the standby server.

25.2.4. Setting Up a Standby Server

To set up the standby server, restore the base backup taken from primary server (see Section 24.3.4). Create a recovery command file `recovery.conf` in the standby's cluster data directory, and turn on `standby_mode`. Set `restore_command` to a simple command to copy files from the WAL archive. If you

plan to have multiple standby servers for high availability purposes, set `recovery_target_timeline` to `latest`, to make the standby server follow the timeline change that occurs at failover to another standby.

> **Note:** Do not use pg_standby or similar tools with the built-in standby mode described here. `restore_command` should return immediately if the file does not exist; the server will retry the command again if necessary. See Section 25.4 for using tools like pg_standby.

If you want to use streaming replication, fill in `primary_conninfo` with a libpq connection string, including the host name (or IP address) and any additional details needed to connect to the primary server. If the primary needs a password for authentication, the password needs to be specified in `primary_conninfo` as well.

If you're setting up the standby server for high availability purposes, set up WAL archiving, connections and authentication like the primary server, because the standby server will work as a primary server after failover.

If you're using a WAL archive, its size can be minimized using the archive_cleanup_command parameter to remove files that are no longer required by the standby server. The pg_archivecleanup utility is designed specifically to be used with `archive_cleanup_command` in typical single-standby configurations, see pg_archivecleanup. Note however, that if you're using the archive for backup purposes, you need to retain files needed to recover from at least the latest base backup, even if they're no longer needed by the standby.

A simple example of a `recovery.conf` is:

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
restore_command = 'cp /path/to/archive/%f %p'
archive_cleanup_command = 'pg_archivecleanup /path/to/archive %r'
```

You can have any number of standby servers, but if you use streaming replication, make sure you set `max_wal_senders` high enough in the primary to allow them to be connected simultaneously.

25.2.5. Streaming Replication

Streaming replication allows a standby server to stay more up-to-date than is possible with file-based log shipping. The standby connects to the primary, which streams WAL records to the standby as they're generated, without waiting for the WAL file to be filled.

Streaming replication is asynchronous by default (see Section 25.2.8), in which case there is a small delay between committing a transaction in the primary and the changes becoming visible in the standby. This delay is however much smaller than with file-based log shipping, typically under one second assuming the standby is powerful enough to keep up with the load. With streaming replication, `archive_timeout` is not required to reduce the data loss window.

If you use streaming replication without file-based continuous archiving, the server might recycle old WAL segments before the standby has received them. If this occurs, the standby will need to be reinitialized from a new base backup. You can avoid this by setting `wal_keep_segments` to a value large enough to ensure that WAL segments are not recycled too early, or by configuring a replication slot for the

standby. If you set up a WAL archive that's accessible from the standby, these solutions are not required, since the standby can always use the archive to catch up provided it retains enough segments.

To use streaming replication, set up a file-based log-shipping standby server as described in Section 25.2. The step that turns a file-based log-shipping standby into streaming replication standby is setting `primary_conninfo` setting in the `recovery.conf` file to point to the primary server. Set listen_addresses and authentication options (see `pg_hba.conf`) on the primary so that the standby server can connect to the `replication` pseudo-database on the primary server (see Section 25.2.5.1).

On systems that support the keepalive socket option, setting tcp_keepalives_idle, tcp_keepalives_interval and tcp_keepalives_count helps the primary promptly notice a broken connection.

Set the maximum number of concurrent connections from the standby servers (see max_wal_senders for details).

When the standby is started and `primary_conninfo` is set correctly, the standby will connect to the primary after replaying all WAL files available in the archive. If the connection is established successfully, you will see a walreceiver process in the standby, and a corresponding walsender process in the primary.

25.2.5.1. Authentication

It is very important that the access privileges for replication be set up so that only trusted users can read the WAL stream, because it is easy to extract privileged information from it. Standby servers must authenticate to the primary as a superuser or an account that has the `REPLICATION` privilege. It is recommended to create a dedicated user account with `REPLICATION` and `LOGIN` privileges for replication. While `REPLICATION` privilege gives very high permissions, it does not allow the user to modify any data on the primary system, which the `SUPERUSER` privilege does.

Client authentication for replication is controlled by a `pg_hba.conf` record specifying `replication` in the *`database`* field. For example, if the standby is running on host IP `192.168.1.100` and the account name for replication is `foo`, the administrator can add the following line to the `pg_hba.conf` file on the primary:

```
# Allow the user "foo" from host 192.168.1.100 to connect to the primary
# as a replication standby if the user's password is correctly supplied.
#
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    replication     foo             192.168.1.100/32        md5
```

The host name and port number of the primary, connection user name, and password are specified in the `recovery.conf` file. The password can also be set in the `~/.pgpass` file on the standby (specify `replication` in the *`database`* field). For example, if the primary is running on host IP `192.168.1.50`, port `5432`, the account name for replication is `foo`, and the password is `foopass`, the administrator can add the following line to the `recovery.conf` file on the standby:

```
# The standby connects to the primary that is running on host 192.168.1.50
# and port 5432 as the user "foo" whose password is "foopass".
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
```

25.2.5.2. Monitoring

An important health indicator of streaming replication is the amount of WAL records generated in the primary, but not yet applied in the standby. You can calculate this lag by comparing the current WAL write location on the primary with the last WAL location received by the standby. They can be retrieved using `pg_current_xlog_location` on the primary and the `pg_last_xlog_receive_location` on the standby, respectively (see Table 9-68 and Table 9-69 for details). The last WAL receive location in the standby is also displayed in the process status of the WAL receiver process, displayed using the `ps` command (see Section 27.1 for details).

You can retrieve a list of WAL sender processes via the `pg_stat_replication` view. Large differences between `pg_current_xlog_location` and `sent_location` field might indicate that the master server is under heavy load, while differences between `sent_location` and `pg_last_xlog_receive_location` on the standby might indicate network delay, or that the standby is under heavy load.

25.2.6. Replication Slots

Replication slots provide an automated way to ensure that the master does not remove WAL segments until they have been received by all standbys, and that the master does not remove rows which could cause a recovery conflict even when the standby is disconnected.

In lieu of using replication slots, it is possible to prevent the removal of old WAL segments using wal_keep_segments, or by storing the segments in an archive using archive_command. However, these methods often result in retaining more WAL segments than required, whereas replication slots retain only the number of segments known to be needed. An advantage of these methods is that they bound the space requirement for `pg_xlog`; there is currently no way to do this using replication slots.

Similarly, hot_standby_feedback and vacuum_defer_cleanup_age provide protection against relevant rows being removed by vacuum, but the former provides no protection during any time period when the standby is not connected, and the latter often needs to be set to a high value to provide adequate protection. Replication slots overcome these disadvantages.

25.2.6.1. Querying and manipulating replication slots

Each replication slot has a name, which can contain lower-case letters, numbers, and the underscore character.

Existing replication slots and their state can be seen in the `pg_replication_slots` view.

Slots can be created and dropped either via the streaming replication protocol (see Section 50.3) or via SQL functions (see Section 9.26.6).

25.2.6.2. Configuration Example

You can create a replication slot like this:

```
postgres=# SELECT * FROM pg_create_physical_replication_slot('node_a_slot');
  slot_name  | xlog_position
```

```
-------------+---------------
 node_a_slot |

postgres=# SELECT slot_name, slot_type, active FROM pg_replication_slots;
  slot_name  | slot_type | active
-------------+-----------+--------
 node_a_slot | physical  | f
(1 row)
```

To configure the standby to use this slot, `primary_slot_name` should be configured in the standby's `recovery.conf`. Here is a simple example:

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.1.50 port=5432 user=foo password=foopass'
primary_slot_name = 'node_a_slot'
```

25.2.7. Cascading Replication

The cascading replication feature allows a standby server to accept replication connections and stream WAL records to other standbys, acting as a relay. This can be used to reduce the number of direct connections to the master and also to minimize inter-site bandwidth overheads.

A standby acting as both a receiver and a sender is known as a cascading standby. Standbys that are more directly connected to the master are known as upstream servers, while those standby servers further away are downstream servers. Cascading replication does not place limits on the number or arrangement of downstream servers, though each standby connects to only one upstream server which eventually links to a single master/primary server.

A cascading standby sends not only WAL records received from the master but also those restored from the archive. So even if the replication connection in some upstream connection is terminated, streaming replication continues downstream for as long as new WAL records are available.

Cascading replication is currently asynchronous. Synchronous replication (see Section 25.2.8) settings have no effect on cascading replication at present.

Hot Standby feedback propagates upstream, whatever the cascaded arrangement.

If an upstream standby server is promoted to become new master, downstream servers will continue to stream from the new master if `recovery_target_timeline` is set to `'latest'`.

To use cascading replication, set up the cascading standby so that it can accept replication connections (that is, set max_wal_senders and hot_standby, and configure host-based authentication). You will also need to set `primary_conninfo` in the downstream standby to point to the cascading standby.

25.2.8. Synchronous Replication

PostgreSQL streaming replication is asynchronous by default. If the primary server crashes then some transactions that were committed may not have been replicated to the standby server, causing data loss. The amount of data loss is proportional to the replication delay at the time of failover.

Synchronous replication offers the ability to confirm that all changes made by a transaction have been transferred to one synchronous standby server. This extends the standard level of durability offered by a transaction commit. This level of protection is referred to as 2-safe replication in computer science theory.

When requesting synchronous replication, each commit of a write transaction will wait until confirmation is received that the commit has been written to the transaction log on disk of both the primary and standby server. The only possibility that data can be lost is if both the primary and the standby suffer crashes at the same time. This can provide a much higher level of durability, though only if the sysadmin is cautious about the placement and management of the two servers. Waiting for confirmation increases the user's confidence that the changes will not be lost in the event of server crashes but it also necessarily increases the response time for the requesting transaction. The minimum wait time is the roundtrip time between primary to standby.

Read only transactions and transaction rollbacks need not wait for replies from standby servers. Subtransaction commits do not wait for responses from standby servers, only top-level commits. Long running actions such as data loading or index building do not wait until the very final commit message. All two-phase commit actions require commit waits, including both prepare and commit.

25.2.8.1. Basic Configuration

Once streaming replication has been configured, configuring synchronous replication requires only one additional configuration step: synchronous_standby_names must be set to a non-empty value. `synchronous_commit` must also be set to `on`, but since this is the default value, typically no change is required. (See Section 18.5.1 and Section 18.6.2.) This configuration will cause each commit to wait for confirmation that the standby has written the commit record to durable storage. `synchronous_commit` can be set by individual users, so it can be configured in the configuration file, for particular users or databases, or dynamically by applications, in order to control the durability guarantee on a per-transaction basis.

After a commit record has been written to disk on the primary, the WAL record is then sent to the standby. The standby sends reply messages each time a new batch of WAL data is written to disk, unless `wal_receiver_status_interval` is set to zero on the standby. If the standby is the first matching standby, as specified in `synchronous_standby_names` on the primary, the reply messages from that standby will be used to wake users waiting for confirmation that the commit record has been received. These parameters allow the administrator to specify which standby servers should be synchronous standbys. Note that the configuration of synchronous replication is mainly on the master. Named standbys must be directly connected to the master; the master knows nothing about downstream standby servers using cascaded replication.

Setting `synchronous_commit` to `remote_write` will cause each commit to wait for confirmation that the standby has received the commit record and written it out to its own operating system, but not for the data to be flushed to disk on the standby. This setting provides a weaker guarantee of durability than `on` does: the standby could lose the data in the event of an operating system crash, though not a PostgreSQL crash. However, it's a useful setting in practice because it can decrease the response time for

the transaction. Data loss could only occur if both the primary and the standby crash and the database of the primary gets corrupted at the same time.

Users will stop waiting if a fast shutdown is requested. However, as when using asynchronous replication, the server will not fully shutdown until all outstanding WAL records are transferred to the currently connected standby servers.

25.2.8.2. Planning for Performance

Synchronous replication usually requires carefully planned and placed standby servers to ensure applications perform acceptably. Waiting doesn't utilize system resources, but transaction locks continue to be held until the transfer is confirmed. As a result, incautious use of synchronous replication will reduce performance for database applications because of increased response times and higher contention.

PostgreSQL allows the application developer to specify the durability level required via replication. This can be specified for the system overall, though it can also be specified for specific users or connections, or even individual transactions.

For example, an application workload might consist of: 10% of changes are important customer details, while 90% of changes are less important data that the business can more easily survive if it is lost, such as chat messages between users.

With synchronous replication options specified at the application level (on the primary) we can offer synchronous replication for the most important changes, without slowing down the bulk of the total workload. Application level options are an important and practical tool for allowing the benefits of synchronous replication for high performance applications.

You should consider that the network bandwidth must be higher than the rate of generation of WAL data.

25.2.8.3. Planning for High Availability

Commits made when `synchronous_commit` is set to `on` or `remote_write` will wait until the synchronous standby responds. The response may never occur if the last, or only, standby should crash.

The best solution for avoiding data loss is to ensure you don't lose your last remaining synchronous standby. This can be achieved by naming multiple potential synchronous standbys using `synchronous_standby_names`. The first named standby will be used as the synchronous standby. Standbys listed after this will take over the role of synchronous standby if the first one should fail.

When a standby first attaches to the primary, it will not yet be properly synchronized. This is described as `catchup` mode. Once the lag between standby and primary reaches zero for the first time we move to real-time `streaming` state. The catch-up duration may be long immediately after the standby has been created. If the standby is shut down, then the catch-up period will increase according to the length of time the standby has been down. The standby is only able to become a synchronous standby once it has reached `streaming` state.

If primary restarts while commits are waiting for acknowledgement, those waiting transactions will be marked fully committed once the primary database recovers. There is no way to be certain that all standbys have received all outstanding WAL data at time of the crash of the primary. Some transactions may not show as committed on the standby, even though they show as committed on the primary. The guarantee

we offer is that the application will not receive explicit acknowledgement of the successful commit of a transaction until the WAL data is known to be safely received by the standby.

If you really do lose your last standby server then you should disable `synchronous_standby_names` and reload the configuration file on the primary server.

If the primary is isolated from remaining standby servers you should fail over to the best candidate of those other remaining standby servers.

If you need to re-create a standby server while transactions are waiting, make sure that the commands pg_start_backup() and pg_stop_backup() are run in a session with `synchronous_commit = off`, otherwise those requests will wait forever for the standby to appear.

25.2.9. Continuous archiving in standby

When continuous WAL archiving is used in a standby, there are two different scenarios: the WAL archive can be shared between the primary and the standby, or the standby can have its own WAL archive. When the standby has its own WAL archive, set `archive_mode` to `always`, and the standby will call the archive command for every WAL segment it receives, whether it's by restoring from the archive or by streaming replication. The shared archive can be handled similarly, but the `archive_command` must test if the file being archived exists already, and if the existing file has identical contents. This requires more care in the `archive_command`, as it must be careful to not overwrite an existing file with different contents, but return success if the exactly same file is archived twice. And all that must be done free of race conditions, if two servers attempt to archive the same file at the same time.

If `archive_mode` is set to `on`, the archiver is not enabled during recovery or standby mode. If the standby server is promoted, it will start archiving after the promotion, but will not archive any WAL or timeline history files that it did not generate itself. To get a complete series of WAL files in the archive, you must ensure that all WAL is archived, before it reaches the standby. This is inherently true with file-based log shipping, as the standby can only restore files that are found in the archive, but not if streaming replication is enabled. When a server is not in recovery mode, there is no difference between `on` and `always` modes.

25.3. Failover

If the primary server fails then the standby server should begin failover procedures.

If the standby server fails then no failover need take place. If the standby server can be restarted, even some time later, then the recovery process can also be restarted immediately, taking advantage of restartable recovery. If the standby server cannot be restarted, then a full new standby server instance should be created.

If the primary server fails and the standby server becomes the new primary, and then the old primary restarts, you must have a mechanism for informing the old primary that it is no longer the primary. This is sometimes known as STONITH (Shoot The Other Node In The Head), which is necessary to avoid situations where both systems think they are the primary, which will lead to confusion and ultimately data loss.

Many failover systems use just two systems, the primary and the standby, connected by some kind of heartbeat mechanism to continually verify the connectivity between the two and the viability of the primary. It is also possible to use a third system (called a witness server) to prevent some cases of inappropriate failover, but the additional complexity might not be worthwhile unless it is set up with sufficient care and rigorous testing.

PostgreSQL does not provide the system software required to identify a failure on the primary and notify the standby database server. Many such tools exist and are well integrated with the operating system facilities required for successful failover, such as IP address migration.

Once failover to the standby occurs, there is only a single server in operation. This is known as a degenerate state. The former standby is now the primary, but the former primary is down and might stay down. To return to normal operation, a standby server must be recreated, either on the former primary system when it comes up, or on a third, possibly new, system. The pg_rewind utility can be used to speed up this process on large clusters. Once complete, the primary and standby can be considered to have switched roles. Some people choose to use a third server to provide backup for the new primary until the new standby server is recreated, though clearly this complicates the system configuration and operational processes.

So, switching from primary to standby server can be fast but requires some time to re-prepare the failover cluster. Regular switching from primary to standby is useful, since it allows regular downtime on each system for maintenance. This also serves as a test of the failover mechanism to ensure that it will really work when you need it. Written administration procedures are advised.

To trigger failover of a log-shipping standby server, run `pg_ctl promote` or create a trigger file with the file name and path specified by the `trigger_file` setting in `recovery.conf`. If you're planning to use `pg_ctl promote` to fail over, `trigger_file` is not required. If you're setting up the reporting servers that are only used to offload read-only queries from the primary, not for high availability purposes, you don't need to promote it.

25.4. Alternative Method for Log Shipping

An alternative to the built-in standby mode described in the previous sections is to use a `restore_command` that polls the archive location. This was the only option available in versions 8.4 and below. In this setup, set `standby_mode` off, because you are implementing the polling required for standby operation yourself. See the pg_standby module for a reference implementation of this.

Note that in this mode, the server will apply WAL one file at a time, so if you use the standby server for queries (see Hot Standby), there is a delay between an action in the master and when the action becomes visible in the standby, corresponding to the time it takes to fill up the WAL file. `archive_timeout` can be used to make that delay shorter. Also note that you can't combine streaming replication with this method.

The operations that occur on both primary and standby servers are normal continuous archiving and recovery tasks. The only point of contact between the two database servers is the archive of WAL files that both share: primary writing to the archive, standby reading from the archive. Care must be taken to ensure that WAL archives from separate primary servers do not become mixed together or confused. The archive need not be large if it is only required for standby operation.

The magic that makes the two loosely coupled servers work together is simply a `restore_command` used on the standby that, when asked for the next WAL file, waits for it to become available from the primary. The `restore_command` is specified in the `recovery.conf` file on the standby server. Normal recovery

processing would request a file from the WAL archive, reporting failure if the file was unavailable. For standby processing it is normal for the next WAL file to be unavailable, so the standby must wait for it to appear. For files ending in `.history` there is no need to wait, and a non-zero return code must be returned. A waiting `restore_command` can be written as a custom script that loops after polling for the existence of the next WAL file. There must also be some way to trigger failover, which should interrupt the `restore_command`, break the loop and return a file-not-found error to the standby server. This ends recovery and the standby will then come up as a normal server.

Pseudocode for a suitable `restore_command` is:

```
triggered = false;
while (!NextWALFileReady() && !triggered)
{
    sleep(100000L);         /* wait for ~0.1 sec */
    if (CheckForExternalTrigger())
        triggered = true;
}
if (!triggered)
        CopyWALFileForRecovery();
```

A working example of a waiting `restore_command` is provided in the pg_standby module. It should be used as a reference on how to correctly implement the logic described above. It can also be extended as needed to support specific configurations and environments.

The method for triggering failover is an important part of planning and design. One potential option is the `restore_command` command. It is executed once for each WAL file, but the process running the `restore_command` is created and dies for each file, so there is no daemon or server process, and signals or a signal handler cannot be used. Therefore, the `restore_command` is not suitable to trigger failover. It is possible to use a simple timeout facility, especially if used in conjunction with a known `archive_timeout` setting on the primary. However, this is somewhat error prone since a network problem or busy primary server might be sufficient to initiate failover. A notification mechanism such as the explicit creation of a trigger file is ideal, if this can be arranged.

25.4.1. Implementation

The short procedure for configuring a standby server using this alternative method is as follows. For full details of each step, refer to previous sections as noted.

1. Set up primary and standby systems as nearly identical as possible, including two identical copies of PostgreSQL at the same release level.
2. Set up continuous archiving from the primary to a WAL archive directory on the standby server. Ensure that archive_mode, archive_command and archive_timeout are set appropriately on the primary (see Section 24.3.1).
3. Make a base backup of the primary server (see Section 24.3.2), and load this data onto the standby.
4. Begin recovery on the standby server from the local WAL archive, using a `recovery.conf` that specifies a `restore_command` that waits as described previously (see Section 24.3.4).

Recovery treats the WAL archive as read-only, so once a WAL file has been copied to the standby system it can be copied to tape at the same time as it is being read by the standby database server. Thus, running a standby server for high availability can be performed at the same time as files are stored for longer term disaster recovery purposes.

For testing purposes, it is possible to run both primary and standby servers on the same system. This does not provide any worthwhile improvement in server robustness, nor would it be described as HA.

25.4.2. Record-based Log Shipping

It is also possible to implement record-based log shipping using this alternative method, though this requires custom development, and changes will still only become visible to hot standby queries after a full WAL file has been shipped.

An external program can call the `pg_xlogfile_name_offset()` function (see Section 9.26) to find out the file name and the exact byte offset within it of the current end of WAL. It can then access the WAL file directly and copy the data from the last known end of WAL through the current end over to the standby servers. With this approach, the window for data loss is the polling cycle time of the copying program, which can be very small, and there is no wasted bandwidth from forcing partially-used segment files to be archived. Note that the standby servers' `restore_command` scripts can only deal with whole WAL files, so the incrementally copied data is not ordinarily made available to the standby servers. It is of use only when the primary dies — then the last partial WAL file is fed to the standby before allowing it to come up. The correct implementation of this process requires cooperation of the `restore_command` script with the data copying program.

Starting with PostgreSQL version 9.0, you can use streaming replication (see Section 25.2.5) to achieve the same benefits with less effort.

25.5. Hot Standby

Hot Standby is the term used to describe the ability to connect to the server and run read-only queries while the server is in archive recovery or standby mode. This is useful both for replication purposes and for restoring a backup to a desired state with great precision. The term Hot Standby also refers to the ability of the server to move from recovery through to normal operation while users continue running queries and/or keep their connections open.

Running queries in hot standby mode is similar to normal query operation, though there are several usage and administrative differences explained below.

25.5.1. User's Overview

When the hot_standby parameter is set to true on a standby server, it will begin accepting connections once the recovery has brought the system to a consistent state. All such connections are strictly read-only; not even temporary tables may be written.

The data on the standby takes some time to arrive from the primary server so there will be a measurable delay between primary and standby. Running the same query nearly simultaneously on both primary and standby might therefore return differing results. We say that data on the standby is *eventually consistent* with the primary. Once the commit record for a transaction is replayed on the standby, the changes made by that transaction will be visible to any new snapshots taken on the standby. Snapshots may be taken at the start of each query or at the start of each transaction, depending on the current transaction isolation level. For more details, see Section 13.2.

Transactions started during hot standby may issue the following commands:

- Query access - `SELECT`, `COPY TO`
- Cursor commands - `DECLARE`, `FETCH`, `CLOSE`
- Parameters - `SHOW`, `SET`, `RESET`
- Transaction management commands
  - `BEGIN`, `END`, `ABORT`, `START TRANSACTION`
  - `SAVEPOINT`, `RELEASE`, `ROLLBACK TO SAVEPOINT`
  - `EXCEPTION` blocks and other internal subtransactions
- `LOCK TABLE`, though only when explicitly in one of these modes: `ACCESS SHARE`, `ROW SHARE` or `ROW EXCLUSIVE`.
- Plans and resources - `PREPARE`, `EXECUTE`, `DEALLOCATE`, `DISCARD`
- Plugins and extensions - `LOAD`
- `UNLISTEN`

Transactions started during hot standby will never be assigned a transaction ID and cannot write to the system write-ahead log. Therefore, the following actions will produce error messages:

- Data Manipulation Language (DML) - `INSERT`, `UPDATE`, `DELETE`, `COPY FROM`, `TRUNCATE`. Note that there are no allowed actions that result in a trigger being executed during recovery. This restriction applies even to temporary tables, because table rows cannot be read or written without assigning a transaction ID, which is currently not possible in a Hot Standby environment.
- Data Definition Language (DDL) - `CREATE`, `DROP`, `ALTER`, `COMMENT`. This restriction applies even to temporary tables, because carrying out these operations would require updating the system catalog tables.
- `SELECT ... FOR SHARE | UPDATE`, because row locks cannot be taken without updating the underlying data files.
- Rules on `SELECT` statements that generate DML commands.
- `LOCK` that explicitly requests a mode higher than `ROW EXCLUSIVE MODE`.
- `LOCK` in short default form, since it requests `ACCESS EXCLUSIVE MODE`.
- Transaction management commands that explicitly set non-read-only state:
  - `BEGIN READ WRITE`, `START TRANSACTION READ WRITE`

  - SET TRANSACTION READ WRITE, SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE
  - SET transaction_read_only = off

- Two-phase commit commands - PREPARE TRANSACTION, COMMIT PREPARED, ROLLBACK PREPARED because even read-only transactions need to write WAL in the prepare phase (the first phase of two phase commit).
- Sequence updates - nextval(), setval()
- LISTEN, NOTIFY

In normal operation, "read-only" transactions are allowed to use LISTEN and NOTIFY, so Hot Standby sessions operate under slightly tighter restrictions than ordinary read-only sessions. It is possible that some of these restrictions might be loosened in a future release.

During hot standby, the parameter transaction_read_only is always true and may not be changed. But as long as no attempt is made to modify the database, connections during hot standby will act much like any other database connection. If failover or switchover occurs, the database will switch to normal processing mode. Sessions will remain connected while the server changes mode. Once hot standby finishes, it will be possible to initiate read-write transactions (even from a session begun during hot standby).

Users will be able to tell whether their session is read-only by issuing SHOW transaction_read_only. In addition, a set of functions (Table 9-69) allow users to access information about the standby server. These allow you to write programs that are aware of the current state of the database. These can be used to monitor the progress of recovery, or to allow you to write complex programs that restore the database to particular states.

25.5.2. Handling Query Conflicts

The primary and standby servers are in many ways loosely connected. Actions on the primary will have an effect on the standby. As a result, there is potential for negative interactions or conflicts between them. The easiest conflict to understand is performance: if a huge data load is taking place on the primary then this will generate a similar stream of WAL records on the standby, so standby queries may contend for system resources, such as I/O.

There are also additional types of conflict that can occur with Hot Standby. These conflicts are *hard conflicts* in the sense that queries might need to be canceled and, in some cases, sessions disconnected to resolve them. The user is provided with several ways to handle these conflicts. Conflict cases include:

- Access Exclusive locks taken on the primary server, including both explicit LOCK commands and various DDL actions, conflict with table accesses in standby queries.
- Dropping a tablespace on the primary conflicts with standby queries using that tablespace for temporary work files.
- Dropping a database on the primary conflicts with sessions connected to that database on the standby.
- Application of a vacuum cleanup record from WAL conflicts with standby transactions whose snapshots can still "see" any of the rows to be removed.

- Application of a vacuum cleanup record from WAL conflicts with queries accessing the target page on the standby, whether or not the data to be removed is visible.

On the primary server, these cases simply result in waiting; and the user might choose to cancel either of the conflicting actions. However, on the standby there is no choice: the WAL-logged action already occurred on the primary so the standby must not fail to apply it. Furthermore, allowing WAL application to wait indefinitely may be very undesirable, because the standby's state will become increasingly far behind the primary's. Therefore, a mechanism is provided to forcibly cancel standby queries that conflict with to-be-applied WAL records.

An example of the problem situation is an administrator on the primary server running `DROP TABLE` on a table that is currently being queried on the standby server. Clearly the standby query cannot continue if the `DROP TABLE` is applied on the standby. If this situation occurred on the primary, the `DROP TABLE` would wait until the other query had finished. But when `DROP TABLE` is run on the primary, the primary doesn't have information about what queries are running on the standby, so it will not wait for any such standby queries. The WAL change records come through to the standby while the standby query is still running, causing a conflict. The standby server must either delay application of the WAL records (and everything after them, too) or else cancel the conflicting query so that the `DROP TABLE` can be applied.

When a conflicting query is short, it's typically desirable to allow it to complete by delaying WAL application for a little bit; but a long delay in WAL application is usually not desirable. So the cancel mechanism has parameters, max_standby_archive_delay and max_standby_streaming_delay, that define the maximum allowed delay in WAL application. Conflicting queries will be canceled once it has taken longer than the relevant delay setting to apply any newly-received WAL data. There are two parameters so that different delay values can be specified for the case of reading WAL data from an archive (i.e., initial recovery from a base backup or "catching up" a standby server that has fallen far behind) versus reading WAL data via streaming replication.

In a standby server that exists primarily for high availability, it's best to set the delay parameters relatively short, so that the server cannot fall far behind the primary due to delays caused by standby queries. However, if the standby server is meant for executing long-running queries, then a high or even infinite delay value may be preferable. Keep in mind however that a long-running query could cause other sessions on the standby server to not see recent changes on the primary, if it delays application of WAL records.

Once the delay specified by `max_standby_archive_delay` or `max_standby_streaming_delay` has been exceeded, conflicting queries will be canceled. This usually results just in a cancellation error, although in the case of replaying a `DROP DATABASE` the entire conflicting session will be terminated. Also, if the conflict is over a lock held by an idle transaction, the conflicting session is terminated (this behavior might change in the future).

Canceled queries may be retried immediately (after beginning a new transaction, of course). Since query cancellation depends on the nature of the WAL records being replayed, a query that was canceled may well succeed if it is executed again.

Keep in mind that the delay parameters are compared to the elapsed time since the WAL data was received by the standby server. Thus, the grace period allowed to any one query on the standby is never more than the delay parameter, and could be considerably less if the standby has already fallen behind as a result of waiting for previous queries to complete, or as a result of being unable to keep up with a heavy update load.

The most common reason for conflict between standby queries and WAL replay is "early cleanup". Normally, PostgreSQL allows cleanup of old row versions when there are no transactions that need to see them to ensure correct visibility of data according to MVCC rules. However, this rule can only be applied for transactions executing on the master. So it is possible that cleanup on the master will remove row versions that are still visible to a transaction on the standby.

Experienced users should note that both row version cleanup and row version freezing will potentially conflict with standby queries. Running a manual `VACUUM FREEZE` is likely to cause conflicts even on tables with no updated or deleted rows.

Users should be clear that tables that are regularly and heavily updated on the primary server will quickly cause cancellation of longer running queries on the standby. In such cases the setting of a finite value for `max_standby_archive_delay` or `max_standby_streaming_delay` can be considered similar to setting `statement_timeout`.

Remedial possibilities exist if the number of standby-query cancellations is found to be unacceptable. The first option is to set the parameter `hot_standby_feedback`, which prevents `VACUUM` from removing recently-dead rows and so cleanup conflicts do not occur. If you do this, you should note that this will delay cleanup of dead rows on the primary, which may result in undesirable table bloat. However, the cleanup situation will be no worse than if the standby queries were running directly on the primary server, and you are still getting the benefit of off-loading execution onto the standby. If standby servers connect and disconnect frequently, you might want to make adjustments to handle the period when `hot_standby_feedback` feedback is not being provided. For example, consider increasing `max_standby_archive_delay` so that queries are not rapidly canceled by conflicts in WAL archive files during disconnected periods. You should also consider increasing `max_standby_streaming_delay` to avoid rapid cancellations by newly-arrived streaming WAL entries after reconnection.

Another option is to increase vacuum_defer_cleanup_age on the primary server, so that dead rows will not be cleaned up as quickly as they normally would be. This will allow more time for queries to execute before they are canceled on the standby, without having to set a high `max_standby_streaming_delay`. However it is difficult to guarantee any specific execution-time window with this approach, since `vacuum_defer_cleanup_age` is measured in transactions executed on the primary server.

The number of query cancels and the reason for them can be viewed using the `pg_stat_database_conflicts` system view on the standby server. The `pg_stat_database` system view also contains summary information.

25.5.3. Administrator's Overview

If `hot_standby` is turned `on` in `postgresql.conf` and there is a `recovery.conf` file present, the server will run in Hot Standby mode. However, it may take some time for Hot Standby connections to be allowed, because the server will not accept connections until it has completed sufficient recovery to provide a consistent state against which queries can run. During this period, clients that attempt to connect will be refused with an error message. To confirm the server has come up, either loop trying to connect from the application, or look for these messages in the server logs:

```
LOG:  entering standby mode

... then some time later ...
```

```
LOG:  consistent recovery state reached
LOG:  database system is ready to accept read only connections
```

Consistency information is recorded once per checkpoint on the primary. It is not possible to enable hot standby when reading WAL written during a period when `wal_level` was not set to `hot_standby` or `logical` on the primary. Reaching a consistent state can also be delayed in the presence of both of these conditions:

- A write transaction has more than 64 subtransactions
- Very long-lived write transactions

If you are running file-based log shipping ("warm standby"), you might need to wait until the next WAL file arrives, which could be as long as the `archive_timeout` setting on the primary.

The setting of some parameters on the standby will need reconfiguration if they have been changed on the primary. For these parameters, the value on the standby must be equal to or greater than the value on the primary. If these parameters are not set high enough then the standby will refuse to start. Higher values can then be supplied and the server restarted to begin recovery again. These parameters are:

- `max_connections`
- `max_prepared_transactions`
- `max_locks_per_transaction`
- `max_worker_processes`

It is important that the administrator select appropriate settings for max_standby_archive_delay and max_standby_streaming_delay. The best choices vary depending on business priorities. For example if the server is primarily tasked as a High Availability server, then you will want low delay settings, perhaps even zero, though that is a very aggressive setting. If the standby server is tasked as an additional server for decision support queries then it might be acceptable to set the maximum delay values to many hours, or even -1 which means wait forever for queries to complete.

Transaction status "hint bits" written on the primary are not WAL-logged, so data on the standby will likely re-write the hints again on the standby. Thus, the standby server will still perform disk writes even though all users are read-only; no changes occur to the data values themselves. Users will still write large sort temporary files and re-generate relcache info files, so no part of the database is truly read-only during hot standby mode. Note also that writes to remote databases using dblink module, and other operations outside the database using PL functions will still be possible, even though the transaction is read-only locally.

The following types of administration commands are not accepted during recovery mode:

- Data Definition Language (DDL) - e.g., `CREATE INDEX`
- Privilege and Ownership - `GRANT`, `REVOKE`, `REASSIGN`
- Maintenance commands - `ANALYZE`, `VACUUM`, `CLUSTER`, `REINDEX`

Again, note that some of these commands are actually allowed during "read only" mode transactions on the primary.

As a result, you cannot create additional indexes that exist solely on the standby, nor statistics that exist solely on the standby. If these administration commands are needed, they should be executed on the primary, and eventually those changes will propagate to the standby.

`pg_cancel_backend()` and `pg_terminate_backend()` will work on user backends, but not the Startup process, which performs recovery. `pg_stat_activity` does not show an entry for the Startup process, nor do recovering transactions show as active. As a result, `pg_prepared_xacts` is always empty during recovery. If you wish to resolve in-doubt prepared transactions, view `pg_prepared_xacts` on the primary and issue commands to resolve transactions there.

`pg_locks` will show locks held by backends, as normal. `pg_locks` also shows a virtual transaction managed by the Startup process that owns all `AccessExclusiveLocks` held by transactions being replayed by recovery. Note that the Startup process does not acquire locks to make database changes, and thus locks other than `AccessExclusiveLocks` do not show in `pg_locks` for the Startup process; they are just presumed to exist.

The Nagios plugin check_pgsql will work, because the simple information it checks for exists. The check_postgres monitoring script will also work, though some reported values could give different or confusing results. For example, last vacuum time will not be maintained, since no vacuum occurs on the standby. Vacuums running on the primary do still send their changes to the standby.

WAL file control commands will not work during recovery, e.g., `pg_start_backup`, `pg_switch_xlog` etc.

Dynamically loadable modules work, including `pg_stat_statements`.

Advisory locks work normally in recovery, including deadlock detection. Note that advisory locks are never WAL logged, so it is impossible for an advisory lock on either the primary or the standby to conflict with WAL replay. Nor is it possible to acquire an advisory lock on the primary and have it initiate a similar advisory lock on the standby. Advisory locks relate only to the server on which they are acquired.

Trigger-based replication systems such as Slony, Londiste and Bucardo won't run on the standby at all, though they will run happily on the primary server as long as the changes are not sent to standby servers to be applied. WAL replay is not trigger-based so you cannot relay from the standby to any system that requires additional database writes or relies on the use of triggers.

New OIDs cannot be assigned, though some UUID generators may still work as long as they do not rely on writing new status to the database.

Currently, temporary table creation is not allowed during read only transactions, so in some cases existing scripts will not run correctly. This restriction might be relaxed in a later release. This is both a SQL Standard compliance issue and a technical issue.

`DROP TABLESPACE` can only succeed if the tablespace is empty. Some standby users may be actively using the tablespace via their `temp_tablespaces` parameter. If there are temporary files in the tablespace, all active queries are canceled to ensure that temporary files are removed, so the tablespace can be removed and WAL replay can continue.

Running `DROP DATABASE` or `ALTER DATABASE ... SET TABLESPACE` on the primary will generate a WAL entry that will cause all users connected to that database on the standby to be forcibly disconnected. This action occurs immediately, whatever the setting of `max_standby_streaming_delay`. Note

that ALTER DATABASE ... RENAME does not disconnect users, which in most cases will go unnoticed, though might in some cases cause a program confusion if it depends in some way upon database name.

In normal (non-recovery) mode, if you issue DROP USER or DROP ROLE for a role with login capability while that user is still connected then nothing happens to the connected user - they remain connected. The user cannot reconnect however. This behavior applies in recovery also, so a DROP USER on the primary does not disconnect that user on the standby.

The statistics collector is active during recovery. All scans, reads, blocks, index usage, etc., will be recorded normally on the standby. Replayed actions will not duplicate their effects on primary, so replaying an insert will not increment the Inserts column of pg_stat_user_tables. The stats file is deleted at the start of recovery, so stats from primary and standby will differ; this is considered a feature, not a bug.

Autovacuum is not active during recovery. It will start normally at the end of recovery.

The checkpointer process and the background writer process are active during recovery. The checkpointer process will perform restartpoints (similar to checkpoints on the primary) and the background writer process will perform normal block cleaning activities. This can include updates of the hint bit information stored on the standby server. The CHECKPOINT command is accepted during recovery, though it performs a restartpoint rather than a new checkpoint.

25.5.4. Hot Standby Parameter Reference

Various parameters have been mentioned above in Section 25.5.2 and Section 25.5.3.

On the primary, parameters wal_level and vacuum_defer_cleanup_age can be used. max_standby_archive_delay and max_standby_streaming_delay have no effect if set on the primary.

On the standby, parameters hot_standby, max_standby_archive_delay and max_standby_streaming_delay can be used. vacuum_defer_cleanup_age has no effect as long as the server remains in standby mode, though it will become relevant if the standby becomes primary.

25.5.5. Caveats

There are several limitations of Hot Standby. These can and probably will be fixed in future releases:

- Operations on hash indexes are not presently WAL-logged, so replay will not update these indexes.
- Full knowledge of running transactions is required before snapshots can be taken. Transactions that use large numbers of subtransactions (currently greater than 64) will delay the start of read only connections until the completion of the longest running write transaction. If this situation occurs, explanatory messages will be sent to the server log.
- Valid starting points for standby queries are generated at each checkpoint on the master. If the standby is shut down while the master is in a shutdown state, it might not be possible to re-enter Hot Standby until the primary is started up, so that it generates further starting points in the WAL logs. This situation isn't a problem in the most common situations where it might happen. Generally, if the primary is shut down and not available anymore, that's likely due to a serious failure that requires the standby being converted to operate as the new primary anyway. And in situations where the primary is being

intentionally taken down, coordinating to make sure the standby becomes the new primary smoothly is also standard procedure.

- At the end of recovery, `AccessExclusiveLocks` held by prepared transactions will require twice the normal number of lock table entries. If you plan on running either a large number of concurrent prepared transactions that normally take `AccessExclusiveLocks`, or you plan on having one large transaction that takes many `AccessExclusiveLocks`, you are advised to select a larger value of `max_locks_per_transaction`, perhaps as much as twice the value of the parameter on the primary server. You need not consider this at all if your setting of `max_prepared_transactions` is 0.
- The Serializable transaction isolation level is not yet available in hot standby. (See Section 13.2.3 and Section 13.4.1 for details.) An attempt to set a transaction to the serializable isolation level in hot standby mode will generate an error.

Chapter 26. Recovery Configuration

This chapter describes the settings available in the `recovery.conf` file. They apply only for the duration of the recovery. They must be reset for any subsequent recovery you wish to perform. They cannot be changed once recovery has begun.

Settings in `recovery.conf` are specified in the format `name = 'value'`. One parameter is specified per line. Hash marks (`#`) designate the rest of the line as a comment. To embed a single quote in a parameter value, write two quotes (`''`).

A sample file, `share/recovery.conf.sample`, is provided in the installation's `share/` directory.

26.1. Archive Recovery Settings

`restore_command` (`string`)

The local shell command to execute to retrieve an archived segment of the WAL file series. This parameter is required for archive recovery, but optional for streaming replication. Any `%f` in the string is replaced by the name of the file to retrieve from the archive, and any `%p` is replaced by the copy destination path name on the server. (The path name is relative to the current working directory, i.e., the cluster's data directory.) Any `%r` is replaced by the name of the file containing the last valid restart point. That is the earliest file that must be kept to allow a restore to be restartable, so this information can be used to truncate the archive to just the minimum required to support restarting from the current restore. `%r` is typically only used by warm-standby configurations (see Section 25.2). Write `%%` to embed an actual `%` character.

It is important for the command to return a zero exit status only if it succeeds. The command *will* be asked for file names that are not present in the archive; it must return nonzero when so asked. Examples:

```
restore_command = 'cp /mnt/server/archivedir/%f "%p"'
restore_command = 'copy "C:\\server\\archivedir\\%f" "%p"'  # Windows
```

An exception is that if the command was terminated by a signal (other than SIGTERM, which is used as part of a database server shutdown) or an error by the shell (such as command not found), then recovery will abort and the server will not start up.

`archive_cleanup_command` (`string`)

This optional parameter specifies a shell command that will be executed at every restartpoint. The purpose of `archive_cleanup_command` is to provide a mechanism for cleaning up old archived WAL files that are no longer needed by the standby server. Any `%r` is replaced by the name of the file containing the last valid restart point. That is the earliest file that must be *kept* to allow a restore to be restartable, and so all files earlier than `%r` may be safely removed. This information can be used to truncate the archive to just the minimum required to support restart from the current restore. The pg_archivecleanup module is often used in `archive_cleanup_command` for single-standby configurations, for example:

```
archive_cleanup_command = 'pg_archivecleanup /mnt/server/archivedir %r'
```

Note however that if multiple standby servers are restoring from the same archive directory, you will need to ensure that you do not delete WAL files until they are no longer needed by any of the servers.

archive_cleanup_command would typically be used in a warm-standby configuration (see Section 25.2). Write %% to embed an actual % character in the command.

If the command returns a nonzero exit status then a warning log message will be written. An exception is that if the command was terminated by a signal or an error by the shell (such as command not found), a fatal error will be raised.

recovery_end_command (string)

This parameter specifies a shell command that will be executed once only at the end of recovery. This parameter is optional. The purpose of the recovery_end_command is to provide a mechanism for cleanup following replication or recovery. Any %r is replaced by the name of the file containing the last valid restart point, like in archive_cleanup_command.

If the command returns a nonzero exit status then a warning log message will be written and the database will proceed to start up anyway. An exception is that if the command was terminated by a signal or an error by the shell (such as command not found), the database will not proceed with startup.

26.2. Recovery Target Settings

By default, recovery will recover to the end of the WAL log. The following parameters can be used to specify an earlier stopping point. At most one of recovery_target, recovery_target_name, recovery_target_time, or recovery_target_xid can be used; if more than one of these is specified in the configuration file, the last entry will be used.

recovery_target = 'immediate'

This parameter specifies that recovery should end as soon as a consistent state is reached, i.e., as early as possible. When restoring from an online backup, this means the point where taking the backup ended.

Technically, this is a string parameter, but 'immediate' is currently the only allowed value.

recovery_target_name (string)

This parameter specifies the named restore point (created with pg_create_restore_point()) to which recovery will proceed.

recovery_target_time (timestamp)

This parameter specifies the time stamp up to which recovery will proceed. The precise stopping point is also influenced by recovery_target_inclusive.

recovery_target_xid (string)

This parameter specifies the transaction ID up to which recovery will proceed. Keep in mind that while transaction IDs are assigned sequentially at transaction start, transactions can complete in a different numeric order. The transactions that will be recovered are those that committed before (and optionally including) the specified one. The precise stopping point is also influenced by recovery_target_inclusive.

The following options further specify the recovery target, and affect what happens when the target is reached:

recovery_target_inclusive (boolean)

Specifies whether to stop just after the specified recovery target (true), or just before the recovery target (false). Applies when either recovery_target_time or recovery_target_xid is specified. This setting controls whether transactions having exactly the target commit time or ID, respectively, will be included in the recovery. Default is true.

recovery_target_timeline (string)

Specifies recovering into a particular timeline. The default is to recover along the same timeline that was current when the base backup was taken. Setting this to latest recovers to the latest timeline found in the archive, which is useful in a standby server. Other than that you only need to set this parameter in complex re-recovery situations, where you need to return to a state that itself was reached after a point-in-time recovery. See Section 24.3.5 for discussion.

recovery_target_action (enum)

Specifies what action the server should take once the recovery target is reached. The default is pause, which means recovery will be paused. promote means the recovery process will finish and the server will start to accept connections. Finally shutdown will stop the server after reaching the recovery target.

The intended use of the pause setting is to allow queries to be executed against the database to check if this recovery target is the most desirable point for recovery. The paused state can be resumed by using pg_xlog_replay_resume() (see Table 9-70), which then causes recovery to end. If this recovery target is not the desired stopping point, then shut down the server, change the recovery target settings to a later target and restart to continue recovery.

The shutdown setting is useful to have the instance ready at the exact replay point desired. The instance will still be able to replay more WAL records (and in fact will have to replay WAL records since the last checkpoint next time it is started).

Note that because recovery.conf will not be renamed when recovery_target_action is set to shutdown, any subsequent start will end with immediate shutdown unless the configuration is changed or the recovery.conf file is removed manually.

This setting has no effect if no recovery target is set. If hot_standby is not enabled, a setting of pause will act the same as shutdown.

26.3. Standby Server Settings

standby_mode (boolean)

Specifies whether to start the PostgreSQL server as a standby. If this parameter is on, the server will not stop recovery when the end of archived WAL is reached, but will keep trying to continue recovery by fetching new WAL segments using restore_command and/or by connecting to the primary server as specified by the primary_conninfo setting.

primary_conninfo (string)

Specifies a connection string to be used for the standby server to connect with the primary. This string is in the format described in Section 31.1.1. If any option is unspecified in this string, then the

corresponding environment variable (see Section 31.14) is checked. If the environment variable is not set either, then defaults are used.

The connection string should specify the host name (or address) of the primary server, as well as the port number if it is not the same as the standby server's default. Also specify a user name corresponding to a suitably-privileged role on the primary (see Section 25.2.5.1). A password needs to be provided too, if the primary demands password authentication. It can be provided in the `primary_conninfo` string, or in a separate `~/.pgpass` file on the standby server (use `replication` as the database name). Do not specify a database name in the `primary_conninfo` string.

This setting has no effect if `standby_mode` is `off`.

`primary_slot_name` (`string`)

Optionally specifies an existing replication slot to be used when connecting to the primary via streaming replication to control resource removal on the upstream node (see Section 25.2.6). This setting has no effect if `primary_conninfo` is not set.

`trigger_file` (`string`)

Specifies a trigger file whose presence ends recovery in the standby. Even if this value is not set, you can still promote the standby using `pg_ctl promote`. This setting has no effect if `standby_mode` is `off`.

`recovery_min_apply_delay` (`integer`)

By default, a standby server restores WAL records from the primary as soon as possible. It may be useful to have a time-delayed copy of the data, offering opportunities to correct data loss errors. This parameter allows you to delay recovery by a fixed period of time, measured in milliseconds if no unit is specified. For example, if you set this parameter to `5min`, the standby will replay each transaction commit only when the system time on the standby is at least five minutes past the commit time reported by the master.

It is possible that the replication delay between servers exceeds the value of this parameter, in which case no delay is added. Note that the delay is calculated between the WAL time stamp as written on master and the current time on the standby. Delays in transfer because of network lag or cascading replication configurations may reduce the actual wait time significantly. If the system clocks on master and standby are not synchronized, this may lead to recovery applying records earlier than expected; but that is not a major issue because useful settings of this parameter are much larger than typical time deviations between servers.

The delay occurs only on WAL records for transaction commits. Other records are replayed as quickly as possible, which is not a problem because MVCC visibility rules ensure their effects are not visible until the corresponding commit record is applied.

The delay occurs once the database in recovery has reached a consistent state, until the standby is promoted or triggered. After that the standby will end recovery without further waiting.

This parameter is intended for use with streaming replication deployments; however, if the parameter is specified it will be honored in all cases. Synchronous replication is not affected by this setting because there is not yet any setting to request synchronous apply of transaction commits. `hot_standby_feedback` will be delayed by use of this feature which could lead to bloat on the master; use both together with care.

Chapter 27. Monitoring Database Activity

A database administrator frequently wonders, "What is the system doing right now?" This chapter discusses how to find that out.

Several tools are available for monitoring database activity and analyzing performance. Most of this chapter is devoted to describing PostgreSQL's statistics collector, but one should not neglect regular Unix monitoring programs such as `ps`, `top`, `iostat`, and `vmstat`. Also, once one has identified a poorly-performing query, further investigation might be needed using PostgreSQL's EXPLAIN command. Section 14.1 discusses `EXPLAIN` and other methods for understanding the behavior of an individual query.

27.1. Standard Unix Tools

On most Unix platforms, PostgreSQL modifies its command title as reported by `ps`, so that individual server processes can readily be identified. A sample display is

```
$ ps auxww | grep ^postgres
postgres  15551  0.0  0.1  57536  7132 pts/0    S    18:02   0:00 postgres -i
postgres  15554  0.0  0.0  57536  1184 ?        Ss   18:02   0:00 postgres: writer process
postgres  15555  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres: checkpointer pr
postgres  15556  0.0  0.0  57536   916 ?        Ss   18:02   0:00 postgres: wal writer proc
postgres  15557  0.0  0.0  58504  2244 ?        Ss   18:02   0:00 postgres: autovacuum laun
postgres  15558  0.0  0.0  17512  1068 ?        Ss   18:02   0:00 postgres: stats collector
postgres  15582  0.0  0.0  58772  3080 ?        Ss   18:04   0:00 postgres: joe runbug 127.
postgres  15606  0.0  0.0  58772  3052 ?        Ss   18:07   0:00 postgres: tgl regression
postgres  15610  0.0  0.0  58772  3056 ?        Ss   18:07   0:00 postgres: tgl regression
```

(The appropriate invocation of `ps` varies across different platforms, as do the details of what is shown. This example is from a recent Linux system.) The first process listed here is the master server process. The command arguments shown for it are the same ones used when it was launched. The next five processes are background worker processes automatically launched by the master process. (The "stats collector" process will not be present if you have set the system not to start the statistics collector; likewise the "autovacuum launcher" process can be disabled.) Each of the remaining processes is a server process handling one client connection. Each such process sets its command line display in the form

```
postgres: user database host activity
```

The user, database, and (client) host items remain the same for the life of the client connection, but the activity indicator changes. The activity can be `idle` (i.e., waiting for a client command), `idle in transaction` (waiting for client inside a `BEGIN` block), or a command type name such as `SELECT`. Also, `waiting` is appended if the server process is presently waiting on a lock held by another session. In the above example we can infer that process 15606 is waiting for process 15610 to complete its transaction and thereby release some lock. (Process 15610 must be the blocker, because there is no other active session. In more complicated cases it would be necessary to look into the `pg_locks` system view to determine who is blocking whom.)

If cluster_name has been configured the cluster name will also be shown in `ps` output:

```
$ psql -c 'SHOW cluster_name'
 cluster_name
```

```
--------------
 server1
(1 row)

$ ps aux|grep server1
postgres   27093  0.0  0.0  30096  2752 ?        Ss   11:34   0:00 postgres: server1: write
...
```

If you have turned off update_process_title then the activity indicator is not updated; the process title is set only once when a new process is launched. On some platforms this saves a measurable amount of per-command overhead; on others it's insignificant.

> **Tip:** Solaris requires special handling. You must use `/usr/ucb/ps`, rather than `/bin/ps`. You also must use two `w` flags, not just one. In addition, your original invocation of the `postgres` command must have a shorter `ps` status display than that provided by each server process. If you fail to do all three things, the `ps` output for each server process will be the original `postgres` command line.

27.2. The Statistics Collector

PostgreSQL's *statistics collector* is a subsystem that supports collection and reporting of information about server activity. Presently, the collector can count accesses to tables and indexes in both disk-block and individual-row terms. It also tracks the total number of rows in each table, and information about vacuum and analyze actions for each table. It can also count calls to user-defined functions and the total time spent in each one.

PostgreSQL also supports reporting dynamic information about exactly what is going on in the system right now, such as the exact command currently being executed by other server processes, and which other connections exist in the system. This facility is independent of the collector process.

27.2.1. Statistics Collection Configuration

Since collection of statistics adds some overhead to query execution, the system can be configured to collect or not collect information. This is controlled by configuration parameters that are normally set in `postgresql.conf`. (See Chapter 18 for details about setting configuration parameters.)

The parameter track_activities enables monitoring of the current command being executed by any server process.

The parameter track_counts controls whether statistics are collected about table and index accesses.

The parameter track_functions enables tracking of usage of user-defined functions.

The parameter track_io_timing enables monitoring of block read and write times.

Normally these parameters are set in `postgresql.conf` so that they apply to all server processes, but it is possible to turn them on or off in individual sessions using the SET command. (To prevent ordinary users

from hiding their activity from the administrator, only superusers are allowed to change these parameters with SET.)

The statistics collector transmits the collected information to other PostgreSQL processes through temporary files. These files are stored in the directory named by the stats_temp_directory parameter, pg_stat_tmp by default. For better performance, stats_temp_directory can be pointed at a RAM-based file system, decreasing physical I/O requirements. When the server shuts down cleanly, a permanent copy of the statistics data is stored in the pg_stat subdirectory, so that statistics can be retained across server restarts. When recovery is performed at server start (e.g., after immediate shutdown, server crash, and point-in-time recovery), all statistics counters are reset.

27.2.2. Viewing Statistics

Several predefined views, listed in Table 27-1, are available to show the current state of the system. There are also several other views, listed in Table 27-2, available to show the results of statistics collection. Alternatively, one can build custom views using the underlying statistics functions, as discussed in Section 27.2.3.

When using the statistics to monitor collected data, it is important to realize that the information does not update instantaneously. Each individual server process transmits new statistical counts to the collector just before going idle; so a query or transaction still in progress does not affect the displayed totals. Also, the collector itself emits a new report at most once per PGSTAT_STAT_INTERVAL milliseconds (500 ms unless altered while building the server). So the displayed information lags behind actual activity. However, current-query information collected by track_activities is always up-to-date.

Another important point is that when a server process is asked to display any of these statistics, it first fetches the most recent report emitted by the collector process and then continues to use this snapshot for all statistical views and functions until the end of its current transaction. So the statistics will show static information as long as you continue the current transaction. Similarly, information about the current queries of all sessions is collected when any such information is first requested within a transaction, and the same information will be displayed throughout the transaction. This is a feature, not a bug, because it allows you to perform several queries on the statistics and correlate the results without worrying that the numbers are changing underneath you. But if you want to see new results with each query, be sure to do the queries outside any transaction block. Alternatively, you can invoke pg_stat_clear_snapshot(), which will discard the current transaction's statistics snapshot (if any). The next use of statistical information will cause a new snapshot to be fetched.

A transaction can also see its own statistics (as yet untransmitted to the collector) in the views pg_stat_xact_all_tables, pg_stat_xact_sys_tables, pg_stat_xact_user_tables, and pg_stat_xact_user_functions. These numbers do not act as stated above; instead they update continuously throughout the transaction.

Table 27-1. Dynamic Statistics Views

View Name	Description
pg_stat_activity	One row per server process, showing information related to the current activity of that process, such as state and current query. See pg_stat_activity for details.

View Name	Description
pg_stat_replication	One row per WAL sender process, showing statistics about replication to that sender's connected standby server. See pg_stat_replication for details.
pg_stat_ssl	One row per connection (regular and replication), showing information about SSL used on this connection. See pg_stat_ssl for details.

Table 27-2. Collected Statistics Views

View Name	Description
pg_stat_archiver	One row only, showing statistics about the WAL archiver process's activity. See pg_stat_archiver for details.
pg_stat_bgwriter	One row only, showing statistics about the background writer process's activity. See pg_stat_bgwriter for details.
pg_stat_database	One row per database, showing database-wide statistics. See pg_stat_database for details.
pg_stat_database_conflicts	One row per database, showing database-wide statistics about query cancels due to conflict with recovery on standby servers. See pg_stat_database_conflicts for details.
pg_stat_all_tables	One row for each table in the current database, showing statistics about accesses to that specific table. See pg_stat_all_tables for details.
pg_stat_sys_tables	Same as pg_stat_all_tables, except that only system tables are shown.
pg_stat_user_tables	Same as pg_stat_all_tables, except that only user tables are shown.
pg_stat_xact_all_tables	Similar to pg_stat_all_tables, but counts actions taken so far within the current transaction (which are *not* yet included in pg_stat_all_tables and related views). The columns for numbers of live and dead rows and vacuum and analyze actions are not present in this view.
pg_stat_xact_sys_tables	Same as pg_stat_xact_all_tables, except that only system tables are shown.
pg_stat_xact_user_tables	Same as pg_stat_xact_all_tables, except that only user tables are shown.

View Name	Description
pg_stat_all_indexes	One row for each index in the current database, showing statistics about accesses to that specific index. See pg_stat_all_indexes for details.
pg_stat_sys_indexes	Same as pg_stat_all_indexes, except that only indexes on system tables are shown.
pg_stat_user_indexes	Same as pg_stat_all_indexes, except that only indexes on user tables are shown.
pg_statio_all_tables	One row for each table in the current database, showing statistics about I/O on that specific table. See pg_statio_all_tables for details.
pg_statio_sys_tables	Same as pg_statio_all_tables, except that only system tables are shown.
pg_statio_user_tables	Same as pg_statio_all_tables, except that only user tables are shown.
pg_statio_all_indexes	One row for each index in the current database, showing statistics about I/O on that specific index. See pg_statio_all_indexes for details.
pg_statio_sys_indexes	Same as pg_statio_all_indexes, except that only indexes on system tables are shown.
pg_statio_user_indexes	Same as pg_statio_all_indexes, except that only indexes on user tables are shown.
pg_statio_all_sequences	One row for each sequence in the current database, showing statistics about I/O on that specific sequence. See pg_statio_all_sequences for details.
pg_statio_sys_sequences	Same as pg_statio_all_sequences, except that only system sequences are shown. (Presently, no system sequences are defined, so this view is always empty.)
pg_statio_user_sequences	Same as pg_statio_all_sequences, except that only user sequences are shown.
pg_stat_user_functions	One row for each tracked function, showing statistics about executions of that function. See pg_stat_user_functions for details.
pg_stat_xact_user_functions	Similar to pg_stat_user_functions, but counts only calls during the current transaction (which are *not* yet included in pg_stat_user_functions).

The per-index statistics are particularly useful to determine which indexes are being used and how effective they are.

The pg_statio_ views are primarily useful to determine the effectiveness of the buffer cache. When the number of actual disk reads is much smaller than the number of buffer hits, then the cache is satisfying most read requests without invoking a kernel call. However, these statistics do not give the entire story:

due to the way in which PostgreSQL handles disk I/O, data that is not in the PostgreSQL buffer cache might still reside in the kernel's I/O cache, and might therefore still be fetched without requiring a physical read. Users interested in obtaining more detailed information on PostgreSQL I/O behavior are advised to use the PostgreSQL statistics collector in combination with operating system utilities that allow insight into the kernel's handling of I/O.

Table 27-3. `pg_stat_activity` View

Column	Type	Description
`datid`	`oid`	OID of the database this backend is connected to
`datname`	`name`	Name of the database this backend is connected to
`pid`	`integer`	Process ID of this backend
`usesysid`	`oid`	OID of the user logged into this backend
`usename`	`name`	Name of the user logged into this backend
`application_name`	`text`	Name of the application that is connected to this backend
`client_addr`	`inet`	IP address of the client connected to this backend. If this field is null, it indicates either that the client is connected via a Unix socket on the server machine or that this is an internal process such as autovacuum.
`client_hostname`	`text`	Host name of the connected client, as reported by a reverse DNS lookup of `client_addr`. This field will only be non-null for IP connections, and only when log_hostname is enabled.
`client_port`	`integer`	TCP port number that the client is using for communication with this backend, or `-1` if a Unix socket is used
`backend_start`	`timestamp with time zone`	Time when this process was started, i.e., when the client connected to the server

Column	Type	Description
xact_start	timestamp with time zone	Time when this process' current transaction was started, or null if no transaction is active. If the current query is the first of its transaction, this column is equal to the query_start column.
query_start	timestamp with time zone	Time when the currently active query was started, or if state is not active, when the last query was started
state_change	timestamp with time zone	Time when the state was last changed
waiting	boolean	True if this backend is currently waiting on a lock
state	text	Current overall state of this backend. Possible values are: • active: The backend is executing a query. • idle: The backend is waiting for a new client command. • idle in transaction: The backend is in a transaction, but is not currently executing a query. • idle in transaction (aborted): This state is similar to idle in transaction, except one of the statements in the transaction caused an error. • fastpath function call: The backend is executing a fast-path function. • disabled: This state is reported if track_activities is disabled in this backend.
backend_xid	xid	Top-level transaction identifier of this backend, if any.

Column	Type	Description
backend_xmin	xid	The current backend's xmin horizon.
query	text	Text of this backend's most recent query. If state is active this field shows the currently executing query. In all other states, it shows the last query that was executed.

The pg_stat_activity view will have one row per server process, showing information related to the current activity of that process.

> **Note:** The waiting and state columns are independent. If a backend is in the active state, it may or may not be waiting. If the state is active and waiting is true, it means that a query is being executed, but is being blocked by a lock somewhere in the system.

Table 27-4. pg_stat_replication View

Column	Type	Description
pid	integer	Process ID of a WAL sender process
usesysid	oid	OID of the user logged into this WAL sender process
usename	name	Name of the user logged into this WAL sender process
application_name	text	Name of the application that is connected to this WAL sender
client_addr	inet	IP address of the client connected to this WAL sender. If this field is null, it indicates that the client is connected via a Unix socket on the server machine.
client_hostname	text	Host name of the connected client, as reported by a reverse DNS lookup of client_addr. This field will only be non-null for IP connections, and only when log_hostname is enabled.
client_port	integer	TCP port number that the client is using for communication with this WAL sender, or -1 if a Unix socket is used

Column	Type	Description
backend_start	timestamp with time zone	Time when this process was started, i.e., when the client connected to this WAL sender
backend_xmin	xid	This standby's xmin horizon reported by hot_standby_feedback.
state	text	Current WAL sender state
sent_location	pg_lsn	Last transaction log position sent on this connection
write_location	pg_lsn	Last transaction log position written to disk by this standby server
flush_location	pg_lsn	Last transaction log position flushed to disk by this standby server
replay_location	pg_lsn	Last transaction log position replayed into the database on this standby server
sync_priority	integer	Priority of this standby server for being chosen as the synchronous standby
sync_state	text	Synchronous state of this standby server

The pg_stat_replication view will contain one row per WAL sender process, showing statistics about replication to that sender's connected standby server. Only directly connected standbys are listed; no information is available about downstream standby servers.

Table 27-5. pg_stat_ssl View

Column	Type	Description
pid	integer	Process ID of a backend or WAL sender process
ssl	boolean	True if SSL is used on this connection
version	text	Version of SSL in use, or NULL if SSL is not in use on this connection
cipher	text	Name of SSL cipher in use, or NULL if SSL is not in use on this connection
bits	integer	Number of bits in the encryption algorithm used, or NULL if SSL is not used on this connection

Column	Type	Description
compression	boolean	True if SSL compression is in use, false if not, or NULL if SSL is not in use on this connection
clientdn	text	Distinguished Name (DN) field from the client certificate used, or NULL if no client certificate was supplied or if SSL is not in use on this connection. This field is truncated if the DN field is longer than NAMEDATALEN (64 characters in a standard build)

The pg_stat_ssl view will contain one row per backend or WAL sender process, showing statistics about SSL usage on this connection. It can be joined to pg_stat_activity or pg_stat_replication on the pid column to get more details about the connection.

Table 27-6. pg_stat_archiver View

Column	Type	Description
archived_count	bigint	Number of WAL files that have been successfully archived
last_archived_wal	text	Name of the last WAL file successfully archived
last_archived_time	timestamp with time zone	Time of the last successful archive operation
failed_count	bigint	Number of failed attempts for archiving WAL files
last_failed_wal	text	Name of the WAL file of the last failed archival operation
last_failed_time	timestamp with time zone	Time of the last failed archival operation
stats_reset	timestamp with time zone	Time at which these statistics were last reset

The pg_stat_archiver view will always have a single row, containing data about the archiver process of the cluster.

Table 27-7. pg_stat_bgwriter View

Column	Type	Description
checkpoints_timed	bigint	Number of scheduled checkpoints that have been performed

Column	Type	Description
checkpoints_req	bigint	Number of requested checkpoints that have been performed
checkpoint_write_time	double precision	Total amount of time that has been spent in the portion of checkpoint processing where files are written to disk, in milliseconds
checkpoint_sync_time	double precision	Total amount of time that has been spent in the portion of checkpoint processing where files are synchronized to disk, in milliseconds
buffers_checkpoint	bigint	Number of buffers written during checkpoints
buffers_clean	bigint	Number of buffers written by the background writer
maxwritten_clean	bigint	Number of times the background writer stopped a cleaning scan because it had written too many buffers
buffers_backend	bigint	Number of buffers written directly by a backend
buffers_backend_fsync	bigint	Number of times a backend had to execute its own fsync call (normally the background writer handles those even when the backend does its own write)
buffers_alloc	bigint	Number of buffers allocated
stats_reset	timestamp with time zone	Time at which these statistics were last reset

The pg_stat_bgwriter view will always have a single row, containing global data for the cluster.

Table 27-8. pg_stat_database View

Column	Type	Description
datid	oid	OID of a database
datname	name	Name of this database

Column	Type	Description
numbackends	integer	Number of backends currently connected to this database. This is the only column in this view that returns a value reflecting current state; all other columns return the accumulated values since the last reset.
xact_commit	bigint	Number of transactions in this database that have been committed
xact_rollback	bigint	Number of transactions in this database that have been rolled back
blks_read	bigint	Number of disk blocks read in this database
blks_hit	bigint	Number of times disk blocks were found already in the buffer cache, so that a read was not necessary (this only includes hits in the PostgreSQL buffer cache, not the operating system's file system cache)
tup_returned	bigint	Number of rows returned by queries in this database
tup_fetched	bigint	Number of rows fetched by queries in this database
tup_inserted	bigint	Number of rows inserted by queries in this database
tup_updated	bigint	Number of rows updated by queries in this database
tup_deleted	bigint	Number of rows deleted by queries in this database
conflicts	bigint	Number of queries canceled due to conflicts with recovery in this database. (Conflicts occur only on standby servers; see pg_stat_database_conflicts for details.)

Column	Type	Description
temp_files	bigint	Number of temporary files created by queries in this database. All temporary files are counted, regardless of why the temporary file was created (e.g., sorting or hashing), and regardless of the log_temp_files setting.
temp_bytes	bigint	Total amount of data written to temporary files by queries in this database. All temporary files are counted, regardless of why the temporary file was created, and regardless of the log_temp_files setting.
deadlocks	bigint	Number of deadlocks detected in this database
blk_read_time	double precision	Time spent reading data file blocks by backends in this database, in milliseconds
blk_write_time	double precision	Time spent writing data file blocks by backends in this database, in milliseconds
stats_reset	timestamp with time zone	Time at which these statistics were last reset

The pg_stat_database view will contain one row for each database in the cluster, showing database-wide statistics.

Table 27-9. pg_stat_database_conflicts View

Column	Type	Description
datid	oid	OID of a database
datname	name	Name of this database
confl_tablespace	bigint	Number of queries in this database that have been canceled due to dropped tablespaces
confl_lock	bigint	Number of queries in this database that have been canceled due to lock timeouts
confl_snapshot	bigint	Number of queries in this database that have been canceled due to old snapshots

Column	Type	Description
confl_bufferpin	bigint	Number of queries in this database that have been canceled due to pinned buffers
confl_deadlock	bigint	Number of queries in this database that have been canceled due to deadlocks

The pg_stat_database_conflicts view will contain one row per database, showing database-wide statistics about query cancels occurring due to conflicts with recovery on standby servers. This view will only contain information on standby servers, since conflicts do not occur on master servers.

Table 27-10. pg_stat_all_tables View

Column	Type	Description
relid	oid	OID of a table
schemaname	name	Name of the schema that this table is in
relname	name	Name of this table
seq_scan	bigint	Number of sequential scans initiated on this table
seq_tup_read	bigint	Number of live rows fetched by sequential scans
idx_scan	bigint	Number of index scans initiated on this table
idx_tup_fetch	bigint	Number of live rows fetched by index scans
n_tup_ins	bigint	Number of rows inserted
n_tup_upd	bigint	Number of rows updated (includes HOT updated rows)
n_tup_del	bigint	Number of rows deleted
n_tup_hot_upd	bigint	Number of rows HOT updated (i.e., with no separate index update required)
n_live_tup	bigint	Estimated number of live rows
n_dead_tup	bigint	Estimated number of dead rows
n_mod_since_analyze	bigint	Estimated number of rows modified since this table was last analyzed
last_vacuum	timestamp with time zone	Last time at which this table was manually vacuumed (not counting VACUUM FULL)

Column	Type	Description
last_autovacuum	timestamp with time zone	Last time at which this table was vacuumed by the autovacuum daemon
last_analyze	timestamp with time zone	Last time at which this table was manually analyzed
last_autoanalyze	timestamp with time zone	Last time at which this table was analyzed by the autovacuum daemon
vacuum_count	bigint	Number of times this table has been manually vacuumed (not counting VACUUM FULL)
autovacuum_count	bigint	Number of times this table has been vacuumed by the autovacuum daemon
analyze_count	bigint	Number of times this table has been manually analyzed
autoanalyze_count	bigint	Number of times this table has been analyzed by the autovacuum daemon

The pg_stat_all_tables view will contain one row for each table in the current database (including TOAST tables), showing statistics about accesses to that specific table. The pg_stat_user_tables and pg_stat_sys_tables views contain the same information, but filtered to only show user and system tables respectively.

Table 27-11. pg_stat_all_indexes View

Column	Type	Description
relid	oid	OID of the table for this index
indexrelid	oid	OID of this index
schemaname	name	Name of the schema this index is in
relname	name	Name of the table for this index
indexrelname	name	Name of this index
idx_scan	bigint	Number of index scans initiated on this index
idx_tup_read	bigint	Number of index entries returned by scans on this index
idx_tup_fetch	bigint	Number of live table rows fetched by simple index scans using this index

The pg_stat_all_indexes view will contain one row for each index in the current database, showing statistics about accesses to that specific index. The pg_stat_user_indexes and

pg_stat_sys_indexes views contain the same information, but filtered to only show user and system indexes respectively.

Indexes can be used by simple index scans, "bitmap" index scans, and the optimizer. In a bitmap scan the output of several indexes can be combined via AND or OR rules, so it is difficult to associate individual heap row fetches with specific indexes when a bitmap scan is used. Therefore, a bitmap scan increments the pg_stat_all_indexes.idx_tup_read count(s) for the index(es) it uses, and it increments the pg_stat_all_tables.idx_tup_fetch count for the table, but it does not affect pg_stat_all_indexes.idx_tup_fetch. The optimizer also accesses indexes to check for supplied constants whose values are outside the recorded range of the optimizer statistics because the optimizer statistics might be stale.

> **Note:** The idx_tup_read and idx_tup_fetch counts can be different even without any use of bitmap scans, because idx_tup_read counts index entries retrieved from the index while idx_tup_fetch counts live rows fetched from the table. The latter will be less if any dead or not-yet-committed rows are fetched using the index, or if any heap fetches are avoided by means of an index-only scan.

Table 27-12. pg_statio_all_tables View

Column	Type	Description
relid	oid	OID of a table
schemaname	name	Name of the schema that this table is in
relname	name	Name of this table
heap_blks_read	bigint	Number of disk blocks read from this table
heap_blks_hit	bigint	Number of buffer hits in this table
idx_blks_read	bigint	Number of disk blocks read from all indexes on this table
idx_blks_hit	bigint	Number of buffer hits in all indexes on this table
toast_blks_read	bigint	Number of disk blocks read from this table's TOAST table (if any)
toast_blks_hit	bigint	Number of buffer hits in this table's TOAST table (if any)
tidx_blks_read	bigint	Number of disk blocks read from this table's TOAST table indexes (if any)
tidx_blks_hit	bigint	Number of buffer hits in this table's TOAST table indexes (if any)

The pg_statio_all_tables view will contain one row for each table in the current database (including TOAST tables), showing statistics about I/O on that specific table. The pg_statio_user_tables and

pg_statio_sys_tables views contain the same information, but filtered to only show user and system tables respectively.

Table 27-13. pg_statio_all_indexes View

Column	Type	Description
relid	oid	OID of the table for this index
indexrelid	oid	OID of this index
schemaname	name	Name of the schema this index is in
relname	name	Name of the table for this index
indexrelname	name	Name of this index
idx_blks_read	bigint	Number of disk blocks read from this index
idx_blks_hit	bigint	Number of buffer hits in this index

The pg_statio_all_indexes view will contain one row for each index in the current database, showing statistics about I/O on that specific index. The pg_statio_user_indexes and pg_statio_sys_indexes views contain the same information, but filtered to only show user and system indexes respectively.

Table 27-14. pg_statio_all_sequences View

Column	Type	Description
relid	oid	OID of a sequence
schemaname	name	Name of the schema this sequence is in
relname	name	Name of this sequence
blks_read	bigint	Number of disk blocks read from this sequence
blks_hit	bigint	Number of buffer hits in this sequence

The pg_statio_all_sequences view will contain one row for each sequence in the current database, showing statistics about I/O on that specific sequence.

Table 27-15. pg_stat_user_functions View

Column	Type	Description
funcid	oid	OID of a function
schemaname	name	Name of the schema this function is in
funcname	name	Name of this function

Column	Type	Description
calls	bigint	Number of times this function has been called
total_time	double precision	Total time spent in this function and all other functions called by it, in milliseconds
self_time	double precision	Total time spent in this function itself, not including other functions called by it, in milliseconds

The pg_stat_user_functions view will contain one row for each tracked function, showing statistics about executions of that function. The track_functions parameter controls exactly which functions are tracked.

27.2.3. Statistics Functions

Other ways of looking at the statistics can be set up by writing queries that use the same underlying statistics access functions used by the standard views shown above. For details such as the functions' names, consult the definitions of the standard views. (For example, in psql you could issue \d+ pg_stat_activity.) The access functions for per-database statistics take a database OID as an argument to identify which database to report on. The per-table and per-index functions take a table or index OID. The functions for per-function statistics take a function OID. Note that only tables, indexes, and functions in the current database can be seen with these functions.

Additional functions related to statistics collection are listed in Table 27-16.

Table 27-16. Additional Statistics Functions

Function	Return Type	Description
pg_backend_pid()	integer	Process ID of the server process handling the current session
pg_stat_get_activity(integer)	setof record	Returns a record of information about the backend with the specified PID, or one record for each active backend in the system if NULL is specified. The fields returned are a subset of those in the pg_stat_activity view.
pg_stat_get_snapshot_timestamp()	timestamp with time zone	Returns the timestamp of the current statistics snapshot
pg_stat_clear_snapshot()	void	Discard the current statistics snapshot

Function	Return Type	Description
pg_stat_reset()	void	Reset all statistics counters for the current database to zero (requires superuser privileges)
pg_stat_reset_shared(text)	void	Reset some cluster-wide statistics counters to zero, depending on the argument (requires superuser privileges). Calling pg_stat_reset_shared('bgwriter') will zero all the counters shown in the pg_stat_bgwriter view. Calling pg_stat_reset_shared('archiver') will zero all the counters shown in the pg_stat_archiver view.
pg_stat_reset_single_table_counters(oid)	void	Reset statistics for a single table or index in the current database to zero (requires superuser privileges)
pg_stat_reset_single_function_counters(oid)	void	Reset statistics for a single function in the current database to zero (requires superuser privileges)

pg_stat_get_activity, the underlying function of the pg_stat_activity view, returns a set of records containing all the available information about each backend process. Sometimes it may be more convenient to obtain just a subset of this information. In such cases, an older set of per-backend statistics access functions can be used; these are shown in Table 27-17. These access functions use a backend ID number, which ranges from one to the number of currently active backends. The function pg_stat_get_backend_idset provides a convenient way to generate one row for each active backend for invoking these functions. For example, to show the PIDs and current queries of all backends:

```
SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
       pg_stat_get_backend_activity(s.backendid) AS query
    FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

Table 27-17. Per-Backend Statistics Functions

Function	Return Type	Description
pg_stat_get_backend_idset()	setof integer	Set of currently active backend ID numbers (from 1 to the number of active backends)

Function	Return Type	Description
pg_stat_get_backend_activity(integer)	text	Text of this backend's most recent query
pg_stat_get_backend_activity_start(integer)	timestamp with time zone	Time when the most recent query was started
pg_stat_get_backend_client_addr(integer)	inet	IP address of the client connected to this backend
pg_stat_get_backend_client_port(integer)	integer	TCP port number that the client is using for communication
pg_stat_get_backend_dbid(integer)	oid	OID of the database this backend is connected to
pg_stat_get_backend_pid(integer)	integer	Process ID of this backend
pg_stat_get_backend_start(integer)	timestamp with time zone	Time when this process was started
pg_stat_get_backend_userid(integer)	oid	OID of the user logged into this backend
pg_stat_get_backend_waiting(integer)	boolean	True if this backend is currently waiting on a lock
pg_stat_get_backend_xact_start(integer)	timestamp with time zone	Time when the current transaction was started

27.3. Viewing Locks

Another useful tool for monitoring database activity is the pg_locks system table. It allows the database administrator to view information about the outstanding locks in the lock manager. For example, this capability can be used to:

- View all the locks currently outstanding, all the locks on relations in a particular database, all the locks on a particular relation, or all the locks held by a particular PostgreSQL session.
- Determine the relation in the current database with the most ungranted locks (which might be a source of contention among database clients).
- Determine the effect of lock contention on overall database performance, as well as the extent to which contention varies with overall database traffic.

Details of the pg_locks view appear in Section 49.63. For more information on locking and managing concurrency with PostgreSQL, refer to Chapter 13.

27.4. Dynamic Tracing

PostgreSQL provides facilities to support dynamic tracing of the database server. This allows an external utility to be called at specific points in the code and thereby trace execution.

A number of probes or trace points are already inserted into the source code. These probes are intended to be used by database developers and administrators. By default the probes are not compiled into PostgreSQL; the user needs to explicitly tell the configure script to make the probes available.

Currently, the DTrace[1] utility is supported, which, at the time of this writing, is available on Solaris, OS X, FreeBSD, NetBSD, and Oracle Linux. The SystemTap[2] project for Linux provides a DTrace equivalent and can also be used. Supporting other dynamic tracing utilities is theoretically possible by changing the definitions for the macros in `src/include/utils/probes.h`.

27.4.1. Compiling for Dynamic Tracing

By default, probes are not available, so you will need to explicitly tell the configure script to make the probes available in PostgreSQL. To include DTrace support specify `--enable-dtrace` to configure. See Section 15.4 for further information.

27.4.2. Built-in Probes

A number of standard probes are provided in the source code, as shown in Table 27-18; Table 27-19 shows the types used in the probes. More probes can certainly be added to enhance PostgreSQL's observability.

Table 27-18. Built-in DTrace Probes

Name	Parameters	Description
transaction-start	(LocalTransactionId)	Probe that fires at the start of a new transaction. arg0 is the transaction ID.
transaction-commit	(LocalTransactionId)	Probe that fires when a transaction completes successfully. arg0 is the transaction ID.
transaction-abort	(LocalTransactionId)	Probe that fires when a transaction completes unsuccessfully. arg0 is the transaction ID.
query-start	(const char *)	Probe that fires when the processing of a query is started. arg0 is the query string.

1. https://en.wikipedia.org/wiki/DTrace
2. https://sourceware.org/systemtap/

Name	Parameters	Description
query-done	(const char *)	Probe that fires when the processing of a query is complete. arg0 is the query string.
query-parse-start	(const char *)	Probe that fires when the parsing of a query is started. arg0 is the query string.
query-parse-done	(const char *)	Probe that fires when the parsing of a query is complete. arg0 is the query string.
query-rewrite-start	(const char *)	Probe that fires when the rewriting of a query is started. arg0 is the query string.
query-rewrite-done	(const char *)	Probe that fires when the rewriting of a query is complete. arg0 is the query string.
query-plan-start	()	Probe that fires when the planning of a query is started.
query-plan-done	()	Probe that fires when the planning of a query is complete.
query-execute-start	()	Probe that fires when the execution of a query is started.
query-execute-done	()	Probe that fires when the execution of a query is complete.
statement-status	(const char *)	Probe that fires anytime the server process updates its `pg_stat_activity.status`. arg0 is the new status string.
checkpoint-start	(int)	Probe that fires when a checkpoint is started. arg0 holds the bitwise flags used to distinguish different checkpoint types, such as shutdown, immediate or force.
checkpoint-done	(int, int, int, int, int)	Probe that fires when a checkpoint is complete. (The probes listed next fire in sequence during checkpoint processing.) arg0 is the number of buffers written. arg1 is the total number of buffers. arg2, arg3 and arg4 contain the number of xlog file(s) added, removed and recycled respectively.

Name	Parameters	Description
clog-checkpoint-start	(bool)	Probe that fires when the CLOG portion of a checkpoint is started. arg0 is true for normal checkpoint, false for shutdown checkpoint.
clog-checkpoint-done	(bool)	Probe that fires when the CLOG portion of a checkpoint is complete. arg0 has the same meaning as for clog-checkpoint-start.
subtrans-checkpoint-start	(bool)	Probe that fires when the SUBTRANS portion of a checkpoint is started. arg0 is true for normal checkpoint, false for shutdown checkpoint.
subtrans-checkpoint-done	(bool)	Probe that fires when the SUBTRANS portion of a checkpoint is complete. arg0 has the same meaning as for subtrans-checkpoint-start.
multixact-checkpoint-start	(bool)	Probe that fires when the MultiXact portion of a checkpoint is started. arg0 is true for normal checkpoint, false for shutdown checkpoint.
multixact-checkpoint-done	(bool)	Probe that fires when the MultiXact portion of a checkpoint is complete. arg0 has the same meaning as for multixact-checkpoint-start.
buffer-checkpoint-start	(int)	Probe that fires when the buffer-writing portion of a checkpoint is started. arg0 holds the bitwise flags used to distinguish different checkpoint types, such as shutdown, immediate or force.
buffer-sync-start	(int, int)	Probe that fires when we begin to write dirty buffers during checkpoint (after identifying which buffers must be written). arg0 is the total number of buffers. arg1 is the number that are currently dirty and need to be written.

Name	Parameters	Description
buffer-sync-written	(int)	Probe that fires after each buffer is written during checkpoint. arg0 is the ID number of the buffer.
buffer-sync-done	(int, int, int)	Probe that fires when all dirty buffers have been written. arg0 is the total number of buffers. arg1 is the number of buffers actually written by the checkpoint process. arg2 is the number that were expected to be written (arg1 of buffer-sync-start); any difference reflects other processes flushing buffers during the checkpoint.
buffer-checkpoint-sync-start	()	Probe that fires after dirty buffers have been written to the kernel, and before starting to issue fsync requests.
buffer-checkpoint-done	()	Probe that fires when syncing of buffers to disk is complete.
twophase-checkpoint-start	()	Probe that fires when the two-phase portion of a checkpoint is started.
twophase-checkpoint-done	()	Probe that fires when the two-phase portion of a checkpoint is complete.
buffer-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool)	Probe that fires when a buffer read is started. arg0 and arg1 contain the fork and block numbers of the page (but arg1 will be -1 if this is a relation extension request). arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or InvalidBackendId (-1) for a shared buffer. arg6 is true for a relation extension request, false for normal read.

Name	Parameters	Description
buffer-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, bool, bool)	Probe that fires when a buffer read is complete. arg0 and arg1 contain the fork and block numbers of the page (if this is a relation extension request, arg1 now contains the block number of the newly added block). arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or InvalidBackendId (-1) for a shared buffer. arg6 is true for a relation extension request, false for normal read. arg7 is true if the buffer was found in the pool, false if not.
buffer-flush-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Probe that fires before issuing any write request for a shared buffer. arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation.
buffer-flush-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Probe that fires when a write request is complete. (Note that this just reflects the time to pass the data to the kernel; it's typically not actually been written to disk yet.) The arguments are the same as for buffer-flush-start.

Name	Parameters	Description
buffer-write-dirty-start	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Probe that fires when a server process begins to write a dirty buffer. (If this happens often, it implies that shared_buffers is too small or the bgwriter control parameters need adjustment.) arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation.
buffer-write-dirty-done	(ForkNumber, BlockNumber, Oid, Oid, Oid)	Probe that fires when a dirty-buffer write is complete. The arguments are the same as for buffer-write-dirty-start.
wal-buffer-write-dirty-start	()	Probe that fires when a server process begins to write a dirty WAL buffer because no more WAL buffer space is available. (If this happens often, it implies that wal_buffers is too small.)
wal-buffer-write-dirty-done	()	Probe that fires when a dirty WAL buffer write is complete.
xlog-insert	(unsigned char, unsigned char)	Probe that fires when a WAL record is inserted. arg0 is the resource manager (rmid) for the record. arg1 contains the info flags.
xlog-switch	()	Probe that fires when a WAL segment switch is requested.
smgr-md-read-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	Probe that fires when beginning to read a block from a relation. arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or InvalidBackendId (-1) for a shared buffer.

Name	Parameters	Description
smgr-md-read-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	Probe that fires when a block read is complete. arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or InvalidBackendId (-1) for a shared buffer. arg6 is the number of bytes actually read, while arg7 is the number requested (if these are different it indicates trouble).
smgr-md-write-start	(ForkNumber, BlockNumber, Oid, Oid, Oid, int)	Probe that fires when beginning to write a block to a relation. arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or InvalidBackendId (-1) for a shared buffer.
smgr-md-write-done	(ForkNumber, BlockNumber, Oid, Oid, Oid, int, int, int)	Probe that fires when a block write is complete. arg0 and arg1 contain the fork and block numbers of the page. arg2, arg3, and arg4 contain the tablespace, database, and relation OIDs identifying the relation. arg5 is the ID of the backend which created the temporary relation for a local buffer, or InvalidBackendId (-1) for a shared buffer. arg6 is the number of bytes actually written, while arg7 is the number requested (if these are different it indicates trouble).

Name	Parameters	Description
sort-start	(int, bool, int, int, bool)	Probe that fires when a sort operation is started. arg0 indicates heap, index or datum sort. arg1 is true for unique-value enforcement. arg2 is the number of key columns. arg3 is the number of kilobytes of work memory allowed. arg4 is true if random access to the sort result is required.
sort-done	(bool, long)	Probe that fires when a sort is complete. arg0 is true for external sort, false for internal sort. arg1 is the number of disk blocks used for an external sort, or kilobytes of memory used for an internal sort.
lwlock-acquire	(char *, int, LWLockMode)	Probe that fires when an LWLock has been acquired. arg0 is the LWLock's tranche. arg1 is the LWLock's offset within its tranche. arg2 is the requested lock mode, either exclusive or shared.
lwlock-release	(char *, int)	Probe that fires when an LWLock has been released (but note that any released waiters have not yet been awakened). arg0 is the LWLock's tranche. arg1 is the LWLock's offset within its tranche.
lwlock-wait-start	(char *, int, LWLockMode)	Probe that fires when an LWLock was not immediately available and a server process has begun to wait for the lock to become available. arg0 is the LWLock's tranche. arg1 is the LWLock's offset within its tranche. arg2 is the requested lock mode, either exclusive or shared.

Name	Parameters	Description
lwlock-wait-done	(char *, int, LWLockMode)	Probe that fires when a server process has been released from its wait for an LWLock (it does not actually have the lock yet). arg0 is the LWLock's tranche. arg1 is the LWLock's offset within its tranche. arg2 is the requested lock mode, either exclusive or shared.
lwlock-condacquire	(char *, int, LWLockMode)	Probe that fires when an LWLock was successfully acquired when the caller specified no waiting. arg0 is the LWLock's tranche. arg1 is the LWLock's offset within its tranche. arg2 is the requested lock mode, either exclusive or shared.
lwlock-condacquire-fail	(char *, int, LWLockMode)	Probe that fires when an LWLock was not successfully acquired when the caller specified no waiting. arg0 is the LWLock's tranche. arg1 is the LWLock's offset within its tranche. arg2 is the requested lock mode, either exclusive or shared.
lock-wait-start	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	Probe that fires when a request for a heavyweight lock (lmgr lock) has begun to wait because the lock is not available. arg0 through arg3 are the tag fields identifying the object being locked. arg4 indicates the type of object being locked. arg5 indicates the lock type being requested.
lock-wait-done	(unsigned int, unsigned int, unsigned int, unsigned int, unsigned int, LOCKMODE)	Probe that fires when a request for a heavyweight lock (lmgr lock) has finished waiting (i.e., has acquired the lock). The arguments are the same as for lock-wait-start.
deadlock-found	()	Probe that fires when a deadlock is found by the deadlock detector.

Table 27-19. Defined Types Used in Probe Parameters

Type	Definition
LocalTransactionId	unsigned int
LWLockMode	int
LOCKMODE	int
BlockNumber	unsigned int
Oid	unsigned int
ForkNumber	int
bool	char

27.4.3. Using Probes

The example below shows a DTrace script for analyzing transaction counts in the system, as an alternative to snapshotting `pg_stat_database` before and after a performance test:

```
#!/usr/sbin/dtrace -qs

postgresql$1:::transaction-start
{
      @start["Start"] = count();
      self->ts  = timestamp;
}

postgresql$1:::transaction-abort
{
      @abort["Abort"] = count();
}

postgresql$1:::transaction-commit
/self->ts/
{
      @commit["Commit"] = count();
      @time["Total time (ns)"] = sum(timestamp - self->ts);
      self->ts=0;
}
```

When executed, the example D script gives output such as:

```
# ./txn_count.d `pgrep -n postgres` or ./txn_count.d <PID>
^C

Start                                          71
Commit                                         70
Total time (ns)                        2312105013
```

Note: SystemTap uses a different notation for trace scripts than DTrace does, even though the underlying trace points are compatible. One point worth noting is that at this writing, SystemTap scripts must reference probe names using double underscores in place of hyphens. This is expected to be fixed in future SystemTap releases.

You should remember that DTrace scripts need to be carefully written and debugged, otherwise the trace information collected might be meaningless. In most cases where problems are found it is the instrumentation that is at fault, not the underlying system. When discussing information found using dynamic tracing, be sure to enclose the script used to allow that too to be checked and discussed.

27.4.4. Defining New Probes

New probes can be defined within the code wherever the developer desires, though this will require a recompilation. Below are the steps for inserting new probes:

1. Decide on probe names and data to be made available through the probes
2. Add the probe definitions to `src/backend/utils/probes.d`
3. Include `pg_trace.h` if it is not already present in the module(s) containing the probe points, and insert `TRACE_POSTGRESQL` probe macros at the desired locations in the source code
4. Recompile and verify that the new probes are available

Example: Here is an example of how you would add a probe to trace all new transactions by transaction ID.

1. Decide that the probe will be named `transaction-start` and requires a parameter of type LocalTransactionId
2. Add the probe definition to `src/backend/utils/probes.d`:

   ```
   probe transaction__start(LocalTransactionId);
   ```

   Note the use of the double underline in the probe name. In a DTrace script using the probe, the double underline needs to be replaced with a hyphen, so `transaction-start` is the name to document for users.
3. At compile time, `transaction__start` is converted to a macro called `TRACE_POSTGRESQL_TRANSACTION_START` (notice the underscores are single here), which is available by including `pg_trace.h`. Add the macro call to the appropriate location in the source code. In this case, it looks like the following:

   ```
   TRACE_POSTGRESQL_TRANSACTION_START(vxid.localTransactionId);
   ```
4. After recompiling and running the new binary, check that your newly added probe is available by executing the following DTrace command. You should see similar output:

   ```
   # dtrace -ln transaction-start
      ID    PROVIDER          MODULE           FUNCTION NAME
   18705 postgresql49878     postgres     StartTransactionCommand transaction-start
   18755 postgresql49877     postgres     StartTransactionCommand transaction-start
   18805 postgresql49876     postgres     StartTransactionCommand transaction-start
   ```

```
  18855 postgresql49875     postgres     StartTransactionCommand transaction-start
  18986 postgresql49873     postgres     StartTransactionCommand transaction-start
```

There are a few things to be careful about when adding trace macros to the C code:

- You should take care that the data types specified for a probe's parameters match the data types of the variables used in the macro. Otherwise, you will get compilation errors.
- On most platforms, if PostgreSQL is built with `--enable-dtrace`, the arguments to a trace macro will be evaluated whenever control passes through the macro, *even if no tracing is being done*. This is usually not worth worrying about if you are just reporting the values of a few local variables. But beware of putting expensive function calls into the arguments. If you need to do that, consider protecting the macro with a check to see if the trace is actually enabled:

  ```
  if (TRACE_POSTGRESQL_TRANSACTION_START_ENABLED())
      TRACE_POSTGRESQL_TRANSACTION_START(some_function(...));
  ```

  Each trace macro has a corresponding `ENABLED` macro.

Chapter 28. Monitoring Disk Usage

This chapter discusses how to monitor the disk usage of a PostgreSQL database system.

28.1. Determining Disk Usage

Each table has a primary heap disk file where most of the data is stored. If the table has any columns with potentially-wide values, there also might be a TOAST file associated with the table, which is used to store values too wide to fit comfortably in the main table (see Section 63.2). There will be one valid index on the TOAST table, if present. There also might be indexes associated with the base table. Each table and index is stored in a separate disk file — possibly more than one file, if the file would exceed one gigabyte. Naming conventions for these files are described in Section 63.1.

You can monitor disk space in three ways: using the SQL functions listed in Table 9-73, using the oid2name module, or using manual inspection of the system catalogs. The SQL functions are the easiest to use and are generally recommended. The remainder of this section shows how to do it by inspection of the system catalogs.

Using psql on a recently vacuumed or analyzed database, you can issue queries to see the disk usage of any table:

```
SELECT pg_relation_filepath(oid), relpages FROM pg_class WHERE relname = 'customer';

 pg_relation_filepath | relpages
----------------------+----------
 base/16384/16806     |       60
(1 row)
```

Each page is typically 8 kilobytes. (Remember, `relpages` is only updated by `VACUUM`, `ANALYZE`, and a few DDL commands such as `CREATE INDEX`.) The file path name is of interest if you want to examine the table's disk file directly.

To show the space used by TOAST tables, use a query like the following:

```
SELECT relname, relpages
FROM pg_class,
     (SELECT reltoastrelid
      FROM pg_class
      WHERE relname = 'customer') AS ss
WHERE oid = ss.reltoastrelid OR
      oid = (SELECT indexrelid
             FROM pg_index
             WHERE indrelid = ss.reltoastrelid)
ORDER BY relname;

       relname        | relpages
----------------------+----------
 pg_toast_16806       |        0
 pg_toast_16806_index |        1
```

You can easily display index sizes, too:

```
SELECT c2.relname, c2.relpages
FROM pg_class c, pg_class c2, pg_index i
WHERE c.relname = 'customer' AND
      c.oid = i.indrelid AND
      c2.oid = i.indexrelid
ORDER BY c2.relname;

       relname        | relpages
----------------------+----------
 customer_id_indexdex |       26
```

It is easy to find your largest tables and indexes using this information:

```
SELECT relname, relpages
FROM pg_class
ORDER BY relpages DESC;

       relname        | relpages
----------------------+----------
 bigtable             |     3290
 customer             |     3144
```

28.2. Disk Full Failure

The most important disk monitoring task of a database administrator is to make sure the disk doesn't become full. A filled data disk will not result in data corruption, but it might prevent useful activity from occurring. If the disk holding the WAL files grows full, database server panic and consequent shutdown might occur.

If you cannot free up additional space on the disk by deleting other things, you can move some of the database files to other file systems by making use of tablespaces. See Section 21.6 for more information about that.

> **Tip:** Some file systems perform badly when they are almost full, so do not wait until the disk is completely full to take action.

If your system supports per-user disk quotas, then the database will naturally be subject to whatever quota is placed on the user the server runs as. Exceeding the quota will have the same bad effects as running out of disk space entirely.

Chapter 29. Reliability and the Write-Ahead Log

This chapter explains how the Write-Ahead Log is used to obtain efficient, reliable operation.

29.1. Reliability

Reliability is an important property of any serious database system, and PostgreSQL does everything possible to guarantee reliable operation. One aspect of reliable operation is that all data recorded by a committed transaction should be stored in a nonvolatile area that is safe from power loss, operating system failure, and hardware failure (except failure of the nonvolatile area itself, of course). Successfully writing the data to the computer's permanent storage (disk drive or equivalent) ordinarily meets this requirement. In fact, even if a computer is fatally damaged, if the disk drives survive they can be moved to another computer with similar hardware and all committed transactions will remain intact.

While forcing data to the disk platters periodically might seem like a simple operation, it is not. Because disk drives are dramatically slower than main memory and CPUs, several layers of caching exist between the computer's main memory and the disk platters. First, there is the operating system's buffer cache, which caches frequently requested disk blocks and combines disk writes. Fortunately, all operating systems give applications a way to force writes from the buffer cache to disk, and PostgreSQL uses those features. (See the wal_sync_method parameter to adjust how this is done.)

Next, there might be a cache in the disk drive controller; this is particularly common on RAID controller cards. Some of these caches are *write-through*, meaning writes are sent to the drive as soon as they arrive. Others are *write-back*, meaning data is sent to the drive at some later time. Such caches can be a reliability hazard because the memory in the disk controller cache is volatile, and will lose its contents in a power failure. Better controller cards have *battery-backup units* (BBUs), meaning the card has a battery that maintains power to the cache in case of system power loss. After power is restored the data will be written to the disk drives.

And finally, most disk drives have caches. Some are write-through while some are write-back, and the same concerns about data loss exist for write-back drive caches as for disk controller caches. Consumer-grade IDE and SATA drives are particularly likely to have write-back caches that will not survive a power failure. Many solid-state drives (SSD) also have volatile write-back caches.

These caches can typically be disabled; however, the method for doing this varies by operating system and drive type:

- On Linux, IDE and SATA drives can be queried using `hdparm -I`; write caching is enabled if there is a `*` next to `Write cache`. `hdparm -W 0` can be used to turn off write caching. SCSI drives can be queried using sdparm[1]. Use `sdparm --get=WCE` to check whether the write cache is enabled and `sdparm --clear=WCE` to disable it.
- On FreeBSD, IDE drives can be queried using `atacontrol` and write caching turned off using `hw.ata.wc=0` in `/boot/loader.conf`; SCSI drives can be queried using `camcontrol identify`, and the write cache both queried and changed using `sdparm` when available.
- On Solaris, the disk write cache is controlled by `format -e`. (The Solaris ZFS file system is safe with disk write-cache enabled because it issues its own disk cache flush commands.)

1. http://sg.danny.cz/sg/sdparm.html

- On Windows, if `wal_sync_method` is `open_datasync` (the default), write caching can be disabled by unchecking `My Computer\Open\`*`disk drive`*`\Properties\Hardware\Properties\Policies\Enable write caching on the disk`. Alternatively, set `wal_sync_method` to `fsync` or `fsync_writethrough`, which prevent write caching.
- On OS X, write caching can be prevented by setting `wal_sync_method` to `fsync_writethrough`.

Recent SATA drives (those following ATAPI-6 or later) offer a drive cache flush command (`FLUSH CACHE EXT`), while SCSI drives have long supported a similar command `SYNCHRONIZE CACHE`. These commands are not directly accessible to PostgreSQL, but some file systems (e.g., ZFS, ext4) can use them to flush data to the platters on write-back-enabled drives. Unfortunately, such file systems behave suboptimally when combined with battery-backup unit (BBU) disk controllers. In such setups, the synchronize command forces all data from the controller cache to the disks, eliminating much of the benefit of the BBU. You can run the pg_test_fsync program to see if you are affected. If you are affected, the performance benefits of the BBU can be regained by turning off write barriers in the file system or reconfiguring the disk controller, if that is an option. If write barriers are turned off, make sure the battery remains functional; a faulty battery can potentially lead to data loss. Hopefully file system and disk controller designers will eventually address this suboptimal behavior.

When the operating system sends a write request to the storage hardware, there is little it can do to make sure the data has arrived at a truly non-volatile storage area. Rather, it is the administrator's responsibility to make certain that all storage components ensure integrity for both data and file-system metadata. Avoid disk controllers that have non-battery-backed write caches. At the drive level, disable write-back caching if the drive cannot guarantee the data will be written before shutdown. If you use SSDs, be aware that many of these do not honor cache flush commands by default. You can test for reliable I/O subsystem behavior using `diskchecker.pl`[2].

Another risk of data loss is posed by the disk platter write operations themselves. Disk platters are divided into sectors, commonly 512 bytes each. Every physical read or write operation processes a whole sector. When a write request arrives at the drive, it might be for some multiple of 512 bytes (PostgreSQL typically writes 8192 bytes, or 16 sectors, at a time), and the process of writing could fail due to power loss at any time, meaning some of the 512-byte sectors were written while others were not. To guard against such failures, PostgreSQL periodically writes full page images to permanent WAL storage *before* modifying the actual page on disk. By doing this, during crash recovery PostgreSQL can restore partially-written pages from WAL. If you have file-system software that prevents partial page writes (e.g., ZFS), you can turn off this page imaging by turning off the full_page_writes parameter. Battery-Backed Unit (BBU) disk controllers do not prevent partial page writes unless they guarantee that data is written to the BBU as full (8kB) pages.

PostgreSQL also protects against some kinds of data corruption on storage devices that may occur because of hardware errors or media failure over time, such as reading/writing garbage data.

- Each individual record in a WAL file is protected by a CRC-32 (32-bit) check that allows us to tell if record contents are correct. The CRC value is set when we write each WAL record and checked during crash recovery, archive recovery and replication.
- Data pages are not currently checksummed by default, though full page images recorded in WAL records will be protected; see initdb for details about enabling data page checksums.

2. http://brad.livejournal.com/2116715.html

- Internal data structures such as `pg_clog`, `pg_subtrans`, `pg_multixact`, `pg_serial`, `pg_notify`, `pg_stat`, `pg_snapshots` are not directly checksummed, nor are pages protected by full page writes. However, where such data structures are persistent, WAL records are written that allow recent changes to be accurately rebuilt at crash recovery and those WAL records are protected as discussed above.
- Individual state files in `pg_twophase` are protected by CRC-32.
- Temporary data files used in larger SQL queries for sorts, materializations and intermediate results are not currently checksummed, nor will WAL records be written for changes to those files.

PostgreSQL does not protect against correctable memory errors and it is assumed you will operate using RAM that uses industry standard Error Correcting Codes (ECC) or better protection.

29.2. Write-Ahead Logging (WAL)

Write-Ahead Logging (WAL) is a standard method for ensuring data integrity. A detailed description can be found in most (if not all) books about transaction processing. Briefly, WAL's central concept is that changes to data files (where tables and indexes reside) must be written only after those changes have been logged, that is, after log records describing the changes have been flushed to permanent storage. If we follow this procedure, we do not need to flush data pages to disk on every transaction commit, because we know that in the event of a crash we will be able to recover the database using the log: any changes that have not been applied to the data pages can be redone from the log records. (This is roll-forward recovery, also known as REDO.)

> **Tip:** Because WAL restores database file contents after a crash, journaled file systems are not necessary for reliable storage of the data files or WAL files. In fact, journaling overhead can reduce performance, especially if journaling causes file system *data* to be flushed to disk. Fortunately, data flushing during journaling can often be disabled with a file system mount option, e.g., `data=writeback` on a Linux ext3 file system. Journaled file systems do improve boot speed after a crash.

Using WAL results in a significantly reduced number of disk writes, because only the log file needs to be flushed to disk to guarantee that a transaction is committed, rather than every data file changed by the transaction. The log file is written sequentially, and so the cost of syncing the log is much less than the cost of flushing the data pages. This is especially true for servers handling many small transactions touching different parts of the data store. Furthermore, when the server is processing many small concurrent transactions, one `fsync` of the log file may suffice to commit many transactions.

WAL also makes it possible to support on-line backup and point-in-time recovery, as described in Section 24.3. By archiving the WAL data we can support reverting to any time instant covered by the available WAL data: we simply install a prior physical backup of the database, and replay the WAL log just as far as the desired time. What's more, the physical backup doesn't have to be an instantaneous snapshot of the database state — if it is made over some period of time, then replaying the WAL log for that period will fix any internal inconsistencies.

29.3. Asynchronous Commit

Asynchronous commit is an option that allows transactions to complete more quickly, at the cost that the most recent transactions may be lost if the database should crash. In many applications this is an acceptable trade-off.

As described in the previous section, transaction commit is normally *synchronous*: the server waits for the transaction's WAL records to be flushed to permanent storage before returning a success indication to the client. The client is therefore guaranteed that a transaction reported to be committed will be preserved, even in the event of a server crash immediately after. However, for short transactions this delay is a major component of the total transaction time. Selecting asynchronous commit mode means that the server returns success as soon as the transaction is logically completed, before the WAL records it generated have actually made their way to disk. This can provide a significant boost in throughput for small transactions.

Asynchronous commit introduces the risk of data loss. There is a short time window between the report of transaction completion to the client and the time that the transaction is truly committed (that is, it is guaranteed not to be lost if the server crashes). Thus asynchronous commit should not be used if the client will take external actions relying on the assumption that the transaction will be remembered. As an example, a bank would certainly not use asynchronous commit for a transaction recording an ATM's dispensing of cash. But in many scenarios, such as event logging, there is no need for a strong guarantee of this kind.

The risk that is taken by using asynchronous commit is of data loss, not data corruption. If the database should crash, it will recover by replaying WAL up to the last record that was flushed. The database will therefore be restored to a self-consistent state, but any transactions that were not yet flushed to disk will not be reflected in that state. The net effect is therefore loss of the last few transactions. Because the transactions are replayed in commit order, no inconsistency can be introduced — for example, if transaction B made changes relying on the effects of a previous transaction A, it is not possible for A's effects to be lost while B's effects are preserved.

The user can select the commit mode of each transaction, so that it is possible to have both synchronous and asynchronous commit transactions running concurrently. This allows flexible trade-offs between performance and certainty of transaction durability. The commit mode is controlled by the user-settable parameter synchronous_commit, which can be changed in any of the ways that a configuration parameter can be set. The mode used for any one transaction depends on the value of `synchronous_commit` when transaction commit begins.

Certain utility commands, for instance `DROP TABLE`, are forced to commit synchronously regardless of the setting of `synchronous_commit`. This is to ensure consistency between the server's file system and the logical state of the database. The commands supporting two-phase commit, such as `PREPARE TRANSACTION`, are also always synchronous.

If the database crashes during the risk window between an asynchronous commit and the writing of the transaction's WAL records, then changes made during that transaction *will* be lost. The duration of the risk window is limited because a background process (the "WAL writer") flushes unwritten WAL records to disk every wal_writer_delay milliseconds. The actual maximum duration of the risk window is three times `wal_writer_delay` because the WAL writer is designed to favor writing whole pages at a time during busy periods.

> **Caution**
>
> An immediate-mode shutdown is equivalent to a server crash, and will therefore cause loss of any unflushed asynchronous commits.

Asynchronous commit provides behavior different from setting fsync = off. `fsync` is a server-wide setting that will alter the behavior of all transactions. It disables all logic within PostgreSQL that attempts to synchronize writes to different portions of the database, and therefore a system crash (that is, a hardware or operating system crash, not a failure of PostgreSQL itself) could result in arbitrarily bad corruption of the database state. In many scenarios, asynchronous commit provides most of the performance improvement that could be obtained by turning off `fsync`, but without the risk of data corruption.

commit_delay also sounds very similar to asynchronous commit, but it is actually a synchronous commit method (in fact, `commit_delay` is ignored during an asynchronous commit). `commit_delay` causes a delay just before a transaction flushes WAL to disk, in the hope that a single flush executed by one such transaction can also serve other transactions committing at about the same time. The setting can be thought of as a way of increasing the time window in which transactions can join a group about to participate in a single flush, to amortize the cost of the flush among multiple transactions.

29.4. WAL Configuration

There are several WAL-related configuration parameters that affect database performance. This section explains their use. Consult Chapter 18 for general information about setting server configuration parameters.

Checkpoints are points in the sequence of transactions at which it is guaranteed that the heap and index data files have been updated with all information written before that checkpoint. At checkpoint time, all dirty data pages are flushed to disk and a special checkpoint record is written to the log file. (The change records were previously flushed to the WAL files.) In the event of a crash, the crash recovery procedure looks at the latest checkpoint record to determine the point in the log (known as the redo record) from which it should start the REDO operation. Any changes made to data files before that point are guaranteed to be already on disk. Hence, after a checkpoint, log segments preceding the one containing the redo record are no longer needed and can be recycled or removed. (When WAL archiving is being done, the log segments must be archived before being recycled or removed.)

The checkpoint requirement of flushing all dirty data pages to disk can cause a significant I/O load. For this reason, checkpoint activity is throttled so that I/O begins at checkpoint start and completes before the next checkpoint is due to start; this minimizes performance degradation during checkpoints.

The server's checkpointer process automatically performs a checkpoint every so often. A checkpoint is begun every checkpoint_timeout seconds, or if max_wal_size is about to be exceeded, whichever comes first. The default settings are 5 minutes and 1 GB, respectively. If no WAL has been written since the previous checkpoint, new checkpoints will be skipped even if `checkpoint_timeout` has passed. (If WAL archiving is being used and you want to put a lower limit on how often files are archived in order to bound potential data loss, you should adjust the archive_timeout parameter rather than the checkpoint parameters.) It is also possible to force a checkpoint by using the SQL command `CHECKPOINT`.

Reducing `checkpoint_timeout` and/or `max_wal_size` causes checkpoints to occur more often. This allows faster after-crash recovery, since less work will need to be redone. However, one must balance

this against the increased cost of flushing dirty data pages more often. If full_page_writes is set (as is the default), there is another factor to consider. To ensure data page consistency, the first modification of a data page after each checkpoint results in logging the entire page content. In that case, a smaller checkpoint interval increases the volume of output to the WAL log, partially negating the goal of using a smaller interval, and in any case causing more disk I/O.

Checkpoints are fairly expensive, first because they require writing out all currently dirty buffers, and second because they result in extra subsequent WAL traffic as discussed above. It is therefore wise to set the checkpointing parameters high enough so that checkpoints don't happen too often. As a simple sanity check on your checkpointing parameters, you can set the checkpoint_warning parameter. If checkpoints happen closer together than `checkpoint_warning` seconds, a message will be output to the server log recommending increasing `max_wal_size`. Occasional appearance of such a message is not cause for alarm, but if it appears often then the checkpoint control parameters should be increased. Bulk operations such as large `COPY` transfers might cause a number of such warnings to appear if you have not set `max_wal_size` high enough.

To avoid flooding the I/O system with a burst of page writes, writing dirty buffers during a checkpoint is spread over a period of time. That period is controlled by checkpoint_completion_target, which is given as a fraction of the checkpoint interval. The I/O rate is adjusted so that the checkpoint finishes when the given fraction of `checkpoint_timeout` seconds have elapsed, or before `max_wal_size` is exceeded, whichever is sooner. With the default value of 0.5, PostgreSQL can be expected to complete each checkpoint in about half the time before the next checkpoint starts. On a system that's very close to maximum I/O throughput during normal operation, you might want to increase `checkpoint_completion_target` to reduce the I/O load from checkpoints. The disadvantage of this is that prolonging checkpoints affects recovery time, because more WAL segments will need to be kept around for possible use in recovery. Although `checkpoint_completion_target` can be set as high as 1.0, it is best to keep it less than that (perhaps 0.9 at most) since checkpoints include some other activities besides writing dirty buffers. A setting of 1.0 is quite likely to result in checkpoints not being completed on time, which would result in performance loss due to unexpected variation in the number of WAL segments needed.

The number of WAL segment files in `pg_xlog` directory depends on `min_wal_size`, `max_wal_size` and the amount of WAL generated in previous checkpoint cycles. When old log segment files are no longer needed, they are removed or recycled (that is, renamed to become future segments in the numbered sequence). If, due to a short-term peak of log output rate, `max_wal_size` is exceeded, the unneeded segment files will be removed until the system gets back under this limit. Below that limit, the system recycles enough WAL files to cover the estimated need until the next checkpoint, and removes the rest. The estimate is based on a moving average of the number of WAL files used in previous checkpoint cycles. The moving average is increased immediately if the actual usage exceeds the estimate, so it accommodates peak usage rather than average usage to some extent. `min_wal_size` puts a minimum on the amount of WAL files recycled for future usage; that much WAL is always recycled for future use, even if the system is idle and the WAL usage estimate suggests that little WAL is needed.

Independently of `max_wal_size`, wal_keep_segments + 1 most recent WAL files are kept at all times. Also, if WAL archiving is used, old segments can not be removed or recycled until they are archived. If WAL archiving cannot keep up with the pace that WAL is generated, or if `archive_command` fails repeatedly, old WAL files will accumulate in `pg_xlog` until the situation is resolved. A slow or failed standby server that uses a replication slot will have the same effect (see Section 25.2.6).

In archive recovery or standby mode, the server periodically performs *restartpoints*, which are similar to checkpoints in normal operation: the server forces all its state to disk, updates the `pg_control` file to

indicate that the already-processed WAL data need not be scanned again, and then recycles any old log segment files in the `pg_xlog` directory. Restartpoints can't be performed more frequently than checkpoints in the master because restartpoints can only be performed at checkpoint records. A restartpoint is triggered when a checkpoint record is reached if at least `checkpoint_timeout` seconds have passed since the last restartpoint, or if WAL size is about to exceed `max_wal_size`. However, because of limitations on when a restartpoint can be performed, `max_wal_size` is often exceeded during recovery, by up to one checkpoint cycle's worth of WAL. (`max_wal_size` is never a hard limit anyway, so you should always leave plenty of headroom to avoid running out of disk space.)

There are two commonly used internal WAL functions: `XLogInsertRecord` and `XLogFlush`. `XLogInsertRecord` is used to place a new record into the WAL buffers in shared memory. If there is no space for the new record, `XLogInsertRecord` will have to write (move to kernel cache) a few filled WAL buffers. This is undesirable because `XLogInsertRecord` is used on every database low level modification (for example, row insertion) at a time when an exclusive lock is held on affected data pages, so the operation needs to be as fast as possible. What is worse, writing WAL buffers might also force the creation of a new log segment, which takes even more time. Normally, WAL buffers should be written and flushed by an `XLogFlush` request, which is made, for the most part, at transaction commit time to ensure that transaction records are flushed to permanent storage. On systems with high log output, `XLogFlush` requests might not occur often enough to prevent `XLogInsertRecord` from having to do writes. On such systems one should increase the number of WAL buffers by modifying the wal_buffers parameter. When full_page_writes is set and the system is very busy, setting `wal_buffers` higher will help smooth response times during the period immediately following each checkpoint.

The commit_delay parameter defines for how many microseconds a group commit leader process will sleep after acquiring a lock within `XLogFlush`, while group commit followers queue up behind the leader. This delay allows other server processes to add their commit records to the WAL buffers so that all of them will be flushed by the leader's eventual sync operation. No sleep will occur if fsync is not enabled, or if fewer than commit_siblings other sessions are currently in active transactions; this avoids sleeping when it's unlikely that any other session will commit soon. Note that on some platforms, the resolution of a sleep request is ten milliseconds, so that any nonzero `commit_delay` setting between 1 and 10000 microseconds would have the same effect. Note also that on some platforms, sleep operations may take slightly longer than requested by the parameter.

Since the purpose of `commit_delay` is to allow the cost of each flush operation to be amortized across concurrently committing transactions (potentially at the expense of transaction latency), it is necessary to quantify that cost before the setting can be chosen intelligently. The higher that cost is, the more effective `commit_delay` is expected to be in increasing transaction throughput, up to a point. The pg_test_fsync program can be used to measure the average time in microseconds that a single WAL flush operation takes. A value of half of the average time the program reports it takes to flush after a single 8kB write operation is often the most effective setting for `commit_delay`, so this value is recommended as the starting point to use when optimizing for a particular workload. While tuning `commit_delay` is particularly useful when the WAL log is stored on high-latency rotating disks, benefits can be significant even on storage media with very fast sync times, such as solid-state drives or RAID arrays with a battery-backed write cache; but this should definitely be tested against a representative workload. Higher values of `commit_siblings` should be used in such cases, whereas smaller `commit_siblings` values are often helpful on higher latency media. Note that it is quite possible that a setting of `commit_delay` that is too high can increase transaction latency by so much that total transaction throughput suffers.

When `commit_delay` is set to zero (the default), it is still possible for a form of group commit to occur, but each group will consist only of sessions that reach the point where they need to flush their com-

mit records during the window in which the previous flush operation (if any) is occurring. At higher client counts a "gangway effect" tends to occur, so that the effects of group commit become significant even when `commit_delay` is zero, and thus explicitly setting `commit_delay` tends to help less. Setting `commit_delay` can only help when (1) there are some concurrently committing transactions, and (2) throughput is limited to some degree by commit rate; but with high rotational latency this setting can be effective in increasing transaction throughput with as few as two clients (that is, a single committing client with one sibling transaction).

The wal_sync_method parameter determines how PostgreSQL will ask the kernel to force WAL updates out to disk. All the options should be the same in terms of reliability, with the exception of `fsync_writethrough`, which can sometimes force a flush of the disk cache even when other options do not do so. However, it's quite platform-specific which one will be the fastest. You can test the speeds of different options using the pg_test_fsync program. Note that this parameter is irrelevant if `fsync` has been turned off.

Enabling the wal_debug configuration parameter (provided that PostgreSQL has been compiled with support for it) will result in each `XLogInsertRecord` and `XLogFlush` WAL call being logged to the server log. This option might be replaced by a more general mechanism in the future.

29.5. WAL Internals

WAL is automatically enabled; no action is required from the administrator except ensuring that the disk-space requirements for the WAL logs are met, and that any necessary tuning is done (see Section 29.4).

WAL logs are stored in the directory `pg_xlog` under the data directory, as a set of segment files, normally each 16 MB in size (but the size can be changed by altering the `--with-wal-segsize` configure option when building the server). Each segment is divided into pages, normally 8 kB each (this size can be changed via the `--with-wal-blocksize` configure option). The log record headers are described in `access/xlogrecord.h`; the record content is dependent on the type of event that is being logged. Segment files are given ever-increasing numbers as names, starting at `000000010000000000000001`. The numbers do not wrap, but it will take a very, very long time to exhaust the available stock of numbers.

It is advantageous if the log is located on a different disk from the main database files. This can be achieved by moving the `pg_xlog` directory to another location (while the server is shut down, of course) and creating a symbolic link from the original location in the main data directory to the new location.

The aim of WAL is to ensure that the log is written before database records are altered, but this can be subverted by disk drives that falsely report a successful write to the kernel, when in fact they have only cached the data and not yet stored it on the disk. A power failure in such a situation might lead to irrecoverable data corruption. Administrators should try to ensure that disks holding PostgreSQL's WAL log files do not make such false reports. (See Section 29.1.)

After a checkpoint has been made and the log flushed, the checkpoint's position is saved in the file `pg_control`. Therefore, at the start of recovery, the server first reads `pg_control` and then the checkpoint record; then it performs the REDO operation by scanning forward from the log position indicated in the checkpoint record. Because the entire content of data pages is saved in the log on the first page modification after a checkpoint (assuming full_page_writes is not disabled), all pages changed since the checkpoint will be restored to a consistent state.

To deal with the case where `pg_control` is corrupt, we should support the possibility of scanning existing log segments in reverse order — newest to oldest — in order to find the latest checkpoint. This has not been implemented yet. `pg_control` is small enough (less than one disk page) that it is not subject to partial-write problems, and as of this writing there have been no reports of database failures due solely to the inability to read `pg_control` itself. So while it is theoretically a weak spot, `pg_control` does not seem to be a problem in practice.

Chapter 30. Regression Tests

The regression tests are a comprehensive set of tests for the SQL implementation in PostgreSQL. They test standard SQL operations as well as the extended capabilities of PostgreSQL.

30.1. Running the Tests

The regression tests can be run against an already installed and running server, or using a temporary installation within the build tree. Furthermore, there is a "parallel" and a "sequential" mode for running the tests. The sequential method runs each test script alone, while the parallel method starts up multiple server processes to run groups of tests in parallel. Parallel testing adds confidence that interprocess communication and locking are working correctly.

30.1.1. Running the Tests Against a Temporary Installation

To run the parallel regression tests after building but before installation, type:

```
make check
```

in the top-level directory. (Or you can change to `src/test/regress` and run the command there.) At the end you should see something like:

```
=======================
 All 115 tests passed.
=======================
```

or otherwise a note about which tests failed. See Section 30.2 below before assuming that a "failure" represents a serious problem.

Because this test method runs a temporary server, it will not work if you did the build as the root user, since the server will not start as root. Recommended procedure is not to do the build as root, or else to perform testing after completing the installation.

If you have configured PostgreSQL to install into a location where an older PostgreSQL installation already exists, and you perform `make check` before installing the new version, you might find that the tests fail because the new programs try to use the already-installed shared libraries. (Typical symptoms are complaints about undefined symbols.) If you wish to run the tests before overwriting the old installation, you'll need to build with `configure --disable-rpath`. It is not recommended that you use this option for the final installation, however.

The parallel regression test starts quite a few processes under your user ID. Presently, the maximum concurrency is twenty parallel test scripts, which means forty processes: there's a server process and a psql process for each test script. So if your system enforces a per-user limit on the number of processes, make sure this limit is at least fifty or so, else you might get random-seeming failures in the parallel test. If you are not in a position to raise the limit, you can cut down the degree of parallelism by setting the `MAX_CONNECTIONS` parameter. For example:

```
make MAX_CONNECTIONS=10 check
```

runs no more than ten tests concurrently.

30.1.2. Running the Tests Against an Existing Installation

To run the tests after installation (see Chapter 15), initialize a data area and start the server as explained in Chapter 17, then type:

```
make installcheck
```

or for a parallel test:

```
make installcheck-parallel
```

The tests will expect to contact the server at the local host and the default port number, unless directed otherwise by `PGHOST` and `PGPORT` environment variables. The tests will be run in a database named `regression`; any existing database by this name will be dropped. The tests will also transiently create some cluster-wide objects, such as user identities named `regressuser`*N*.

30.1.3. Additional Test Suites

The `make check` and `make installcheck` commands run only the "core" regression tests, which test built-in functionality of the PostgreSQL server. The source distribution also contains additional test suites, most of them having to do with add-on functionality such as optional procedural languages.

To run all test suites applicable to the modules that have been selected to be built, including the core tests, type one of these commands at the top of the build tree:

```
make check-world
make installcheck-world
```

These commands run the tests using temporary servers or an already-installed server, respectively, just as previously explained for `make check` and `make installcheck`. Other considerations are the same as previously explained for each method. Note that `make check-world` builds a separate temporary installation tree for each tested module, so it requires a great deal more time and disk space than `make installcheck-world`.

Alternatively, you can run individual test suites by typing `make check` or `make installcheck` in the appropriate subdirectory of the build tree. Keep in mind that `make installcheck` assumes you've installed the relevant module(s), not only the core server.

The additional tests that can be invoked this way include:

- Regression tests for optional procedural languages (other than PL/pgSQL, which is tested by the core tests). These are located under `src/pl`.
- Regression tests for `contrib` modules, located under `contrib`. Not all `contrib` modules have tests.
- Regression tests for the ECPG interface library, located in `src/interfaces/ecpg/test`.
- Tests stressing behavior of concurrent sessions, located in `src/test/isolation`.
- Tests of client programs under `src/bin`. See also Section 30.4.

When using `installcheck` mode, these tests will destroy any existing databases named `pl_regression`, `contrib_regression`, `isolationtest`, `regress1`, or `connectdb`, as well as `regression`.

30.1.4. Locale and Encoding

By default, tests using a temporary installation use the locale defined in the current environment and the corresponding database encoding as determined by `initdb`. It can be useful to test different locales by setting the appropriate environment variables, for example:

```
make check LANG=C
make check LC_COLLATE=en_US.utf8 LC_CTYPE=fr_CA.utf8
```

For implementation reasons, setting `LC_ALL` does not work for this purpose; all the other locale-related environment variables do work.

When testing against an existing installation, the locale is determined by the existing database cluster and cannot be set separately for the test run.

You can also choose the database encoding explicitly by setting the variable `ENCODING`, for example:

```
make check LANG=C ENCODING=EUC_JP
```

Setting the database encoding this way typically only makes sense if the locale is C; otherwise the encoding is chosen automatically from the locale, and specifying an encoding that does not match the locale will result in an error.

The database encoding can be set for tests against either a temporary or an existing installation, though in the latter case it must be compatible with the installation's locale.

30.1.5. Extra Tests

The core regression test suite contains a few test files that are not run by default, because they might be platform-dependent or take a very long time to run. You can run these or other extra test files by setting the variable `EXTRA_TESTS`. For example, to run the `numeric_big` test:

```
make check EXTRA_TESTS=numeric_big
```

To run the collation tests:

```
make check EXTRA_TESTS=collate.linux.utf8 LANG=en_US.utf8
```

The `collate.linux.utf8` test works only on Linux/glibc platforms, and only when run in a database that uses UTF-8 encoding.

30.1.6. Testing Hot Standby

The source distribution also contains regression tests for the static behavior of Hot Standby. These tests require a running primary server and a running standby server that is accepting new WAL changes from

the primary (using either file-based log shipping or streaming replication). Those servers are not automatically created for you, nor is replication setup documented here. Please check the various sections of the documentation devoted to the required commands and related issues.

To run the Hot Standby tests, first create a database called `regression` on the primary:

```
psql -h primary -c "CREATE DATABASE regression"
```

Next, run the preparatory script `src/test/regress/sql/hs_primary_setup.sql` on the primary in the regression database, for example:

```
psql -h primary -f src/test/regress/sql/hs_primary_setup.sql regression
```

Allow these changes to propagate to the standby.

Now arrange for the default database connection to be to the standby server under test (for example, by setting the `PGHOST` and `PGPORT` environment variables). Finally, run `make standbycheck` in the regression directory:

```
cd src/test/regress
make standbycheck
```

Some extreme behaviors can also be generated on the primary using the script `src/test/regress/sql/hs_primary_extremes.sql` to allow the behavior of the standby to be tested.

30.2. Test Evaluation

Some properly installed and fully functional PostgreSQL installations can "fail" some of these regression tests due to platform-specific artifacts such as varying floating-point representation and message wording. The tests are currently evaluated using a simple `diff` comparison against the outputs generated on a reference system, so the results are sensitive to small system differences. When a test is reported as "failed", always examine the differences between expected and actual results; you might find that the differences are not significant. Nonetheless, we still strive to maintain accurate reference files across all supported platforms, so it can be expected that all tests pass.

The actual outputs of the regression tests are in files in the `src/test/regress/results` directory. The test script uses `diff` to compare each output file against the reference outputs stored in the `src/test/regress/expected` directory. Any differences are saved for your inspection in `src/test/regress/regression.diffs`. (When running a test suite other than the core tests, these files of course appear in the relevant subdirectory, not `src/test/regress`.)

If you don't like the `diff` options that are used by default, set the environment variable `PG_REGRESS_DIFF_OPTS`, for instance `PG_REGRESS_DIFF_OPTS='-u'`. (Or you can run `diff` yourself, if you prefer.)

If for some reason a particular platform generates a "failure" for a given test, but inspection of the output convinces you that the result is valid, you can add a new comparison file to silence the failure report in future test runs. See Section 30.3 for details.

30.2.1. Error Message Differences

Some of the regression tests involve intentional invalid input values. Error messages can come from either the PostgreSQL code or from the host platform system routines. In the latter case, the messages can vary between platforms, but should reflect similar information. These differences in messages will result in a "failed" regression test that can be validated by inspection.

30.2.2. Locale Differences

If you run the tests against a server that was initialized with a collation-order locale other than C, then there might be differences due to sort order and subsequent failures. The regression test suite is set up to handle this problem by providing alternate result files that together are known to handle a large number of locales.

To run the tests in a different locale when using the temporary-installation method, pass the appropriate locale-related environment variables on the `make` command line, for example:

```
make check LANG=de_DE.utf8
```

(The regression test driver unsets `LC_ALL`, so it does not work to choose the locale using that variable.) To use no locale, either unset all locale-related environment variables (or set them to `C`) or use the following special invocation:

```
make check NO_LOCALE=1
```

When running the tests against an existing installation, the locale setup is determined by the existing installation. To change it, initialize the database cluster with a different locale by passing the appropriate options to `initdb`.

In general, it is advisable to try to run the regression tests in the locale setup that is wanted for production use, as this will exercise the locale- and encoding-related code portions that will actually be used in production. Depending on the operating system environment, you might get failures, but then you will at least know what locale-specific behaviors to expect when running real applications.

30.2.3. Date and Time Differences

Most of the date and time results are dependent on the time zone environment. The reference files are generated for time zone `PST8PDT` (Berkeley, California), and there will be apparent failures if the tests are not run with that time zone setting. The regression test driver sets environment variable `PGTZ` to `PST8PDT`, which normally ensures proper results.

30.2.4. Floating-Point Differences

Some of the tests involve computing 64-bit floating-point numbers (`double precision`) from table columns. Differences in results involving mathematical functions of `double precision` columns have been observed. The `float8` and `geometry` tests are particularly prone to small differences across platforms, or even with different compiler optimization settings. Human eyeball comparison is needed to

determine the real significance of these differences which are usually 10 places to the right of the decimal point.

Some systems display minus zero as `-0`, while others just show `0`.

Some systems signal errors from `pow()` and `exp()` differently from the mechanism expected by the current PostgreSQL code.

30.2.5. Row Ordering Differences

You might see differences in which the same rows are output in a different order than what appears in the expected file. In most cases this is not, strictly speaking, a bug. Most of the regression test scripts are not so pedantic as to use an `ORDER BY` for every single `SELECT`, and so their result row orderings are not well-defined according to the SQL specification. In practice, since we are looking at the same queries being executed on the same data by the same software, we usually get the same result ordering on all platforms, so the lack of `ORDER BY` is not a problem. Some queries do exhibit cross-platform ordering differences, however. When testing against an already-installed server, ordering differences can also be caused by non-C locale settings or non-default parameter settings, such as custom values of `work_mem` or the planner cost parameters.

Therefore, if you see an ordering difference, it's not something to worry about, unless the query does have an `ORDER BY` that your result is violating. However, please report it anyway, so that we can add an `ORDER BY` to that particular query to eliminate the bogus "failure" in future releases.

You might wonder why we don't order all the regression test queries explicitly to get rid of this issue once and for all. The reason is that that would make the regression tests less useful, not more, since they'd tend to exercise query plan types that produce ordered results to the exclusion of those that don't.

30.2.6. Insufficient Stack Depth

If the `errors` test results in a server crash at the `select infinite_recurse()` command, it means that the platform's limit on process stack size is smaller than the max_stack_depth parameter indicates. This can be fixed by running the server under a higher stack size limit (4MB is recommended with the default value of `max_stack_depth`). If you are unable to do that, an alternative is to reduce the value of `max_stack_depth`.

On platforms supporting `getrlimit()`, the server should automatically choose a safe value of `max_stack_depth`; so unless you've manually overridden this setting, a failure of this kind is a reportable bug.

30.2.7. The "random" Test

The `random` test script is intended to produce random results. In very rare cases, this causes that regression test to fail. Typing:

```
diff results/random.out expected/random.out
```

should produce only one or a few lines of differences. You need not worry unless the random test fails repeatedly.

30.2.8. Configuration Parameters

When running the tests against an existing installation, some non-default parameter settings could cause the tests to fail. For example, changing parameters such as `enable_seqscan` or `enable_indexscan` could cause plan changes that would affect the results of tests that use `EXPLAIN`.

30.3. Variant Comparison Files

Since some of the tests inherently produce environment-dependent results, we have provided ways to specify alternate "expected" result files. Each regression test can have several comparison files showing possible results on different platforms. There are two independent mechanisms for determining which comparison file is used for each test.

The first mechanism allows comparison files to be selected for specific platforms. There is a mapping file, `src/test/regress/resultmap`, that defines which comparison file to use for each platform. To eliminate bogus test "failures" for a particular platform, you first choose or make a variant result file, and then add a line to the `resultmap` file.

Each line in the mapping file is of the form

```
testname:output:platformpattern=comparisonfilename
```

The test name is just the name of the particular regression test module. The output value indicates which output file to check. For the standard regression tests, this is always `out`. The value corresponds to the file extension of the output file. The platform pattern is a pattern in the style of the Unix tool `expr` (that is, a regular expression with an implicit `^` anchor at the start). It is matched against the platform name as printed by `config.guess`. The comparison file name is the base name of the substitute result comparison file.

For example: some systems interpret very small floating-point values as zero, rather than reporting an underflow error. This causes a few differences in the `float8` regression test. Therefore, we provide a variant comparison file, `float8-small-is-zero.out`, which includes the results to be expected on these systems. To silence the bogus "failure" message on OpenBSD platforms, `resultmap` includes:

```
float8:out:i.86-.*-openbsd=float8-small-is-zero.out
```

which will trigger on any machine where the output of `config.guess` matches `i.86-.*-openbsd`. Other lines in `resultmap` select the variant comparison file for other platforms where it's appropriate.

The second selection mechanism for variant comparison files is much more automatic: it simply uses the "best match" among several supplied comparison files. The regression test driver script considers both the standard comparison file for a test, *`testname`*`.out`, and variant files named *`testname_digit`*`.out` (where the *`digit`* is any single digit `0-9`). If any such file is an exact match, the test is considered to pass; otherwise, the one that generates the shortest diff is used to create the failure report. (If `resultmap` includes an entry for the particular test, then the base *`testname`* is the substitute name given in `resultmap`.)

For example, for the `char` test, the comparison file `char.out` contains results that are expected in the `C` and `POSIX` locales, while the file `char_1.out` contains results sorted as they appear in many other locales.

The best-match mechanism was devised to cope with locale-dependent results, but it can be used in any situation where the test results cannot be predicted easily from the platform name alone. A limitation of this mechanism is that the test driver cannot tell which variant is actually "correct" for the current environment; it will just pick the variant that seems to work best. Therefore it is safest to use this mechanism only for variant results that you are willing to consider equally valid in all contexts.

30.4. TAP Tests

The client program tests under `src/bin` use the Perl TAP tools and are run by `prove`. You can pass command-line options to `prove` by setting the `make` variable `PROVE_FLAGS`, for example:

```
make -C src/bin check PROVE_FLAGS='--reverse'
```

The default is `--verbose`. See the manual page of `prove` for more information.

The tests written in Perl require the Perl module `IPC::Run`. This module is available from CPAN or an operating system package.

30.5. Test Coverage Examination

The PostgreSQL source code can be compiled with coverage testing instrumentation, so that it becomes possible to examine which parts of the code are covered by the regression tests or any other test suite that is run with the code. This is currently supported when compiling with GCC and requires the `gcov` and `lcov` programs.

A typical workflow would look like this:

```
./configure --enable-coverage ... OTHER OPTIONS ...
make
make check # or other test suite
make coverage-html
```

Then point your HTML browser to `coverage/index.html`. The `make` commands also work in subdirectories.

To reset the execution counts between test runs, run:

```
make coverage-clean
```

IV. Client Interfaces

This part describes the client programming interfaces distributed with PostgreSQL. Each of these chapters can be read independently. Note that there are many other programming interfaces for client programs that are distributed separately and contain their own documentation (Appendix H lists some of the more popular ones). Readers of this part should be familiar with using SQL commands to manipulate and query the database (see Part II) and of course with the programming language that the interface uses.

Chapter 31. libpq - C Library

libpq is the C application programmer's interface to PostgreSQL. libpq is a set of library functions that allow client programs to pass queries to the PostgreSQL backend server and to receive the results of these queries.

libpq is also the underlying engine for several other PostgreSQL application interfaces, including those written for C++, Perl, Python, Tcl and ECPG. So some aspects of libpq's behavior will be important to you if you use one of those packages. In particular, Section 31.14, Section 31.15 and Section 31.18 describe behavior that is visible to the user of any application that uses libpq.

Some short programs are included at the end of this chapter (Section 31.21) to show how to write programs that use libpq. There are also several complete examples of libpq applications in the directory `src/test/examples` in the source code distribution.

Client programs that use libpq must include the header file `libpq-fe.h` and must link with the libpq library.

31.1. Database Connection Control Functions

The following functions deal with making a connection to a PostgreSQL backend server. An application program can have several backend connections open at one time. (One reason to do that is to access more than one database.) Each connection is represented by a `PGconn` object, which is obtained from the function `PQconnectdb`, `PQconnectdbParams`, or `PQsetdbLogin`. Note that these functions will always return a non-null object pointer, unless perhaps there is too little memory even to allocate the `PGconn` object. The `PQstatus` function should be called to check the return value for a successful connection before queries are sent via the connection object.

Warning

If untrusted users have access to a database that has not adopted a secure schema usage pattern, begin each session by removing publicly-writable schemas from `search_path`. One can set parameter key word `options` to value `-csearch_path=`. Alternately, one can issue `PQexec(`*`conn`*`, "SELECT pg_catalog.set_config('search_path', '', false)")` after connecting. This consideration is not specific to libpq; it applies to every interface for executing arbitrary SQL commands.

Warning

On Unix, forking a process with open libpq connections can lead to unpredictable results because the parent and child processes share the same sockets and operating system resources. For this reason, such usage is not recommended, though doing an `exec` from the child process to load a new executable is safe.

PQconnectdbParams

Makes a new connection to the database server.

```
PGconn *PQconnectdbParams(const char * const *keywords,
                          const char * const *values,
                          int expand_dbname);
```

This function opens a new database connection using the parameters taken from two NULL-terminated arrays. The first, keywords, is defined as an array of strings, each one being a key word. The second, values, gives the value for each key word. Unlike PQsetdbLogin below, the parameter set can be extended without changing the function signature, so use of this function (or its nonblocking analogs PQconnectStartParams and PQconnectPoll) is preferred for new application programming.

The currently recognized parameter key words are listed in Section 31.1.2.

When expand_dbname is non-zero, the dbname key word value is allowed to be recognized as a connection string. Only the first occurrence of dbname is expanded this way, any subsequent dbname value is processed as plain database name. More details on the possible connection string formats appear in Section 31.1.1.

The passed arrays can be empty to use all default parameters, or can contain one or more parameter settings. They should be matched in length. Processing will stop at the first NULL element in the keywords array.

If any parameter is NULL or an empty string, the corresponding environment variable (see Section 31.14) is checked. If the environment variable is not set either, then the indicated built-in defaults are used.

In general key words are processed from the beginning of these arrays in index order. The effect of this is that when key words are repeated, the last processed value is retained. Therefore, through careful placement of the dbname key word, it is possible to determine what may be overridden by a conninfo string, and what may not.

PQconnectdb

Makes a new connection to the database server.

```
PGconn *PQconnectdb(const char *conninfo);
```

This function opens a new database connection using the parameters taken from the string conninfo.

The passed string can be empty to use all default parameters, or it can contain one or more parameter settings separated by whitespace, or it can contain a URI. See Section 31.1.1 for details.

PQsetdbLogin

Makes a new connection to the database server.

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
                     const char *login,
                     const char *pwd);
```

This is the predecessor of PQconnectdb with a fixed set of parameters. It has the same functionality except that the missing parameters will always take on default values. Write NULL or an empty string for any one of the fixed parameters that is to be defaulted.

If the dbName contains an = sign or has a valid connection URI prefix, it is taken as a conninfo string in exactly the same way as if it had been passed to PQconnectdb, and the remaining parameters are then applied as specified for PQconnectdbParams.

PQsetdb

Makes a new connection to the database server.

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName);
```

This is a macro that calls PQsetdbLogin with null pointers for the login and pwd parameters. It is provided for backward compatibility with very old programs.

PQconnectStartParams
PQconnectStart
PQconnectPoll

Make a connection to the database server in a nonblocking manner.

```
PGconn *PQconnectStartParams(const char * const *keywords,
                             const char * const *values,
                             int expand_dbname);

PGconn *PQconnectStart(const char *conninfo);

PostgresPollingStatusType PQconnectPoll(PGconn *conn);
```

These three functions are used to open a connection to a database server such that your application's thread of execution is not blocked on remote I/O whilst doing so. The point of this approach is that the waits for I/O to complete can occur in the application's main loop, rather than down inside PQconnectdbParams or PQconnectdb, and so the application can manage this operation in parallel with other activities.

With PQconnectStartParams, the database connection is made using the parameters taken from the keywords and values arrays, and controlled by expand_dbname, as described above for PQconnectdbParams.

With PQconnectStart, the database connection is made using the parameters taken from the string conninfo as described above for PQconnectdb.

Neither PQconnectStartParams nor PQconnectStart nor PQconnectPoll will block, so long as a number of restrictions are met:

- The hostaddr and host parameters are used appropriately to ensure that name and reverse name queries are not made. See the documentation of these parameters in Section 31.1.2 for details.
- If you call PQtrace, ensure that the stream object into which you trace will not block.
- You ensure that the socket is in the appropriate state before calling PQconnectPoll, as described below.

Note: use of PQconnectStartParams is analogous to PQconnectStart shown below.

To begin a nonblocking connection request, call conn = PQconnectStart("*connection_info_string*"). If conn is null, then libpq has been unable to allocate a new PGconn structure. Otherwise, a valid PGconn pointer is returned (though not yet representing a valid connection to the database). On return from PQconnectStart, call status = PQstatus(conn). If status equals CONNECTION_BAD, PQconnectStart has failed.

If PQconnectStart succeeds, the next stage is to poll libpq so that it can proceed with the connection sequence. Use PQsocket(conn) to obtain the descriptor of the socket underlying the database connection. Loop thus: If PQconnectPoll(conn) last returned PGRES_POLLING_READING, wait until the socket is ready to read (as indicated by select(), poll(), or similar system function). Then call PQconnectPoll(conn) again. Conversely, if PQconnectPoll(conn) last returned PGRES_POLLING_WRITING, wait until the socket is ready to write, then call PQconnectPoll(conn) again. If you have yet to call PQconnectPoll, i.e., just after the call to PQconnectStart, behave as if it last returned PGRES_POLLING_WRITING. Continue this loop until PQconnectPoll(conn) returns PGRES_POLLING_FAILED, indicating the connection procedure has failed, or PGRES_POLLING_OK, indicating the connection has been successfully made.

At any time during connection, the status of the connection can be checked by calling PQstatus. If this call returns CONNECTION_BAD, then the connection procedure has failed; if the call returns CONNECTION_OK, then the connection is ready. Both of these states are equally detectable from the return value of PQconnectPoll, described above. Other states might also occur during (and only during) an asynchronous connection procedure. These indicate the current stage of the connection procedure and might be useful to provide feedback to the user for example. These statuses are:

CONNECTION_STARTED

Waiting for connection to be made.

CONNECTION_MADE

Connection OK; waiting to send.

CONNECTION_AWAITING_RESPONSE

Waiting for a response from the server.

CONNECTION_AUTH_OK

Received authentication; waiting for backend start-up to finish.

CONNECTION_SSL_STARTUP

Negotiating SSL encryption.

CONNECTION_SETENV

Negotiating environment-driven parameter settings.

Note that, although these constants will remain (in order to maintain compatibility), an application should never rely upon these occurring in a particular order, or at all, or on the status always being one of these documented values. An application might do something like this:

```
switch(PQstatus(conn))
```

```
{
        case CONNECTION_STARTED:
            feedback = "Connecting...";
            break;

        case CONNECTION_MADE:
            feedback = "Connected to server...";
            break;
.
.
.
        default:
            feedback = "Connecting...";
}
```

The `connect_timeout` connection parameter is ignored when using `PQconnectPoll`; it is the application's responsibility to decide whether an excessive amount of time has elapsed. Otherwise, `PQconnectStart` followed by a `PQconnectPoll` loop is equivalent to `PQconnectdb`.

Note that if `PQconnectStart` returns a non-null pointer, you must call `PQfinish` when you are finished with it, in order to dispose of the structure and any associated memory blocks. This must be done even if the connection attempt fails or is abandoned.

`PQconndefaults`

Returns the default connection options.

```
PQconninfoOption *PQconndefaults(void);

typedef struct
{
    char   *keyword;   /* The keyword of the option */
    char   *envvar;    /* Fallback environment variable name */
    char   *compiled;  /* Fallback compiled in default value */
    char   *val;       /* Option's current value, or NULL */
    char   *label;     /* Label for field in connect dialog */
    char   *dispchar;  /* Indicates how to display this field
                          in a connect dialog. Values are:
                          ""        Display entered value as is
                          "*"       Password field - hide value
                          "D"       Debug option - don't show by default */
    int     dispsize;  /* Field size in characters for dialog */
} PQconninfoOption;
```

Returns a connection options array. This can be used to determine all possible `PQconnectdb` options and their current default values. The return value points to an array of `PQconninfoOption` structures, which ends with an entry having a null `keyword` pointer. The null pointer is returned if memory could not be allocated. Note that the current default values (`val` fields) will depend on environment variables and other context. A missing or invalid service file will be silently ignored. Callers must treat the connection options data as read-only.

After processing the options array, free it by passing it to `PQconninfoFree`. If this is not done, a small amount of memory is leaked for each call to `PQconndefaults`.

PQconninfo

Returns the connection options used by a live connection.

```
PQconninfoOption *PQconninfo(PGconn *conn);
```

Returns a connection options array. This can be used to determine all possible PQconnectdb options and the values that were used to connect to the server. The return value points to an array of PQconninfoOption structures, which ends with an entry having a null keyword pointer. All notes above for PQconndefaults also apply to the result of PQconninfo.

PQconninfoParse

Returns parsed connection options from the provided connection string.

```
PQconninfoOption *PQconninfoParse(const char *conninfo, char **errmsg);
```

Parses a connection string and returns the resulting options as an array; or returns NULL if there is a problem with the connection string. This function can be used to extract the PQconnectdb options in the provided connection string. The return value points to an array of PQconninfoOption structures, which ends with an entry having a null keyword pointer.

All legal options will be present in the result array, but the PQconninfoOption for any option not present in the connection string will have val set to NULL; default values are not inserted.

If errmsg is not NULL, then *errmsg is set to NULL on success, else to a malloc'd error string explaining the problem. (It is also possible for *errmsg to be set to NULL and the function to return NULL; this indicates an out-of-memory condition.)

After processing the options array, free it by passing it to PQconninfoFree. If this is not done, some memory is leaked for each call to PQconninfoParse. Conversely, if an error occurs and errmsg is not NULL, be sure to free the error string using PQfreemem.

PQfinish

Closes the connection to the server. Also frees memory used by the PGconn object.

```
void PQfinish(PGconn *conn);
```

Note that even if the server connection attempt fails (as indicated by PQstatus), the application should call PQfinish to free the memory used by the PGconn object. The PGconn pointer must not be used again after PQfinish has been called.

PQreset

Resets the communication channel to the server.

```
void PQreset(PGconn *conn);
```

This function will close the connection to the server and attempt to reestablish a new connection to the same server, using all the same parameters previously used. This might be useful for error recovery if a working connection is lost.

PQresetStart
PQresetPoll

Reset the communication channel to the server, in a nonblocking manner.

```
int PQresetStart(PGconn *conn);

PostgresPollingStatusType PQresetPoll(PGconn *conn);
```

These functions will close the connection to the server and attempt to reestablish a new connection to the same server, using all the same parameters previously used. This can be useful for error recovery if a working connection is lost. They differ from `PQreset` (above) in that they act in a nonblocking manner. These functions suffer from the same restrictions as `PQconnectStartParams`, `PQconnectStart` and `PQconnectPoll`.

To initiate a connection reset, call `PQresetStart`. If it returns 0, the reset has failed. If it returns 1, poll the reset using `PQresetPoll` in exactly the same way as you would create the connection using `PQconnectPoll`.

`PQpingParams`

`PQpingParams` reports the status of the server. It accepts connection parameters identical to those of `PQconnectdbParams`, described above. It is not necessary to supply correct user name, password, or database name values to obtain the server status; however, if incorrect values are provided, the server will log a failed connection attempt.

```
PGPing PQpingParams(const char * const *keywords,
                    const char * const *values,
                    int expand_dbname);
```

The function returns one of the following values:

`PQPING_OK`

The server is running and appears to be accepting connections.

`PQPING_REJECT`

The server is running but is in a state that disallows connections (startup, shutdown, or crash recovery).

`PQPING_NO_RESPONSE`

The server could not be contacted. This might indicate that the server is not running, or that there is something wrong with the given connection parameters (for example, wrong port number), or that there is a network connectivity problem (for example, a firewall blocking the connection request).

`PQPING_NO_ATTEMPT`

No attempt was made to contact the server, because the supplied parameters were obviously incorrect or there was some client-side problem (for example, out of memory).

`PQping`

`PQping` reports the status of the server. It accepts connection parameters identical to those of `PQconnectdb`, described above. It is not necessary to supply correct user name, password, or database name values to obtain the server status; however, if incorrect values are provided, the server will log a failed connection attempt.

```
PGPing PQping(const char *conninfo);
```

The return values are the same as for `PQpingParams`.

31.1.1. Connection Strings

Several libpq functions parse a user-specified string to obtain connection parameters. There are two accepted formats for these strings: plain `keyword = value` strings and RFC 3986[1] URIs.

31.1.1.1. Keyword/Value Connection Strings

In the first format, each parameter setting is in the form `keyword = value`. Spaces around the equal sign are optional. To write an empty value, or a value containing spaces, surround it with single quotes, e.g., `keyword = 'a value'`. Single quotes and backslashes within the value must be escaped with a backslash, i.e., `\'` and `\\`.

Example:

```
host=localhost port=5432 dbname=mydb connect_timeout=10
```

The recognized parameter key words are listed in Section 31.1.2.

31.1.1.2. Connection URIs

The general form for a connection URI is:

```
postgresql://[user[:password]@][netloc][:port][/dbname][?param1=value1&...]
```

The URI scheme designator can be either `postgresql://` or `postgres://`. Each of the URI parts is optional. The following examples illustrate valid URI syntax uses:

```
postgresql://
postgresql://localhost
postgresql://localhost:5433
postgresql://localhost/mydb
postgresql://user@localhost
postgresql://user:secret@localhost
postgresql://other@localhost/otherdb?connect_timeout=10&application_name=myapp
```

Components of the hierarchical part of the URI can also be given as parameters. For example:

```
postgresql:///mydb?host=localhost&port=5433
```

Percent-encoding may be used to include symbols with special meaning in any of the URI parts.

Any connection parameters not corresponding to key words listed in Section 31.1.2 are ignored and a warning message about them is sent to `stderr`.

For improved compatibility with JDBC connection URIs, instances of parameter `ssl=true` are translated into `sslmode=require`.

1. https://tools.ietf.org/html/rfc3986

The host part may be either host name or an IP address. To specify an IPv6 host address, enclose it in square brackets:

```
postgresql://[2001:db8::1234]/database
```

The host component is interpreted as described for the parameter host. In particular, a Unix-domain socket connection is chosen if the host part is either empty or starts with a slash, otherwise a TCP/IP connection is initiated. Note, however, that the slash is a reserved character in the hierarchical part of the URI. So, to specify a non-standard Unix-domain socket directory, either omit the host specification in the URI and specify the host as a parameter, or percent-encode the path in the host component of the URI:

```
postgresql:///dbname?host=/var/lib/postgresql
postgresql://%2Fvar%2Flib%2Fpostgresql/dbname
```

31.1.2. Parameter Key Words

The currently recognized parameter key words are:

`host`

Name of host to connect to. If this begins with a slash, it specifies Unix-domain communication rather than TCP/IP communication; the value is the name of the directory in which the socket file is stored. The default behavior when `host` is not specified is to connect to a Unix-domain socket in `/tmp` (or whatever socket directory was specified when PostgreSQL was built). On machines without Unix-domain sockets, the default is to connect to `localhost`.

`hostaddr`

Numeric IP address of host to connect to. This should be in the standard IPv4 address format, e.g., `172.28.40.9`. If your machine supports IPv6, you can also use those addresses. TCP/IP communication is always used when a nonempty string is specified for this parameter.

Using `hostaddr` instead of `host` allows the application to avoid a host name look-up, which might be important in applications with time constraints. However, a host name is required for GSSAPI or SSPI authentication methods, as well as for `verify-full` SSL certificate verification. The following rules are used:

- If `host` is specified without `hostaddr`, a host name lookup occurs.
- If `hostaddr` is specified without `host`, the value for `hostaddr` gives the server network address. The connection attempt will fail if the authentication method requires a host name.
- If both `host` and `hostaddr` are specified, the value for `hostaddr` gives the server network address. The value for `host` is ignored unless the authentication method requires it, in which case it will be used as the host name.

Note that authentication is likely to fail if `host` is not the name of the server at network address `hostaddr`. Also, note that `host` rather than `hostaddr` is used to identify the connection in `~/.pgpass` (see Section 31.15).

Without either a host name or host address, libpq will connect using a local Unix-domain socket; or on machines without Unix-domain sockets, it will attempt to connect to `localhost`.

`port`

Port number to connect to at the server host, or socket file name extension for Unix-domain connections.

`dbname`

The database name. Defaults to be the same as the user name. In certain contexts, the value is checked for extended formats; see Section 31.1.1 for more details on those.

`user`

PostgreSQL user name to connect as. Defaults to be the same as the operating system name of the user running the application.

`password`

Password to be used if the server demands password authentication.

`connect_timeout`

Maximum wait for connection, in seconds (write as a decimal integer string). Zero or not specified means wait indefinitely. It is not recommended to use a timeout of less than 2 seconds.

`client_encoding`

This sets the `client_encoding` configuration parameter for this connection. In addition to the values accepted by the corresponding server option, you can use `auto` to determine the right encoding from the current locale in the client (`LC_CTYPE` environment variable on Unix systems).

`options`

Specifies command-line options to send to the server at connection start. For example, setting this to `-c geqo=off` sets the session's value of the `geqo` parameter to `off`. Spaces within this string are considered to separate command-line arguments, unless escaped with a backslash (`\`); write `\\` to represent a literal backslash. For a detailed discussion of the available options, consult Chapter 18.

`application_name`

Specifies a value for the application_name configuration parameter.

`fallback_application_name`

Specifies a fallback value for the application_name configuration parameter. This value will be used if no value has been given for `application_name` via a connection parameter or the `PGAPPNAME` environment variable. Specifying a fallback name is useful in generic utility programs that wish to set a default application name but allow it to be overridden by the user.

`keepalives`

Controls whether client-side TCP keepalives are used. The default value is 1, meaning on, but you can change this to 0, meaning off, if keepalives are not wanted. This parameter is ignored for connections made via a Unix-domain socket.

`keepalives_idle`

Controls the number of seconds of inactivity after which TCP should send a keepalive message to the server. A value of zero uses the system default. This parameter is ignored for connections

made via a Unix-domain socket, or if keepalives are disabled. It is only supported on systems where TCP_KEEPIDLE or an equivalent socket option is available, and on Windows; on other systems, it has no effect.

keepalives_interval

Controls the number of seconds after which a TCP keepalive message that is not acknowledged by the server should be retransmitted. A value of zero uses the system default. This parameter is ignored for connections made via a Unix-domain socket, or if keepalives are disabled. It is only supported on systems where TCP_KEEPINTVL or an equivalent socket option is available, and on Windows; on other systems, it has no effect.

keepalives_count

Controls the number of TCP keepalives that can be lost before the client's connection to the server is considered dead. A value of zero uses the system default. This parameter is ignored for connections made via a Unix-domain socket, or if keepalives are disabled. It is only supported on systems where TCP_KEEPCNT or an equivalent socket option is available; on other systems, it has no effect.

tty

Ignored (formerly, this specified where to send server debug output).

sslmode

This option determines whether or with what priority a secure SSL TCP/IP connection will be negotiated with the server. There are six modes:

disable

only try a non-SSL connection

allow

first try a non-SSL connection; if that fails, try an SSL connection

prefer (default)

first try an SSL connection; if that fails, try a non-SSL connection

require

only try an SSL connection. If a root CA file is present, verify the certificate in the same way as if verify-ca was specified

verify-ca

only try an SSL connection, and verify that the server certificate is issued by a trusted certificate authority (CA)

verify-full

only try an SSL connection, verify that the server certificate is issued by a trusted CA and that the requested server host name matches that in the certificate

See Section 31.18 for a detailed description of how these options work.

sslmode is ignored for Unix domain socket communication. If PostgreSQL is compiled without SSL support, using options require, verify-ca, or verify-full will cause an error, while options allow and prefer will be accepted but libpq will not actually attempt an SSL connection.

`requiressl`

This option is deprecated in favor of the `sslmode` setting.

If set to 1, an SSL connection to the server is required (this is equivalent to `sslmode require`). libpq will then refuse to connect if the server does not accept an SSL connection. If set to 0 (default), libpq will negotiate the connection type with the server (equivalent to `sslmode prefer`). This option is only available if PostgreSQL is compiled with SSL support.

`sslcompression`

If set to 1 (default), data sent over SSL connections will be compressed (this requires OpenSSL version 0.9.8 or later). If set to 0, compression will be disabled (this requires OpenSSL 1.0.0 or later). This parameter is ignored if a connection without SSL is made, or if the version of OpenSSL used does not support it.

Compression uses CPU time, but can improve throughput if the network is the bottleneck. Disabling compression can improve response time and throughput if CPU performance is the limiting factor.

`sslcert`

This parameter specifies the file name of the client SSL certificate, replacing the default `~/.postgresql/postgresql.crt`. This parameter is ignored if an SSL connection is not made.

`sslkey`

This parameter specifies the location for the secret key used for the client certificate. It can either specify a file name that will be used instead of the default `~/.postgresql/postgresql.key`, or it can specify a key obtained from an external "engine" (engines are OpenSSL loadable modules). An external engine specification should consist of a colon-separated engine name and an engine-specific key identifier. This parameter is ignored if an SSL connection is not made.

`sslrootcert`

This parameter specifies the name of a file containing SSL certificate authority (CA) certificate(s). If the file exists, the server's certificate will be verified to be signed by one of these authorities. The default is `~/.postgresql/root.crt`.

`sslcrl`

This parameter specifies the file name of the SSL certificate revocation list (CRL). Certificates listed in this file, if it exists, will be rejected while attempting to authenticate the server's certificate. The default is `~/.postgresql/root.crl`.

`requirepeer`

This parameter specifies the operating-system user name of the server, for example `requirepeer=postgres`. When making a Unix-domain socket connection, if this parameter is set, the client checks at the beginning of the connection that the server process is running under the specified user name; if it is not, the connection is aborted with an error. This parameter can be used to provide server authentication similar to that available with SSL certificates on TCP/IP connections. (Note that if the Unix-domain socket is in `/tmp` or another publicly writable location, any user could start a server listening there. Use this parameter to ensure that you are connected to a server run by a trusted user.) This option is only supported on platforms for which the `peer` authentication method is implemented; see Section 19.3.6.

krbsrvname

Kerberos service name to use when authenticating with GSSAPI. This must match the service name specified in the server configuration for Kerberos authentication to succeed. (See also Section 19.3.3.)

gsslib

GSS library to use for GSSAPI authentication. Currently this is disregarded except on Windows builds that include both GSSAPI and SSPI support. In that case, set this to `gssapi` to cause libpq to use the GSSAPI library for authentication instead of the default SSPI.

service

Service name to use for additional parameters. It specifies a service name in `pg_service.conf` that holds additional connection parameters. This allows applications to specify only a service name so connection parameters can be centrally maintained. See Section 31.16.

31.2. Connection Status Functions

These functions can be used to interrogate the status of an existing database connection object.

Tip: libpq application programmers should be careful to maintain the `PGconn` abstraction. Use the accessor functions described below to get at the contents of `PGconn`. Reference to internal `PGconn` fields using `libpq-int.h` is not recommended because they are subject to change in the future.

The following functions return parameter values established at connection. These values are fixed for the life of the `PGconn` object.

PQdb

Returns the database name of the connection.

```
char *PQdb(const PGconn *conn);
```

PQuser

Returns the user name of the connection.

```
char *PQuser(const PGconn *conn);
```

PQpass

Returns the password of the connection.

```
char *PQpass(const PGconn *conn);
```

PQhost

Returns the server host name of the connection.

```
char *PQhost(const PGconn *conn);
```

PQport

Returns the port of the connection.

```
char *PQport(const PGconn *conn);
```

PQtty

Returns the debug TTY of the connection. (This is obsolete, since the server no longer pays attention to the TTY setting, but the function remains for backward compatibility.)

```
char *PQtty(const PGconn *conn);
```

PQoptions

Returns the command-line options passed in the connection request.

```
char *PQoptions(const PGconn *conn);
```

The following functions return status data that can change as operations are executed on the PGconn object.

PQstatus

Returns the status of the connection.

```
ConnStatusType PQstatus(const PGconn *conn);
```

The status can be one of a number of values. However, only two of these are seen outside of an asynchronous connection procedure: CONNECTION_OK and CONNECTION_BAD. A good connection to the database has the status CONNECTION_OK. A failed connection attempt is signaled by status CONNECTION_BAD. Ordinarily, an OK status will remain so until PQfinish, but a communications failure might result in the status changing to CONNECTION_BAD prematurely. In that case the application could try to recover by calling PQreset.

See the entry for PQconnectStartParams, PQconnectStart and PQconnectPoll with regards to other status codes that might be returned.

PQtransactionStatus

Returns the current in-transaction status of the server.

```
PGTransactionStatusType PQtransactionStatus(const PGconn *conn);
```

The status can be PQTRANS_IDLE (currently idle), PQTRANS_ACTIVE (a command is in progress), PQTRANS_INTRANS (idle, in a valid transaction block), or PQTRANS_INERROR (idle, in a failed transaction block). PQTRANS_UNKNOWN is reported if the connection is bad. PQTRANS_ACTIVE is reported only when a query has been sent to the server and not yet completed.

PQparameterStatus

Looks up a current parameter setting of the server.

```
const char *PQparameterStatus(const PGconn *conn, const char *paramName);
```

Certain parameter values are reported by the server automatically at connection startup or whenever their values change. PQparameterStatus can be used to interrogate these settings. It returns the current value of a parameter if known, or NULL if the parameter is not known.

Parameters reported as of the current release include server_version, server_encoding, client_encoding, application_name, is_superuser, session_authorization,

DateStyle, IntervalStyle, TimeZone, integer_datetimes, and standard_conforming_strings. (server_encoding, TimeZone, and integer_datetimes were not reported by releases before 8.0; standard_conforming_strings was not reported by releases before 8.1; IntervalStyle was not reported by releases before 8.4; application_name was not reported by releases before 9.0.) Note that server_version, server_encoding and integer_datetimes cannot change after startup.

Pre-3.0-protocol servers do not report parameter settings, but libpq includes logic to obtain values for server_version and client_encoding anyway. Applications are encouraged to use PQparameterStatus rather than *ad hoc* code to determine these values. (Beware however that on a pre-3.0 connection, changing client_encoding via SET after connection startup will not be reflected by PQparameterStatus.) For server_version, see also PQserverVersion, which returns the information in a numeric form that is much easier to compare against.

If no value for standard_conforming_strings is reported, applications can assume it is off, that is, backslashes are treated as escapes in string literals. Also, the presence of this parameter can be taken as an indication that the escape string syntax (E'...') is accepted.

Although the returned pointer is declared const, it in fact points to mutable storage associated with the PGconn structure. It is unwise to assume the pointer will remain valid across queries.

PQprotocolVersion

Interrogates the frontend/backend protocol being used.

```
int PQprotocolVersion(const PGconn *conn);
```

Applications might wish to use this function to determine whether certain features are supported. Currently, the possible values are 2 (2.0 protocol), 3 (3.0 protocol), or zero (connection bad). The protocol version will not change after connection startup is complete, but it could theoretically change during a connection reset. The 3.0 protocol will normally be used when communicating with PostgreSQL 7.4 or later servers; pre-7.4 servers support only protocol 2.0. (Protocol 1.0 is obsolete and not supported by libpq.)

PQserverVersion

Returns an integer representing the backend version.

```
int PQserverVersion(const PGconn *conn);
```

Applications might use this function to determine the version of the database server they are connected to. The number is formed by converting the major, minor, and revision numbers into two-decimal-digit numbers and appending them together. For example, version 8.1.5 will be returned as 80105, and version 8.2 will be returned as 80200 (leading zeroes are not shown). Zero is returned if the connection is bad.

PQerrorMessage

Returns the error message most recently generated by an operation on the connection.

```
char *PQerrorMessage(const PGconn *conn);
```

Nearly all libpq functions will set a message for PQerrorMessage if they fail. Note that by libpq convention, a nonempty PQerrorMessage result can consist of multiple lines, and will include a trailing newline. The caller should not free the result directly. It will be freed when the associated PGconn handle is passed to PQfinish. The result string should not be expected to remain the same across operations on the PGconn structure.

`PQsocket`

Obtains the file descriptor number of the connection socket to the server. A valid descriptor will be greater than or equal to 0; a result of -1 indicates that no server connection is currently open. (This will not change during normal operation, but could change during connection setup or reset.)

```
int PQsocket(const PGconn *conn);
```

`PQbackendPID`

Returns the process ID (PID) of the backend process handling this connection.

```
int PQbackendPID(const PGconn *conn);
```

The backend PID is useful for debugging purposes and for comparison to `NOTIFY` messages (which include the PID of the notifying backend process). Note that the PID belongs to a process executing on the database server host, not the local host!

`PQconnectionNeedsPassword`

Returns true (1) if the connection authentication method required a password, but none was available. Returns false (0) if not.

```
int PQconnectionNeedsPassword(const PGconn *conn);
```

This function can be applied after a failed connection attempt to decide whether to prompt the user for a password.

`PQconnectionUsedPassword`

Returns true (1) if the connection authentication method used a password. Returns false (0) if not.

```
int PQconnectionUsedPassword(const PGconn *conn);
```

This function can be applied after either a failed or successful connection attempt to detect whether the server demanded a password.

The following functions return information related to SSL. This information usually doesn't change after a connection is established.

`PQsslInUse`

Returns true (1) if the connection uses SSL, false (0) if not.

```
int PQsslInUse(const PGconn *conn);
```

`PQsslAttribute`

Returns SSL-related information about the connection.

```
const char *PQsslAttribute(const PGconn *conn, const char *attribute_name);
```

The list of available attributes varies depending on the SSL library being used, and the type of connection. If an attribute is not available, returns NULL.

The following attributes are commonly available:

`library`

Name of the SSL implementation in use. (Currently, only `"OpenSSL"` is implemented)

`protocol`

SSL/TLS version in use. Common values are `"SSLv2"`, `"SSLv3"`, `"TLSv1"`, `"TLSv1.1"` and `"TLSv1.2"`, but an implementation may return other strings if some other protocol is used.

`key_bits`

Number of key bits used by the encryption algorithm.

`cipher`

A short name of the ciphersuite used, e.g., `"DHE-RSA-DES-CBC3-SHA"`. The names are specific to each SSL implementation.

`compression`

If SSL compression is in use, returns the name of the compression algorithm, or "on" if compression is used but the algorithm is not known. If compression is not in use, returns "off".

`PQsslAttributeNames`

Return an array of SSL attribute names available. The array is terminated by a NULL pointer.

```
const char * const * PQsslAttributeNames(const PGconn *conn);
```

`PQsslStruct`

Return a pointer to an SSL-implementation-specific object describing the connection.

```
void *PQsslStruct(const PGconn *conn, const char *struct_name);
```

The struct(s) available depend on the SSL implementation in use. For OpenSSL, there is one struct, available under the name "OpenSSL", and it returns a pointer to the OpenSSL `SSL` struct. To use this function, code along the following lines could be used:

```
#include <libpq-fe.h>
#include <openssl/ssl.h>

...

    SSL *ssl;

    dbconn = PQconnectdb(...);
    ...

    ssl = PQsslStruct(dbconn, "OpenSSL");
    if (ssl)
    {
        /* use OpenSSL functions to access ssl */
    }
```

This structure can be used to verify encryption levels, check server certificates, and more. Refer to the OpenSSL documentation for information about this structure.

`PQgetssl`

Returns the SSL structure used in the connection, or null if SSL is not in use.

```
void *PQgetssl(const PGconn *conn);
```

This function is equivalent to `PQsslStruct(conn, "OpenSSL")`. It should not be used in new applications, because the returned struct is specific to OpenSSL and will not be available if another SSL implementation is used. To check if a connection uses SSL, call `PQsslInUse` instead, and for more details about the connection, use `PQsslAttribute`.

31.3. Command Execution Functions

Once a connection to a database server has been successfully established, the functions described here are used to perform SQL queries and commands.

31.3.1. Main Functions

`PQexec`

Submits a command to the server and waits for the result.

```
PGresult *PQexec(PGconn *conn, const char *command);
```

Returns a `PGresult` pointer or possibly a null pointer. A non-null pointer will generally be returned except in out-of-memory conditions or serious errors such as inability to send the command to the server. The `PQresultStatus` function should be called to check the return value for any errors (including the value of a null pointer, in which case it will return `PGRES_FATAL_ERROR`). Use `PQerrorMessage` to get more information about such errors.

The command string can include multiple SQL commands (separated by semicolons). Multiple queries sent in a single `PQexec` call are processed in a single transaction, unless there are explicit `BEGIN`/`COMMIT` commands included in the query string to divide it into multiple transactions. Note however that the returned `PGresult` structure describes only the result of the last command executed from the string. Should one of the commands fail, processing of the string stops with it and the returned `PGresult` describes the error condition.

`PQexecParams`

Submits a command to the server and waits for the result, with the ability to pass parameters separately from the SQL command text.

```
PGresult *PQexecParams(PGconn *conn,
                       const char *command,
                       int nParams,
                       const Oid *paramTypes,
                       const char * const *paramValues,
                       const int *paramLengths,
                       const int *paramFormats,
                       int resultFormat);
```

`PQexecParams` is like `PQexec`, but offers additional functionality: parameter values can be specified separately from the command string proper, and query results can be requested in either text or binary

format. PQexecParams is supported only in protocol 3.0 and later connections; it will fail when using protocol 2.0.

The function arguments are:

conn

The connection object to send the command through.

command

The SQL command string to be executed. If parameters are used, they are referred to in the command string as $1, $2, etc.

nParams

The number of parameters supplied; it is the length of the arrays paramTypes[], paramValues[], paramLengths[], and paramFormats[]. (The array pointers can be NULL when nParams is zero.)

paramTypes[]

Specifies, by OID, the data types to be assigned to the parameter symbols. If paramTypes is NULL, or any particular element in the array is zero, the server infers a data type for the parameter symbol in the same way it would do for an untyped literal string.

paramValues[]

Specifies the actual values of the parameters. A null pointer in this array means the corresponding parameter is null; otherwise the pointer points to a zero-terminated text string (for text format) or binary data in the format expected by the server (for binary format).

paramLengths[]

Specifies the actual data lengths of binary-format parameters. It is ignored for null parameters and text-format parameters. The array pointer can be null when there are no binary parameters.

paramFormats[]

Specifies whether parameters are text (put a zero in the array entry for the corresponding parameter) or binary (put a one in the array entry for the corresponding parameter). If the array pointer is null then all parameters are presumed to be text strings.

Values passed in binary format require knowledge of the internal representation expected by the backend. For example, integers must be passed in network byte order. Passing numeric values requires knowledge of the server storage format, as implemented in src/backend/utils/adt/numeric.c::numeric_send() and src/backend/utils/adt/numeric.c::numeric_recv().

resultFormat

Specify zero to obtain results in text format, or one to obtain results in binary format. (There is not currently a provision to obtain different result columns in different formats, although that is possible in the underlying protocol.)

The primary advantage of PQexecParams over PQexec is that parameter values can be separated from the command string, thus avoiding the need for tedious and error-prone quoting and escaping.

Unlike PQexec, PQexecParams allows at most one SQL command in the given string. (There can be semicolons in it, but not more than one nonempty command.) This is a limitation of the underlying protocol, but has some usefulness as an extra defense against SQL-injection attacks.

> **Tip:** Specifying parameter types via OIDs is tedious, particularly if you prefer not to hard-wire particular OID values into your program. However, you can avoid doing so even in cases where the server by itself cannot determine the type of the parameter, or chooses a different type than you want. In the SQL command text, attach an explicit cast to the parameter symbol to show what data type you will send. For example:
>
> ```
> SELECT * FROM mytable WHERE x = $1::bigint;
> ```
>
> This forces parameter $1 to be treated as bigint, whereas by default it would be assigned the same type as x. Forcing the parameter type decision, either this way or by specifying a numeric type OID, is strongly recommended when sending parameter values in binary format, because binary format has less redundancy than text format and so there is less chance that the server will detect a type mismatch mistake for you.

PQprepare

Submits a request to create a prepared statement with the given parameters, and waits for completion.

```
PGresult *PQprepare(PGconn *conn,
                    const char *stmtName,
                    const char *query,
                    int nParams,
                    const Oid *paramTypes);
```

PQprepare creates a prepared statement for later execution with PQexecPrepared. This feature allows commands that will be used repeatedly to be parsed and planned just once, rather than each time they are executed. PQprepare is supported only in protocol 3.0 and later connections; it will fail when using protocol 2.0.

The function creates a prepared statement named stmtName from the query string, which must contain a single SQL command. stmtName can be "" to create an unnamed statement, in which case any pre-existing unnamed statement is automatically replaced; otherwise it is an error if the statement name is already defined in the current session. If any parameters are used, they are referred to in the query as $1, $2, etc. nParams is the number of parameters for which types are pre-specified in the array paramTypes[]. (The array pointer can be NULL when nParams is zero.) paramTypes[] specifies, by OID, the data types to be assigned to the parameter symbols. If paramTypes is NULL, or any particular element in the array is zero, the server assigns a data type to the parameter symbol in the same way it would do for an untyped literal string. Also, the query can use parameter symbols with numbers higher than nParams; data types will be inferred for these symbols as well. (See PQdescribePrepared for a means to find out what data types were inferred.)

As with PQexec, the result is normally a PGresult object whose contents indicate server-side success or failure. A null result indicates out-of-memory or inability to send the command at all. Use PQerrorMessage to get more information about such errors.

Prepared statements for use with PQexecPrepared can also be created by executing SQL PREPARE statements. Also, although there is no libpq function for deleting a prepared statement, the SQL DEALLOCATE statement can be used for that purpose.

PQexecPrepared

Sends a request to execute a prepared statement with given parameters, and waits for the result.

```
PGresult *PQexecPrepared(PGconn *conn,
                         const char *stmtName,
                         int nParams,
                         const char * const *paramValues,
                         const int *paramLengths,
                         const int *paramFormats,
                         int resultFormat);
```

PQexecPrepared is like PQexecParams, but the command to be executed is specified by naming a previously-prepared statement, instead of giving a query string. This feature allows commands that will be used repeatedly to be parsed and planned just once, rather than each time they are executed. The statement must have been prepared previously in the current session. PQexecPrepared is supported only in protocol 3.0 and later connections; it will fail when using protocol 2.0.

The parameters are identical to PQexecParams, except that the name of a prepared statement is given instead of a query string, and the paramTypes[] parameter is not present (it is not needed since the prepared statement's parameter types were determined when it was created).

PQdescribePrepared

Submits a request to obtain information about the specified prepared statement, and waits for completion.

```
PGresult *PQdescribePrepared(PGconn *conn, const char *stmtName);
```

PQdescribePrepared allows an application to obtain information about a previously prepared statement. PQdescribePrepared is supported only in protocol 3.0 and later connections; it will fail when using protocol 2.0.

stmtName can be "" or NULL to reference the unnamed statement, otherwise it must be the name of an existing prepared statement. On success, a PGresult with status PGRES_COMMAND_OK is returned. The functions PQnparams and PQparamtype can be applied to this PGresult to obtain information about the parameters of the prepared statement, and the functions PQnfields, PQfname, PQftype, etc provide information about the result columns (if any) of the statement.

PQdescribePortal

Submits a request to obtain information about the specified portal, and waits for completion.

```
PGresult *PQdescribePortal(PGconn *conn, const char *portalName);
```

PQdescribePortal allows an application to obtain information about a previously created portal. (libpq does not provide any direct access to portals, but you can use this function to inspect the properties of a cursor created with a DECLARE CURSOR SQL command.) PQdescribePortal is supported only in protocol 3.0 and later connections; it will fail when using protocol 2.0.

portalName can be "" or NULL to reference the unnamed portal, otherwise it must be the name of an existing portal. On success, a PGresult with status PGRES_COMMAND_OK is returned. The functions

PQnfields, PQfname, PQftype, etc can be applied to the PGresult to obtain information about the result columns (if any) of the portal.

The PGresult structure encapsulates the result returned by the server. libpq application programmers should be careful to maintain the PGresult abstraction. Use the accessor functions below to get at the contents of PGresult. Avoid directly referencing the fields of the PGresult structure because they are subject to change in the future.

PQresultStatus

Returns the result status of the command.

```
ExecStatusType PQresultStatus(const PGresult *res);
```

PQresultStatus can return one of the following values:

PGRES_EMPTY_QUERY

The string sent to the server was empty.

PGRES_COMMAND_OK

Successful completion of a command returning no data.

PGRES_TUPLES_OK

Successful completion of a command returning data (such as a SELECT or SHOW).

PGRES_COPY_OUT

Copy Out (from server) data transfer started.

PGRES_COPY_IN

Copy In (to server) data transfer started.

PGRES_BAD_RESPONSE

The server's response was not understood.

PGRES_NONFATAL_ERROR

A nonfatal error (a notice or warning) occurred.

PGRES_FATAL_ERROR

A fatal error occurred.

PGRES_COPY_BOTH

Copy In/Out (to and from server) data transfer started. This feature is currently used only for streaming replication, so this status should not occur in ordinary applications.

PGRES_SINGLE_TUPLE

The PGresult contains a single result tuple from the current command. This status occurs only when single-row mode has been selected for the query (see Section 31.5).

If the result status is PGRES_TUPLES_OK or PGRES_SINGLE_TUPLE, then the functions described below can be used to retrieve the rows returned by the query. Note that a SELECT command that happens to retrieve zero rows still shows PGRES_TUPLES_OK. PGRES_COMMAND_OK is for commands

that can never return rows (INSERT or UPDATE without a RETURNING clause, etc.). A response of PGRES_EMPTY_QUERY might indicate a bug in the client software.

A result of status PGRES_NONFATAL_ERROR will never be returned directly by PQexec or other query execution functions; results of this kind are instead passed to the notice processor (see Section 31.12).

PQresStatus

Converts the enumerated type returned by PQresultStatus into a string constant describing the status code. The caller should not free the result.

```
char *PQresStatus(ExecStatusType status);
```

PQresultErrorMessage

Returns the error message associated with the command, or an empty string if there was no error.

```
char *PQresultErrorMessage(const PGresult *res);
```

If there was an error, the returned string will include a trailing newline. The caller should not free the result directly. It will be freed when the associated PGresult handle is passed to PQclear.

Immediately following a PQexec or PQgetResult call, PQerrorMessage (on the connection) will return the same string as PQresultErrorMessage (on the result). However, a PGresult will retain its error message until destroyed, whereas the connection's error message will change when subsequent operations are done. Use PQresultErrorMessage when you want to know the status associated with a particular PGresult; use PQerrorMessage when you want to know the status from the latest operation on the connection.

PQresultErrorField

Returns an individual field of an error report.

```
char *PQresultErrorField(const PGresult *res, int fieldcode);
```

fieldcode is an error field identifier; see the symbols listed below. NULL is returned if the PGresult is not an error or warning result, or does not include the specified field. Field values will normally not include a trailing newline. The caller should not free the result directly. It will be freed when the associated PGresult handle is passed to PQclear.

The following field codes are available:

PG_DIAG_SEVERITY

The severity; the field contents are ERROR, FATAL, or PANIC (in an error message), or WARNING, NOTICE, DEBUG, INFO, or LOG (in a notice message), or a localized translation of one of these. Always present.

PG_DIAG_SQLSTATE

The SQLSTATE code for the error. The SQLSTATE code identifies the type of error that has occurred; it can be used by front-end applications to perform specific operations (such as error handling) in response to a particular database error. For a list of the possible SQLSTATE codes, see Appendix A. This field is not localizable, and is always present.

PG_DIAG_MESSAGE_PRIMARY

The primary human-readable error message (typically one line). Always present.

PG_DIAG_MESSAGE_DETAIL

Detail: an optional secondary error message carrying more detail about the problem. Might run to multiple lines.

PG_DIAG_MESSAGE_HINT

Hint: an optional suggestion what to do about the problem. This is intended to differ from detail in that it offers advice (potentially inappropriate) rather than hard facts. Might run to multiple lines.

PG_DIAG_STATEMENT_POSITION

A string containing a decimal integer indicating an error cursor position as an index into the original statement string. The first character has index 1, and positions are measured in characters not bytes.

PG_DIAG_INTERNAL_POSITION

This is defined the same as the PG_DIAG_STATEMENT_POSITION field, but it is used when the cursor position refers to an internally generated command rather than the one submitted by the client. The PG_DIAG_INTERNAL_QUERY field will always appear when this field appears.

PG_DIAG_INTERNAL_QUERY

The text of a failed internally-generated command. This could be, for example, a SQL query issued by a PL/pgSQL function.

PG_DIAG_CONTEXT

An indication of the context in which the error occurred. Presently this includes a call stack traceback of active procedural language functions and internally-generated queries. The trace is one entry per line, most recent first.

PG_DIAG_SCHEMA_NAME

If the error was associated with a specific database object, the name of the schema containing that object, if any.

PG_DIAG_TABLE_NAME

If the error was associated with a specific table, the name of the table. (Refer to the schema name field for the name of the table's schema.)

PG_DIAG_COLUMN_NAME

If the error was associated with a specific table column, the name of the column. (Refer to the schema and table name fields to identify the table.)

PG_DIAG_DATATYPE_NAME

If the error was associated with a specific data type, the name of the data type. (Refer to the schema name field for the name of the data type's schema.)

PG_DIAG_CONSTRAINT_NAME

If the error was associated with a specific constraint, the name of the constraint. Refer to fields listed above for the associated table or domain. (For this purpose, indexes are treated as constraints, even if they weren't created with constraint syntax.)

PG_DIAG_SOURCE_FILE

The file name of the source-code location where the error was reported.

PG_DIAG_SOURCE_LINE

The line number of the source-code location where the error was reported.

PG_DIAG_SOURCE_FUNCTION

The name of the source-code function reporting the error.

Note: The fields for schema name, table name, column name, data type name, and constraint name are supplied only for a limited number of error types; see Appendix A. Do not assume that the presence of any of these fields guarantees the presence of another field. Core error sources observe the interrelationships noted above, but user-defined functions may use these fields in other ways. In the same vein, do not assume that these fields denote contemporary objects in the current database.

The client is responsible for formatting displayed information to meet its needs; in particular it should break long lines as needed. Newline characters appearing in the error message fields should be treated as paragraph breaks, not line breaks.

Errors generated internally by libpq will have severity and primary message, but typically no other fields. Errors returned by a pre-3.0-protocol server will include severity and primary message, and sometimes a detail message, but no other fields.

Note that error fields are only available from PGresult objects, not PGconn objects; there is no PQerrorField function.

PQclear

Frees the storage associated with a PGresult. Every command result should be freed via PQclear when it is no longer needed.

```
void PQclear(PGresult *res);
```

You can keep a PGresult object around for as long as you need it; it does not go away when you issue a new command, nor even if you close the connection. To get rid of it, you must call PQclear. Failure to do this will result in memory leaks in your application.

31.3.2. Retrieving Query Result Information

These functions are used to extract information from a PGresult object that represents a successful query result (that is, one that has status PGRES_TUPLES_OK or PGRES_SINGLE_TUPLE). They can also be used to extract information from a successful Describe operation: a Describe's result has all the same column information that actual execution of the query would provide, but it has zero rows. For objects with other status values, these functions will act as though the result has zero rows and zero columns.

PQntuples

Returns the number of rows (tuples) in the query result. (Note that PGresult objects are limited to no more than INT_MAX rows, so an int result is sufficient.)

```
int PQntuples(const PGresult *res);
```

PQnfields

Returns the number of columns (fields) in each row of the query result.

```
int PQnfields(const PGresult *res);
```

PQfname

Returns the column name associated with the given column number. Column numbers start at 0. The caller should not free the result directly. It will be freed when the associated PGresult handle is passed to PQclear.

```
char *PQfname(const PGresult *res,
              int column_number);
```

NULL is returned if the column number is out of range.

PQfnumber

Returns the column number associated with the given column name.

```
int PQfnumber(const PGresult *res,
              const char *column_name);
```

-1 is returned if the given name does not match any column.

The given name is treated like an identifier in an SQL command, that is, it is downcased unless double-quoted. For example, given a query result generated from the SQL command:

```
SELECT 1 AS FOO, 2 AS "BAR";
```

we would have the results:

```
PQfname(res, 0)             foo
PQfname(res, 1)             BAR
PQfnumber(res, "FOO")       0
PQfnumber(res, "foo")       0
PQfnumber(res, "BAR")       -1
PQfnumber(res, "\"BAR\"")   1
```

PQftable

Returns the OID of the table from which the given column was fetched. Column numbers start at 0.

```
Oid PQftable(const PGresult *res,
             int column_number);
```

InvalidOid is returned if the column number is out of range, or if the specified column is not a simple reference to a table column, or when using pre-3.0 protocol. You can query the system table pg_class to determine exactly which table is referenced.

The type Oid and the constant InvalidOid will be defined when you include the libpq header file. They will both be some integer type.

`PQftablecol`

Returns the column number (within its table) of the column making up the specified query result column. Query-result column numbers start at 0, but table columns have nonzero numbers.

```
int PQftablecol(const PGresult *res,
                int column_number);
```

Zero is returned if the column number is out of range, or if the specified column is not a simple reference to a table column, or when using pre-3.0 protocol.

`PQfformat`

Returns the format code indicating the format of the given column. Column numbers start at 0.

```
int PQfformat(const PGresult *res,
              int column_number);
```

Format code zero indicates textual data representation, while format code one indicates binary representation. (Other codes are reserved for future definition.)

`PQftype`

Returns the data type associated with the given column number. The integer returned is the internal OID number of the type. Column numbers start at 0.

```
Oid PQftype(const PGresult *res,
            int column_number);
```

You can query the system table `pg_type` to obtain the names and properties of the various data types. The OIDs of the built-in data types are defined in the file `src/include/catalog/pg_type.h` in the source tree.

`PQfmod`

Returns the type modifier of the column associated with the given column number. Column numbers start at 0.

```
int PQfmod(const PGresult *res,
           int column_number);
```

The interpretation of modifier values is type-specific; they typically indicate precision or size limits. The value -1 is used to indicate "no information available". Most data types do not use modifiers, in which case the value is always -1.

`PQfsize`

Returns the size in bytes of the column associated with the given column number. Column numbers start at 0.

```
int PQfsize(const PGresult *res,
            int column_number);
```

`PQfsize` returns the space allocated for this column in a database row, in other words the size of the server's internal representation of the data type. (Accordingly, it is not really very useful to clients.) A negative value indicates the data type is variable-length.

`PQbinaryTuples`

Returns 1 if the `PGresult` contains binary data and 0 if it contains text data.

```
int PQbinaryTuples(const PGresult *res);
```

This function is deprecated (except for its use in connection with COPY), because it is possible for a single PGresult to contain text data in some columns and binary data in others. PQfformat is preferred. PQbinaryTuples returns 1 only if all columns of the result are binary (format 1).

PQgetvalue

Returns a single field value of one row of a PGresult. Row and column numbers start at 0. The caller should not free the result directly. It will be freed when the associated PGresult handle is passed to PQclear.

```
char *PQgetvalue(const PGresult *res,
                 int row_number,
                 int column_number);
```

For data in text format, the value returned by PQgetvalue is a null-terminated character string representation of the field value. For data in binary format, the value is in the binary representation determined by the data type's typsend and typreceive functions. (The value is actually followed by a zero byte in this case too, but that is not ordinarily useful, since the value is likely to contain embedded nulls.)

An empty string is returned if the field value is null. See PQgetisnull to distinguish null values from empty-string values.

The pointer returned by PQgetvalue points to storage that is part of the PGresult structure. One should not modify the data it points to, and one must explicitly copy the data into other storage if it is to be used past the lifetime of the PGresult structure itself.

PQgetisnull

Tests a field for a null value. Row and column numbers start at 0.

```
int PQgetisnull(const PGresult *res,
                int row_number,
                int column_number);
```

This function returns 1 if the field is null and 0 if it contains a non-null value. (Note that PQgetvalue will return an empty string, not a null pointer, for a null field.)

PQgetlength

Returns the actual length of a field value in bytes. Row and column numbers start at 0.

```
int PQgetlength(const PGresult *res,
                int row_number,
                int column_number);
```

This is the actual data length for the particular data value, that is, the size of the object pointed to by PQgetvalue. For text data format this is the same as strlen(). For binary format this is essential information. Note that one should *not* rely on PQfsize to obtain the actual data length.

PQnparams

Returns the number of parameters of a prepared statement.

```
int PQnparams(const PGresult *res);
```

This function is only useful when inspecting the result of PQdescribePrepared. For other types of queries it will return zero.

PQparamtype

Returns the data type of the indicated statement parameter. Parameter numbers start at 0.

```
Oid PQparamtype(const PGresult *res, int param_number);
```

This function is only useful when inspecting the result of PQdescribePrepared. For other types of queries it will return zero.

PQprint

Prints out all the rows and, optionally, the column names to the specified output stream.

```
void PQprint(FILE *fout,      /* output stream */
             const PGresult *res,
             const PQprintOpt *po);
typedef struct
{
    pqbool  header;      /* print output field headings and row count */
    pqbool  align;       /* fill align the fields */
    pqbool  standard;    /* old brain dead format */
    pqbool  html3;       /* output HTML tables */
    pqbool  expanded;    /* expand tables */
    pqbool  pager;       /* use pager for output if needed */
    char    *fieldSep;   /* field separator */
    char    *tableOpt;   /* attributes for HTML table element */
    char    *caption;    /* HTML table caption */
    char    **fieldName; /* null-terminated array of replacement field names */
} PQprintOpt;
```

This function was formerly used by psql to print query results, but this is no longer the case. Note that it assumes all the data is in text format.

31.3.3. Retrieving Other Result Information

These functions are used to extract other information from PGresult objects.

PQcmdStatus

Returns the command status tag from the SQL command that generated the PGresult.

```
char *PQcmdStatus(PGresult *res);
```

Commonly this is just the name of the command, but it might include additional data such as the number of rows processed. The caller should not free the result directly. It will be freed when the associated PGresult handle is passed to PQclear.

PQcmdTuples

Returns the number of rows affected by the SQL command.

```
char *PQcmdTuples(PGresult *res);
```

This function returns a string containing the number of rows affected by the SQL statement that generated the PGresult. This function can only be used following the execution of a SELECT, CREATE TABLE AS, INSERT, UPDATE, DELETE, MOVE, FETCH, or COPY statement, or an EXECUTE of a prepared query that contains an INSERT, UPDATE, or DELETE statement. If the command that

generated the PGresult was anything else, PQcmdTuples returns an empty string. The caller should not free the return value directly. It will be freed when the associated PGresult handle is passed to PQclear.

PQoidValue

Returns the OID of the inserted row, if the SQL command was an INSERT that inserted exactly one row into a table that has OIDs, or a EXECUTE of a prepared query containing a suitable INSERT statement. Otherwise, this function returns InvalidOid. This function will also return InvalidOid if the table affected by the INSERT statement does not contain OIDs.

```
Oid PQoidValue(const PGresult *res);
```

PQoidStatus

This function is deprecated in favor of PQoidValue and is not thread-safe. It returns a string with the OID of the inserted row, while PQoidValue returns the OID value.

```
char *PQoidStatus(const PGresult *res);
```

31.3.4. Escaping Strings for Inclusion in SQL Commands

PQescapeLiteral

```
char *PQescapeLiteral(PGconn *conn, const char *str, size_t length);
```

PQescapeLiteral escapes a string for use within an SQL command. This is useful when inserting data values as literal constants in SQL commands. Certain characters (such as quotes and backslashes) must be escaped to prevent them from being interpreted specially by the SQL parser. PQescapeLiteral performs this operation.

PQescapeLiteral returns an escaped version of the str parameter in memory allocated with malloc(). This memory should be freed using PQfreemem() when the result is no longer needed. A terminating zero byte is not required, and should not be counted in length. (If a terminating zero byte is found before length bytes are processed, PQescapeLiteral stops at the zero; the behavior is thus rather like strncpy.) The return string has all special characters replaced so that they can be properly processed by the PostgreSQL string literal parser. A terminating zero byte is also added. The single quotes that must surround PostgreSQL string literals are included in the result string.

On error, PQescapeLiteral returns NULL and a suitable message is stored in the conn object.

> **Tip:** It is especially important to do proper escaping when handling strings that were received from an untrustworthy source. Otherwise there is a security risk: you are vulnerable to "SQL injection" attacks wherein unwanted SQL commands are fed to your database.

Note that it is neither necessary nor correct to do escaping when a data value is passed as a separate parameter in PQexecParams or its sibling routines.

PQescapeIdentifier

```
char *PQescapeIdentifier(PGconn *conn, const char *str, size_t length);
```

PQescapeIdentifier escapes a string for use as an SQL identifier, such as a table, column, or function name. This is useful when a user-supplied identifier might contain special characters that would otherwise not be interpreted as part of the identifier by the SQL parser, or when the identifier might contain upper case characters whose case should be preserved.

PQescapeIdentifier returns a version of the str parameter escaped as an SQL identifier in memory allocated with malloc(). This memory must be freed using PQfreemem() when the result is no longer needed. A terminating zero byte is not required, and should not be counted in length. (If a terminating zero byte is found before length bytes are processed, PQescapeIdentifier stops at the zero; the behavior is thus rather like strncpy.) The return string has all special characters replaced so that it will be properly processed as an SQL identifier. A terminating zero byte is also added. The return string will also be surrounded by double quotes.

On error, PQescapeIdentifier returns NULL and a suitable message is stored in the conn object.

> **Tip:** As with string literals, to prevent SQL injection attacks, SQL identifiers must be escaped when they are received from an untrustworthy source.

PQescapeStringConn

```
size_t PQescapeStringConn(PGconn *conn,
                          char *to, const char *from, size_t length,
                          int *error);
```

PQescapeStringConn escapes string literals, much like PQescapeLiteral. Unlike PQescapeLiteral, the caller is responsible for providing an appropriately sized buffer. Furthermore, PQescapeStringConn does not generate the single quotes that must surround PostgreSQL string literals; they should be provided in the SQL command that the result is inserted into. The parameter from points to the first character of the string that is to be escaped, and the length parameter gives the number of bytes in this string. A terminating zero byte is not required, and should not be counted in length. (If a terminating zero byte is found before length bytes are processed, PQescapeStringConn stops at the zero; the behavior is thus rather like strncpy.) to shall point to a buffer that is able to hold at least one more byte than twice the value of length, otherwise the behavior is undefined. Behavior is likewise undefined if the to and from strings overlap.

If the error parameter is not NULL, then *error is set to zero on success, nonzero on error. Presently the only possible error conditions involve invalid multibyte encoding in the source string. The output string is still generated on error, but it can be expected that the server will reject it as malformed. On error, a suitable message is stored in the conn object, whether or not error is NULL.

PQescapeStringConn returns the number of bytes written to to, not including the terminating zero byte.

PQescapeString

PQescapeString is an older, deprecated version of PQescapeStringConn.

```
size_t PQescapeString (char *to, const char *from, size_t length);
```

The only difference from PQescapeStringConn is that PQescapeString does not take PGconn or error parameters. Because of this, it cannot adjust its behavior depending on the connection

properties (such as character encoding) and therefore *it might give the wrong results*. Also, it has no way to report error conditions.

`PQescapeString` can be used safely in client programs that work with only one PostgreSQL connection at a time (in this case it can find out what it needs to know "behind the scenes"). In other contexts it is a security hazard and should be avoided in favor of `PQescapeStringConn`.

`PQescapeByteaConn`

Escapes binary data for use within an SQL command with the type `bytea`. As with `PQescapeStringConn`, this is only used when inserting data directly into an SQL command string.

```
unsigned char *PQescapeByteaConn(PGconn *conn,
                                 const unsigned char *from,
                                 size_t from_length,
                                 size_t *to_length);
```

Certain byte values must be escaped when used as part of a `bytea` literal in an SQL statement. `PQescapeByteaConn` escapes bytes using either hex encoding or backslash escaping. See Section 8.4 for more information.

The `from` parameter points to the first byte of the string that is to be escaped, and the `from_length` parameter gives the number of bytes in this binary string. (A terminating zero byte is neither necessary nor counted.) The `to_length` parameter points to a variable that will hold the resultant escaped string length. This result string length includes the terminating zero byte of the result.

`PQescapeByteaConn` returns an escaped version of the `from` parameter binary string in memory allocated with `malloc()`. This memory should be freed using `PQfreemem()` when the result is no longer needed. The return string has all special characters replaced so that they can be properly processed by the PostgreSQL string literal parser, and the `bytea` input function. A terminating zero byte is also added. The single quotes that must surround PostgreSQL string literals are not part of the result string.

On error, a null pointer is returned, and a suitable error message is stored in the `conn` object. Currently, the only possible error is insufficient memory for the result string.

`PQescapeBytea`

`PQescapeBytea` is an older, deprecated version of `PQescapeByteaConn`.

```
unsigned char *PQescapeBytea(const unsigned char *from,
                             size_t from_length,
                             size_t *to_length);
```

The only difference from `PQescapeByteaConn` is that `PQescapeBytea` does not take a `PGconn` parameter. Because of this, `PQescapeBytea` can only be used safely in client programs that use a single PostgreSQL connection at a time (in this case it can find out what it needs to know "behind the scenes"). It *might give the wrong results* if used in programs that use multiple database connections (use `PQescapeByteaConn` in such cases).

`PQunescapeBytea`

Converts a string representation of binary data into binary data — the reverse of `PQescapeBytea`. This is needed when retrieving `bytea` data in text format, but not when retrieving it in binary format.

```
unsigned char *PQunescapeBytea(const unsigned char *from, size_t *to_length);
```

The from parameter points to a string such as might be returned by PQgetvalue when applied to a bytea column. PQunescapeBytea converts this string representation into its binary representation. It returns a pointer to a buffer allocated with malloc(), or NULL on error, and puts the size of the buffer in to_length. The result must be freed using PQfreemem when it is no longer needed.

This conversion is not exactly the inverse of PQescapeBytea, because the string is not expected to be "escaped" when received from PQgetvalue. In particular this means there is no need for string quoting considerations, and so no need for a PGconn parameter.

31.4. Asynchronous Command Processing

The PQexec function is adequate for submitting commands in normal, synchronous applications. It has a few deficiencies, however, that can be of importance to some users:

- PQexec waits for the command to be completed. The application might have other work to do (such as maintaining a user interface), in which case it won't want to block waiting for the response.
- Since the execution of the client application is suspended while it waits for the result, it is hard for the application to decide that it would like to try to cancel the ongoing command. (It can be done from a signal handler, but not otherwise.)
- PQexec can return only one PGresult structure. If the submitted command string contains multiple SQL commands, all but the last PGresult are discarded by PQexec.
- PQexec always collects the command's entire result, buffering it in a single PGresult. While this simplifies error-handling logic for the application, it can be impractical for results containing many rows.

Applications that do not like these limitations can instead use the underlying functions that PQexec is built from: PQsendQuery and PQgetResult. There are also PQsendQueryParams, PQsendPrepare, PQsendQueryPrepared, PQsendDescribePrepared, and PQsendDescribePortal, which can be used with PQgetResult to duplicate the functionality of PQexecParams, PQprepare, PQexecPrepared, PQdescribePrepared, and PQdescribePortal respectively.

PQsendQuery

Submits a command to the server without waiting for the result(s). 1 is returned if the command was successfully dispatched and 0 if not (in which case, use PQerrorMessage to get more information about the failure).

```
int PQsendQuery(PGconn *conn, const char *command);
```

After successfully calling PQsendQuery, call PQgetResult one or more times to obtain the results. PQsendQuery cannot be called again (on the same connection) until PQgetResult has returned a null pointer, indicating that the command is done.

PQsendQueryParams

Submits a command and separate parameters to the server without waiting for the result(s).

```
int PQsendQueryParams(PGconn *conn,
```

```
                        const char *command,
                        int nParams,
                        const Oid *paramTypes,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

This is equivalent to `PQsendQuery` except that query parameters can be specified separately from the query string. The function's parameters are handled identically to `PQexecParams`. Like `PQexecParams`, it will not work on 2.0-protocol connections, and it allows only one command in the query string.

`PQsendPrepare`

Sends a request to create a prepared statement with the given parameters, without waiting for completion.

```
int PQsendPrepare(PGconn *conn,
                  const char *stmtName,
                  const char *query,
                  int nParams,
                  const Oid *paramTypes);
```

This is an asynchronous version of `PQprepare`: it returns 1 if it was able to dispatch the request, and 0 if not. After a successful call, call `PQgetResult` to determine whether the server successfully created the prepared statement. The function's parameters are handled identically to `PQprepare`. Like `PQprepare`, it will not work on 2.0-protocol connections.

`PQsendQueryPrepared`

Sends a request to execute a prepared statement with given parameters, without waiting for the result(s).

```
int PQsendQueryPrepared(PGconn *conn,
                        const char *stmtName,
                        int nParams,
                        const char * const *paramValues,
                        const int *paramLengths,
                        const int *paramFormats,
                        int resultFormat);
```

This is similar to `PQsendQueryParams`, but the command to be executed is specified by naming a previously-prepared statement, instead of giving a query string. The function's parameters are handled identically to `PQexecPrepared`. Like `PQexecPrepared`, it will not work on 2.0-protocol connections.

`PQsendDescribePrepared`

Submits a request to obtain information about the specified prepared statement, without waiting for completion.

```
int PQsendDescribePrepared(PGconn *conn, const char *stmtName);
```

This is an asynchronous version of `PQdescribePrepared`: it returns 1 if it was able to dispatch the request, and 0 if not. After a successful call, call `PQgetResult` to obtain the results. The function's parameters are handled identically to `PQdescribePrepared`. Like `PQdescribePrepared`, it will not work on 2.0-protocol connections.

PQsendDescribePortal

Submits a request to obtain information about the specified portal, without waiting for completion.

```
int PQsendDescribePortal(PGconn *conn, const char *portalName);
```

This is an asynchronous version of PQdescribePortal: it returns 1 if it was able to dispatch the request, and 0 if not. After a successful call, call PQgetResult to obtain the results. The function's parameters are handled identically to PQdescribePortal. Like PQdescribePortal, it will not work on 2.0-protocol connections.

PQgetResult

Waits for the next result from a prior PQsendQuery, PQsendQueryParams, PQsendPrepare, PQsendQueryPrepared, PQsendDescribePrepared, or PQsendDescribePortal call, and returns it. A null pointer is returned when the command is complete and there will be no more results.

```
PGresult *PQgetResult(PGconn *conn);
```

PQgetResult must be called repeatedly until it returns a null pointer, indicating that the command is done. (If called when no command is active, PQgetResult will just return a null pointer at once.) Each non-null result from PQgetResult should be processed using the same PGresult accessor functions previously described. Don't forget to free each result object with PQclear when done with it. Note that PQgetResult will block only if a command is active and the necessary response data has not yet been read by PQconsumeInput.

> **Note:** Even when PQresultStatus indicates a fatal error, PQgetResult should be called until it returns a null pointer, to allow libpq to process the error information completely.

Using PQsendQuery and PQgetResult solves one of PQexec's problems: If a command string contains multiple SQL commands, the results of those commands can be obtained individually. (This allows a simple form of overlapped processing, by the way: the client can be handling the results of one command while the server is still working on later queries in the same command string.)

Another frequently-desired feature that can be obtained with PQsendQuery and PQgetResult is retrieving large query results a row at a time. This is discussed in Section 31.5.

By itself, calling PQgetResult will still cause the client to block until the server completes the next SQL command. This can be avoided by proper use of two more functions:

PQconsumeInput

If input is available from the server, consume it.

```
int PQconsumeInput(PGconn *conn);
```

PQconsumeInput normally returns 1 indicating "no error", but returns 0 if there was some kind of trouble (in which case PQerrorMessage can be consulted). Note that the result does not say whether any input data was actually collected. After calling PQconsumeInput, the application can check PQisBusy and/or PQnotifies to see if their state has changed.

PQconsumeInput can be called even if the application is not prepared to deal with a result or notification just yet. The function will read available data and save it in a buffer, thereby causing a

select() read-ready indication to go away. The application can thus use PQconsumeInput to clear the select() condition immediately, and then examine the results at leisure.

PQisBusy

Returns 1 if a command is busy, that is, PQgetResult would block waiting for input. A 0 return indicates that PQgetResult can be called with assurance of not blocking.

```
int PQisBusy(PGconn *conn);
```

PQisBusy will not itself attempt to read data from the server; therefore PQconsumeInput must be invoked first, or the busy state will never end.

A typical application using these functions will have a main loop that uses select() or poll() to wait for all the conditions that it must respond to. One of the conditions will be input available from the server, which in terms of select() means readable data on the file descriptor identified by PQsocket. When the main loop detects input ready, it should call PQconsumeInput to read the input. It can then call PQisBusy, followed by PQgetResult if PQisBusy returns false (0). It can also call PQnotifies to detect NOTIFY messages (see Section 31.8).

A client that uses PQsendQuery/PQgetResult can also attempt to cancel a command that is still being processed by the server; see Section 31.6. But regardless of the return value of PQcancel, the application must continue with the normal result-reading sequence using PQgetResult. A successful cancellation will simply cause the command to terminate sooner than it would have otherwise.

By using the functions described above, it is possible to avoid blocking while waiting for input from the database server. However, it is still possible that the application will block waiting to send output to the server. This is relatively uncommon but can happen if very long SQL commands or data values are sent. (It is much more probable if the application sends data via COPY IN, however.) To prevent this possibility and achieve completely nonblocking database operation, the following additional functions can be used.

PQsetnonblocking

Sets the nonblocking status of the connection.

```
int PQsetnonblocking(PGconn *conn, int arg);
```

Sets the state of the connection to nonblocking if arg is 1, or blocking if arg is 0. Returns 0 if OK, -1 if error.

In the nonblocking state, calls to PQsendQuery, PQputline, PQputnbytes, PQputCopyData, and PQendcopy will not block but instead return an error if they need to be called again.

Note that PQexec does not honor nonblocking mode; if it is called, it will act in blocking fashion anyway.

PQisnonblocking

Returns the blocking status of the database connection.

```
int PQisnonblocking(const PGconn *conn);
```

Returns 1 if the connection is set to nonblocking mode and 0 if blocking.

PQflush

Attempts to flush any queued output data to the server. Returns 0 if successful (or if the send queue is empty), -1 if it failed for some reason, or 1 if it was unable to send all the data in the send queue yet (this case can only occur if the connection is nonblocking).

```
int PQflush(PGconn *conn);
```

After sending any command or data on a nonblocking connection, call PQflush. If it returns 1, wait for the socket to become read- or write-ready. If it becomes write-ready, call PQflush again. If it becomes read-ready, call PQconsumeInput, then call PQflush again. Repeat until PQflush returns 0. (It is necessary to check for read-ready and drain the input with PQconsumeInput, because the server can block trying to send us data, e.g., NOTICE messages, and won't read our data until we read its.) Once PQflush returns 0, wait for the socket to be read-ready and then read the response as described above.

31.5. Retrieving Query Results Row-By-Row

Ordinarily, libpq collects a SQL command's entire result and returns it to the application as a single PGresult. This can be unworkable for commands that return a large number of rows. For such cases, applications can use PQsendQuery and PQgetResult in *single-row mode*. In this mode, the result row(s) are returned to the application one at a time, as they are received from the server.

To enter single-row mode, call PQsetSingleRowMode immediately after a successful call of PQsendQuery (or a sibling function). This mode selection is effective only for the currently executing query. Then call PQgetResult repeatedly, until it returns null, as documented in Section 31.4. If the query returns any rows, they are returned as individual PGresult objects, which look like normal query results except for having status code PGRES_SINGLE_TUPLE instead of PGRES_TUPLES_OK. After the last row, or immediately if the query returns zero rows, a zero-row object with status PGRES_TUPLES_OK is returned; this is the signal that no more rows will arrive. (But note that it is still necessary to continue calling PQgetResult until it returns null.) All of these PGresult objects will contain the same row description data (column names, types, etc) that an ordinary PGresult object for the query would have. Each object should be freed with PQclear as usual.

PQsetSingleRowMode

Select single-row mode for the currently-executing query.

```
int PQsetSingleRowMode(PGconn *conn);
```

This function can only be called immediately after PQsendQuery or one of its sibling functions, before any other operation on the connection such as PQconsumeInput or PQgetResult. If called at the correct time, the function activates single-row mode for the current query and returns 1. Otherwise the mode stays unchanged and the function returns 0. In any case, the mode reverts to normal after completion of the current query.

Caution

While processing a query, the server may return some rows and then encounter an error, causing the query to be aborted. Ordinarily, libpq discards any such rows and reports only the error. But in single-row mode, those rows will have already been returned to the application. Hence, the application will see some `PGRES_SINGLE_TUPLE` `PGresult` objects followed by a `PGRES_FATAL_ERROR` object. For proper transactional behavior, the application must be designed to discard or undo whatever has been done with the previously-processed rows, if the query ultimately fails.

31.6. Canceling Queries in Progress

A client application can request cancellation of a command that is still being processed by the server, using the functions described in this section.

`PQgetCancel`

Creates a data structure containing the information needed to cancel a command issued through a particular database connection.

```
PGcancel *PQgetCancel(PGconn *conn);
```

`PQgetCancel` creates a `PGcancel` object given a `PGconn` connection object. It will return `NULL` if the given `conn` is `NULL` or an invalid connection. The `PGcancel` object is an opaque structure that is not meant to be accessed directly by the application; it can only be passed to `PQcancel` or `PQfreeCancel`.

`PQfreeCancel`

Frees a data structure created by `PQgetCancel`.

```
void PQfreeCancel(PGcancel *cancel);
```

`PQfreeCancel` frees a data object previously created by `PQgetCancel`.

`PQcancel`

Requests that the server abandon processing of the current command.

```
int PQcancel(PGcancel *cancel, char *errbuf, int errbufsize);
```

The return value is 1 if the cancel request was successfully dispatched and 0 if not. If not, `errbuf` is filled with an explanatory error message. `errbuf` must be a char array of size `errbufsize` (the recommended size is 256 bytes).

Successful dispatch is no guarantee that the request will have any effect, however. If the cancellation is effective, the current command will terminate early and return an error result. If the cancellation fails (say, because the server was already done processing the command), then there will be no visible result at all.

`PQcancel` can safely be invoked from a signal handler, if the `errbuf` is a local variable in the signal handler. The `PGcancel` object is read-only as far as `PQcancel` is concerned, so it can also be invoked from a thread that is separate from the one manipulating the `PGconn` object.

`PQrequestCancel`

`PQrequestCancel` is a deprecated variant of `PQcancel`.

```
int PQrequestCancel(PGconn *conn);
```

Requests that the server abandon processing of the current command. It operates directly on the `PGconn` object, and in case of failure stores the error message in the `PGconn` object (whence it can be retrieved by `PQerrorMessage`). Although the functionality is the same, this approach creates hazards for multiple-thread programs and signal handlers, since it is possible that overwriting the `PGconn`'s error message will mess up the operation currently in progress on the connection.

31.7. The Fast-Path Interface

PostgreSQL provides a fast-path interface to send simple function calls to the server.

> **Tip:** This interface is somewhat obsolete, as one can achieve similar performance and greater functionality by setting up a prepared statement to define the function call. Then, executing the statement with binary transmission of parameters and results substitutes for a fast-path function call.

The function `PQfn` requests execution of a server function via the fast-path interface:

```
PGresult *PQfn(PGconn *conn,
               int fnid,
               int *result_buf,
               int *result_len,
               int result_is_int,
               const PQArgBlock *args,
               int nargs);

typedef struct
{
    int len;
    int isint;
    union
    {
        int *ptr;
        int integer;
    } u;
} PQArgBlock;
```

The `fnid` argument is the OID of the function to be executed. `args` and `nargs` define the parameters to be passed to the function; they must match the declared function argument list. When the `isint` field of a parameter structure is true, the `u.integer` value is sent to the server as an integer of the indicated length (this must be 2 or 4 bytes); proper byte-swapping occurs. When `isint` is false, the indicated number of bytes at `*u.ptr` are sent with no processing; the data must be in the format expected by the server for binary transmission of the function's argument data type. (The declaration of `u.ptr` as being of type

int * is historical; it would be better to consider it void *.) result_buf points to the buffer in which to place the function's return value. The caller must have allocated sufficient space to store the return value. (There is no check!) The actual result length in bytes will be returned in the integer pointed to by result_len. If a 2- or 4-byte integer result is expected, set result_is_int to 1, otherwise set it to 0. Setting result_is_int to 1 causes libpq to byte-swap the value if necessary, so that it is delivered as a proper int value for the client machine; note that a 4-byte integer is delivered into *result_buf for either allowed result size. When result_is_int is 0, the binary-format byte string sent by the server is returned unmodified. (In this case it's better to consider result_buf as being of type void *.)

PQfn always returns a valid PGresult pointer. The result status should be checked before the result is used. The caller is responsible for freeing the PGresult with PQclear when it is no longer needed.

Note that it is not possible to handle null arguments, null results, nor set-valued results when using this interface.

31.8. Asynchronous Notification

PostgreSQL offers asynchronous notification via the LISTEN and NOTIFY commands. A client session registers its interest in a particular notification channel with the LISTEN command (and can stop listening with the UNLISTEN command). All sessions listening on a particular channel will be notified asynchronously when a NOTIFY command with that channel name is executed by any session. A "payload" string can be passed to communicate additional data to the listeners.

libpq applications submit LISTEN, UNLISTEN, and NOTIFY commands as ordinary SQL commands. The arrival of NOTIFY messages can subsequently be detected by calling PQnotifies.

The function PQnotifies returns the next notification from a list of unhandled notification messages received from the server. It returns a null pointer if there are no pending notifications. Once a notification is returned from PQnotifies, it is considered handled and will be removed from the list of notifications.

```
PGnotify *PQnotifies(PGconn *conn);

typedef struct pgNotify
{
    char *relname;              /* notification channel name */
    int  be_pid;                /* process ID of notifying server process */
    char *extra;                /* notification payload string */
} PGnotify;
```

After processing a PGnotify object returned by PQnotifies, be sure to free it with PQfreemem. It is sufficient to free the PGnotify pointer; the relname and extra fields do not represent separate allocations. (The names of these fields are historical; in particular, channel names need not have anything to do with relation names.)

Example 31-2 gives a sample program that illustrates the use of asynchronous notification.

PQnotifies does not actually read data from the server; it just returns messages previously absorbed by another libpq function. In ancient releases of libpq, the only way to ensure timely receipt of NOTIFY messages was to constantly submit commands, even empty ones, and then check PQnotifies after each PQexec. While this still works, it is deprecated as a waste of processing power.

A better way to check for NOTIFY messages when you have no useful commands to execute is to call PQconsumeInput, then check PQnotifies. You can use select() to wait for data to arrive from the server, thereby using no CPU power unless there is something to do. (See PQsocket to obtain the file descriptor number to use with select().) Note that this will work OK whether you submit commands with PQsendQuery/PQgetResult or simply use PQexec. You should, however, remember to check PQnotifies after each PQgetResult or PQexec, to see if any notifications came in during the processing of the command.

31.9. Functions Associated with the COPY Command

The COPY command in PostgreSQL has options to read from or write to the network connection used by libpq. The functions described in this section allow applications to take advantage of this capability by supplying or consuming copied data.

The overall process is that the application first issues the SQL COPY command via PQexec or one of the equivalent functions. The response to this (if there is no error in the command) will be a PGresult object bearing a status code of PGRES_COPY_OUT or PGRES_COPY_IN (depending on the specified copy direction). The application should then use the functions of this section to receive or transmit data rows. When the data transfer is complete, another PGresult object is returned to indicate success or failure of the transfer. Its status will be PGRES_COMMAND_OK for success or PGRES_FATAL_ERROR if some problem was encountered. At this point further SQL commands can be issued via PQexec. (It is not possible to execute other SQL commands using the same connection while the COPY operation is in progress.)

If a COPY command is issued via PQexec in a string that could contain additional commands, the application must continue fetching results via PQgetResult after completing the COPY sequence. Only when PQgetResult returns NULL is it certain that the PQexec command string is done and it is safe to issue more commands.

The functions of this section should be executed only after obtaining a result status of PGRES_COPY_OUT or PGRES_COPY_IN from PQexec or PQgetResult.

A PGresult object bearing one of these status values carries some additional data about the COPY operation that is starting. This additional data is available using functions that are also used in connection with query results:

PQnfields

Returns the number of columns (fields) to be copied.

PQbinaryTuples

0 indicates the overall copy format is textual (rows separated by newlines, columns separated by separator characters, etc). 1 indicates the overall copy format is binary. See COPY for more information.

PQfformat

Returns the format code (0 for text, 1 for binary) associated with each column of the copy operation. The per-column format codes will always be zero when the overall copy format is textual, but the binary format can support both text and binary columns. (However, as of the current implementation of COPY, only binary columns appear in a binary copy; so the per-column formats always match the overall format at present.)

Note: These additional data values are only available when using protocol 3.0. When using protocol 2.0, all these functions will return 0.

31.9.1. Functions for Sending COPY Data

These functions are used to send data during COPY FROM STDIN. They will fail if called when the connection is not in COPY_IN state.

PQputCopyData

Sends data to the server during COPY_IN state.

```
int PQputCopyData(PGconn *conn,
                  const char *buffer,
                  int nbytes);
```

Transmits the COPY data in the specified buffer, of length nbytes, to the server. The result is 1 if the data was queued, zero if it was not queued because of full buffers (this will only happen in nonblocking mode), or -1 if an error occurred. (Use PQerrorMessage to retrieve details if the return value is -1. If the value is zero, wait for write-ready and try again.)

The application can divide the COPY data stream into buffer loads of any convenient size. Buffer-load boundaries have no semantic significance when sending. The contents of the data stream must match the data format expected by the COPY command; see COPY for details.

PQputCopyEnd

Sends end-of-data indication to the server during COPY_IN state.

```
int PQputCopyEnd(PGconn *conn,
                 const char *errormsg);
```

Ends the COPY_IN operation successfully if errormsg is NULL. If errormsg is not NULL then the COPY is forced to fail, with the string pointed to by errormsg used as the error message. (One should not assume that this exact error message will come back from the server, however, as the server might have already failed the COPY for its own reasons. Also note that the option to force failure does not work when using pre-3.0-protocol connections.)

The result is 1 if the termination message was sent; or in nonblocking mode, this may only indicate that the termination message was successfully queued. (In nonblocking mode, to be certain that the data has been sent, you should next wait for write-ready and call PQflush, repeating until it returns zero.) Zero indicates that the function could not queue the termination message because of full buffers; this will only happen in nonblocking mode. (In this case, wait for write-ready and try the PQputCopyEnd call again.) If a hard error occurs, -1 is returned; you can use PQerrorMessage to retrieve details.

After successfully calling PQputCopyEnd, call PQgetResult to obtain the final result status of the COPY command. One can wait for this result to be available in the usual way. Then return to normal operation.

31.9.2. Functions for Receiving COPY Data

These functions are used to receive data during COPY TO STDOUT. They will fail if called when the connection is not in COPY_OUT state.

PQgetCopyData

Receives data from the server during COPY_OUT state.

```
int PQgetCopyData(PGconn *conn,
                  char **buffer,
                  int async);
```

Attempts to obtain another row of data from the server during a COPY. Data is always returned one data row at a time; if only a partial row is available, it is not returned. Successful return of a data row involves allocating a chunk of memory to hold the data. The buffer parameter must be non-NULL. *buffer is set to point to the allocated memory, or to NULL in cases where no buffer is returned. A non-NULL result buffer should be freed using PQfreemem when no longer needed.

When a row is successfully returned, the return value is the number of data bytes in the row (this will always be greater than zero). The returned string is always null-terminated, though this is probably only useful for textual COPY. A result of zero indicates that the COPY is still in progress, but no row is yet available (this is only possible when async is true). A result of -1 indicates that the COPY is done. A result of -2 indicates that an error occurred (consult PQerrorMessage for the reason).

When async is true (not zero), PQgetCopyData will not block waiting for input; it will return zero if the COPY is still in progress but no complete row is available. (In this case wait for read-ready and then call PQconsumeInput before calling PQgetCopyData again.) When async is false (zero), PQgetCopyData will block until data is available or the operation completes.

After PQgetCopyData returns -1, call PQgetResult to obtain the final result status of the COPY command. One can wait for this result to be available in the usual way. Then return to normal operation.

31.9.3. Obsolete Functions for COPY

These functions represent older methods of handling COPY. Although they still work, they are deprecated due to poor error handling, inconvenient methods of detecting end-of-data, and lack of support for binary or nonblocking transfers.

PQgetline

Reads a newline-terminated line of characters (transmitted by the server) into a buffer string of size length.

```
int PQgetline(PGconn *conn,
              char *buffer,
              int length);
```

This function copies up to length-1 characters into the buffer and converts the terminating newline into a zero byte. PQgetline returns EOF at the end of input, 0 if the entire line has been read, and 1 if the buffer is full but the terminating newline has not yet been read.

Note that the application must check to see if a new line consists of the two characters \., which indicates that the server has finished sending the results of the COPY command. If the application might receive lines that are more than length-1 characters long, care is needed to be sure it recognizes the \. line correctly (and does not, for example, mistake the end of a long data line for a terminator line).

PQgetlineAsync

Reads a row of COPY data (transmitted by the server) into a buffer without blocking.

```
int PQgetlineAsync(PGconn *conn,
                   char *buffer,
                   int bufsize);
```

This function is similar to PQgetline, but it can be used by applications that must read COPY data asynchronously, that is, without blocking. Having issued the COPY command and gotten a PGRES_COPY_OUT response, the application should call PQconsumeInput and PQgetlineAsync until the end-of-data signal is detected.

Unlike PQgetline, this function takes responsibility for detecting end-of-data.

On each call, PQgetlineAsync will return data if a complete data row is available in libpq's input buffer. Otherwise, no data is returned until the rest of the row arrives. The function returns -1 if the end-of-copy-data marker has been recognized, or 0 if no data is available, or a positive number giving the number of bytes of data returned. If -1 is returned, the caller must next call PQendcopy, and then return to normal processing.

The data returned will not extend beyond a data-row boundary. If possible a whole row will be returned at one time. But if the buffer offered by the caller is too small to hold a row sent by the server, then a partial data row will be returned. With textual data this can be detected by testing whether the last returned byte is \n or not. (In a binary COPY, actual parsing of the COPY data format will be needed to make the equivalent determination.) The returned string is not null-terminated. (If you want to add a terminating null, be sure to pass a bufsize one smaller than the room actually available.)

PQputline

Sends a null-terminated string to the server. Returns 0 if OK and EOF if unable to send the string.

```
int PQputline(PGconn *conn,
              const char *string);
```

The COPY data stream sent by a series of calls to PQputline has the same format as that returned by PQgetlineAsync, except that applications are not obliged to send exactly one data row per PQputline call; it is okay to send a partial line or multiple lines per call.

> **Note:** Before PostgreSQL protocol 3.0, it was necessary for the application to explicitly send the two characters \. as a final line to indicate to the server that it had finished sending COPY data. While this still works, it is deprecated and the special meaning of \. can be expected to be removed in a future release. It is sufficient to call PQendcopy after having sent the actual data.

`PQputnbytes`

Sends a non-null-terminated string to the server. Returns 0 if OK and `EOF` if unable to send the string.

```
int PQputnbytes(PGconn *conn,
                const char *buffer,
                int nbytes);
```

This is exactly like `PQputline`, except that the data buffer need not be null-terminated since the number of bytes to send is specified directly. Use this procedure when sending binary data.

`PQendcopy`

Synchronizes with the server.

```
int PQendcopy(PGconn *conn);
```

This function waits until the server has finished the copying. It should either be issued when the last string has been sent to the server using `PQputline` or when the last string has been received from the server using `PQgetline`. It must be issued or the server will get "out of sync" with the client. Upon return from this function, the server is ready to receive the next SQL command. The return value is 0 on successful completion, nonzero otherwise. (Use `PQerrorMessage` to retrieve details if the return value is nonzero.)

When using `PQgetResult`, the application should respond to a `PGRES_COPY_OUT` result by executing `PQgetline` repeatedly, followed by `PQendcopy` after the terminator line is seen. It should then return to the `PQgetResult` loop until `PQgetResult` returns a null pointer. Similarly a `PGRES_COPY_IN` result is processed by a series of `PQputline` calls followed by `PQendcopy`, then return to the `PQgetResult` loop. This arrangement will ensure that a `COPY` command embedded in a series of SQL commands will be executed correctly.

Older applications are likely to submit a `COPY` via `PQexec` and assume that the transaction is done after `PQendcopy`. This will work correctly only if the `COPY` is the only SQL command in the command string.

31.10. Control Functions

These functions control miscellaneous details of libpq's behavior.

`PQclientEncoding`

Returns the client encoding.

```
int PQclientEncoding(const PGconn *conn);
```

Note that it returns the encoding ID, not a symbolic string such as `EUC_JP`. If unsuccessful, it returns -1. To convert an encoding ID to an encoding name, you can use:

```
char *pg_encoding_to_char(int encoding_id);
```

`PQsetClientEncoding`

Sets the client encoding.

```
int PQsetClientEncoding(PGconn *conn, const char *encoding);
```

conn is a connection to the server, and *encoding* is the encoding you want to use. If the function successfully sets the encoding, it returns 0, otherwise -1. The current encoding for this connection can be determined by using PQclientEncoding.

PQsetErrorVerbosity

Determines the verbosity of messages returned by PQerrorMessage and PQresultErrorMessage.

```
typedef enum
{
    PQERRORS_TERSE,
    PQERRORS_DEFAULT,
    PQERRORS_VERBOSE
} PGVerbosity;
```

```
PGVerbosity PQsetErrorVerbosity(PGconn *conn, PGVerbosity verbosity);
```

PQsetErrorVerbosity sets the verbosity mode, returning the connection's previous setting. In *TERSE* mode, returned messages include severity, primary text, and position only; this will normally fit on a single line. The default mode produces messages that include the above plus any detail, hint, or context fields (these might span multiple lines). The *VERBOSE* mode includes all available fields. Changing the verbosity does not affect the messages available from already-existing PGresult objects, only subsequently-created ones.

PQtrace

Enables tracing of the client/server communication to a debugging file stream.

```
void PQtrace(PGconn *conn, FILE *stream);
```

> **Note:** On Windows, if the libpq library and an application are compiled with different flags, this function call will crash the application because the internal representation of the FILE pointers differ. Specifically, multithreaded/single-threaded, release/debug, and static/dynamic flags should be the same for the library and all applications using that library.

PQuntrace

Disables tracing started by PQtrace.

```
void PQuntrace(PGconn *conn);
```

31.11. Miscellaneous Functions

As always, there are some functions that just don't fit anywhere.

PQfreemem

Frees memory allocated by libpq.

```
void PQfreemem(void *ptr);
```

Frees memory allocated by libpq, particularly PQescapeByteaConn, PQescapeBytea, PQunescapeBytea, and PQnotifies. It is particularly important that this function, rather than

free(), be used on Microsoft Windows. This is because allocating memory in a DLL and releasing it in the application works only if multithreaded/single-threaded, release/debug, and static/dynamic flags are the same for the DLL and the application. On non-Microsoft Windows platforms, this function is the same as the standard library function free().

PQconninfoFree

Frees the data structures allocated by PQconndefaults or PQconninfoParse.

```
void PQconninfoFree(PQconninfoOption *connOptions);
```

A simple PQfreemem will not do for this, since the array contains references to subsidiary strings.

PQencryptPassword

Prepares the encrypted form of a PostgreSQL password.

```
char * PQencryptPassword(const char *passwd, const char *user);
```

This function is intended to be used by client applications that wish to send commands like ALTER USER joe PASSWORD 'pwd'. It is good practice not to send the original cleartext password in such a command, because it might be exposed in command logs, activity displays, and so on. Instead, use this function to convert the password to encrypted form before it is sent. The arguments are the cleartext password, and the SQL name of the user it is for. The return value is a string allocated by malloc, or NULL if out of memory. The caller can assume the string doesn't contain any special characters that would require escaping. Use PQfreemem to free the result when done with it.

PQmakeEmptyPGresult

Constructs an empty PGresult object with the given status.

```
PGresult *PQmakeEmptyPGresult(PGconn *conn, ExecStatusType status);
```

This is libpq's internal function to allocate and initialize an empty PGresult object. This function returns NULL if memory could not be allocated. It is exported because some applications find it useful to generate result objects (particularly objects with error status) themselves. If conn is not null and status indicates an error, the current error message of the specified connection is copied into the PGresult. Also, if conn is not null, any event procedures registered in the connection are copied into the PGresult. (They do not get PGEVT_RESULTCREATE calls, but see PQfireResultCreateEvents.) Note that PQclear should eventually be called on the object, just as with a PGresult returned by libpq itself.

PQfireResultCreateEvents

Fires a PGEVT_RESULTCREATE event (see Section 31.13) for each event procedure registered in the PGresult object. Returns non-zero for success, zero if any event procedure fails.

```
int PQfireResultCreateEvents(PGconn *conn, PGresult *res);
```

The conn argument is passed through to event procedures but not used directly. It can be NULL if the event procedures won't use it.

Event procedures that have already received a PGEVT_RESULTCREATE or PGEVT_RESULTCOPY event for this object are not fired again.

The main reason that this function is separate from PQmakeEmptyPGresult is that it is often appropriate to create a PGresult and fill it with data before invoking the event procedures.

PQcopyResult

Makes a copy of a PGresult object. The copy is not linked to the source result in any way and PQclear must be called when the copy is no longer needed. If the function fails, NULL is returned.

```
PGresult *PQcopyResult(const PGresult *src, int flags);
```

This is not intended to make an exact copy. The returned result is always put into PGRES_TUPLES_OK status, and does not copy any error message in the source. (It does copy the command status string, however.) The flags argument determines what else is copied. It is a bitwise OR of several flags. PG_COPYRES_ATTRS specifies copying the source result's attributes (column definitions). PG_COPYRES_TUPLES specifies copying the source result's tuples. (This implies copying the attributes, too.) PG_COPYRES_NOTICEHOOKS specifies copying the source result's notify hooks. PG_COPYRES_EVENTS specifies copying the source result's events. (But any instance data associated with the source is not copied.)

PQsetResultAttrs

Sets the attributes of a PGresult object.

```
int PQsetResultAttrs(PGresult *res, int numAttributes, PGresAttDesc *attDescs);
```

The provided attDescs are copied into the result. If the attDescs pointer is NULL or numAttributes is less than one, the request is ignored and the function succeeds. If res already contains attributes, the function will fail. If the function fails, the return value is zero. If the function succeeds, the return value is non-zero.

PQsetvalue

Sets a tuple field value of a PGresult object.

```
int PQsetvalue(PGresult *res, int tup_num, int field_num, char *value, int len);
```

The function will automatically grow the result's internal tuples array as needed. However, the tup_num argument must be less than or equal to PQntuples, meaning this function can only grow the tuples array one tuple at a time. But any field of any existing tuple can be modified in any order. If a value at field_num already exists, it will be overwritten. If len is -1 or value is NULL, the field value will be set to an SQL null value. The value is copied into the result's private storage, thus is no longer needed after the function returns. If the function fails, the return value is zero. If the function succeeds, the return value is non-zero.

PQresultAlloc

Allocate subsidiary storage for a PGresult object.

```
void *PQresultAlloc(PGresult *res, size_t nBytes);
```

Any memory allocated with this function will be freed when res is cleared. If the function fails, the return value is NULL. The result is guaranteed to be adequately aligned for any type of data, just as for malloc.

PQlibVersion

Return the version of libpq that is being used.

```
int PQlibVersion(void);
```

The result of this function can be used to determine, at run time, if specific functionality is available in the currently loaded version of libpq. The function can be used, for example, to determine which

connection options are available for PQconnectdb or if the hex bytea output added in PostgreSQL 9.0 is supported.

The number is formed by converting the major, minor, and revision numbers into two-decimal-digit numbers and appending them together. For example, version 9.1 will be returned as 90100, and version 9.1.2 will be returned as 90102 (leading zeroes are not shown).

Note: This function appeared in PostgreSQL version 9.1, so it cannot be used to detect required functionality in earlier versions, since linking to it will create a link dependency on version 9.1.

31.12. Notice Processing

Notice and warning messages generated by the server are not returned by the query execution functions, since they do not imply failure of the query. Instead they are passed to a notice handling function, and execution continues normally after the handler returns. The default notice handling function prints the message on stderr, but the application can override this behavior by supplying its own handling function.

For historical reasons, there are two levels of notice handling, called the notice receiver and notice processor. The default behavior is for the notice receiver to format the notice and pass a string to the notice processor for printing. However, an application that chooses to provide its own notice receiver will typically ignore the notice processor layer and just do all the work in the notice receiver.

The function PQsetNoticeReceiver sets or examines the current notice receiver for a connection object. Similarly, PQsetNoticeProcessor sets or examines the current notice processor.

```
typedef void (*PQnoticeReceiver) (void *arg, const PGresult *res);

PQnoticeReceiver
PQsetNoticeReceiver(PGconn *conn,
                    PQnoticeReceiver proc,
                    void *arg);

typedef void (*PQnoticeProcessor) (void *arg, const char *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

Each of these functions returns the previous notice receiver or processor function pointer, and sets the new value. If you supply a null function pointer, no action is taken, but the current pointer is returned.

When a notice or warning message is received from the server, or generated internally by libpq, the notice receiver function is called. It is passed the message in the form of a PGRES_NONFATAL_ERROR PGresult. (This allows the receiver to extract individual fields using PQresultErrorField, or the complete preformatted message using PQresultErrorMessage.) The same void pointer passed to

PQsetNoticeReceiver is also passed. (This pointer can be used to access application-specific state if needed.)

The default notice receiver simply extracts the message (using PQresultErrorMessage) and passes it to the notice processor.

The notice processor is responsible for handling a notice or warning message given in text form. It is passed the string text of the message (including a trailing newline), plus a void pointer that is the same one passed to PQsetNoticeProcessor. (This pointer can be used to access application-specific state if needed.)

The default notice processor is simply:

```
static void
defaultNoticeProcessor(void *arg, const char *message)
{
    fprintf(stderr, "%s", message);
}
```

Once you have set a notice receiver or processor, you should expect that that function could be called as long as either the PGconn object or PGresult objects made from it exist. At creation of a PGresult, the PGconn's current notice handling pointers are copied into the PGresult for possible use by functions like PQgetvalue.

31.13. Event System

libpq's event system is designed to notify registered event handlers about interesting libpq events, such as the creation or destruction of PGconn and PGresult objects. A principal use case is that this allows applications to associate their own data with a PGconn or PGresult and ensure that that data is freed at an appropriate time.

Each registered event handler is associated with two pieces of data, known to libpq only as opaque void * pointers. There is a *passthrough* pointer that is provided by the application when the event handler is registered with a PGconn. The passthrough pointer never changes for the life of the PGconn and all PGresults generated from it; so if used, it must point to long-lived data. In addition there is an *instance data* pointer, which starts out NULL in every PGconn and PGresult. This pointer can be manipulated using the PQinstanceData, PQsetInstanceData, PQresultInstanceData and PQsetResultInstanceData functions. Note that unlike the passthrough pointer, instance data of a PGconn is not automatically inherited by PGresults created from it. libpq does not know what passthrough and instance data pointers point to (if anything) and will never attempt to free them — that is the responsibility of the event handler.

31.13.1. Event Types

The enum PGEventId names the types of events handled by the event system. All its values have names beginning with PGEVT. For each event type, there is a corresponding event info structure that carries the parameters passed to the event handlers. The event types are:

PGEVT_REGISTER

The register event occurs when PQregisterEventProc is called. It is the ideal time to initialize any instanceData an event procedure may need. Only one register event will be fired per event handler per connection. If the event procedure fails, the registration is aborted.

```
typedef struct
{
    PGconn *conn;
} PGEventRegister;
```

When a PGEVT_REGISTER event is received, the evtInfo pointer should be cast to a PGEventRegister *. This structure contains a PGconn that should be in the CONNECTION_OK status; guaranteed if one calls PQregisterEventProc right after obtaining a good PGconn. When returning a failure code, all cleanup must be performed as no PGEVT_CONNDESTROY event will be sent.

PGEVT_CONNRESET

The connection reset event is fired on completion of PQreset or PQresetPoll. In both cases, the event is only fired if the reset was successful. If the event procedure fails, the entire connection reset will fail; the PGconn is put into CONNECTION_BAD status and PQresetPoll will return PGRES_POLLING_FAILED.

```
typedef struct
{
    PGconn *conn;
} PGEventConnReset;
```

When a PGEVT_CONNRESET event is received, the evtInfo pointer should be cast to a PGEventConnReset *. Although the contained PGconn was just reset, all event data remains unchanged. This event should be used to reset/reload/requery any associated instanceData. Note that even if the event procedure fails to process PGEVT_CONNRESET, it will still receive a PGEVT_CONNDESTROY event when the connection is closed.

PGEVT_CONNDESTROY

The connection destroy event is fired in response to PQfinish. It is the event procedure's responsibility to properly clean up its event data as libpq has no ability to manage this memory. Failure to clean up will lead to memory leaks.

```
typedef struct
{
    PGconn *conn;
} PGEventConnDestroy;
```

When a PGEVT_CONNDESTROY event is received, the evtInfo pointer should be cast to a PGEventConnDestroy *. This event is fired prior to PQfinish performing any other cleanup. The return value of the event procedure is ignored since there is no way of indicating a failure from PQfinish. Also, an event procedure failure should not abort the process of cleaning up unwanted memory.

PGEVT_RESULTCREATE

The result creation event is fired in response to any query execution function that generates a result, including PQgetResult. This event will only be fired after the result has been created successfully.

```
typedef struct
{
```

```
    PGconn *conn;
    PGresult *result;
} PGEventResultCreate;
```

When a PGEVT_RESULTCREATE event is received, the evtInfo pointer should be cast to a PGEventResultCreate *. The conn is the connection used to generate the result. This is the ideal place to initialize any instanceData that needs to be associated with the result. If the event procedure fails, the result will be cleared and the failure will be propagated. The event procedure must not try to PQclear the result object for itself. When returning a failure code, all cleanup must be performed as no PGEVT_RESULTDESTROY event will be sent.

PGEVT_RESULTCOPY

The result copy event is fired in response to PQcopyResult. This event will only be fired after the copy is complete. Only event procedures that have successfully handled the PGEVT_RESULTCREATE or PGEVT_RESULTCOPY event for the source result will receive PGEVT_RESULTCOPY events.

```
typedef struct
{
    const PGresult *src;
    PGresult *dest;
} PGEventResultCopy;
```

When a PGEVT_RESULTCOPY event is received, the evtInfo pointer should be cast to a PGEventResultCopy *. The src result is what was copied while the dest result is the copy destination. This event can be used to provide a deep copy of instanceData, since PQcopyResult cannot do that. If the event procedure fails, the entire copy operation will fail and the dest result will be cleared. When returning a failure code, all cleanup must be performed as no PGEVT_RESULTDESTROY event will be sent for the destination result.

PGEVT_RESULTDESTROY

The result destroy event is fired in response to a PQclear. It is the event procedure's responsibility to properly clean up its event data as libpq has no ability to manage this memory. Failure to clean up will lead to memory leaks.

```
typedef struct
{
    PGresult *result;
} PGEventResultDestroy;
```

When a PGEVT_RESULTDESTROY event is received, the evtInfo pointer should be cast to a PGEventResultDestroy *. This event is fired prior to PQclear performing any other cleanup. The return value of the event procedure is ignored since there is no way of indicating a failure from PQclear. Also, an event procedure failure should not abort the process of cleaning up unwanted memory.

31.13.2. Event Callback Procedure

PGEventProc

PGEventProc is a typedef for a pointer to an event procedure, that is, the user callback function that receives events from libpq. The signature of an event procedure must be

```
int eventproc(PGEventId evtId, void *evtInfo, void *passThrough)
```

The evtId parameter indicates which PGEVT event occurred. The evtInfo pointer must be cast to the appropriate structure type to obtain further information about the event. The passThrough parameter is the pointer provided to PQregisterEventProc when the event procedure was registered. The function should return a non-zero value if it succeeds and zero if it fails.

A particular event procedure can be registered only once in any PGconn. This is because the address of the procedure is used as a lookup key to identify the associated instance data.

Caution

On Windows, functions can have two different addresses: one visible from outside a DLL and another visible from inside the DLL. One should be careful that only one of these addresses is used with libpq's event-procedure functions, else confusion will result. The simplest rule for writing code that will work is to ensure that event procedures are declared static. If the procedure's address must be available outside its own source file, expose a separate function to return the address.

31.13.3. Event Support Functions

PQregisterEventProc

Registers an event callback procedure with libpq.

```
int PQregisterEventProc(PGconn *conn, PGEventProc proc,
                        const char *name, void *passThrough);
```

An event procedure must be registered once on each PGconn you want to receive events about. There is no limit, other than memory, on the number of event procedures that can be registered with a connection. The function returns a non-zero value if it succeeds and zero if it fails.

The proc argument will be called when a libpq event is fired. Its memory address is also used to lookup instanceData. The name argument is used to refer to the event procedure in error messages. This value cannot be NULL or a zero-length string. The name string is copied into the PGconn, so what is passed need not be long-lived. The passThrough pointer is passed to the proc whenever an event occurs. This argument can be NULL.

PQsetInstanceData

Sets the connection conn's instanceData for procedure proc to data. This returns non-zero for success and zero for failure. (Failure is only possible if proc has not been properly registered in conn.)

```
int PQsetInstanceData(PGconn *conn, PGEventProc proc, void *data);
```

PQinstanceData

Returns the connection conn's instanceData associated with procedure proc, or NULL if there is none.

```
void *PQinstanceData(const PGconn *conn, PGEventProc proc);
```

`PQresultSetInstanceData`

Sets the result's `instanceData` for `proc` to `data`. This returns non-zero for success and zero for failure. (Failure is only possible if `proc` has not been properly registered in the result.)

```
int PQresultSetInstanceData(PGresult *res, PGEventProc proc, void *data);
```

`PQresultInstanceData`

Returns the result's `instanceData` associated with `proc`, or `NULL` if there is none.

```
void *PQresultInstanceData(const PGresult *res, PGEventProc proc);
```

31.13.4. Event Example

Here is a skeleton example of managing private data associated with libpq connections and results.

```
/* required header for libpq events (note: includes libpq-fe.h) */
#include <libpq-events.h>

/* The instanceData */
typedef struct
{
    int n;
    char *str;
} mydata;

/* PGEventProc */
static int myEventProc(PGEventId evtId, void *evtInfo, void *passThrough);

int
main(void)
{
    mydata *data;
    PGresult *res;
    PGconn *conn =
        PQconnectdb("dbname=postgres options=-csearch_path=");

    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        PQfinish(conn);
        return 1;
    }

    /* called once on any connection that should receive events.
     * Sends a PGEVT_REGISTER to myEventProc.
     */
    if (!PQregisterEventProc(conn, myEventProc, "mydata_proc", NULL))
    {
        fprintf(stderr, "Cannot register PGEventProc\n");
        PQfinish(conn);
        return 1;
```

```
    }

    /* conn instanceData is available */
    data = PQinstanceData(conn, myEventProc);

    /* Sends a PGEVT_RESULTCREATE to myEventProc */
    res = PQexec(conn, "SELECT 1 + 1");

    /* result instanceData is available */
    data = PQresultInstanceData(res, myEventProc);

    /* If PG_COPYRES_EVENTS is used, sends a PGEVT_RESULTCOPY to myEventProc */
    res_copy = PQcopyResult(res, PG_COPYRES_TUPLES | PG_COPYRES_EVENTS);

    /* result instanceData is available if PG_COPYRES_EVENTS was
     * used during the PQcopyResult call.
     */
    data = PQresultInstanceData(res_copy, myEventProc);

    /* Both clears send a PGEVT_RESULTDESTROY to myEventProc */
    PQclear(res);
    PQclear(res_copy);

    /* Sends a PGEVT_CONNDESTROY to myEventProc */
    PQfinish(conn);

    return 0;
}

static int
myEventProc(PGEventId evtId, void *evtInfo, void *passThrough)
{
    switch (evtId)
    {
        case PGEVT_REGISTER:
        {
            PGEventRegister *e = (PGEventRegister *)evtInfo;
            mydata *data = get_mydata(e->conn);

            /* associate app specific data with connection */
            PQsetInstanceData(e->conn, myEventProc, data);
            break;
        }

        case PGEVT_CONNRESET:
        {
            PGEventConnReset *e = (PGEventConnReset *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            if (data)
              memset(data, 0, sizeof(mydata));
            break;
        }
```

```
        case PGEVT_CONNDESTROY:
        {
            PGEventConnDestroy *e = (PGEventConnDestroy *)evtInfo;
            mydata *data = PQinstanceData(e->conn, myEventProc);

            /* free instance data because the conn is being destroyed */
            if (data)
              free_mydata(data);
            break;
        }

        case PGEVT_RESULTCREATE:
        {
            PGEventResultCreate *e = (PGEventResultCreate *)evtInfo;
            mydata *conn_data = PQinstanceData(e->conn, myEventProc);
            mydata *res_data = dup_mydata(conn_data);

            /* associate app specific data with result (copy it from conn) */
            PQsetResultInstanceData(e->result, myEventProc, res_data);
            break;
        }

        case PGEVT_RESULTCOPY:
        {
            PGEventResultCopy *e = (PGEventResultCopy *)evtInfo;
            mydata *src_data = PQresultInstanceData(e->src, myEventProc);
            mydata *dest_data = dup_mydata(src_data);

            /* associate app specific data with result (copy it from a result) */
            PQsetResultInstanceData(e->dest, myEventProc, dest_data);
            break;
        }

        case PGEVT_RESULTDESTROY:
        {
            PGEventResultDestroy *e = (PGEventResultDestroy *)evtInfo;
            mydata *data = PQresultInstanceData(e->result, myEventProc);

            /* free instance data because the result is being destroyed */
            if (data)
              free_mydata(data);
            break;
        }

        /* unknown event ID, just return TRUE. */
        default:
            break;
    }

    return TRUE; /* event processing succeeded */
}
```

31.14. Environment Variables

The following environment variables can be used to select default connection parameter values, which will be used by `PQconnectdb`, `PQsetdbLogin` and `PQsetdb` if no value is directly specified by the calling code. These are useful to avoid hard-coding database connection information into simple client applications, for example.

- `PGHOST` behaves the same as the host connection parameter.
- `PGHOSTADDR` behaves the same as the hostaddr connection parameter. This can be set instead of or in addition to `PGHOST` to avoid DNS lookup overhead.
- `PGPORT` behaves the same as the port connection parameter.
- `PGDATABASE` behaves the same as the dbname connection parameter.
- `PGUSER` behaves the same as the user connection parameter.
- `PGPASSWORD` behaves the same as the password connection parameter. Use of this environment variable is not recommended for security reasons, as some operating systems allow non-root users to see process environment variables via ps; instead consider using the `~/.pgpass` file (see Section 31.15).
- `PGPASSFILE` specifies the name of the password file to use for lookups. If not set, it defaults to `~/.pgpass` (see Section 31.15).
- `PGSERVICE` behaves the same as the service connection parameter.
- `PGSERVICEFILE` specifies the name of the per-user connection service file. If not set, it defaults to `~/.pg_service.conf` (see Section 31.16).
- `PGOPTIONS` behaves the same as the options connection parameter.
- `PGAPPNAME` behaves the same as the application_name connection parameter.
- `PGSSLMODE` behaves the same as the sslmode connection parameter.
- `PGREQUIRESSL` behaves the same as the requiressl connection parameter. This environment variable is deprecated in favor of the `PGSSLMODE` variable; setting both variables suppresses the effect of this one.
- `PGSSLCOMPRESSION` behaves the same as the sslcompression connection parameter.
- `PGSSLCERT` behaves the same as the sslcert connection parameter.
- `PGSSLKEY` behaves the same as the sslkey connection parameter.
- `PGSSLROOTCERT` behaves the same as the sslrootcert connection parameter.
- `PGSSLCRL` behaves the same as the sslcrl connection parameter.
- `PGREQUIREPEER` behaves the same as the requirepeer connection parameter.
- `PGKRBSRVNAME` behaves the same as the krbsrvname connection parameter.
- `PGGSSLIB` behaves the same as the gsslib connection parameter.
- `PGCONNECT_TIMEOUT` behaves the same as the connect_timeout connection parameter.
- `PGCLIENTENCODING` behaves the same as the client_encoding connection parameter.

The following environment variables can be used to specify default behavior for each PostgreSQL session. (See also the ALTER ROLE and ALTER DATABASE commands for ways to set default behavior on a per-user or per-database basis.)

- `PGDATESTYLE` sets the default style of date/time representation. (Equivalent to `SET datestyle TO ...`.)
- `PGTZ` sets the default time zone. (Equivalent to `SET timezone TO ...`.)
- `PGGEQO` sets the default mode for the genetic query optimizer. (Equivalent to `SET geqo TO ...`.)

Refer to the SQL command SET for information on correct values for these environment variables.

The following environment variables determine internal behavior of libpq; they override compiled-in defaults.

- `PGSYSCONFDIR` sets the directory containing the `pg_service.conf` file and in a future version possibly other system-wide configuration files.
- `PGLOCALEDIR` sets the directory containing the `locale` files for message localization.

31.15. The Password File

The file `.pgpass` in a user's home directory or the file referenced by `PGPASSFILE` can contain passwords to be used if the connection requires a password (and no password has been specified otherwise). On Microsoft Windows the file is named `%APPDATA%\postgresql\pgpass.conf` (where `%APPDATA%` refers to the Application Data subdirectory in the user's profile).

This file should contain lines of the following format:

hostname:*port*:*database*:*username*:*password*

(You can add a reminder comment to the file by copying the line above and preceding it with `#`.) Each of the first four fields can be a literal value, or `*`, which matches anything. The password field from the first line that matches the current connection parameters will be used. (Therefore, put more-specific entries first when you are using wildcards.) If an entry needs to contain `:` or `\`, escape this character with `\`. A host name of `localhost` matches both TCP (host name `localhost`) and Unix domain socket (`pghost` empty or the default socket directory) connections coming from the local machine. In a standby server, a database name of `replication` matches streaming replication connections made to the master server. The `database` field is of limited usefulness because users have the same password for all databases in the same cluster.

On Unix systems, the permissions on `.pgpass` must disallow any access to world or group; achieve this by the command `chmod 0600 ~/.pgpass`. If the permissions are less strict than this, the file will be ignored. On Microsoft Windows, it is assumed that the file is stored in a directory that is secure, so no special permissions check is made.

31.16. The Connection Service File

The connection service file allows libpq connection parameters to be associated with a single service name. That service name can then be specified by a libpq connection, and the associated settings will be used. This allows connection parameters to be modified without requiring a recompile of the libpq application. The service name can also be specified using the `PGSERVICE` environment variable.

The connection service file can be a per-user service file at `~/.pg_service.conf` or the location specified by the environment variable `PGSERVICEFILE`, or it can be a system-wide file at `` `pg_config --sysconfdir`/pg_service.conf `` or in the directory specified by the environment variable `PGSYSCONFDIR`. If service definitions with the same name exist in the user and the system file, the user file takes precedence.

The file uses an "INI file" format where the section name is the service name and the parameters are connection parameters; see Section 31.1.2 for a list. For example:

```
# comment
[mydb]
host=somehost
port=5433
user=admin
```

An example file is provided at `share/pg_service.conf.sample`.

31.17. LDAP Lookup of Connection Parameters

If libpq has been compiled with LDAP support (option `--with-ldap` for `configure`) it is possible to retrieve connection options like `host` or `dbname` via LDAP from a central server. The advantage is that if the connection parameters for a database change, the connection information doesn't have to be updated on all client machines.

LDAP connection parameter lookup uses the connection service file `pg_service.conf` (see Section 31.16). A line in a `pg_service.conf` stanza that starts with `ldap://` will be recognized as an LDAP URL and an LDAP query will be performed. The result must be a list of `keyword = value` pairs which will be used to set connection options. The URL must conform to RFC 1959 and be of the form

ldap://[*hostname*[:*port*]]/*search_base*?*attribute*?*search_scope*?*filter*

where *hostname* defaults to `localhost` and *port* defaults to 389.

Processing of `pg_service.conf` is terminated after a successful LDAP lookup, but is continued if the LDAP server cannot be contacted. This is to provide a fallback with further LDAP URL lines that point to different LDAP servers, classical `keyword = value` pairs, or default connection options. If you would rather get an error message in this case, add a syntactically incorrect line after the LDAP URL.

A sample LDAP entry that has been created with the LDIF file

```
version:1
dn:cn=mydatabase,dc=mycompany,dc=com
changetype:add
objectclass:top
objectclass:device
```

```
cn:mydatabase
description:host=dbserver.mycompany.com
description:port=5439
description:dbname=mydb
description:user=mydb_user
description:sslmode=require
```

might be queried with the following LDAP URL:

```
ldap://ldap.mycompany.com/dc=mycompany,dc=com?description?one?(cn=mydatabase)
```

You can also mix regular service file entries with LDAP lookups. A complete example for a stanza in `pg_service.conf` would be:

```
# only host and port are stored in LDAP, specify dbname and user explicitly
[customerdb]
dbname=customer
user=appuser
ldap://ldap.acme.com/cn=dbserver,cn=hosts?pgconnectinfo?base?(objectclass=*)
```

31.18. SSL Support

PostgreSQL has native support for using SSL connections to encrypt client/server communications for increased security. See Section 17.9 for details about the server-side SSL functionality.

libpq reads the system-wide OpenSSL configuration file. By default, this file is named `openssl.cnf` and is located in the directory reported by `openssl version -d`. This default can be overridden by setting environment variable `OPENSSL_CONF` to the name of the desired configuration file.

31.18.1. Client Verification of Server Certificates

By default, PostgreSQL will not perform any verification of the server certificate. This means that it is possible to spoof the server identity (for example by modifying a DNS record or by taking over the server IP address) without the client knowing. In order to prevent spoofing, the client must be able to verify the server's identity via a chain of trust. A chain of trust is established by placing a root (self-signed) certificate authority (CA) certificate on one computer and a leaf certificate *signed* by the root certificate on another computer. It is also possible to use an "intermediate" certificate which is signed by the root certificate and signs leaf certificates.

To allow the client to verify the identity of the server, place a root certificate on the client and a leaf certificate signed by the root certificate on the server. To allow the server to verify the identity of the client, place a root certificate on the server and a leaf certificate signed by the root certificate on the client. One or more intermediate certificates (usually stored with the leaf certificate) can also be used to link the leaf certificate to the root certificate.

Once a chain of trust has been established, there are two ways for the client to validate the leaf certificate sent by the server. If the parameter `sslmode` is set to `verify-ca`, libpq will verify that the server is trustworthy by checking the certificate chain up to the root certificate stored on the client. If `sslmode` is set to `verify-full`, libpq will *also* verify that the server host name matches the name stored in the server certificate. The SSL connection will fail if the server certificate cannot be verified. `verify-full` is recommended in most security-sensitive environments.

In `verify-full` mode, the host name is matched against the certificate's Subject Alternative Name attribute(s), or against the Common Name attribute if no Subject Alternative Name of type dNSName is present. If the certificate's name attribute starts with an asterisk (`*`), the asterisk will be treated as a wildcard, which will match all characters *except* a dot (`.`). This means the certificate will not match subdomains. If the connection is made using an IP address instead of a host name, the IP address will be matched (without doing any DNS lookups).

To allow server certificate verification, one or more root certificates must be placed in the file `~/.postgresql/root.crt` in the user's home directory. (On Microsoft Windows the file is named `%APPDATA%\postgresql\root.crt`.) Intermediate certificates should also be added to the file if they are needed to link the certificate chain sent by the server to the root certificates stored on the client.

Certificate Revocation List (CRL) entries are also checked if the file `~/.postgresql/root.crl` exists (`%APPDATA%\postgresql\root.crl` on Microsoft Windows).

The location of the root certificate file and the CRL can be changed by setting the connection parameters `sslrootcert` and `sslcrl` or the environment variables `PGSSLROOTCERT` and `PGSSLCRL`.

> **Note:** For backwards compatibility with earlier versions of PostgreSQL, if a root CA file exists, the behavior of `sslmode=require` will be the same as that of `verify-ca`, meaning the server certificate is validated against the CA. Relying on this behavior is discouraged, and applications that need certificate validation should always use `verify-ca` or `verify-full`.

31.18.2. Client Certificates

If the server attempts to verify the identity of the client by requesting the client's leaf certificate, libpq will send the certificates stored in file `~/.postgresql/postgresql.crt` in the user's home directory. The certificates must chain to the root certificate trusted by the server. A matching private key file `~/.postgresql/postgresql.key` must also be present. The private key file must not allow any access to world or group; achieve this by the command `chmod 0600 ~/.postgresql/postgresql.key`. On Microsoft Windows these files are named `%APPDATA%\postgresql\postgresql.crt` and `%APPDATA%\postgresql\postgresql.key`, and there is no special permissions check since the directory is presumed secure. The location of the certificate and key files can be overridden by the connection parameters `sslcert` and `sslkey` or the environment variables `PGSSLCERT` and `PGSSLKEY`.

The first certificate in `postgresql.crt` must be the client's certificate because it must match the client's private key. "Intermediate" certificates can be optionally appended to the file — doing so avoids requiring storage of intermediate certificates on the server's `root.crt` file.

For instructions on creating certificates, see Section 17.9.3.

31.18.3. Protection Provided in Different Modes

The different values for the `sslmode` parameter provide different levels of protection. SSL can provide protection against three types of attacks:

Eavesdropping

If a third party can examine the network traffic between the client and the server, it can read both connection information (including the user name and password) and the data that is passed. SSL uses encryption to prevent this.

Man in the middle (MITM)

If a third party can modify the data while passing between the client and server, it can pretend to be the server and therefore see and modify data *even if it is encrypted*. The third party can then forward the connection information and data to the original server, making it impossible to detect this attack. Common vectors to do this include DNS poisoning and address hijacking, whereby the client is directed to a different server than intended. There are also several other attack methods that can accomplish this. SSL uses certificate verification to prevent this, by authenticating the server to the client.

Impersonation

If a third party can pretend to be an authorized client, it can simply access data it should not have access to. Typically this can happen through insecure password management. SSL uses client certificates to prevent this, by making sure that only holders of valid certificates can access the server.

For a connection to be known secure, SSL usage must be configured on *both the client and the server* before the connection is made. If it is only configured on the server, the client may end up sending sensitive information (e.g., passwords) before it knows that the server requires high security. In libpq, secure connections can be ensured by setting the `sslmode` parameter to `verify-full` or `verify-ca`, and providing the system with a root certificate to verify against. This is analogous to using an `https` URL for encrypted web browsing.

Once the server has been authenticated, the client can pass sensitive data. This means that up until this point, the client does not need to know if certificates will be used for authentication, making it safe to specify that only in the server configuration.

All SSL options carry overhead in the form of encryption and key-exchange, so there is a tradeoff that has to be made between performance and security. Table 31-1 illustrates the risks the different `sslmode` values protect against, and what statement they make about security and overhead.

Table 31-1. SSL Mode Descriptions

`sslmode`	Eavesdropping protection	MITM protection	Statement
`disable`	No	No	I don't care about security, and I don't want to pay the overhead of encryption.

sslmode	Eavesdropping protection	MITM protection	Statement
allow	Maybe	No	I don't care about security, but I will pay the overhead of encryption if the server insists on it.
prefer	Maybe	No	I don't care about encryption, but I wish to pay the overhead of encryption if the server supports it.
require	Yes	No	I want my data to be encrypted, and I accept the overhead. I trust that the network will make sure I always connect to the server I want.
verify-ca	Yes	Depends on CA policy	I want my data encrypted, and I accept the overhead. I want to be sure that I connect to a server that I trust.
verify-full	Yes	Yes	I want my data encrypted, and I accept the overhead. I want to be sure that I connect to a server I trust, and that it's the one I specify.

The difference between verify-ca and verify-full depends on the policy of the root CA. If a public CA is used, verify-ca allows connections to a server that *somebody else* may have registered with the CA. In this case, verify-full should always be used. If a local CA is used, or even a self-signed certificate, using verify-ca often provides enough protection.

The default value for sslmode is prefer. As is shown in the table, this makes no sense from a security point of view, and it only promises performance overhead if possible. It is only provided as the default for backward compatibility, and is not recommended in secure deployments.

31.18.4. SSL Client File Usage

Table 31-2 summarizes the files that are relevant to the SSL setup on the client.

Table 31-2. Libpq/Client SSL File Usage

File	Contents	Effect

File	Contents	Effect
~/.postgresql/postgresql.crt	client certificate	sent to server
~/.postgresql/postgresql.key	client private key	proves client certificate sent by owner; does not indicate certificate owner is trustworthy
~/.postgresql/root.crt	trusted certificate authorities	checks that server certificate is signed by a trusted certificate authority
~/.postgresql/root.crl	certificates revoked by certificate authorities	server certificate must not be on this list

31.18.5. SSL Library Initialization

If your application initializes libssl and/or libcrypto libraries and libpq is built with SSL support, you should call PQinitOpenSSL to tell libpq that the libssl and/or libcrypto libraries have been initialized by your application, so that libpq will not also initialize those libraries.

PQinitOpenSSL

Allows applications to select which security libraries to initialize.

```
void PQinitOpenSSL(int do_ssl, int do_crypto);
```

When do_ssl is non-zero, libpq will initialize the OpenSSL library before first opening a database connection. When do_crypto is non-zero, the libcrypto library will be initialized. By default (if PQinitOpenSSL is not called), both libraries are initialized. When SSL support is not compiled in, this function is present but does nothing.

If your application uses and initializes either OpenSSL or its underlying libcrypto library, you *must* call this function with zeroes for the appropriate parameter(s) before first opening a database connection. Also be sure that you have done that initialization before opening a database connection.

PQinitSSL

Allows applications to select which security libraries to initialize.

```
void PQinitSSL(int do_ssl);
```

This function is equivalent to PQinitOpenSSL(do_ssl, do_ssl). It is sufficient for applications that initialize both or neither of OpenSSL and libcrypto.

PQinitSSL has been present since PostgreSQL 8.0, while PQinitOpenSSL was added in PostgreSQL 8.4, so PQinitSSL might be preferable for applications that need to work with older versions of libpq.

31.19. Behavior in Threaded Programs

libpq is reentrant and thread-safe by default. You might need to use special compiler command-line options when you compile your application code. Refer to your system's documentation for information about how to build thread-enabled applications, or look in `src/Makefile.global` for `PTHREAD_CFLAGS` and `PTHREAD_LIBS`. This function allows the querying of libpq's thread-safe status:

`PQisthreadsafe`

Returns the thread safety status of the libpq library.

```
int PQisthreadsafe();
```

Returns 1 if the libpq is thread-safe and 0 if it is not.

One thread restriction is that no two threads attempt to manipulate the same `PGconn` object at the same time. In particular, you cannot issue concurrent commands from different threads through the same connection object. (If you need to run concurrent commands, use multiple connections.)

`PGresult` objects are normally read-only after creation, and so can be passed around freely between threads. However, if you use any of the `PGresult`-modifying functions described in Section 31.11 or Section 31.13, it's up to you to avoid concurrent operations on the same `PGresult`, too.

The deprecated functions `PQrequestCancel` and `PQoidStatus` are not thread-safe and should not be used in multithread programs. `PQrequestCancel` can be replaced by `PQcancel`. `PQoidStatus` can be replaced by `PQoidValue`.

If you are using Kerberos inside your application (in addition to inside libpq), you will need to do locking around Kerberos calls because Kerberos functions are not thread-safe. See function `PQregisterThreadLock` in the libpq source code for a way to do cooperative locking between libpq and your application.

31.20. Building libpq Programs

To build (i.e., compile and link) a program using libpq you need to do all of the following things:

- Include the `libpq-fe.h` header file:

  ```
  #include <libpq-fe.h>
  ```

  If you failed to do that then you will normally get error messages from your compiler similar to:

  ```
  foo.c: In function 'main':
  foo.c:34: 'PGconn' undeclared (first use in this function)
  foo.c:35: 'PGresult' undeclared (first use in this function)
  foo.c:54: 'CONNECTION_BAD' undeclared (first use in this function)
  foo.c:68: 'PGRES_COMMAND_OK' undeclared (first use in this function)
  foo.c:95: 'PGRES_TUPLES_OK' undeclared (first use in this function)
  ```

- Point your compiler to the directory where the PostgreSQL header files were installed, by supplying the `-I`*`directory`* option to your compiler. (In some cases the compiler will look into the directory in question by default, so you can omit this option.) For instance, your compile command line could look like:

```
cc -c -I/usr/local/pgsql/include testprog.c
```

If you are using makefiles then add the option to the `CPPFLAGS` variable:

```
CPPFLAGS += -I/usr/local/pgsql/include
```

If there is any chance that your program might be compiled by other users then you should not hardcode the directory location like that. Instead, you can run the utility `pg_config` to find out where the header files are on the local system:

```
$ pg_config --includedir
/usr/local/include
```

If you have `pkg-config` installed, you can run instead:

```
$ pkg-config --cflags libpq
-I/usr/local/include
```

Note that this will already include the `-I` in front of the path.

Failure to specify the correct option to the compiler will result in an error message such as:

```
testlibpq.c:8:22: libpq-fe.h: No such file or directory
```

- When linking the final program, specify the option `-lpq` so that the libpq library gets pulled in, as well as the option `-L`*directory* to point the compiler to the directory where the libpq library resides. (Again, the compiler will search some directories by default.) For maximum portability, put the `-L` option before the `-lpq` option. For example:

  ```
  cc -o testprog testprog1.o testprog2.o -L/usr/local/pgsql/lib -lpq
  ```

  You can find out the library directory using `pg_config` as well:

  ```
  $ pg_config --libdir
  /usr/local/pgsql/lib
  ```

  Or again use `pkg-config`:

  ```
  $ pkg-config --libs libpq
  -L/usr/local/pgsql/lib -lpq
  ```

  Note again that this prints the full options, not only the path.

  Error messages that point to problems in this area could look like the following:

  ```
  testlibpq.o: In function 'main':
  testlibpq.o(.text+0x60): undefined reference to 'PQsetdbLogin'
  testlibpq.o(.text+0x71): undefined reference to 'PQstatus'
  testlibpq.o(.text+0xa4): undefined reference to 'PQerrorMessage'
  ```

  This means you forgot `-lpq`.

  ```
  /usr/bin/ld: cannot find -lpq
  ```

  This means you forgot the `-L` option or did not specify the right directory.

31.21. Example Programs

These examples and others can be found in the directory `src/test/examples` in the source code distribution.

Example 31-1. libpq Example Program 1

```
/*
 * src/test/examples/testlibpq.c
 *
 *
 * testlibpq.c
 *
 *      Test the C version of libpq, the PostgreSQL frontend library.
 */
#include <stdio.h>
#include <stdlib.h>
#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    int         nFields;
    int         i,
                j;

    /*
     * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
     * environment variables or defaults for all other connection parameters.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* Make a connection to the database */
    conn = PQconnectdb(conninfo);

    /* Check to see that the backend connection was successfully made */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* Set always-secure search path, so malicious users can't take control. */
```

```
res = PQexec(conn,
             "SELECT pg_catalog.set_config('search_path', '', false)");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/*
 * Should PQclear PGresult whenever it is no longer needed to avoid memory
 * leaks
 */
PQclear(res);

/*
 * Our test case here involves using a cursor, for which we must be inside
 * a transaction block.  We could do the whole thing with a single
 * PQexec() of "select * from pg_database", but that's too trivial to make
 * a good example.
 */

/* Start a transaction block */
res = PQexec(conn, "BEGIN");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "BEGIN command failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

/*
 * Fetch rows from pg_database, the system catalog of databases
 */
res = PQexec(conn, "DECLARE myportal CURSOR FOR select * from pg_database");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "DECLARE CURSOR failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

res = PQexec(conn, "FETCH ALL in myportal");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "FETCH ALL failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/* first, print out the attribute names */
```

```
    nFields = PQnfields(res);
    for (i = 0; i < nFields; i++)
        printf("%-15s", PQfname(res, i));
    printf("\n\n");

    /* next, print out the rows */
    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }

    PQclear(res);

    /* close the portal ... we don't bother to check for errors ... */
    res = PQexec(conn, "CLOSE myportal");
    PQclear(res);

    /* end the transaction */
    res = PQexec(conn, "END");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Example 31-2. libpq Example Program 2

```
/*
 * src/test/examples/testlibpq2.c
 *
 *
 * testlibpq2.c
 *      Test of the asynchronous notification interface
 *
 * Start this program, then from psql in another window do
 *   NOTIFY TBL2;
 * Repeat four times to get this program to exit.
 *
 * Or, if you want to get fancy, try this:
 * populate a database with the following commands
 * (provided in src/test/examples/testlibpq2.sql):
 *
 *   CREATE SCHEMA TESTLIBPQ2;
 *   SET search_path = TESTLIBPQ2;
 *   CREATE TABLE TBL1 (i int4);
 *   CREATE TABLE TBL2 (i int4);
 *   CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *     (INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);
```

```
 *
 * Start this program, then from psql do this four times:
 *
 *   INSERT INTO TESTLIBPQ2.TBL1 VALUES (10);
 */

#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#ifdef HAVE_SYS_SELECT_H
#include <sys/select.h>
#endif

#include "libpq-fe.h"

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;
    int         nnotifies;

    /*
     * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
     * environment variables or defaults for all other connection parameters.
     */
    if (argc > 1)
        conninfo = argv[1];
    else
        conninfo = "dbname = postgres";

    /* Make a connection to the database */
    conn = PQconnectdb(conninfo);

    /* Check to see that the backend connection was successfully made */
    if (PQstatus(conn) != CONNECTION_OK)
    {
```

```
    fprintf(stderr, "Connection to database failed: %s",
            PQerrorMessage(conn));
    exit_nicely(conn);
}

/* Set always-secure search path, so malicious users can't take control. */
res = PQexec(conn,
             "SELECT pg_catalog.set_config('search_path', '', false)");
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}

/*
 * Should PQclear PGresult whenever it is no longer needed to avoid memory
 * leaks
 */
PQclear(res);

/*
 * Issue LISTEN command to enable notifications from the rule's NOTIFY.
 */
res = PQexec(conn, "LISTEN TBL2");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "LISTEN command failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

/* Quit after four notifies are received. */
nnotifies = 0;
while (nnotifies < 4)
{
    /*
     * Sleep until something happens on the connection.  We use select(2)
     * to wait for input, but you could also use poll() or similar
     * facilities.
     */
    int         sock;
    fd_set      input_mask;

    sock = PQsocket(conn);

    if (sock < 0)
        break;              /* shouldn't happen */

    FD_ZERO(&input_mask);
    FD_SET(sock, &input_mask);
```

```
        if (select(sock + 1, &input_mask, NULL, NULL, NULL) < 0)
        {
            fprintf(stderr, "select() failed: %s\n", strerror(errno));
            exit_nicely(conn);
        }

        /* Now check for input */
        PQconsumeInput(conn);
        while ((notify = PQnotifies(conn)) != NULL)
        {
            fprintf(stderr,
                    "ASYNC NOTIFY of '%s' received from backend PID %d\n",
                    notify->relname, notify->be_pid);
            PQfreemem(notify);
            nnotifies++;
            PQconsumeInput(conn);
        }
    }

    fprintf(stderr, "Done.\n");

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Example 31-3. libpq Example Program 3

```
/*
 * src/test/examples/testlibpq3.c
 *
 *
 * testlibpq3.c
 *      Test out-of-line parameters and binary I/O.
 *
 * Before running this, populate a database with the following commands
 * (provided in src/test/examples/testlibpq3.sql):
 *
 * CREATE SCHEMA testlibpq3;
 * SET search_path = testlibpq3;
 * CREATE TABLE test1 (i int4, t text, b bytea);
 * INSERT INTO test1 values (1, 'joe"s place', '\\000\\001\\002\\003\\004');
 * INSERT INTO test1 values (2, 'ho there', '\\004\\003\\002\\001\\000');
 *
 * The expected output is:
 *
 * tuple 0: got
 *  i = (4 bytes) 1
 *  t = (11 bytes) 'joe's place'
 *  b = (5 bytes) \000\001\002\003\004
 *
```

```
 * tuple 0: got
 *  i = (4 bytes) 2
 *  t = (8 bytes) 'ho there'
 *  b = (5 bytes) \004\003\002\001\000
 */

#ifdef WIN32
#include <windows.h>
#endif

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <sys/types.h>
#include "libpq-fe.h"

/* for ntohl/htonl */
#include <netinet/in.h>
#include <arpa/inet.h>


static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

/*
 * This function prints a query result that is a binary-format fetch from
 * a table defined as in the comment above.  We split it out because the
 * main() function uses it twice.
 */
static void
show_binary_results(PGresult *res)
{
    int         i,
                j;
    int         i_fnum,
                t_fnum,
                b_fnum;

    /* Use PQfnumber to avoid assumptions about field order in result */
    i_fnum = PQfnumber(res, "i");
    t_fnum = PQfnumber(res, "t");
    b_fnum = PQfnumber(res, "b");

    for (i = 0; i < PQntuples(res); i++)
    {
        char       *iptr;
        char       *tptr;
        char       *bptr;
```

```
        int         blen;
        int         ival;

        /* Get the field values (we ignore possibility they are null!) */
        iptr = PQgetvalue(res, i, i_fnum);
        tptr = PQgetvalue(res, i, t_fnum);
        bptr = PQgetvalue(res, i, b_fnum);

        /*
         * The binary representation of INT4 is in network byte order, which
         * we'd better coerce to the local byte order.
         */
        ival = ntohl(*((uint32_t *) iptr));

        /*
         * The binary representation of TEXT is, well, text, and since libpq
         * was nice enough to append a zero byte to it, it'll work just fine
         * as a C string.
         *
         * The binary representation of BYTEA is a bunch of bytes, which could
         * include embedded nulls so we have to pay attention to field length.
         */
        blen = PQgetlength(res, i, b_fnum);

        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d\n",
               PQgetlength(res, i, i_fnum), ival);
        printf(" t = (%d bytes) '%s'\n",
               PQgetlength(res, i, t_fnum), tptr);
        printf(" b = (%d bytes) ", blen);
        for (j = 0; j < blen; j++)
            printf("\\%03o", bptr[j]);
        printf("\n\n");
    }
}

int
main(int argc, char **argv)
{
    const char *conninfo;
    PGconn     *conn;
    PGresult   *res;
    const char *paramValues[1];
    int         paramLengths[1];
    int         paramFormats[1];
    uint32_t    binaryIntVal;

    /*
     * If the user supplies a parameter on the command line, use it as the
     * conninfo string; otherwise default to setting dbname=postgres and using
     * environment variables or defaults for all other connection parameters.
     */
    if (argc > 1)
```

```
    conninfo = argv[1];
else
    conninfo = "dbname = postgres";

/* Make a connection to the database */
conn = PQconnectdb(conninfo);

/* Check to see that the backend connection was successfully made */
if (PQstatus(conn) != CONNECTION_OK)
{
    fprintf(stderr, "Connection to database failed: %s",
            PQerrorMessage(conn));
    exit_nicely(conn);
}

/* Set always-secure search path, so malicious users can't take control. */
res = PQexec(conn, "SET search_path = testlibpq3");
if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
}
PQclear(res);

/*
 * The point of this program is to illustrate use of PQexecParams() with
 * out-of-line parameters, as well as binary transmission of data.
 *
 * This first example transmits the parameters as text, but receives the
 * results in binary format.  By using out-of-line parameters we can avoid
 * a lot of tedious mucking about with quoting and escaping, even though
 * the data is text.  Notice how we don't have to do anything special with
 * the quote mark in the parameter value.
 */

/* Here is our out-of-line parameter value */
paramValues[0] = "joe's place";

res = PQexecParams(conn,
                   "SELECT * FROM test1 WHERE t = $1",
                   1,       /* one param */
                   NULL,    /* let the backend deduce param type */
                   paramValues,
                   NULL,    /* don't need param lengths since text */
                   NULL,    /* default to all text params */
                   1);      /* ask for binary results */

if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
    fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
    PQclear(res);
    exit_nicely(conn);
```

```
    }

    show_binary_results(res);

    PQclear(res);

    /*
     * In this second example we transmit an integer parameter in binary form,
     * and again retrieve the results in binary form.
     *
     * Although we tell PQexecParams we are letting the backend deduce
     * parameter type, we really force the decision by casting the parameter
     * symbol in the query text.  This is a good safety measure when sending
     * binary parameters.
     */

    /* Convert integer value "2" to network byte order */
    binaryIntVal = htonl((uint32_t) 2);

    /* Set up parameter arrays for PQexecParams */
    paramValues[0] = (char *) &binaryIntVal;
    paramLengths[0] = sizeof(binaryIntVal);
    paramFormats[0] = 1;        /* binary */

    res = PQexecParams(conn,
                       "SELECT * FROM test1 WHERE i = $1::int4",
                       1,       /* one param */
                       NULL,    /* let the backend deduce param type */
                       paramValues,
                       paramLengths,
                       paramFormats,
                       1);      /* ask for binary results */

    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SELECT failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }

    show_binary_results(res);

    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Chapter 32. Large Objects

PostgreSQL has a *large object* facility, which provides stream-style access to user data that is stored in a special large-object structure. Streaming access is useful when working with data values that are too large to manipulate conveniently as a whole.

This chapter describes the implementation and the programming and query language interfaces to PostgreSQL large object data. We use the libpq C library for the examples in this chapter, but most programming interfaces native to PostgreSQL support equivalent functionality. Other interfaces might use the large object interface internally to provide generic support for large values. This is not described here.

32.1. Introduction

All large objects are stored in a single system table named `pg_largeobject`. Each large object also has an entry in the system table `pg_largeobject_metadata`. Large objects can be created, modified, and deleted using a read/write API that is similar to standard operations on files.

PostgreSQL also supports a storage system called "TOAST", which automatically stores values larger than a single database page into a secondary storage area per table. This makes the large object facility partially obsolete. One remaining advantage of the large object facility is that it allows values up to 4 TB in size, whereas TOASTed fields can be at most 1 GB. Also, reading and updating portions of a large object can be done efficiently, while most operations on a TOASTed field will read or write the whole value as a unit.

32.2. Implementation Features

The large object implementation breaks large objects up into "chunks" and stores the chunks in rows in the database. A B-tree index guarantees fast searches for the correct chunk number when doing random access reads and writes.

The chunks stored for a large object do not have to be contiguous. For example, if an application opens a new large object, seeks to offset 1000000, and writes a few bytes there, this does not result in allocation of 1000000 bytes worth of storage; only of chunks covering the range of data bytes actually written. A read operation will, however, read out zeroes for any unallocated locations preceding the last existing chunk. This corresponds to the common behavior of "sparsely allocated" files in Unix file systems.

As of PostgreSQL 9.0, large objects have an owner and a set of access permissions, which can be managed using GRANT and REVOKE. `SELECT` privileges are required to read a large object, and `UPDATE` privileges are required to write or truncate it. Only the large object's owner (or a database superuser) can delete, comment on, or change the owner of a large object. To adjust this behavior for compatibility with prior releases, see the lo_compat_privileges run-time parameter.

32.3. Client Interfaces

This section describes the facilities that PostgreSQL's libpq client interface library provides for accessing

large objects. The PostgreSQL large object interface is modeled after the Unix file-system interface, with analogues of `open`, `read`, `write`, `lseek`, etc.

All large object manipulation using these functions *must* take place within an SQL transaction block, since large object file descriptors are only valid for the duration of a transaction.

If an error occurs while executing any one of these functions, the function will return an otherwise-impossible value, typically 0 or -1. A message describing the error is stored in the connection object and can be retrieved with `PQerrorMessage`.

Client applications that use these functions should include the header file `libpq/libpq-fs.h` and link with the libpq library.

32.3.1. Creating a Large Object

The function

```
Oid lo_creat(PGconn *conn, int mode);
```

creates a new large object. The return value is the OID that was assigned to the new large object, or `InvalidOid` (zero) on failure. *`mode`* is unused and ignored as of PostgreSQL 8.1; however, for backward compatibility with earlier releases it is best to set it to `INV_READ`, `INV_WRITE`, or `INV_READ | INV_WRITE`. (These symbolic constants are defined in the header file `libpq/libpq-fs.h`.)

An example:

```
inv_oid = lo_creat(conn, INV_READ|INV_WRITE);
```

The function

```
Oid lo_create(PGconn *conn, Oid lobjId);
```

also creates a new large object. The OID to be assigned can be specified by *`lobjId`*; if so, failure occurs if that OID is already in use for some large object. If *`lobjId`* is `InvalidOid` (zero) then `lo_create` assigns an unused OID (this is the same behavior as `lo_creat`). The return value is the OID that was assigned to the new large object, or `InvalidOid` (zero) on failure.

`lo_create` is new as of PostgreSQL 8.1; if this function is run against an older server version, it will fail and return `InvalidOid`.

An example:

```
inv_oid = lo_create(conn, desired_oid);
```

32.3.2. Importing a Large Object

To import an operating system file as a large object, call

```
Oid lo_import(PGconn *conn, const char *filename);
```

filename specifies the operating system name of the file to be imported as a large object. The return value is the OID that was assigned to the new large object, or InvalidOid (zero) on failure. Note that the file is read by the client interface library, not by the server; so it must exist in the client file system and be readable by the client application.

The function

```
Oid lo_import_with_oid(PGconn *conn, const char *filename, Oid lobjId);
```

also imports a new large object. The OID to be assigned can be specified by *lobjId*; if so, failure occurs if that OID is already in use for some large object. If *lobjId* is InvalidOid (zero) then lo_import_with_oid assigns an unused OID (this is the same behavior as lo_import). The return value is the OID that was assigned to the new large object, or InvalidOid (zero) on failure.

lo_import_with_oid is new as of PostgreSQL 8.4 and uses lo_create internally which is new in 8.1; if this function is run against 8.0 or before, it will fail and return InvalidOid.

32.3.3. Exporting a Large Object

To export a large object into an operating system file, call

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename);
```

The lobjId argument specifies the OID of the large object to export and the filename argument specifies the operating system name of the file. Note that the file is written by the client interface library, not by the server. Returns 1 on success, -1 on failure.

32.3.4. Opening an Existing Large Object

To open an existing large object for reading or writing, call

```
int lo_open(PGconn *conn, Oid lobjId, int mode);
```

The lobjId argument specifies the OID of the large object to open. The mode bits control whether the object is opened for reading (INV_READ), writing (INV_WRITE), or both. (These symbolic constants are defined in the header file libpq/libpq-fs.h.) lo_open returns a (non-negative) large object descriptor for later use in lo_read, lo_write, lo_lseek, lo_lseek64, lo_tell, lo_tell64, lo_truncate, lo_truncate64, and lo_close. The descriptor is only valid for the duration of the current transaction. On failure, -1 is returned.

The server currently does not distinguish between modes INV_WRITE and INV_READ | INV_WRITE: you are allowed to read from the descriptor in either case. However there is a significant difference between these modes and INV_READ alone: with INV_READ you cannot write on the descriptor, and the data read from it will reflect the contents of the large object at the time of the transaction snapshot that was active when lo_open was executed, regardless of later writes by this or other transactions. Reading from a descriptor opened with INV_WRITE returns data that reflects all writes of other committed transactions as well as writes of the current transaction. This is similar to the behavior of REPEATABLE READ versus READ COMMITTED transaction modes for ordinary SQL SELECT commands.

An example:

```
inv_fd = lo_open(conn, inv_oid, INV_READ|INV_WRITE);
```

32.3.5. Writing Data to a Large Object

The function

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len);
```

writes `len` bytes from `buf` (which must be of size `len`) to large object descriptor `fd`. The `fd` argument must have been returned by a previous `lo_open`. The number of bytes actually written is returned (in the current implementation, this will always equal `len` unless there is an error). In the event of an error, the return value is -1.

Although the `len` parameter is declared as `size_t`, this function will reject length values larger than `INT_MAX`. In practice, it's best to transfer data in chunks of at most a few megabytes anyway.

32.3.6. Reading Data from a Large Object

The function

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len);
```

reads up to `len` bytes from large object descriptor `fd` into `buf` (which must be of size `len`). The `fd` argument must have been returned by a previous `lo_open`. The number of bytes actually read is returned; this will be less than `len` if the end of the large object is reached first. In the event of an error, the return value is -1.

Although the `len` parameter is declared as `size_t`, this function will reject length values larger than `INT_MAX`. In practice, it's best to transfer data in chunks of at most a few megabytes anyway.

32.3.7. Seeking in a Large Object

To change the current read or write location associated with a large object descriptor, call

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence);
```

This function moves the current location pointer for the large object descriptor identified by `fd` to the new location specified by `offset`. The valid values for `whence` are `SEEK_SET` (seek from object start), `SEEK_CUR` (seek from current position), and `SEEK_END` (seek from object end). The return value is the new location pointer, or -1 on error.

When dealing with large objects that might exceed 2GB in size, instead use

```
pg_int64 lo_lseek64(PGconn *conn, int fd, pg_int64 offset, int whence);
```

This function has the same behavior as `lo_lseek`, but it can accept an `offset` larger than 2GB and/or deliver a result larger than 2GB. Note that `lo_lseek` will fail if the new location pointer would be greater than 2GB.

`lo_lseek64` is new as of PostgreSQL 9.3. If this function is run against an older server version, it will fail and return -1.

32.3.8. Obtaining the Seek Position of a Large Object

To obtain the current read or write location of a large object descriptor, call

```
int lo_tell(PGconn *conn, int fd);
```

If there is an error, the return value is -1.

When dealing with large objects that might exceed 2GB in size, instead use

```
pg_int64 lo_tell64(PGconn *conn, int fd);
```

This function has the same behavior as `lo_tell`, but it can deliver a result larger than 2GB. Note that `lo_tell` will fail if the current read/write location is greater than 2GB.

`lo_tell64` is new as of PostgreSQL 9.3. If this function is run against an older server version, it will fail and return -1.

32.3.9. Truncating a Large Object

To truncate a large object to a given length, call

```
int lo_truncate(PGcon *conn, int fd, size_t len);
```

This function truncates the large object descriptor `fd` to length `len`. The `fd` argument must have been returned by a previous `lo_open`. If `len` is greater than the large object's current length, the large object is extended to the specified length with null bytes ('\0'). On success, `lo_truncate` returns zero. On error, the return value is -1.

The read/write location associated with the descriptor `fd` is not changed.

Although the `len` parameter is declared as `size_t`, `lo_truncate` will reject length values larger than `INT_MAX`.

When dealing with large objects that might exceed 2GB in size, instead use

```
int lo_truncate64(PGcon *conn, int fd, pg_int64 len);
```

This function has the same behavior as `lo_truncate`, but it can accept a `len` value exceeding 2GB.

`lo_truncate` is new as of PostgreSQL 8.3; if this function is run against an older server version, it will fail and return -1.

`lo_truncate64` is new as of PostgreSQL 9.3; if this function is run against an older server version, it will fail and return -1.

32.3.10. Closing a Large Object Descriptor

A large object descriptor can be closed by calling

```
int lo_close(PGconn *conn, int fd);
```

where `fd` is a large object descriptor returned by `lo_open`. On success, `lo_close` returns zero. On error, the return value is -1.

Any large object descriptors that remain open at the end of a transaction will be closed automatically.

32.3.11. Removing a Large Object

To remove a large object from the database, call

```
int lo_unlink(PGconn *conn, Oid lobjId);
```

The `lobjId` argument specifies the OID of the large object to remove. Returns 1 if successful, -1 on failure.

32.4. Server-side Functions

Server-side functions tailored for manipulating large objects from SQL are listed in Table 32-1.

Table 32-1. SQL-oriented Large Object Functions

Function	Return Type	Description	Example	Result
`lo_from_bytea(loid oid, string bytea)`	`oid`	Create a large object and store data there, returning its OID. Pass `0` to have the system choose an OID.	`lo_from_bytea(0, '\xffffff00')`	`24528`
`lo_put(loid oid, offset bigint, str bytea)`	`void`	Write data at the given offset.	`lo_put(24528, 1, '\xaa')`	
`lo_get(loid oid [, from bigint, for int])`	`bytea`	Extract contents or a substring thereof.	`lo_get(24528, 0, 3)`	`\xffaaff`

There are additional server-side functions corresponding to each of the client-side functions described earlier; indeed, for the most part the client-side functions are simply interfaces to the equivalent server-side functions. The ones just as convenient to call via SQL commands are `lo_creat`, `lo_create`, `lo_unlink`, `lo_import`, and `lo_export`. Here are examples of their use:

```
CREATE TABLE image (
    name            text,
    raster          oid
);

SELECT lo_creat(-1);       -- returns OID of new, empty large object

SELECT lo_create(43213);   -- attempts to create large object with OID 43213

SELECT lo_unlink(173454);  -- deletes large object with OID 173454

INSERT INTO image (name, raster)
    VALUES ('beautiful image', lo_import('/etc/motd'));

INSERT INTO image (name, raster)  -- same as above, but specify OID to use
    VALUES ('beautiful image', lo_import('/etc/motd', 68583));

SELECT lo_export(image.raster, '/tmp/motd') FROM image
    WHERE name = 'beautiful image';
```

The server-side `lo_import` and `lo_export` functions behave considerably differently from their client-side analogs. These two functions read and write files in the server's file system, using the permissions of the database's owning user. Therefore, their use is restricted to superusers. In contrast, the client-side import and export functions read and write files in the client's file system, using the permissions of the client program. The client-side functions do not require superuser privilege.

The functionality of `lo_read` and `lo_write` is also available via server-side calls, but the names of the server-side functions differ from the client side interfaces in that they do not contain underscores. You must call these functions as `loread` and `lowrite`.

32.5. Example Program

Example 32-1 is a sample program which shows how the large object interface in libpq can be used. Parts of the program are commented out but are left in the source for the reader's benefit. This program can also be found in `src/test/examples/testlo.c` in the source distribution.

Example 32-1. Large Objects with libpq Example Program

```
/*--------------------------------------------------------------------------
 *
 * testlo.c
 *    test using large objects with libpq
 *
 * Portions Copyright (c) 1996-2015, PostgreSQL Global Development Group
 * Portions Copyright (c) 1994, Regents of the University of California
 *
 *
 * IDENTIFICATION
 *    src/test/examples/testlo.c
```

```
 *
 *-------------------------------------------------------------------------
 */
#include <stdio.h>
#include <stdlib.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE         1024

/*
 * importFile -
 *    import file "in_filename" into database as large object "lobjOid"
 *
 */
static Oid
importFile(PGconn *conn, char *filename)
{
    Oid         lobjId;
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * open the file to be read in
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)
    {                           /* error */
        fprintf(stderr, "cannot open unix file\"%s\"\n", filename);
    }

    /*
     * create the large object
     */
    lobjId = lo_creat(conn, INV_READ | INV_WRITE);
    if (lobjId == 0)
        fprintf(stderr, "cannot create large object");

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);

    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0)
```

```
    {
        tmp = lo_write(conn, lobj_fd, buf, nbytes);
        if (tmp < nbytes)
            fprintf(stderr, "error while reading \"%s\"", filename);
    }

    close(fd);
    lo_close(conn, lobj_fd);

    return lobjId;
}

static void
pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    nread = 0;
    while (len - nread > 0)
    {
        nbytes = lo_read(conn, lobj_fd, buf, len - nread);
        buf[nbytes] = '\0';
        fprintf(stderr, ">>> %s", buf);
        nread += nbytes;
        if (nbytes <= 0)
            break;              /* no more data? */
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}

static void
overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int         lobj_fd;
    char       *buf;
    int         nbytes;
    int         nwritten;
    int         i;

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);
    if (lobj_fd < 0)
```

```
        fprintf(stderr, "cannot open large object %u", lobjId);

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len + 1);

    for (i = 0; i < len; i++)
        buf[i] = 'X';
    buf[i] = '\0';

    nwritten = 0;
    while (len - nwritten > 0)
    {
        nbytes = lo_write(conn, lobj_fd, buf + nwritten, len - nwritten);
        nwritten += nbytes;
        if (nbytes <= 0)
        {
            fprintf(stderr, "\nWRITE FAILED!\n");
            break;
        }
    }
    free(buf);
    fprintf(stderr, "\n");
    lo_close(conn, lobj_fd);
}


/*
 * exportFile -
 *    export large object "lobjOid" to file "out_filename"
 *
 */
static void
exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int         lobj_fd;
    char        buf[BUFSIZE];
    int         nbytes,
                tmp;
    int         fd;

    /*
     * open the large object
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0)
        fprintf(stderr, "cannot open large object %u", lobjId);

    /*
     * open the file to be written to
     */
    fd = open(filename, O_CREAT | O_WRONLY | O_TRUNC, 0666);
    if (fd < 0)
    {                           /* error */
```

```
        fprintf(stderr, "cannot open unix file\"%s\"",
                filename);
    }

    /*
     * read in from the inversion file and write to the Unix file
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0)
    {
        tmp = write(fd, buf, nbytes);
        if (tmp < nbytes)
        {
            fprintf(stderr, "error while writing \"%s\"",
                    filename);
        }
    }

    lo_close(conn, lobj_fd);
    close(fd);

    return;
}

static void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

int
main(int argc, char **argv)
{
    char       *in_filename,
               *out_filename;
    char       *database;
    Oid         lobjOid;
    PGconn     *conn;
    PGresult   *res;

    if (argc != 4)
    {
        fprintf(stderr, "Usage: %s database_name in_filename out_filename\n",
                argv[0]);
        exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * set up the connection
```

```
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* check to see that the backend connection was successfully made */
    if (PQstatus(conn) != CONNECTION_OK)
    {
        fprintf(stderr, "Connection to database failed: %s",
                PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* Set always-secure search path, so malicious users can't take control. */
    res = PQexec(conn,
                 "SELECT pg_catalog.set_config('search_path', '', false)");
    if (PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "begin");
    PQclear(res);
    printf("importing file \"%s\" ...\n", in_filename);
/*  lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
    if (lobjOid == 0)
        fprintf(stderr, "%s\n", PQerrorMessage(conn));
    else
    {
        printf("\tas large object %u.\n", lobjOid);

        printf("picking out bytes 1000-2000 of the large object\n");
        pickout(conn, lobjOid, 1000, 1000);

        printf("overwriting bytes 1000-2000 of the large object with X's\n");
        overwrite(conn, lobjOid, 1000, 1000);

        printf("exporting large object to file \"%s\" ...\n", out_filename);
/*      exportFile(conn, lobjOid, out_filename); */
        if (lo_export(conn, lobjOid, out_filename) < 0)
            fprintf(stderr, "%s\n", PQerrorMessage(conn));
    }

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    return 0;
}
```

Chapter 33. ECPG - Embedded SQL in C

This chapter describes the embedded SQL package for PostgreSQL. It was written by Linus Tolke (<linus@epact.se>) and Michael Meskes (<meskes@postgresql.org>). Originally it was written to work with C. It also works with C++, but it does not recognize all C++ constructs yet.

This documentation is quite incomplete. But since this interface is standardized, additional information can be found in many resources about SQL.

33.1. The Concept

An embedded SQL program consists of code written in an ordinary programming language, in this case C, mixed with SQL commands in specially marked sections. To build the program, the source code (`*.pgc`) is first passed through the embedded SQL preprocessor, which converts it to an ordinary C program (`*.c`), and afterwards it can be processed by a C compiler. (For details about the compiling and linking see Section 33.10). Converted ECPG applications call functions in the libpq library through the embedded SQL library (ecpglib), and communicate with the PostgreSQL server using the normal frontend-backend protocol.

Embedded SQL has advantages over other methods for handling SQL commands from C code. First, it takes care of the tedious passing of information to and from variables in your C program. Second, the SQL code in the program is checked at build time for syntactical correctness. Third, embedded SQL in C is specified in the SQL standard and supported by many other SQL database systems. The PostgreSQL implementation is designed to match this standard as much as possible, and it is usually possible to port embedded SQL programs written for other SQL databases to PostgreSQL with relative ease.

As already stated, programs written for the embedded SQL interface are normal C programs with special code inserted to perform database-related actions. This special code always has the form:

```
EXEC SQL ...;
```

These statements syntactically take the place of a C statement. Depending on the particular statement, they can appear at the global level or within a function. Embedded SQL statements follow the case-sensitivity rules of normal SQL code, and not those of C. Also they allow nested C-style comments that are part of the SQL standard. The C part of the program, however, follows the C standard of not accepting nested comments.

The following sections explain all the embedded SQL statements.

33.2. Managing Database Connections

This section describes how to open, close, and switch database connections.

33.2.1. Connecting to the Database Server

One connects to a database using the following statement:

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name];
```

The *target* can be specified in the following ways:

- *dbname*[@*hostname*][:*port*]
- tcp:postgresql://*hostname*[:*port*][/*dbname*][?*options*]
- unix:postgresql://*hostname*[:*port*][/*dbname*][?*options*]
- an SQL string literal containing one of the above forms
- a reference to a character variable containing one of the above forms (see examples)
- DEFAULT

If you specify the connection target literally (that is, not through a variable reference) and you don't quote the value, then the case-insensitivity rules of normal SQL are applied. In that case you can also double-quote the individual parameters separately as needed. In practice, it is probably less error-prone to use a (single-quoted) string literal or a variable reference. The connection target DEFAULT initiates a connection to the default database under the default user name. No separate user name or connection name can be specified in that case.

There are also different ways to specify the user name:

- *username*
- *username*/*password*
- *username* IDENTIFIED BY *password*
- *username* USING *password*

As above, the parameters *username* and *password* can be an SQL identifier, an SQL string literal, or a reference to a character variable.

If the connection target includes any *options*, those consist of *keyword*=*value* specifications separated by ampersands (&). The allowed key words are the same ones recognized by libpq (see Section 31.1.2). Spaces are ignored before any *keyword* or *value*, though not within or after one. Note that there is no way to write & within a *value*.

The *connection-name* is used to handle multiple connections in one program. It can be omitted if a program uses only one connection. The most recently opened connection becomes the current connection, which is used by default when an SQL statement is to be executed (see later in this chapter).

If untrusted users have access to a database that has not adopted a secure schema usage pattern, begin each session by removing publicly-writable schemas from search_path. For example, add options=-c search_path= to *options*, or issue EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); after connecting. This consideration is not specific to ECPG; it applies to every interface for executing arbitrary SQL commands.

Here are some examples of CONNECT statements:

```
EXEC SQL CONNECT TO mydb@sql.mydomain.com;

EXEC SQL CONNECT TO unix:postgresql://sql.mydomain.com/mydb AS myconnection USER john;

EXEC SQL BEGIN DECLARE SECTION;
const char *target = "mydb@sql.mydomain.com";
const char *user = "john";
const char *passwd = "secret";
EXEC SQL END DECLARE SECTION;
 ...
EXEC SQL CONNECT TO :target USER :user USING :passwd;
/* or EXEC SQL CONNECT TO :target USER :user/:passwd; */
```

The last form makes use of the variant referred to above as character variable reference. You will see in later sections how C variables can be used in SQL statements when you prefix them with a colon.

Be advised that the format of the connection target is not specified in the SQL standard. So if you want to develop portable applications, you might want to use something based on the last example above to encapsulate the connection target string somewhere.

33.2.2. Choosing a Connection

SQL statements in embedded SQL programs are by default executed on the current connection, that is, the most recently opened one. If an application needs to manage multiple connections, then there are two ways to handle this.

The first option is to explicitly choose a connection for each SQL statement, for example:

```
EXEC SQL AT connection-name SELECT ...;
```

This option is particularly suitable if the application needs to use several connections in mixed order.

If your application uses multiple threads of execution, they cannot share a connection concurrently. You must either explicitly control access to the connection (using mutexes) or use a connection for each thread.

The second option is to execute a statement to switch the current connection. That statement is:

```
EXEC SQL SET CONNECTION connection-name;
```

This option is particularly convenient if many statements are to be executed on the same connection.

Here is an example program managing multiple database connections:

```
#include <stdio.h>

EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
EXEC SQL END DECLARE SECTION;

int
main()
{
    EXEC SQL CONNECT TO testdb1 AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
```

```
    EXEC SQL CONNECT TO testdb2 AS con2 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL CONNECT TO testdb3 AS con3 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    /* This query would be executed in the last opened database "testdb3". */
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb3)\n", dbname);

    /* Using "AT" to run a query in "testdb2" */
    EXEC SQL AT con2 SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb2)\n", dbname);

    /* Switch the current connection to "testdb1". */
    EXEC SQL SET CONNECTION con1;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current=%s (should be testdb1)\n", dbname);

    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

This example would produce this output:

```
current=testdb3 (should be testdb3)
current=testdb2 (should be testdb2)
current=testdb1 (should be testdb1)
```

33.2.3. Closing a Connection

To close a connection, use the following statement:

```
EXEC SQL DISCONNECT [connection];
```

The *connection* can be specified in the following ways:

- *connection-name*
- DEFAULT
- CURRENT
- ALL

If no connection name is specified, the current connection is closed.

It is good style that an application always explicitly disconnect from every connection it opened.

33.3. Running SQL Commands

Any SQL command can be run from within an embedded SQL application. Below are some examples of how to do that.

33.3.1. Executing SQL Statements

Creating a table:

```
EXEC SQL CREATE TABLE foo (number integer, ascii char(16));
EXEC SQL CREATE UNIQUE INDEX num1 ON foo(number);
EXEC SQL COMMIT;
```

Inserting rows:

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

Deleting rows:

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

Updates:

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE number = 9999;
EXEC SQL COMMIT;
```

`SELECT` statements that return a single result row can also be executed using `EXEC SQL` directly. To handle result sets with multiple rows, an application has to use a cursor; see Section 33.3.2 below. (As a special case, an application can fetch multiple rows at once into an array host variable; see Section 33.4.4.3.1.)

Single-row select:

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad';
```

Also, a configuration parameter can be retrieved with the `SHOW` command:

```
EXEC SQL SHOW search_path INTO :var;
```

The tokens of the form `:`*`something`* are *host variables*, that is, they refer to variables in the C program. They are explained in Section 33.4.

33.3.2. Using Cursors

To retrieve a result set holding multiple rows, an application has to declare a cursor and fetch each row from the cursor. The steps to use a cursor are the following: declare a cursor, open it, fetch a row from the cursor, repeat, and finally close it.

Select using cursors:

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii;
EXEC SQL OPEN foo_bar;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

For more details about declaration of the cursor, see DECLARE, and see FETCH for `FETCH` command details.

> **Note:** The ECPG `DECLARE` command does not actually cause a statement to be sent to the PostgreSQL backend. The cursor is opened in the backend (using the backend's `DECLARE` command) at the point when the `OPEN` command is executed.

33.3.3. Managing Transactions

In the default mode, statements are committed only when `EXEC SQL COMMIT` is issued. The embedded SQL interface also supports autocommit of transactions (similar to psql's default behavior) via the `-t` command-line option to `ecpg` (see ecpg) or via the `EXEC SQL SET AUTOCOMMIT TO ON` statement. In autocommit mode, each command is automatically committed unless it is inside an explicit transaction block. This mode can be explicitly turned off using `EXEC SQL SET AUTOCOMMIT TO OFF`.

The following transaction management commands are available:

`EXEC SQL COMMIT`

Commit an in-progress transaction.

`EXEC SQL ROLLBACK`

Roll back an in-progress transaction.

`EXEC SQL SET AUTOCOMMIT TO ON`

Enable autocommit mode.

`SET AUTOCOMMIT TO OFF`

Disable autocommit mode. This is the default.

33.3.4. Prepared Statements

When the values to be passed to an SQL statement are not known at compile time, or the same statement is going to be used many times, then prepared statements can be useful.

The statement is prepared using the command `PREPARE`. For the values that are not known yet, use the placeholder "`?`":

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid, datname FROM pg_database WHERE oid = ?";
```

If a statement returns a single row, the application can call `EXECUTE` after `PREPARE` to execute the statement, supplying the actual values for the placeholders with a `USING` clause:

```
EXEC SQL EXECUTE stmt1 INTO :dboid, :dbname USING 1;
```

If a statement returns multiple rows, the application can use a cursor declared based on the prepared statement. To bind input parameters, the cursor must be opened with a `USING` clause:

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid,datname FROM pg_database WHERE oid > ?";
EXEC SQL DECLARE foo_bar CURSOR FOR stmt1;

/* when end of result set reached, break out of while loop */
EXEC SQL WHENEVER NOT FOUND DO BREAK;

EXEC SQL OPEN foo_bar USING 100;
...
while (1)
{
    EXEC SQL FETCH NEXT FROM foo_bar INTO :dboid, :dbname;
    ...
}
EXEC SQL CLOSE foo_bar;
```

When you don't need the prepared statement anymore, you should deallocate it:

```
EXEC SQL DEALLOCATE PREPARE name;
```

For more details about `PREPARE`, see PREPARE. Also see Section 33.5 for more details about using placeholders and input parameters.

33.4. Using Host Variables

In Section 33.3 you saw how you can execute SQL statements from an embedded SQL program. Some of those statements only used fixed values and did not provide a way to insert user-supplied values into

statements or have the program process the values returned by the query. Those kinds of statements are not really useful in real applications. This section explains in detail how you can pass data between your C program and the embedded SQL statements using a simple mechanism called *host variables*. In an embedded SQL program we consider the SQL statements to be *guests* in the C program code which is the *host language*. Therefore the variables of the C program are called *host variables*.

Another way to exchange values between PostgreSQL backends and ECPG applications is the use of SQL descriptors, described in Section 33.7.

33.4.1. Overview

Passing data between the C program and the SQL statements is particularly simple in embedded SQL. Instead of having the program paste the data into the statement, which entails various complications, such as properly quoting the value, you can simply write the name of a C variable into the SQL statement, prefixed by a colon. For example:

```
EXEC SQL INSERT INTO sometable VALUES (:v1, 'foo', :v2);
```

This statement refers to two C variables named `v1` and `v2` and also uses a regular SQL string literal, to illustrate that you are not restricted to use one kind of data or the other.

This style of inserting C variables in SQL statements works anywhere a value expression is expected in an SQL statement.

33.4.2. Declare Sections

To pass data from the program to the database, for example as parameters in a query, or to pass data from the database back to the program, the C variables that are intended to contain this data need to be declared in specially marked sections, so the embedded SQL preprocessor is made aware of them.

This section starts with:

```
EXEC SQL BEGIN DECLARE SECTION;
```

and ends with:

```
EXEC SQL END DECLARE SECTION;
```

Between those lines, there must be normal C variable declarations, such as:

```
int   x = 4;
char  foo[16], bar[16];
```

As you can see, you can optionally assign an initial value to the variable. The variable's scope is determined by the location of its declaring section within the program. You can also declare variables with the following syntax which implicitly creates a declare section:

```
EXEC SQL int i = 4;
```

You can have as many declare sections in a program as you like.

The declarations are also echoed to the output file as normal C variables, so there's no need to declare them again. Variables that are not intended to be used in SQL commands can be declared normally outside these special sections.

The definition of a structure or union also must be listed inside a `DECLARE` section. Otherwise the preprocessor cannot handle these types since it does not know the definition.

33.4.3. Retrieving Query Results

Now you should be able to pass data generated by your program into an SQL command. But how do you retrieve the results of a query? For that purpose, embedded SQL provides special variants of the usual commands `SELECT` and `FETCH`. These commands have a special `INTO` clause that specifies which host variables the retrieved values are to be stored in. `SELECT` is used for a query that returns only single row, and `FETCH` is used for a query that returns multiple rows, using a cursor.

Here is an example:

```
/*
 * assume this table:
 * CREATE TABLE test1 (a int, b varchar(50));
 */

EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL SELECT a, b INTO :v1, :v2 FROM test;
```

So the `INTO` clause appears between the select list and the `FROM` clause. The number of elements in the select list and the list after `INTO` (also called the target list) must be equal.

Here is an example using the command `FETCH`:

```
EXEC SQL BEGIN DECLARE SECTION;
int v1;
VARCHAR v2;
EXEC SQL END DECLARE SECTION;

 ...

EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test;

 ...

do
{
    ...
    EXEC SQL FETCH NEXT FROM foo INTO :v1, :v2;
    ...
```

```
} while (...);
```

Here the `INTO` clause appears after all the normal clauses.

33.4.4. Type Mapping

When ECPG applications exchange values between the PostgreSQL server and the C application, such as when retrieving query results from the server or executing SQL statements with input parameters, the values need to be converted between PostgreSQL data types and host language variable types (C language data types, concretely). One of the main points of ECPG is that it takes care of this automatically in most cases.

In this respect, there are two kinds of data types: Some simple PostgreSQL data types, such as `integer` and `text`, can be read and written by the application directly. Other PostgreSQL data types, such as `timestamp` and `numeric` can only be accessed through special library functions; see Section 33.4.4.2.

Table 33-1 shows which PostgreSQL data types correspond to which C data types. When you wish to send or receive a value of a given PostgreSQL data type, you should declare a C variable of the corresponding C data type in the declare section.

Table 33-1. Mapping Between PostgreSQL Data Types and C Variable Types

PostgreSQL data type	Host variable type
`smallint`	`short`
`integer`	`int`
`bigint`	`long long int`
`decimal`	`decimal`[a]
`numeric`	`numeric`[a]
`real`	`float`
`double precision`	`double`
`smallserial`	`short`
`serial`	`int`
`bigserial`	`long long int`
`oid`	`unsigned int`
`character(`*n*`)`, `varchar(`*n*`)`, `text`	`char[`*n*`+1]`, `VARCHAR[`*n*`+1]`[b]
`name`	`char[NAMEDATALEN]`
`timestamp`	`timestamp`[a]
`interval`	`interval`[a]
`date`	`date`[a]
`boolean`	`bool`[c]

Notes:
a. This type can only be accessed through special library functions; see Section 33.4.4.2.
b. declared in `ecpglib.h`
c. declared in `ecpglib.h` if not native

33.4.4.1. Handling Character Strings

To handle SQL character string data types, such as `varchar` and `text`, there are two possible ways to declare the host variables.

One way is using `char[]`, an array of `char`, which is the most common way to handle character data in C.

```
EXEC SQL BEGIN DECLARE SECTION;
    char str[50];
EXEC SQL END DECLARE SECTION;
```

Note that you have to take care of the length yourself. If you use this host variable as the target variable of a query which returns a string with more than 49 characters, a buffer overflow occurs.

The other way is using the `VARCHAR` type, which is a special type provided by ECPG. The definition on an array of type `VARCHAR` is converted into a named `struct` for every variable. A declaration like:

```
VARCHAR var[180];
```

is converted into:

```
struct varchar_var { int len; char arr[180]; } var;
```

The member `arr` hosts the string including a terminating zero byte. Thus, to store a string in a `VARCHAR` host variable, the host variable has to be declared with the length including the zero byte terminator. The member `len` holds the length of the string stored in the `arr` without the terminating zero byte. When a host variable is used as input for a query, if `strlen(arr)` and `len` are different, the shorter one is used.

`VARCHAR` can be written in upper or lower case, but not in mixed case.

`char` and `VARCHAR` host variables can also hold values of other SQL types, which will be stored in their string forms.

33.4.4.2. Accessing Special Data Types

ECPG contains some special types that help you to interact easily with some special data types from the PostgreSQL server. In particular, it has implemented support for the `numeric`, `decimal`, `date`, `timestamp`, and `interval` types. These data types cannot usefully be mapped to primitive host variable types (such as `int`, `long long int`, or `char[]`), because they have a complex internal structure. Applications deal with these types by declaring host variables in special types and accessing them using functions in the pgtypes library. The pgtypes library, described in detail in Section 33.6 contains basic functions to deal with those types, such that you do not need to send a query to the SQL server just for adding an interval to a time stamp for example.

The follow subsections describe these special data types. For more details about pgtypes library functions, see Section 33.6.

33.4.4.2.1. timestamp, date

Here is a pattern for handling `timestamp` variables in the ECPG host application.

First, the program has to include the header file for the `timestamp` type:

```
#include <pgtypes_timestamp.h>
```

Next, declare a host variable as type `timestamp` in the declare section:

```
EXEC SQL BEGIN DECLARE SECTION;
timestamp ts;
EXEC SQL END DECLARE SECTION;
```

And after reading a value into the host variable, process it using pgtypes library functions. In following example, the `timestamp` value is converted into text (ASCII) form with the `PGTYPEStimestamp_to_asc()` function:

```
EXEC SQL SELECT now()::timestamp INTO :ts;

printf("ts = %s\n", PGTYPEStimestamp_to_asc(ts));
```

This example will show some result like following:

```
ts = 2010-06-27 18:03:56.949343
```

In addition, the DATE type can be handled in the same way. The program has to include `pgtypes_date.h`, declare a host variable as the date type and convert a DATE value into a text form using `PGTYPESdate_to_asc()` function. For more details about the pgtypes library functions, see Section 33.6.

33.4.4.2.2. interval

The handling of the `interval` type is also similar to the `timestamp` and `date` types. It is required, however, to allocate memory for an `interval` type value explicitly. In other words, the memory space for the variable has to be allocated in the heap memory, not in the stack memory.

Here is an example program:

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_interval.h>

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    interval *in;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    in = PGTYPESinterval_new();
```

```
    EXEC SQL SELECT '1 min'::interval INTO :in;
    printf("interval = %s\n", PGTYPESinterval_to_asc(in));
    PGTYPESinterval_free(in);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

33.4.4.2.3. numeric, decimal

The handling of the `numeric` and `decimal` types is similar to the `interval` type: It requires defining a pointer, allocating some memory space on the heap, and accessing the variable using the pgtypes library functions. For more details about the pgtypes library functions, see Section 33.6.

No functions are provided specifically for the `decimal` type. An application has to convert it to a `numeric` variable using a pgtypes library function to do further processing.

Here is an example program handling `numeric` and `decimal` type variables.

```
#include <stdio.h>
#include <stdlib.h>
#include <pgtypes_numeric.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    numeric *num;
    numeric *num2;
    decimal *dec;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    num = PGTYPESnumeric_new();
    dec = PGTYPESdecimal_new();

    EXEC SQL SELECT 12.345::numeric(4,2), 23.456::decimal(4,2) INTO :num, :dec;

    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 0));
    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 1));
    printf("numeric = %s\n", PGTYPESnumeric_to_asc(num, 2));

    /* Convert decimal to numeric to show a decimal value. */
    num2 = PGTYPESnumeric_new();
    PGTYPESnumeric_from_decimal(dec, num2);
```

```
    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 0));
    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 1));
    printf("decimal = %s\n", PGTYPESnumeric_to_asc(num2, 2));

    PGTYPESnumeric_free(num2);
    PGTYPESdecimal_free(dec);
    PGTYPESnumeric_free(num);

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

33.4.4.3. Host Variables with Nonprimitive Types

As a host variable you can also use arrays, typedefs, structs, and pointers.

33.4.4.3.1. Arrays

There are two use cases for arrays as host variables. The first is a way to store some text string in `char[]` or `VARCHAR[]`, as explained in Section 33.4.4.1. The second use case is to retrieve multiple rows from a query result without using a cursor. Without an array, to process a query result consisting of multiple rows, it is required to use a cursor and the `FETCH` command. But with array host variables, multiple rows can be received at once. The length of the array has to be defined to be able to accommodate all rows, otherwise a buffer overflow will likely occur.

Following example scans the `pg_database` system table and shows all OIDs and names of the available databases:

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int dbid[8];
    char dbname[8][16];
    int i;
EXEC SQL END DECLARE SECTION;

    memset(dbname, 0, sizeof(char)* 16 * 8);
    memset(dbid, 0, sizeof(int) * 8);

    EXEC SQL CONNECT TO testdb;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    /* Retrieve multiple rows into arrays at once. */
    EXEC SQL SELECT oid,datname INTO :dbid, :dbname FROM pg_database;

    for (i = 0; i < 8; i++)
        printf("oid=%d, dbname=%s\n", dbid[i], dbname[i]);
```

```
    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

This example shows following result. (The exact values depend on local circumstances.)

```
oid=1, dbname=template1
oid=11510, dbname=template0
oid=11511, dbname=postgres
oid=313780, dbname=testdb
oid=0, dbname=
oid=0, dbname=
oid=0, dbname=
```

33.4.4.3.2. Structures

A structure whose member names match the column names of a query result, can be used to retrieve multiple columns at once. The structure enables handling multiple column values in a single host variable.

The following example retrieves OIDs, names, and sizes of the available databases from the `pg_database` system table and using the `pg_database_size()` function. In this example, a structure variable `dbinfo_t` with members whose names match each column in the `SELECT` result is used to retrieve one result row without putting multiple host variables in the `FETCH` statement.

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
       int oid;
       char datname[65];
       long long int size;
    } dbinfo_t;

    dbinfo_t dbval;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size FRO
    EXEC SQL OPEN cur1;

    /* when end of result set reached, break out of while loop */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        /* Fetch multiple columns into one structure. */
        EXEC SQL FETCH FROM cur1 INTO :dbval;

        /* Print members of the structure. */
```

```
        printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname, dbval.size);
    }

    EXEC SQL CLOSE cur1;
```

This example shows following result. (The exact values depend on local circumstances.)

```
oid=1, datname=template1, size=4324580
oid=11510, datname=template0, size=4243460
oid=11511, datname=postgres, size=4324580
oid=313780, datname=testdb, size=8183012
```

Structure host variables "absorb" as many columns as the structure as fields. Additional columns can be assigned to other host variables. For example, the above program could also be restructured like this, with the `size` variable outside the structure:

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef struct
    {
       int oid;
       char datname[65];
    } dbinfo_t;

    dbinfo_t dbval;
    long long int size;
EXEC SQL END DECLARE SECTION;

    memset(&dbval, 0, sizeof(dbinfo_t));

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size FRO
    EXEC SQL OPEN cur1;

    /* when end of result set reached, break out of while loop */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        /* Fetch multiple columns into one structure. */
        EXEC SQL FETCH FROM cur1 INTO :dbval, :size;

        /* Print members of the structure. */
        printf("oid=%d, datname=%s, size=%lld\n", dbval.oid, dbval.datname, size);
    }

    EXEC SQL CLOSE cur1;
```

33.4.4.3.3. Typedefs

Use the `typedef` keyword to map new types to already existing types.

```
EXEC SQL BEGIN DECLARE SECTION;
    typedef char mychartype[40];
    typedef long serial_t;
EXEC SQL END DECLARE SECTION;
```

Note that you could also use:

```
EXEC SQL TYPE serial_t IS long;
```

This declaration does not need to be part of a declare section.

33.4.4.3.4. Pointers

You can declare pointers to the most common types. Note however that you cannot use pointers as target variables of queries without auto-allocation. See Section 33.7 for more information on auto-allocation.

```
EXEC SQL BEGIN DECLARE SECTION;
    int   *intp;
    char **charp;
EXEC SQL END DECLARE SECTION;
```

33.4.5. Handling Nonprimitive SQL Data Types

This section contains information on how to handle nonscalar and user-defined SQL-level data types in ECPG applications. Note that this is distinct from the handling of host variables of nonprimitive types, described in the previous section.

33.4.5.1. Arrays

Multi-dimensional SQL-level arrays are not directly supported in ECPG. One-dimensional SQL-level arrays can be mapped into C array host variables and vice-versa. However, when creating a statement ecpg does not know the types of the columns, so that it cannot check if a C array is input into a corresponding SQL-level array. When processing the output of a SQL statement, ecpg has the necessary information and thus checks if both are arrays.

If a query accesses *elements* of an array separately, then this avoids the use of arrays in ECPG. Then, a host variable with a type that can be mapped to the element type should be used. For example, if a column type is array of `integer`, a host variable of type `int` can be used. Also if the element type is `varchar` or `text`, a host variable of type `char[]` or `VARCHAR[]` can be used.

Here is an example. Assume the following table:

```
CREATE TABLE t3 (
```

```
    ii integer[]
);

testdb=> SELECT * FROM t3;
     ii
-------------
 {1,2,3,4,5}
(1 row)
```

The following example program retrieves the 4th element of the array and stores it into a host variable of type `int`:

```
EXEC SQL BEGIN DECLARE SECTION;
int ii;
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[4] FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :ii ;
    printf("ii=%d\n", ii);
}

EXEC SQL CLOSE cur1;
```

This example shows the following result:

```
ii=4
```

To map multiple array elements to the multiple elements in an array type host variables each element of array column and each element of the host variable array have to be managed separately, for example:

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[1], ii[2], ii[3], ii[4] FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH FROM cur1 INTO :ii_a[0], :ii_a[1], :ii_a[2], :ii_a[3];
    ...
}
```

Note again that

```
EXEC SQL BEGIN DECLARE SECTION;
int ii_a[8];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii FROM t3;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* WRONG */
    EXEC SQL FETCH FROM cur1 INTO :ii_a;
    ...
}
```

would not work correctly in this case, because you cannot map an array type column to an array host variable directly.

Another workaround is to store arrays in their external string representation in host variables of type `char[]` or `VARCHAR[]`. For more details about this representation, see Section 8.15.2. Note that this means that the array cannot be accessed naturally as an array in the host program (without further processing that parses the text representation).

33.4.5.2. Composite Types

Composite types are not directly supported in ECPG, but an easy workaround is possible. The available workarounds are similar to the ones described for arrays above: Either access each attribute separately or use the external string representation.

For the following examples, assume the following type and table:

```
CREATE TYPE comp_t AS (intval integer, textval varchar(32));
CREATE TABLE t4 (compval comp_t);
INSERT INTO t4 VALUES ( (256, 'PostgreSQL') );
```

The most obvious solution is to access each attribute separately. The following program retrieves data from the example table by selecting each attribute of the type `comp_t` separately:

```
EXEC SQL BEGIN DECLARE SECTION;
int intval;
varchar textval[33];
EXEC SQL END DECLARE SECTION;

/* Put each element of the composite type column in the SELECT list. */
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
```

```
{
    /* Fetch each element of the composite type column into host variables. */
    EXEC SQL FETCH FROM cur1 INTO :intval, :textval;

    printf("intval=%d, textval=%s\n", intval, textval.arr);
}

EXEC SQL CLOSE cur1;
```

To enhance this example, the host variables to store values in the `FETCH` command can be gathered into one structure. For more details about the host variable in the structure form, see Section 33.4.4.3.2. To switch to the structure, the example can be modified as below. The two host variables, `intval` and `textval`, become members of the `comp_t` structure, and the structure is specified on the `FETCH` command.

```
EXEC SQL BEGIN DECLARE SECTION;
typedef struct
{
    int intval;
    varchar textval[33];
} comp_t;

comp_t compval;
EXEC SQL END DECLARE SECTION;

/* Put each element of the composite type column in the SELECT list. */
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    /* Put all values in the SELECT list into one structure. */
    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}

EXEC SQL CLOSE cur1;
```

Although a structure is used in the `FETCH` command, the attribute names in the `SELECT` clause are specified one by one. This can be enhanced by using a `*` to ask for all attributes of the composite type value.

```
...
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).* FROM t4;
EXEC SQL OPEN cur1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
```

```
    /* Put all values in the SELECT list into one structure. */
    EXEC SQL FETCH FROM cur1 INTO :compval;

    printf("intval=%d, textval=%s\n", compval.intval, compval.textval.arr);
}
...
```

This way, composite types can be mapped into structures almost seamlessly, even though ECPG does not understand the composite type itself.

Finally, it is also possible to store composite type values in their external string representation in host variables of type `char[]` or `VARCHAR[]`. But that way, it is not easily possible to access the fields of the value from the host program.

33.4.5.3. User-defined Base Types

New user-defined base types are not directly supported by ECPG. You can use the external string representation and host variables of type `char[]` or `VARCHAR[]`, and this solution is indeed appropriate and sufficient for many types.

Here is an example using the data type `complex` from the example in Section 35.11. The external string representation of that type is `(%f,%f)`, which is defined in the functions `complex_in()` and `complex_out()` functions in Section 35.11. The following example inserts the complex type values `(1,1)` and `(3,3)` into the columns `a` and `b`, and select them from the table after that.

```
EXEC SQL BEGIN DECLARE SECTION;
    varchar a[64];
    varchar b[64];
EXEC SQL END DECLARE SECTION;

    EXEC SQL INSERT INTO test_complex VALUES ('(1,1)', '(3,3)');

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT a, b FROM test_complex;
    EXEC SQL OPEN cur1;

    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        EXEC SQL FETCH FROM cur1 INTO :a, :b;
        printf("a=%s, b=%s\n", a.arr, b.arr);
    }

    EXEC SQL CLOSE cur1;
```

This example shows following result:

```
a=(1,1), b=(3,3)
```

Another workaround is avoiding the direct use of the user-defined types in ECPG and instead create a function or cast that converts between the user-defined type and a primitive type that ECPG can handle.

Note, however, that type casts, especially implicit ones, should be introduced into the type system very carefully.

For example,

```
CREATE FUNCTION create_complex(r double, i double) RETURNS complex
LANGUAGE SQL
IMMUTABLE
AS $$ SELECT $1 * complex '(1,0)' + $2 * complex '(0,1)' $$;
```

After this definition, the following

```
EXEC SQL BEGIN DECLARE SECTION;
double a, b, c, d;
EXEC SQL END DECLARE SECTION;

a = 1;
b = 2;
c = 3;
d = 4;

EXEC SQL INSERT INTO test_complex VALUES (create_complex(:a, :b), create_complex(:c, :d));
```

has the same effect as

```
EXEC SQL INSERT INTO test_complex VALUES ('(1,2)', '(3,4)');
```

33.4.6. Indicators

The examples above do not handle null values. In fact, the retrieval examples will raise an error if they fetch a null value from the database. To be able to pass null values to the database or retrieve null values from the database, you need to append a second host variable specification to each host variable that contains data. This second host variable is called the *indicator* and contains a flag that tells whether the datum is null, in which case the value of the real host variable is ignored. Here is an example that handles the retrieval of null values correctly:

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR val;
int val_ind;
EXEC SQL END DECLARE SECTION:

 ...

EXEC SQL SELECT b INTO :val :val_ind FROM test1;
```

The indicator variable `val_ind` will be zero if the value was not null, and it will be negative if the value was null.

The indicator has another function: if the indicator value is positive, it means that the value is not null, but it was truncated when it was stored in the host variable.

If the argument `-r no_indicator` is passed to the preprocessor `ecpg`, it works in "no-indicator" mode. In no-indicator mode, if no indicator variable is specified, null values are signaled (on input and output) for character string types as empty string and for integer types as the lowest possible value for type (for example, `INT_MIN` for `int`).

33.5. Dynamic SQL

In many cases, the particular SQL statements that an application has to execute are known at the time the application is written. In some cases, however, the SQL statements are composed at run time or provided by an external source. In these cases you cannot embed the SQL statements directly into the C source code, but there is a facility that allows you to call arbitrary SQL statements that you provide in a string variable.

33.5.1. Executing Statements without a Result Set

The simplest way to execute an arbitrary SQL statement is to use the command `EXECUTE IMMEDIATE`. For example:

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "CREATE TABLE test1 (...);";
EXEC SQL END DECLARE SECTION;

EXEC SQL EXECUTE IMMEDIATE :stmt;
```

`EXECUTE IMMEDIATE` can be used for SQL statements that do not return a result set (e.g., DDL, `INSERT`, `UPDATE`, `DELETE`). You cannot execute statements that retrieve data (e.g., `SELECT`) this way. The next section describes how to do that.

33.5.2. Executing a Statement with Input Parameters

A more powerful way to execute arbitrary SQL statements is to prepare them once and execute the prepared statement as often as you like. It is also possible to prepare a generalized version of a statement and then execute specific versions of it by substituting parameters. When preparing the statement, write question marks where you want to substitute parameters later. For example:

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "INSERT INTO test1 VALUES(?, ?);";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt USING 42, 'foobar';
```

When you don't need the prepared statement anymore, you should deallocate it:

```
EXEC SQL DEALLOCATE PREPARE name;
```

33.5.3. Executing a Statement with a Result Set

To execute an SQL statement with a single result row, `EXECUTE` can be used. To save the result, add an `INTO` clause.

```
EXEC SQL BEGIN DECLARE SECTION;
const char *stmt = "SELECT a, b, c FROM test1 WHERE a > ?";
int v1, v2;
VARCHAR v3[50];
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE mystmt FROM :stmt;
 ...
EXEC SQL EXECUTE mystmt INTO :v1, :v2, :v3 USING 37;
```

An `EXECUTE` command can have an `INTO` clause, a `USING` clause, both, or neither.

If a query is expected to return more than one result row, a cursor should be used, as in the following example. (See Section 33.3.2 for more details about the cursor.)

```
EXEC SQL BEGIN DECLARE SECTION;
char dbaname[128];
char datname[128];
char *stmt = "SELECT u.usename as dbaname, d.datname "
             "  FROM pg_database d, pg_user u "
             "  WHERE d.datdba = u.usesysid";
EXEC SQL END DECLARE SECTION;

EXEC SQL CONNECT TO testdb AS con1 USER testuser;
EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

EXEC SQL PREPARE stmt1 FROM :stmt;

EXEC SQL DECLARE cursor1 CURSOR FOR stmt1;
EXEC SQL OPEN cursor1;

EXEC SQL WHENEVER NOT FOUND DO BREAK;

while (1)
{
    EXEC SQL FETCH cursor1 INTO :dbaname,:datname;
    printf("dbaname=%s, datname=%s\n", dbaname, datname);
}

EXEC SQL CLOSE cursor1;
```

```
EXEC SQL COMMIT;
EXEC SQL DISCONNECT ALL;
```

33.6. pgtypes Library

The pgtypes library maps PostgreSQL database types to C equivalents that can be used in C programs. It also offers functions to do basic calculations with those types within C, i.e., without the help of the PostgreSQL server. See the following example:

```
EXEC SQL BEGIN DECLARE SECTION;
   date date1;
   timestamp ts1, tsout;
   interval iv1;
   char *out;
EXEC SQL END DECLARE SECTION;

PGTYPESdate_today(&date1);
EXEC SQL SELECT started, duration INTO :ts1, :iv1 FROM datetbl WHERE d=:date1;
PGTYPEStimestamp_add_interval(&ts1, &iv1, &tsout);
out = PGTYPEStimestamp_to_asc(&tsout);
printf("Started + duration: %s\n", out);
PGTYPESchar_free(out);
```

33.6.1. Character Strings

Some functions such as `PGTYPESnumeric_to_asc` return a pointer to a freshly allocated character string. These results should be freed with `PGTYPESchar_free` instead of `free`. (This is important only on Windows, where memory allocation and release sometimes need to be done by the same library.)

33.6.2. The numeric Type

The numeric type offers to do calculations with arbitrary precision. See Section 8.1 for the equivalent type in the PostgreSQL server. Because of the arbitrary precision this variable needs to be able to expand and shrink dynamically. That's why you can only create numeric variables on the heap, by means of the `PGTYPESnumeric_new` and `PGTYPESnumeric_free` functions. The decimal type, which is similar but limited in precision, can be created on the stack as well as on the heap.

The following functions can be used to work with the numeric type:

`PGTYPESnumeric_new`

Request a pointer to a newly allocated numeric variable.

```
numeric *PGTYPESnumeric_new(void);
```

`PGTYPESnumeric_free`

Free a numeric type, release all of its memory.

```
void PGTYPESnumeric_free(numeric *var);
```

`PGTYPESnumeric_from_asc`

Parse a numeric type from its string notation.

```
numeric *PGTYPESnumeric_from_asc(char *str, char **endptr);
```

Valid formats are for example: `-2`, `.794`, `+3.44`, `592.49E07` or `-32.84e-4`. If the value could be parsed successfully, a valid pointer is returned, else the NULL pointer. At the moment ECPG always parses the complete string and so it currently does not support to store the address of the first invalid character in `*endptr`. You can safely set `endptr` to NULL.

`PGTYPESnumeric_to_asc`

Returns a pointer to a string allocated by `malloc` that contains the string representation of the numeric type `num`.

```
char *PGTYPESnumeric_to_asc(numeric *num, int dscale);
```

The numeric value will be printed with `dscale` decimal digits, with rounding applied if necessary. The result must be freed with `PGTYPESchar_free()`.

`PGTYPESnumeric_add`

Add two numeric variables into a third one.

```
int PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result);
```

The function adds the variables `var1` and `var2` into the result variable `result`. The function returns 0 on success and -1 in case of error.

`PGTYPESnumeric_sub`

Subtract two numeric variables and return the result in a third one.

```
int PGTYPESnumeric_sub(numeric *var1, numeric *var2, numeric *result);
```

The function subtracts the variable `var2` from the variable `var1`. The result of the operation is stored in the variable `result`. The function returns 0 on success and -1 in case of error.

`PGTYPESnumeric_mul`

Multiply two numeric variables and return the result in a third one.

```
int PGTYPESnumeric_mul(numeric *var1, numeric *var2, numeric *result);
```

The function multiplies the variables `var1` and `var2`. The result of the operation is stored in the variable `result`. The function returns 0 on success and -1 in case of error.

`PGTYPESnumeric_div`

Divide two numeric variables and return the result in a third one.

```
int PGTYPESnumeric_div(numeric *var1, numeric *var2, numeric *result);
```

The function divides the variables `var1` by `var2`. The result of the operation is stored in the variable `result`. The function returns 0 on success and -1 in case of error.

`PGTYPESnumeric_cmp`

Compare two numeric variables.

```
int PGTYPESnumeric_cmp(numeric *var1, numeric *var2)
```

This function compares two numeric variables. In case of error, INT_MAX is returned. On success, the function returns one of three possible results:

- 1, if var1 is bigger than var2
- -1, if var1 is smaller than var2
- 0, if var1 and var2 are equal

PGTYPESnumeric_from_int

Convert an int variable to a numeric variable.

```
int PGTYPESnumeric_from_int(signed int int_val, numeric *var);
```

This function accepts a variable of type signed int and stores it in the numeric variable var. Upon success, 0 is returned and -1 in case of a failure.

PGTYPESnumeric_from_long

Convert a long int variable to a numeric variable.

```
int PGTYPESnumeric_from_long(signed long int long_val, numeric *var);
```

This function accepts a variable of type signed long int and stores it in the numeric variable var. Upon success, 0 is returned and -1 in case of a failure.

PGTYPESnumeric_copy

Copy over one numeric variable into another one.

```
int PGTYPESnumeric_copy(numeric *src, numeric *dst);
```

This function copies over the value of the variable that src points to into the variable that dst points to. It returns 0 on success and -1 if an error occurs.

PGTYPESnumeric_from_double

Convert a variable of type double to a numeric.

```
int  PGTYPESnumeric_from_double(double d, numeric *dst);
```

This function accepts a variable of type double and stores the result in the variable that dst points to. It returns 0 on success and -1 if an error occurs.

PGTYPESnumeric_to_double

Convert a variable of type numeric to double.

```
int PGTYPESnumeric_to_double(numeric *nv, double *dp)
```

The function converts the numeric value from the variable that nv points to into the double variable that dp points to. It returns 0 on success and -1 if an error occurs, including overflow. On overflow, the global variable errno will be set to PGTYPES_NUM_OVERFLOW additionally.

PGTYPESnumeric_to_int

Convert a variable of type numeric to int.

```
int PGTYPESnumeric_to_int(numeric *nv, int *ip);
```

The function converts the numeric value from the variable that nv points to into the integer variable that ip points to. It returns 0 on success and -1 if an error occurs, including overflow. On overflow, the global variable errno will be set to PGTYPES_NUM_OVERFLOW additionally.

`PGTYPESnumeric_to_long`

Convert a variable of type numeric to long.

```
int PGTYPESnumeric_to_long(numeric *nv, long *lp);
```

The function converts the numeric value from the variable that `nv` points to into the long integer variable that `lp` points to. It returns 0 on success and -1 if an error occurs, including overflow. On overflow, the global variable `errno` will be set to `PGTYPES_NUM_OVERFLOW` additionally.

`PGTYPESnumeric_to_decimal`

Convert a variable of type numeric to decimal.

```
int PGTYPESnumeric_to_decimal(numeric *src, decimal *dst);
```

The function converts the numeric value from the variable that `src` points to into the decimal variable that `dst` points to. It returns 0 on success and -1 if an error occurs, including overflow. On overflow, the global variable `errno` will be set to `PGTYPES_NUM_OVERFLOW` additionally.

`PGTYPESnumeric_from_decimal`

Convert a variable of type decimal to numeric.

```
int PGTYPESnumeric_from_decimal(decimal *src, numeric *dst);
```

The function converts the decimal value from the variable that `src` points to into the numeric variable that `dst` points to. It returns 0 on success and -1 if an error occurs. Since the decimal type is implemented as a limited version of the numeric type, overflow cannot occur with this conversion.

33.6.3. The date Type

The date type in C enables your programs to deal with data of the SQL type date. See Section 8.5 for the equivalent type in the PostgreSQL server.

The following functions can be used to work with the date type:

`PGTYPESdate_from_timestamp`

Extract the date part from a timestamp.

```
date PGTYPESdate_from_timestamp(timestamp dt);
```

The function receives a timestamp as its only argument and returns the extracted date part from this timestamp.

`PGTYPESdate_from_asc`

Parse a date from its textual representation.

```
date PGTYPESdate_from_asc(char *str, char **endptr);
```

The function receives a C char* string `str` and a pointer to a C char* string `endptr`. At the moment ECPG always parses the complete string and so it currently does not support to store the address of the first invalid character in `*endptr`. You can safely set `endptr` to NULL.

Note that the function always assumes MDY-formatted dates and there is currently no variable to change that within ECPG.

Table 33-2 shows the allowed input formats.

Table 33-2. Valid Input Formats for `PGTYPESdate_from_asc`

Input	Result
January 8, 1999	January 8, 1999
1999-01-08	January 8, 1999
1/8/1999	January 8, 1999
1/18/1999	January 18, 1999
01/02/03	February 1, 2003
1999-Jan-08	January 8, 1999
Jan-08-1999	January 8, 1999
08-Jan-1999	January 8, 1999
99-Jan-08	January 8, 1999
08-Jan-99	January 8, 1999
08-Jan-06	January 8, 2006
Jan-08-99	January 8, 1999
19990108	ISO 8601; January 8, 1999
990108	ISO 8601; January 8, 1999
1999.008	year and day of year
J2451187	Julian day
January 8, 99 BC	year 99 before the Common Era

`PGTYPESdate_to_asc`

Return the textual representation of a date variable.

```
char *PGTYPESdate_to_asc(date dDate);
```

The function receives the date `dDate` as its only parameter. It will output the date in the form `1999-01-18`, i.e., in the `YYYY-MM-DD` format. The result must be freed with `PGTYPESchar_free()`.

`PGTYPESdate_julmdy`

Extract the values for the day, the month and the year from a variable of type date.

```
void PGTYPESdate_julmdy(date d, int *mdy);
```

The function receives the date `d` and a pointer to an array of 3 integer values `mdy`. The variable name indicates the sequential order: `mdy[0]` will be set to contain the number of the month, `mdy[1]` will be set to the value of the day and `mdy[2]` will contain the year.

`PGTYPESdate_mdyjul`

Create a date value from an array of 3 integers that specify the day, the month and the year of the date.

```
void PGTYPESdate_mdyjul(int *mdy, date *jdate);
```

The function receives the array of the 3 integers (`mdy`) as its first argument and as its second argument a pointer to a variable of type date that should hold the result of the operation.

`PGTYPESdate_dayofweek`

Return a number representing the day of the week for a date value.

```
int PGTYPESdate_dayofweek(date d);
```

The function receives the date variable `d` as its only argument and returns an integer that indicates the day of the week for this date.

- 0 - Sunday
- 1 - Monday
- 2 - Tuesday
- 3 - Wednesday
- 4 - Thursday
- 5 - Friday
- 6 - Saturday

`PGTYPESdate_today`

Get the current date.

```
void PGTYPESdate_today(date *d);
```

The function receives a pointer to a date variable (`d`) that it sets to the current date.

`PGTYPESdate_fmt_asc`

Convert a variable of type date to its textual representation using a format mask.

```
int PGTYPESdate_fmt_asc(date dDate, char *fmtstring, char *outbuf);
```

The function receives the date to convert (`dDate`), the format mask (`fmtstring`) and the string that will hold the textual representation of the date (`outbuf`).

On success, 0 is returned and a negative value if an error occurred.

The following literals are the field specifiers you can use:

- `dd` - The number of the day of the month.
- `mm` - The number of the month of the year.
- `yy` - The number of the year as a two digit number.
- `yyyy` - The number of the year as a four digit number.
- `ddd` - The name of the day (abbreviated).
- `mmm` - The name of the month (abbreviated).

All other characters are copied 1:1 to the output string.

Table 33-3 indicates a few possible formats. This will give you an idea of how to use this function. All output lines are based on the same date: November 23, 1959.

Format	Result

Table 33-3. Valid Input Formats for `PGTYPESdate_fmt_asc`

Format	Result
mmddyy	112359
ddmmyy	231159
yymmdd	591123
yy/mm/dd	59/11/23
yy mm dd	59 11 23
yy.mm.dd	59.11.23
.mm.yyyy.dd.	.11.1959.23.
mmm. dd, yyyy	Nov. 23, 1959
mmm dd yyyy	Nov 23 1959
yyyy dd mm	1959 23 11
ddd, mmm. dd, yyyy	Mon, Nov. 23, 1959
(ddd) mmm. dd, yyyy	(Mon) Nov. 23, 1959

`PGTYPESdate_defmt_asc`

Use a format mask to convert a C `char*` string to a value of type date.

```
int PGTYPESdate_defmt_asc(date *d, char *fmt, char *str);
```

The function receives a pointer to the date value that should hold the result of the operation (`d`), the format mask to use for parsing the date (`fmt`) and the C char* string containing the textual representation of the date (`str`). The textual representation is expected to match the format mask. However you do not need to have a 1:1 mapping of the string to the format mask. The function only analyzes the sequential order and looks for the literals `yy` or `yyyy` that indicate the position of the year, `mm` to indicate the position of the month and `dd` to indicate the position of the day.

Table 33-4 indicates a few possible formats. This will give you an idea of how to use this function.

Table 33-4. Valid Input Formats for `rdefmtdate`

Format	String	Result
ddmmyy	21-2-54	1954-02-21
ddmmyy	2-12-54	1954-12-02
ddmmyy	20111954	1954-11-20
ddmmyy	130464	1964-04-13
mmm.dd.yyyy	MAR-12-1967	1967-03-12
yy/mm/dd	1954, February 3rd	1954-02-03
mmm.dd.yyyy	041269	1969-04-12
yy/mm/dd	In the year 2525, in the month of July, mankind will be alive on the 28th day	2525-07-28

Format	String	Result
dd-mm-yy	I said on the 28th of July in the year 2525	2525-07-28
mmm.dd.yyyy	9/14/58	1958-09-14
yy/mm/dd	47/03/29	1947-03-29
mmm.dd.yyyy	oct 28 1975	1975-10-28
mmddyy	Nov 14th, 1985	1985-11-14

33.6.4. The timestamp Type

The timestamp type in C enables your programs to deal with data of the SQL type timestamp. See Section 8.5 for the equivalent type in the PostgreSQL server.

The following functions can be used to work with the timestamp type:

`PGTYPEStimestamp_from_asc`

Parse a timestamp from its textual representation into a timestamp variable.

```
timestamp PGTYPEStimestamp_from_asc(char *str, char **endptr);
```

The function receives the string to parse (`str`) and a pointer to a C char* (`endptr`). At the moment ECPG always parses the complete string and so it currently does not support to store the address of the first invalid character in `*endptr`. You can safely set `endptr` to NULL.

The function returns the parsed timestamp on success. On error, `PGTYPESInvalidTimestamp` is returned and `errno` is set to `PGTYPES_TS_BAD_TIMESTAMP`. See *`PGTYPESInvalidTimestamp`* for important notes on this value.

In general, the input string can contain any combination of an allowed date specification, a whitespace character and an allowed time specification. Note that time zones are not supported by ECPG. It can parse them but does not apply any calculation as the PostgreSQL server does for example. Timezone specifiers are silently discarded.

Table 33-5 contains a few examples for input strings.

Table 33-5. Valid Input Formats for `PGTYPEStimestamp_from_asc`

Input	Result
1999-01-08 04:05:06	1999-01-08 04:05:06
January 8 04:05:06 1999 PST	1999-01-08 04:05:06
1999-Jan-08 04:05:06.789-8	1999-01-08 04:05:06.789 (time zone specifier ignored)
J2451187 04:05-08:00	1999-01-08 04:05:00 (time zone specifier ignored)

`PGTYPEStimestamp_to_asc`

Converts a date to a C char* string.

```
char *PGTYPEStimestamp_to_asc(timestamp tstamp);
```

The function receives the timestamp `tstamp` as its only argument and returns an allocated string that contains the textual representation of the timestamp. The result must be freed with `PGTYPESchar_free()`.

`PGTYPEStimestamp_current`

Retrieve the current timestamp.

```
void PGTYPEStimestamp_current(timestamp *ts);
```

The function retrieves the current timestamp and saves it into the timestamp variable that `ts` points to.

`PGTYPEStimestamp_fmt_asc`

Convert a timestamp variable to a C char* using a format mask.

```
int PGTYPEStimestamp_fmt_asc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

The function receives a pointer to the timestamp to convert as its first argument (`ts`), a pointer to the output buffer (`output`), the maximal length that has been allocated for the output buffer (`str_len`) and the format mask to use for the conversion (`fmtstr`).

Upon success, the function returns 0 and a negative value if an error occurred.

You can use the following format specifiers for the format mask. The format specifiers are the same ones that are used in the `strftime` function in libc. Any non-format specifier will be copied into the output buffer.

- `%A` - is replaced by national representation of the full weekday name.
- `%a` - is replaced by national representation of the abbreviated weekday name.
- `%B` - is replaced by national representation of the full month name.
- `%b` - is replaced by national representation of the abbreviated month name.
- `%C` - is replaced by (year / 100) as decimal number; single digits are preceded by a zero.
- `%c` - is replaced by national representation of time and date.
- `%D` - is equivalent to `%m/%d/%y`.
- `%d` - is replaced by the day of the month as a decimal number (01-31).
- `%E* %O*` - POSIX locale extensions. The sequences `%Ec` `%EC` `%Ex` `%EX` `%Ey` `%EY` `%Od` `%Oe` `%OH` `%OI` `%Om` `%OM` `%OS` `%Ou` `%OU` `%OV` `%Ow` `%OW` `%Oy` are supposed to provide alternative representations.

  Additionally `%OB` implemented to represent alternative months names (used standalone, without day mentioned).
- `%e` - is replaced by the day of month as a decimal number (1-31); single digits are preceded by a blank.
- `%F` - is equivalent to `%Y-%m-%d`.
- `%G` - is replaced by a year as a decimal number with century. This year is the one that contains the greater part of the week (Monday as the first day of the week).
- `%g` - is replaced by the same year as in `%G`, but as a decimal number without century (00-99).

- `%H` - is replaced by the hour (24-hour clock) as a decimal number (00-23).
- `%h` - the same as `%b`.
- `%I` - is replaced by the hour (12-hour clock) as a decimal number (01-12).
- `%j` - is replaced by the day of the year as a decimal number (001-366).
- `%k` - is replaced by the hour (24-hour clock) as a decimal number (0-23); single digits are preceded by a blank.
- `%l` - is replaced by the hour (12-hour clock) as a decimal number (1-12); single digits are preceded by a blank.
- `%M` - is replaced by the minute as a decimal number (00-59).
- `%m` - is replaced by the month as a decimal number (01-12).
- `%n` - is replaced by a newline.
- `%O*` - the same as `%E*`.
- `%p` - is replaced by national representation of either "ante meridiem" or "post meridiem" as appropriate.
- `%R` - is equivalent to `%H:%M`.
- `%r` - is equivalent to `%I:%M:%S %p`.
- `%S` - is replaced by the second as a decimal number (00-60).
- `%s` - is replaced by the number of seconds since the Epoch, UTC.
- `%T` - is equivalent to `%H:%M:%S`
- `%t` - is replaced by a tab.
- `%U` - is replaced by the week number of the year (Sunday as the first day of the week) as a decimal number (00-53).
- `%u` - is replaced by the weekday (Monday as the first day of the week) as a decimal number (1-7).
- `%V` - is replaced by the week number of the year (Monday as the first day of the week) as a decimal number (01-53). If the week containing January 1 has four or more days in the new year, then it is week 1; otherwise it is the last week of the previous year, and the next week is week 1.
- `%v` - is equivalent to `%e-%b-%Y`.
- `%W` - is replaced by the week number of the year (Monday as the first day of the week) as a decimal number (00-53).
- `%w` - is replaced by the weekday (Sunday as the first day of the week) as a decimal number (0-6).
- `%X` - is replaced by national representation of the time.
- `%x` - is replaced by national representation of the date.
- `%Y` - is replaced by the year with century as a decimal number.
- `%y` - is replaced by the year without century as a decimal number (00-99).
- `%Z` - is replaced by the time zone name.

- `%z` - is replaced by the time zone offset from UTC; a leading plus sign stands for east of UTC, a minus sign for west of UTC, hours and minutes follow with two digits each and no delimiter between them (common form for RFC 822 date headers).
- `%+` - is replaced by national representation of the date and time.
- `%-*` - GNU libc extension. Do not do any padding when performing numerical outputs.
- $_* - GNU libc extension. Explicitly specify space for padding.
- `%0*` - GNU libc extension. Explicitly specify zero for padding.
- `%%` - is replaced by `%`.

`PGTYPEStimestamp_sub`

Subtract one timestamp from another one and save the result in a variable of type interval.

```
int PGTYPEStimestamp_sub(timestamp *ts1, timestamp *ts2, interval *iv);
```

The function will subtract the timestamp variable that `ts2` points to from the timestamp variable that `ts1` points to and will store the result in the interval variable that `iv` points to.

Upon success, the function returns 0 and a negative value if an error occurred.

`PGTYPEStimestamp_defmt_asc`

Parse a timestamp value from its textual representation using a formatting mask.

```
int PGTYPEStimestamp_defmt_asc(char *str, char *fmt, timestamp *d);
```

The function receives the textual representation of a timestamp in the variable `str` as well as the formatting mask to use in the variable `fmt`. The result will be stored in the variable that `d` points to.

If the formatting mask `fmt` is NULL, the function will fall back to the default formatting mask which is `%Y-%m-%d %H:%M:%S`.

This is the reverse function to *`PGTYPEStimestamp_fmt_asc`*. See the documentation there in order to find out about the possible formatting mask entries.

`PGTYPEStimestamp_add_interval`

Add an interval variable to a timestamp variable.

```
int PGTYPEStimestamp_add_interval(timestamp *tin, interval *span, timestamp *tout);
```

The function receives a pointer to a timestamp variable `tin` and a pointer to an interval variable `span`. It adds the interval to the timestamp and saves the resulting timestamp in the variable that `tout` points to.

Upon success, the function returns 0 and a negative value if an error occurred.

`PGTYPEStimestamp_sub_interval`

Subtract an interval variable from a timestamp variable.

```
int PGTYPEStimestamp_sub_interval(timestamp *tin, interval *span, timestamp *tout);
```

The function subtracts the interval variable that `span` points to from the timestamp variable that `tin` points to and saves the result into the variable that `tout` points to.

Upon success, the function returns 0 and a negative value if an error occurred.

33.6.5. The interval Type

The interval type in C enables your programs to deal with data of the SQL type interval. See Section 8.5 for the equivalent type in the PostgreSQL server.

The following functions can be used to work with the interval type:

`PGTYPESinterval_new`

Return a pointer to a newly allocated interval variable.

```
interval *PGTYPESinterval_new(void);
```

`PGTYPESinterval_free`

Release the memory of a previously allocated interval variable.

```
void PGTYPESinterval_free(interval *intvl);
```

`PGTYPESinterval_from_asc`

Parse an interval from its textual representation.

```
interval *PGTYPESinterval_from_asc(char *str, char **endptr);
```

The function parses the input string `str` and returns a pointer to an allocated interval variable. At the moment ECPG always parses the complete string and so it currently does not support to store the address of the first invalid character in `*endptr`. You can safely set `endptr` to NULL.

`PGTYPESinterval_to_asc`

Convert a variable of type interval to its textual representation.

```
char *PGTYPESinterval_to_asc(interval *span);
```

The function converts the interval variable that `span` points to into a C char*. The output looks like this example: `@ 1 day 12 hours 59 mins 10 secs`. The result must be freed with `PGTYPESchar_free()`.

`PGTYPESinterval_copy`

Copy a variable of type interval.

```
int PGTYPESinterval_copy(interval *intvlsrc, interval *intvldest);
```

The function copies the interval variable that `intvlsrc` points to into the variable that `intvldest` points to. Note that you need to allocate the memory for the destination variable before.

33.6.6. The decimal Type

The decimal type is similar to the numeric type. However it is limited to a maximum precision of 30 significant digits. In contrast to the numeric type which can be created on the heap only, the decimal type can be created either on the stack or on the heap (by means of the functions `PGTYPESdecimal_new` and `PGTYPESdecimal_free`). There are a lot of other functions that deal with the decimal type in the Informix compatibility mode described in Section 33.15.

The following functions can be used to work with the decimal type and are not only contained in the `libcompat` library.

`PGTYPESdecimal_new`

Request a pointer to a newly allocated decimal variable.

```
decimal *PGTYPESdecimal_new(void);
```

`PGTYPESdecimal_free`

Free a decimal type, release all of its memory.

```
void PGTYPESdecimal_free(decimal *var);
```

33.6.7. errno Values of pgtypeslib

`PGTYPES_NUM_BAD_NUMERIC`

An argument should contain a numeric variable (or point to a numeric variable) but in fact its in-memory representation was invalid.

`PGTYPES_NUM_OVERFLOW`

An overflow occurred. Since the numeric type can deal with almost arbitrary precision, converting a numeric variable into other types might cause overflow.

`PGTYPES_NUM_UNDERFLOW`

An underflow occurred. Since the numeric type can deal with almost arbitrary precision, converting a numeric variable into other types might cause underflow.

`PGTYPES_NUM_DIVIDE_ZERO`

A division by zero has been attempted.

`PGTYPES_DATE_BAD_DATE`

An invalid date string was passed to the `PGTYPESdate_from_asc` function.

`PGTYPES_DATE_ERR_EARGS`

Invalid arguments were passed to the `PGTYPESdate_defmt_asc` function.

`PGTYPES_DATE_ERR_ENOSHORTDATE`

An invalid token in the input string was found by the `PGTYPESdate_defmt_asc` function.

`PGTYPES_INTVL_BAD_INTERVAL`

An invalid interval string was passed to the `PGTYPESinterval_from_asc` function, or an invalid interval value was passed to the `PGTYPESinterval_to_asc` function.

`PGTYPES_DATE_ERR_ENOTDMY`

There was a mismatch in the day/month/year assignment in the `PGTYPESdate_defmt_asc` function.

`PGTYPES_DATE_BAD_DAY`

An invalid day of the month value was found by the `PGTYPESdate_defmt_asc` function.

PGTYPES_DATE_BAD_MONTH

An invalid month value was found by the PGTYPESdate_defmt_asc function.

PGTYPES_TS_BAD_TIMESTAMP

An invalid timestamp string pass passed to the PGTYPEStimestamp_from_asc function, or an invalid timestamp value was passed to the PGTYPEStimestamp_to_asc function.

PGTYPES_TS_ERR_EINFTIME

An infinite timestamp value was encountered in a context that cannot handle it.

33.6.8. Special Constants of pgtypeslib

PGTYPESInvalidTimestamp

A value of type timestamp representing an invalid time stamp. This is returned by the function PGTYPEStimestamp_from_asc on parse error. Note that due to the internal representation of the timestamp data type, PGTYPESInvalidTimestamp is also a valid timestamp at the same time. It is set to 1899-12-31 23:59:59. In order to detect errors, make sure that your application does not only test for PGTYPESInvalidTimestamp but also for errno != 0 after each call to PGTYPEStimestamp_from_asc.

33.7. Using Descriptor Areas

An SQL descriptor area is a more sophisticated method for processing the result of a SELECT, FETCH or a DESCRIBE statement. An SQL descriptor area groups the data of one row of data together with metadata items into one data structure. The metadata is particularly useful when executing dynamic SQL statements, where the nature of the result columns might not be known ahead of time. PostgreSQL provides two ways to use Descriptor Areas: the named SQL Descriptor Areas and the C-structure SQLDAs.

33.7.1. Named SQL Descriptor Areas

A named SQL descriptor area consists of a header, which contains information concerning the entire descriptor, and one or more item descriptor areas, which basically each describe one column in the result row.

Before you can use an SQL descriptor area, you need to allocate one:

```
EXEC SQL ALLOCATE DESCRIPTOR identifier;
```

The identifier serves as the "variable name" of the descriptor area. When you don't need the descriptor anymore, you should deallocate it:

```
EXEC SQL DEALLOCATE DESCRIPTOR identifier;
```

To use a descriptor area, specify it as the storage target in an `INTO` clause, instead of listing host variables:

```
EXEC SQL FETCH NEXT FROM mycursor INTO SQL DESCRIPTOR mydesc;
```

If the result set is empty, the Descriptor Area will still contain the metadata from the query, i.e., the field names.

For not yet executed prepared queries, the `DESCRIBE` statement can be used to get the metadata of the result set:

```
EXEC SQL BEGIN DECLARE SECTION;
char *sql_stmt = "SELECT * FROM table1";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
```

Before PostgreSQL 9.0, the `SQL` keyword was optional, so using `DESCRIPTOR` and `SQL DESCRIPTOR` produced named SQL Descriptor Areas. Now it is mandatory, omitting the `SQL` keyword produces SQLDA Descriptor Areas, see Section 33.7.2.

In `DESCRIBE` and `FETCH` statements, the `INTO` and `USING` keywords can be used to similarly: they produce the result set and the metadata in a Descriptor Area.

Now how do you get the data out of the descriptor area? You can think of the descriptor area as a structure with named fields. To retrieve the value of a field from the header and store it into a host variable, use the following command:

```
EXEC SQL GET DESCRIPTOR name :hostvar = field;
```

Currently, there is only one header field defined: *`COUNT`*, which tells how many item descriptor areas exist (that is, how many columns are contained in the result). The host variable needs to be of an integer type. To get a field from the item descriptor area, use the following command:

```
EXEC SQL GET DESCRIPTOR name VALUE num :hostvar = field;
```

`num` can be a literal integer or a host variable containing an integer. Possible fields are:

`CARDINALITY` (integer)

number of rows in the result set

`DATA`

actual data item (therefore, the data type of this field depends on the query)

`DATETIME_INTERVAL_CODE` (integer)

When `TYPE` is `9`, `DATETIME_INTERVAL_CODE` will have a value of `1` for `DATE`, `2` for `TIME`, `3` for `TIMESTAMP`, `4` for `TIME WITH TIME ZONE`, or `5` for `TIMESTAMP WITH TIME ZONE`.

`DATETIME_INTERVAL_PRECISION` (integer)

not implemented

`INDICATOR` (integer)

the indicator (indicating a null value or a value truncation)

`KEY_MEMBER` (integer)

not implemented

`LENGTH` (integer)

length of the datum in characters

`NAME` (string)

name of the column

`NULLABLE` (integer)

not implemented

`OCTET_LENGTH` (integer)

length of the character representation of the datum in bytes

`PRECISION` (integer)

precision (for type `numeric`)

`RETURNED_LENGTH` (integer)

length of the datum in characters

`RETURNED_OCTET_LENGTH` (integer)

length of the character representation of the datum in bytes

`SCALE` (integer)

scale (for type `numeric`)

`TYPE` (integer)

numeric code of the data type of the column

In `EXECUTE`, `DECLARE` and `OPEN` statements, the effect of the `INTO` and `USING` keywords are different. A Descriptor Area can also be manually built to provide the input parameters for a query or a cursor and `USING SQL DESCRIPTOR` *`name`* is the way to pass the input parameters into a parametrized query. The statement to build a named SQL Descriptor Area is below:

```
EXEC SQL SET DESCRIPTOR name VALUE num field = :hostvar;
```

PostgreSQL supports retrieving more that one record in one `FETCH` statement and storing the data in host variables in this case assumes that the variable is an array. E.g.:

```
EXEC SQL BEGIN DECLARE SECTION;
int id[5];
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL FETCH 5 FROM mycursor INTO SQL DESCRIPTOR mydesc;

EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :id = DATA;
```

33.7.2. SQLDA Descriptor Areas

An SQLDA Descriptor Area is a C language structure which can be also used to get the result set and the metadata of a query. One structure stores one record from the result set.

```
EXEC SQL include sqlda.h;
sqlda_t         *mysqlda;

EXEC SQL FETCH 3 FROM mycursor INTO DESCRIPTOR mysqlda;
```

Note that the `SQL` keyword is omitted. The paragraphs about the use cases of the `INTO` and `USING` keywords in Section 33.7.1 also apply here with an addition. In a `DESCRIBE` statement the `DESCRIPTOR` keyword can be completely omitted if the `INTO` keyword is used:

```
EXEC SQL DESCRIBE prepared_statement INTO mysqlda;
```

The general flow of a program that uses SQLDA is:

1. Prepare a query, and declare a cursor for it.
2. Declare an SQLDA for the result rows.
3. Declare an SQLDA for the input parameters, and initialize them (memory allocation, parameter settings).
4. Open a cursor with the input SQLDA.
5. Fetch rows from the cursor, and store them into an output SQLDA.
6. Read values from the output SQLDA into the host variables (with conversion if necessary).
7. Close the cursor.
8. Free the memory area allocated for the input SQLDA.

33.7.2.1. SQLDA Data Structure

SQLDA uses three data structure types: `sqlda_t`, `sqlvar_t`, and `struct sqlname`.

> **Tip:** PostgreSQL's SQLDA has a similar data structure to the one in IBM DB2 Universal Database, so some technical information on DB2's SQLDA could help understanding PostgreSQL's one better.

33.7.2.1.1. sqlda_t Structure

The structure type `sqlda_t` is the type of the actual SQLDA. It holds one record. And two or more `sqlda_t` structures can be connected in a linked list with the pointer in the `desc_next` field, thus representing an ordered collection of rows. So, when two or more rows are fetched, the application can read them by following the `desc_next` pointer in each `sqlda_t` node.

The definition of `sqlda_t` is:

```
struct sqlda_struct
{
    char            sqldaid[8];
    long            sqldabc;
    short           sqln;
    short           sqld;
    struct sqlda_struct *desc_next;
    struct sqlvar_struct sqlvar[1];
};

typedef struct sqlda_struct sqlda_t;
```

The meaning of the fields is:

`sqldaid`

It contains the literal string `"SQLDA "`.

`sqldabc`

It contains the size of the allocated space in bytes.

`sqln`

It contains the number of input parameters for a parametrized query in case it's passed into `OPEN`, `DECLARE` or `EXECUTE` statements using the `USING` keyword. In case it's used as output of `SELECT`, `EXECUTE` or `FETCH` statements, its value is the same as `sqld` statement

`sqld`

It contains the number of fields in a result set.

`desc_next`

If the query returns more than one record, multiple linked SQLDA structures are returned, and `desc_next` holds a pointer to the next entry in the list.

`sqlvar`

This is the array of the columns in the result set.

33.7.2.1.2. sqlvar_t Structure

The structure type `sqlvar_t` holds a column value and metadata such as type and length. The definition of the type is:

```
struct sqlvar_struct
{
    short          sqltype;
    short          sqllen;
    char          *sqldata;
    short         *sqlind;
    struct sqlname sqlname;
};

typedef struct sqlvar_struct sqlvar_t;
```

The meaning of the fields is:

`sqltype`

Contains the type identifier of the field. For values, see `enum ECPGttype` in `ecpgtype.h`.

`sqllen`

Contains the binary length of the field. e.g., 4 bytes for `ECPGt_int`.

`sqldata`

Points to the data. The format of the data is described in Section 33.4.4.

`sqlind`

Points to the null indicator. 0 means not null, -1 means null.

`sqlname`

The name of the field.

33.7.2.1.3. struct sqlname Structure

A `struct sqlname` structure holds a column name. It is used as a member of the `sqlvar_t` structure. The definition of the structure is:

```
#define NAMEDATALEN 64

struct sqlname
{
        short           length;
        char            data[NAMEDATALEN];
};
```

The meaning of the fields is:

`length`

Contains the length of the field name.

`data`

Contains the actual field name.

33.7.2.2. Retrieving a Result Set Using an SQLDA

The general steps to retrieve a query result set through an SQLDA are:

1. Declare an `sqlda_t` structure to receive the result set.
2. Execute `FETCH`/`EXECUTE`/`DESCRIBE` commands to process a query specifying the declared SQLDA.
3. Check the number of records in the result set by looking at `sqln`, a member of the `sqlda_t` structure.
4. Get the values of each column from `sqlvar[0]`, `sqlvar[1]`, etc., members of the `sqlda_t` structure.
5. Go to next row (`sqlda_t` structure) by following the `desc_next` pointer, a member of the `sqlda_t` structure.
6. Repeat above as you need.

Here is an example retrieving a result set through an SQLDA.

First, declare a `sqlda_t` structure to receive the result set.

```
sqlda_t *sqlda1;
```

Next, specify the SQLDA in a command. This is a `FETCH` command example.

```
EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

Run a loop following the linked list to retrieve the rows.

```
sqlda_t *cur_sqlda;

for (cur_sqlda = sqlda1;
     cur_sqlda != NULL;
     cur_sqlda = cur_sqlda->desc_next)
{
    ...
}
```

Inside the loop, run another loop to retrieve each column data (`sqlvar_t` structure) of the row.

```
for (i = 0; i < cur_sqlda->sqld; i++)
{
    sqlvar_t v = cur_sqlda->sqlvar[i];
    char *sqldata = v.sqldata;
    short sqllen  = v.sqllen;
    ...
}
```

To get a column value, check the `sqltype` value, a member of the `sqlvar_t` structure. Then, switch to an appropriate way, depending on the column type, to copy data from the `sqlvar` field to a host variable.

```
char var_buf[1024];

switch (v.sqltype)
{
    case ECPGt_char:
        memset(&var_buf, 0, sizeof(var_buf));
        memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ? sizeof(var_buf) - 1 : sqllen
        break;

    case ECPGt_int: /* integer */
        memcpy(&intval, sqldata, sqllen);
        snprintf(var_buf, sizeof(var_buf), "%d", intval);
        break;

    ...
}
```

33.7.2.3. Passing Query Parameters Using an SQLDA

The general steps to use an SQLDA to pass input parameters to a prepared query are:

1. Create a prepared query (prepared statement)
2. Declare a sqlda_t structure as an input SQLDA.
3. Allocate memory area (as sqlda_t structure) for the input SQLDA.
4. Set (copy) input values in the allocated memory.
5. Open a cursor with specifying the input SQLDA.

Here is an example.

First, create a prepared statement.

```
EXEC SQL BEGIN DECLARE SECTION;
char query[1024] = "SELECT d.oid, * FROM pg_database d, pg_stat_database s WHERE d.oid = s.
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE stmt1 FROM :query;
```

Next, allocate memory for an SQLDA, and set the number of input parameters in `sqln`, a member variable of the `sqlda_t` structure. When two or more input parameters are required for the prepared query, the application has to allocate additional memory space which is calculated by (nr. of params - 1) * sizeof(sqlvar_t). The example shown here allocates memory space for two input parameters.

```
sqlda_t *sqlda2;

sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));

sqlda2->sqln = 2; /* number of input variables */
```

After memory allocation, store the parameter values into the `sqlvar[]` array. (This is same array used for retrieving column values when the SQLDA is receiving a result set.) In this example, the input parameters are `"postgres"`, having a string type, and `1`, having an integer type.

```
sqlda2->sqlvar[0].sqltype = ECPGt_char;
sqlda2->sqlvar[0].sqldata = "postgres";
sqlda2->sqlvar[0].sqllen  = 8;

int intval = 1;
sqlda2->sqlvar[1].sqltype = ECPGt_int;
sqlda2->sqlvar[1].sqldata = (char *) &intval;
sqlda2->sqlvar[1].sqllen  = sizeof(intval);
```

By opening a cursor and specifying the SQLDA that was set up beforehand, the input parameters are passed to the prepared statement.

```
EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

Finally, after using input SQLDAs, the allocated memory space must be freed explicitly, unlike SQLDAs used for receiving query results.

```
free(sqlda2);
```

33.7.2.4. A Sample Application Using SQLDA

Here is an example program, which describes how to fetch access statistics of the databases, specified by the input parameters, from the system catalogs.

This application joins two system tables, pg_database and pg_stat_database on the database OID, and also fetches and shows the database statistics which are retrieved by two input parameters (a database `postgres`, and OID `1`).

First, declare an SQLDA for input and an SQLDA for output.

```
EXEC SQL include sqlda.h;

sqlda_t *sqlda1; /* an output descriptor */
sqlda_t *sqlda2; /* an input descriptor  */
```

Next, connect to the database, prepare a statement, and declare a cursor for the prepared statement.

```
int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s WHERE d.oid=s
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

Next, put some values in the input SQLDA for the input parameters. Allocate memory for the input SQLDA, and set the number of input parameters to `sqln`. Store type, value, and value length into `sqltype`, `sqldata`, and `sqllen` in the `sqlvar` structure.

```
    /* Create SQLDA structure for input parameters. */
    sqlda2 = (sqlda_t *) malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));
    sqlda2->sqln = 2; /* number of input variables */

    sqlda2->sqlvar[0].sqltype = ECPGt_char;
    sqlda2->sqlvar[0].sqldata = "postgres";
    sqlda2->sqlvar[0].sqllen  = 8;

    intval = 1;
    sqlda2->sqlvar[1].sqltype = ECPGt_int;
    sqlda2->sqlvar[1].sqldata = (char *)&intval;
    sqlda2->sqlvar[1].sqllen  = sizeof(intval);
```

After setting up the input SQLDA, open a cursor with the input SQLDA.

```
    /* Open a cursor with input parameters. */
    EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;
```

Fetch rows into the output SQLDA from the opened cursor. (Generally, you have to call `FETCH` repeatedly in the loop, to fetch all rows in the result set.)

```
while (1)
{
    sqlda_t *cur_sqlda;

    /* Assign descriptor to the cursor  */
    EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;
```

Next, retrieve the fetched records from the SQLDA, by following the linked list of the `sqlda_t` structure.

```
for (cur_sqlda = sqlda1 ;
     cur_sqlda != NULL ;
     cur_sqlda = cur_sqlda->desc_next)
{
    ...
```

Read each columns in the first record. The number of columns is stored in `sqld`, the actual data of the first column is stored in `sqlvar[0]`, both members of the `sqlda_t` structure.

```
    /* Print every column in a row. */
    for (i = 0; i < sqlda1->sqld; i++)
    {
        sqlvar_t v = sqlda1->sqlvar[i];
        char *sqldata = v.sqldata;
        short sqllen  = v.sqllen;

        strncpy(name_buf, v.sqlname.data, v.sqlname.length);
        name_buf[v.sqlname.length] = '\0';
```

Now, the column data is stored in the variable `v`. Copy every datum into host variables, looking at `v.sqltype` for the type of the column.

```
        switch (v.sqltype) {
            int intval;
            double doubleval;
            unsigned long long int longlongval;

            case ECPGt_char:
                memset(&var_buf, 0, sizeof(var_buf));
                memcpy(&var_buf, sqldata, (sizeof(var_buf) <= sqllen ? sizeof(var_buf)-
                break;

            case ECPGt_int: /* integer */
                memcpy(&intval, sqldata, sqllen);
                snprintf(var_buf, sizeof(var_buf), "%d", intval);
                break;

            ...

            default:
```

```
                ...
            }

            printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
        }
```

Close the cursor after processing all of records, and disconnect from the database.

```
    EXEC SQL CLOSE cur1;
    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT ALL;
```

The whole program is shown in Example 33-1.

Example 33-1. Example SQLDA Program

```
#include <stdlib.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

EXEC SQL include sqlda.h;

sqlda_t *sqlda1; /* descriptor for output */
sqlda_t *sqlda2; /* descriptor for input */

EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    EXEC SQL BEGIN DECLARE SECTION;
    char query[1024] = "SELECT d.oid,* FROM pg_database d, pg_stat_database s WHERE d.oid=s

    int intval;
    unsigned long long int longlongval;
    EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO uptimedb AS con1 USER uptime;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    EXEC SQL PREPARE stmt1 FROM :query;
    EXEC SQL DECLARE cur1 CURSOR FOR stmt1;

    /* Create a SQLDA structure for an input parameter */
    sqlda2 = (sqlda_t *)malloc(sizeof(sqlda_t) + sizeof(sqlvar_t));
    memset(sqlda2, 0, sizeof(sqlda_t) + sizeof(sqlvar_t));
```

```
sqlda2->sqln = 2; /* a number of input variables */

sqlda2->sqlvar[0].sqltype = ECPGt_char;
sqlda2->sqlvar[0].sqldata = "postgres";
sqlda2->sqlvar[0].sqllen  = 8;

intval = 1;
sqlda2->sqlvar[1].sqltype = ECPGt_int;
sqlda2->sqlvar[1].sqldata = (char *) &intval;
sqlda2->sqlvar[1].sqllen  = sizeof(intval);

/* Open a cursor with input parameters. */
EXEC SQL OPEN cur1 USING DESCRIPTOR sqlda2;

while (1)
{
    sqlda_t *cur_sqlda;

    /* Assign descriptor to the cursor  */
    EXEC SQL FETCH NEXT FROM cur1 INTO DESCRIPTOR sqlda1;

    for (cur_sqlda = sqlda1 ;
         cur_sqlda != NULL ;
         cur_sqlda = cur_sqlda->desc_next)
    {
        int i;
        char name_buf[1024];
        char var_buf[1024];

        /* Print every column in a row. */
        for (i=0 ; i<cur_sqlda->sqld ; i++)
        {
            sqlvar_t v = cur_sqlda->sqlvar[i];
            char *sqldata = v.sqldata;
            short sqllen  = v.sqllen;

            strncpy(name_buf, v.sqlname.data, v.sqlname.length);
            name_buf[v.sqlname.length] = '\0';

            switch (v.sqltype)
            {
                case ECPGt_char:
                    memset(&var_buf, 0, sizeof(var_buf));
                    memcpy(&var_buf, sqldata, (sizeof(var_buf)<=sqllen ? sizeof(var_buf
                    break;

                case ECPGt_int: /* integer */
                    memcpy(&intval, sqldata, sqllen);
                    snprintf(var_buf, sizeof(var_buf), "%d", intval);
                    break;

                case ECPGt_long_long: /* bigint */
                    memcpy(&longlongval, sqldata, sqllen);
```

```
                        snprintf(var_buf, sizeof(var_buf), "%lld", longlongval);
                        break;

                    default:
                    {
                        int i;
                        memset(var_buf, 0, sizeof(var_buf));
                        for (i = 0; i < sqllen; i++)
                        {
                            char tmpbuf[16];
                            snprintf(tmpbuf, sizeof(tmpbuf), "%02x ", (unsigned char) sqlda
                            strncat(var_buf, tmpbuf, sizeof(var_buf));
                        }
                    }
                        break;
                }

                printf("%s = %s (type: %d)\n", name_buf, var_buf, v.sqltype);
            }

            printf("\n");
        }
    }

    EXEC SQL CLOSE cur1;
    EXEC SQL COMMIT;

    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

The output of this example should look something like the following (some numbers will vary).

```
oid = 1 (type: 1)
datname = template1 (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = t (type: 1)
datallowconn = t (type: 1)
datconnlimit = -1 (type: 5)
datlastsysoid = 11510 (type: 1)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig =  (type: 1)
datacl = {=c/uptime,uptime=CTc/uptime} (type: 1)
datid = 1 (type: 1)
datname = template1 (type: 1)
numbackends = 0 (type: 5)
xact_commit = 113606 (type: 9)
xact_rollback = 0 (type: 9)
blks_read = 130 (type: 9)
blks_hit = 7341714 (type: 9)
tup_returned = 38262679 (type: 9)
```

```
tup_fetched = 1836281 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)

oid = 11511 (type: 1)
datname = postgres (type: 1)
datdba = 10 (type: 1)
encoding = 0 (type: 5)
datistemplate = f (type: 1)
datallowconn = t (type: 1)
datconnlimit = -1 (type: 5)
datlastsysoid = 11510 (type: 1)
datfrozenxid = 379 (type: 1)
dattablespace = 1663 (type: 1)
datconfig =  (type: 1)
datacl =  (type: 1)
datid = 11511 (type: 1)
datname = postgres (type: 1)
numbackends = 0 (type: 5)
xact_commit = 221069 (type: 9)
xact_rollback = 18 (type: 9)
blks_read = 1176 (type: 9)
blks_hit = 13943750 (type: 9)
tup_returned = 77410091 (type: 9)
tup_fetched = 3253694 (type: 9)
tup_inserted = 0 (type: 9)
tup_updated = 0 (type: 9)
tup_deleted = 0 (type: 9)
```

33.8. Error Handling

This section describes how you can handle exceptional conditions and warnings in an embedded SQL program. There are two nonexclusive facilities for this.

- Callbacks can be configured to handle warning and error conditions using the `WHENEVER` command.
- Detailed information about the error or warning can be obtained from the `sqlca` variable.

33.8.1. Setting Callbacks

One simple method to catch errors and warnings is to set a specific action to be executed whenever a particular condition occurs. In general:

```
EXEC SQL WHENEVER condition action;
```

condition can be one of the following:

`SQLERROR`

The specified action is called whenever an error occurs during the execution of an SQL statement.

`SQLWARNING`

The specified action is called whenever a warning occurs during the execution of an SQL statement.

`NOT FOUND`

The specified action is called whenever an SQL statement retrieves or affects zero rows. (This condition is not an error, but you might be interested in handling it specially.)

action can be one of the following:

`CONTINUE`

This effectively means that the condition is ignored. This is the default.

`GOTO` *label*
`GO TO` *label*

Jump to the specified label (using a C `goto` statement).

`SQLPRINT`

Print a message to standard error. This is useful for simple programs or during prototyping. The details of the message cannot be configured.

`STOP`

Call `exit(1)`, which will terminate the program.

`DO BREAK`

Execute the C statement `break`. This should only be used in loops or `switch` statements.

`CALL` *name* `(`*args*`)`
`DO` *name* `(`*args*`)`

Call the specified C functions with the specified arguments.

The SQL standard only provides for the actions `CONTINUE` and `GOTO` (and `GO TO`).

Here is an example that you might want to use in a simple program. It prints a simple message when a warning occurs and aborts the program when an error happens:

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLERROR STOP;
```

The statement `EXEC SQL WHENEVER` is a directive of the SQL preprocessor, not a C statement. The error or warning actions that it sets apply to all embedded SQL statements that appear below the point where the handler is set, unless a different action was set for the same condition between the first `EXEC`

`SQL WHENEVER` and the SQL statement causing the condition, regardless of the flow of control in the C program. So neither of the two following C program excerpts will have the desired effect:

```
/*
 * WRONG
 */
int main(int argc, char *argv[])
{
    ...
    if (verbose) {
        EXEC SQL WHENEVER SQLWARNING SQLPRINT;
    }
    ...
    EXEC SQL SELECT ...;
    ...
}

/*
 * WRONG
 */
int main(int argc, char *argv[])
{
    ...
    set_error_handler();
    ...
    EXEC SQL SELECT ...;
    ...
}

static void set_error_handler(void)
{
    EXEC SQL WHENEVER SQLERROR STOP;
}
```

33.8.2. sqlca

For more powerful error handling, the embedded SQL interface provides a global variable with the name `sqlca` (SQL communication area) that has the following structure:

```
struct
{
    char sqlcaid[8];
    long sqlabc;
    long sqlcode;
    struct
    {
        int sqlerrml;
        char sqlerrmc[SQLERRMC_LEN];
    } sqlerrm;
    char sqlerrp[8];
```

```
    long sqlerrd[6];
    char sqlwarn[8];
    char sqlstate[5];
} sqlca;
```

(In a multithreaded program, every thread automatically gets its own copy of `sqlca`. This works similarly to the handling of the standard C global variable `errno`.)

`sqlca` covers both warnings and errors. If multiple warnings or errors occur during the execution of a statement, then `sqlca` will only contain information about the last one.

If no error occurred in the last SQL statement, `sqlca.sqlcode` will be 0 and `sqlca.sqlstate` will be `"00000"`. If a warning or error occurred, then `sqlca.sqlcode` will be negative and `sqlca.sqlstate` will be different from `"00000"`. A positive `sqlca.sqlcode` indicates a harmless condition, such as that the last query returned zero rows. `sqlcode` and `sqlstate` are two different error code schemes; details appear below.

If the last SQL statement was successful, then `sqlca.sqlerrd[1]` contains the OID of the processed row, if applicable, and `sqlca.sqlerrd[2]` contains the number of processed or returned rows, if applicable to the command.

In case of an error or warning, `sqlca.sqlerrm.sqlerrmc` will contain a string that describes the error. The field `sqlca.sqlerrm.sqlerrml` contains the length of the error message that is stored in `sqlca.sqlerrm.sqlerrmc` (the result of `strlen()`, not really interesting for a C programmer). Note that some messages are too long to fit in the fixed-size `sqlerrmc` array; they will be truncated.

In case of a warning, `sqlca.sqlwarn[2]` is set to `W`. (In all other cases, it is set to something different from `W`.) If `sqlca.sqlwarn[1]` is set to `W`, then a value was truncated when it was stored in a host variable. `sqlca.sqlwarn[0]` is set to `W` if any of the other elements are set to indicate a warning.

The fields `sqlcaid`, `sqlabc`, `sqlerrp`, and the remaining elements of `sqlerrd` and `sqlwarn` currently contain no useful information.

The structure `sqlca` is not defined in the SQL standard, but is implemented in several other SQL database systems. The definitions are similar at the core, but if you want to write portable applications, then you should investigate the different implementations carefully.

Here is one example that combines the use of `WHENEVER` and `sqlca`, printing out the contents of `sqlca` when an error occurs. This is perhaps useful for debugging or prototyping applications, before installing a more “user-friendly” error handler.

```
EXEC SQL WHENEVER SQLERROR CALL print_sqlca();

void
print_sqlca()
{
    fprintf(stderr, "==== sqlca ====\n");
    fprintf(stderr, "sqlcode: %ld\n", sqlca.sqlcode);
    fprintf(stderr, "sqlerrm.sqlerrml: %d\n", sqlca.sqlerrm.sqlerrml);
    fprintf(stderr, "sqlerrm.sqlerrmc: %s\n", sqlca.sqlerrm.sqlerrmc);
    fprintf(stderr, "sqlerrd: %ld %ld %ld %ld %ld %ld\n", sqlca.sqlerrd[0],sqlca.sqlerrd[1]
                                                          sqlca.sqlerrd[3],sqlca.sqlerrd[4]
    fprintf(stderr, "sqlwarn: %d %d %d %d %d %d %d %d\n", sqlca.sqlwarn[0], sqlca.sqlwarn[1
                                                          sqlca.sqlwarn[3], sqlca.sqlwarn[4
                                                          sqlca.sqlwarn[6], sqlca.sqlwarn[7
```

```
    fprintf(stderr, "sqlstate: %5s\n", sqlca.sqlstate);
    fprintf(stderr, "===============\n");
}
```

The result could look as follows (here an error due to a misspelled table name):

```
==== sqlca ====
sqlcode: -400
sqlerrm.sqlerrml: 49
sqlerrm.sqlerrmc: relation "pg_databasep" does not exist on line 38
sqlerrd: 0 0 0 0 0 0
sqlwarn: 0 0 0 0 0 0 0 0
sqlstate: 42P01
===============
```

33.8.3. SQLSTATE vs. SQLCODE

The fields `sqlca.sqlstate` and `sqlca.sqlcode` are two different schemes that provide error codes. Both are derived from the SQL standard, but `SQLCODE` has been marked deprecated in the SQL-92 edition of the standard and has been dropped in later editions. Therefore, new applications are strongly encouraged to use `SQLSTATE`.

`SQLSTATE` is a five-character array. The five characters contain digits or upper-case letters that represent codes of various error and warning conditions. `SQLSTATE` has a hierarchical scheme: the first two characters indicate the general class of the condition, the last three characters indicate a subclass of the general condition. A successful state is indicated by the code `00000`. The `SQLSTATE` codes are for the most part defined in the SQL standard. The PostgreSQL server natively supports `SQLSTATE` error codes; therefore a high degree of consistency can be achieved by using this error code scheme throughout all applications. For further information see Appendix A.

`SQLCODE`, the deprecated error code scheme, is a simple integer. A value of 0 indicates success, a positive value indicates success with additional information, a negative value indicates an error. The SQL standard only defines the positive value +100, which indicates that the last command returned or affected zero rows, and no specific negative values. Therefore, this scheme can only achieve poor portability and does not have a hierarchical code assignment. Historically, the embedded SQL processor for PostgreSQL has assigned some specific `SQLCODE` values for its use, which are listed below with their numeric value and their symbolic name. Remember that these are not portable to other SQL implementations. To simplify the porting of applications to the `SQLSTATE` scheme, the corresponding `SQLSTATE` is also listed. There is, however, no one-to-one or one-to-many mapping between the two schemes (indeed it is many-to-many), so you should consult the global `SQLSTATE` listing in Appendix A in each case.

These are the assigned `SQLCODE` values:

0 (`ECPG_NO_ERROR`)

Indicates no error. (SQLSTATE 00000)

100 (ECPG_NOT_FOUND)

This is a harmless condition indicating that the last command retrieved or processed zero rows, or that you are at the end of the cursor. (SQLSTATE 02000)

When processing a cursor in a loop, you could use this code as a way to detect when to abort the loop, like this:

```
while (1)
{
    EXEC SQL FETCH ... ;
    if (sqlca.sqlcode == ECPG_NOT_FOUND)
        break;
}
```

But WHENEVER NOT FOUND DO BREAK effectively does this internally, so there is usually no advantage in writing this out explicitly.

-12 (ECPG_OUT_OF_MEMORY)

Indicates that your virtual memory is exhausted. The numeric value is defined as -ENOMEM. (SQLSTATE YE001)

-200 (ECPG_UNSUPPORTED)

Indicates the preprocessor has generated something that the library does not know about. Perhaps you are running incompatible versions of the preprocessor and the library. (SQLSTATE YE002)

-201 (ECPG_TOO_MANY_ARGUMENTS)

This means that the command specified more host variables than the command expected. (SQLSTATE 07001 or 07002)

-202 (ECPG_TOO_FEW_ARGUMENTS)

This means that the command specified fewer host variables than the command expected. (SQLSTATE 07001 or 07002)

-203 (ECPG_TOO_MANY_MATCHES)

This means a query has returned multiple rows but the statement was only prepared to store one result row (for example, because the specified variables are not arrays). (SQLSTATE 21000)

-204 (ECPG_INT_FORMAT)

The host variable is of type int and the datum in the database is of a different type and contains a value that cannot be interpreted as an int. The library uses strtol() for this conversion. (SQLSTATE 42804)

-205 (ECPG_UINT_FORMAT)

The host variable is of type unsigned int and the datum in the database is of a different type and contains a value that cannot be interpreted as an unsigned int. The library uses strtoul() for this conversion. (SQLSTATE 42804)

-206 (ECPG_FLOAT_FORMAT)

The host variable is of type float and the datum in the database is of another type and contains a value that cannot be interpreted as a float. The library uses strtod() for this conversion. (SQLSTATE 42804)

-207 (ECPG_NUMERIC_FORMAT)

The host variable is of type numeric and the datum in the database is of another type and contains a value that cannot be interpreted as a numeric value. (SQLSTATE 42804)

-208 (ECPG_INTERVAL_FORMAT)

The host variable is of type interval and the datum in the database is of another type and contains a value that cannot be interpreted as an interval value. (SQLSTATE 42804)

-209 (ECPG_DATE_FORMAT)

The host variable is of type date and the datum in the database is of another type and contains a value that cannot be interpreted as a date value. (SQLSTATE 42804)

-210 (ECPG_TIMESTAMP_FORMAT)

The host variable is of type timestamp and the datum in the database is of another type and contains a value that cannot be interpreted as a timestamp value. (SQLSTATE 42804)

-211 (ECPG_CONVERT_BOOL)

This means the host variable is of type bool and the datum in the database is neither 't' nor 'f'. (SQLSTATE 42804)

-212 (ECPG_EMPTY)

The statement sent to the PostgreSQL server was empty. (This cannot normally happen in an embedded SQL program, so it might point to an internal error.) (SQLSTATE YE002)

-213 (ECPG_MISSING_INDICATOR)

A null value was returned and no null indicator variable was supplied. (SQLSTATE 22002)

-214 (ECPG_NO_ARRAY)

An ordinary variable was used in a place that requires an array. (SQLSTATE 42804)

-215 (ECPG_DATA_NOT_ARRAY)

The database returned an ordinary variable in a place that requires array value. (SQLSTATE 42804)

-220 (ECPG_NO_CONN)

The program tried to access a connection that does not exist. (SQLSTATE 08003)

-221 (ECPG_NOT_CONN)

The program tried to access a connection that does exist but is not open. (This is an internal error.) (SQLSTATE YE002)

-230 (ECPG_INVALID_STMT)

The statement you are trying to use has not been prepared. (SQLSTATE 26000)

-239 (ECPG_INFORMIX_DUPLICATE_KEY)

Duplicate key error, violation of unique constraint (Informix compatibility mode). (SQLSTATE 23505)

-240 (ECPG_UNKNOWN_DESCRIPTOR)

The descriptor specified was not found. The statement you are trying to use has not been prepared. (SQLSTATE 33000)

-241 (ECPG_INVALID_DESCRIPTOR_INDEX)

The descriptor index specified was out of range. (SQLSTATE 07009)

-242 (ECPG_UNKNOWN_DESCRIPTOR_ITEM)

An invalid descriptor item was requested. (This is an internal error.) (SQLSTATE YE002)

-243 (ECPG_VAR_NOT_NUMERIC)

During the execution of a dynamic statement, the database returned a numeric value and the host variable was not numeric. (SQLSTATE 07006)

-244 (ECPG_VAR_NOT_CHAR)

During the execution of a dynamic statement, the database returned a non-numeric value and the host variable was numeric. (SQLSTATE 07006)

-284 (ECPG_INFORMIX_SUBSELECT_NOT_ONE)

A result of the subquery is not single row (Informix compatibility mode). (SQLSTATE 21000)

-400 (ECPG_PGSQL)

Some error caused by the PostgreSQL server. The message contains the error message from the PostgreSQL server.

-401 (ECPG_TRANS)

The PostgreSQL server signaled that we cannot start, commit, or rollback the transaction. (SQLSTATE 08007)

-402 (ECPG_CONNECT)

The connection attempt to the database did not succeed. (SQLSTATE 08001)

-403 (ECPG_DUPLICATE_KEY)

Duplicate key error, violation of unique constraint. (SQLSTATE 23505)

-404 (ECPG_SUBSELECT_NOT_ONE)

A result for the subquery is not single row. (SQLSTATE 21000)

-602 (ECPG_WARNING_UNKNOWN_PORTAL)

An invalid cursor name was specified. (SQLSTATE 34000)

-603 (ECPG_WARNING_IN_TRANSACTION)

Transaction is in progress. (SQLSTATE 25001)

-604 (ECPG_WARNING_NO_TRANSACTION)

There is no active (in-progress) transaction. (SQLSTATE 25P01)

-605 (ECPG_WARNING_PORTAL_EXISTS)

An existing cursor name was specified. (SQLSTATE 42P03)

33.9. Preprocessor Directives

Several preprocessor directives are available that modify how the `ecpg` preprocessor parses and processes a file.

33.9.1. Including Files

To include an external file into your embedded SQL program, use:

```
EXEC SQL INCLUDE filename;
EXEC SQL INCLUDE <filename>;
EXEC SQL INCLUDE "filename";
```

The embedded SQL preprocessor will look for a file named *filename*`.h`, preprocess it, and include it in the resulting C output. Thus, embedded SQL statements in the included file are handled correctly.

The `ecpg` preprocessor will search a file at several directories in following order:

- current directory
- `/usr/local/include`
- PostgreSQL include directory, defined at build time (e.g., `/usr/local/pgsql/include`)
- `/usr/include`

But when `EXEC SQL INCLUDE "`*filename*`"` is used, only the current directory is searched.

In each directory, the preprocessor will first look for the file name as given, and if not found will append `.h` to the file name and try again (unless the specified file name already has that suffix).

Note that `EXEC SQL INCLUDE` is *not* the same as:

```
#include <filename.h>
```

because this file would not be subject to SQL command preprocessing. Naturally, you can continue to use the C `#include` directive to include other header files.

> **Note:** The include file name is case-sensitive, even though the rest of the `EXEC SQL INCLUDE` command follows the normal SQL case-sensitivity rules.

33.9.2. The define and undef Directives

Similar to the directive `#define` that is known from C, embedded SQL has a similar concept:

```
EXEC SQL DEFINE name;
EXEC SQL DEFINE name value;
```

So you can define a name:

```
EXEC SQL DEFINE HAVE_FEATURE;
```

And you can also define constants:

```
EXEC SQL DEFINE MYNUMBER 12;
EXEC SQL DEFINE MYSTRING 'abc';
```

Use `undef` to remove a previous definition:

```
EXEC SQL UNDEF MYNUMBER;
```

Of course you can continue to use the C versions `#define` and `#undef` in your embedded SQL program. The difference is where your defined values get evaluated. If you use `EXEC SQL DEFINE` then the `ecpg` preprocessor evaluates the defines and substitutes the values. For example if you write:

```
EXEC SQL DEFINE MYNUMBER 12;
...
EXEC SQL UPDATE Tbl SET col = MYNUMBER;
```

then `ecpg` will already do the substitution and your C compiler will never see any name or identifier `MYNUMBER`. Note that you cannot use `#define` for a constant that you are going to use in an embedded SQL query because in this case the embedded SQL precompiler is not able to see this declaration.

33.9.3. ifdef, ifndef, else, elif, and endif Directives

You can use the following directives to compile code sections conditionally:

`EXEC SQL ifdef` *`name`*`;`

Checks a *`name`* and processes subsequent lines if *`name`* has been created with `EXEC SQL define` *`name`*.

`EXEC SQL ifndef` *`name`*`;`

Checks a *`name`* and processes subsequent lines if *`name`* has *not* been created with `EXEC SQL define` *`name`*.

`EXEC SQL else;`

Starts processing an alternative section to a section introduced by either `EXEC SQL ifdef` *`name`* or `EXEC SQL ifndef` *`name`*.

`EXEC SQL elif` *`name`*`;`

Checks *`name`* and starts an alternative section if *`name`* has been created with `EXEC SQL define` *`name`*.

`EXEC SQL endif;`

Ends an alternative section.

Example:

```
EXEC SQL ifndef TZVAR;
```

```
EXEC SQL SET TIMEZONE TO 'GMT';
EXEC SQL elif TZNAME;
EXEC SQL SET TIMEZONE TO TZNAME;
EXEC SQL else;
EXEC SQL SET TIMEZONE TO TZVAR;
EXEC SQL endif;
```

33.10. Processing Embedded SQL Programs

Now that you have an idea how to form embedded SQL C programs, you probably want to know how to compile them. Before compiling you run the file through the embedded SQL C preprocessor, which converts the SQL statements you used to special function calls. After compiling, you must link with a special library that contains the needed functions. These functions fetch information from the arguments, perform the SQL command using the libpq interface, and put the result in the arguments specified for output.

The preprocessor program is called `ecpg` and is included in a normal PostgreSQL installation. Embedded SQL programs are typically named with an extension `.pgc`. If you have a program file called `prog1.pgc`, you can preprocess it by simply calling:

```
ecpg prog1.pgc
```

This will create a file called `prog1.c`. If your input files do not follow the suggested naming pattern, you can specify the output file explicitly using the `-o` option.

The preprocessed file can be compiled normally, for example:

```
cc -c prog1.c
```

The generated C source files include header files from the PostgreSQL installation, so if you installed PostgreSQL in a location that is not searched by default, you have to add an option such as `-I/usr/local/pgsql/include` to the compilation command line.

To link an embedded SQL program, you need to include the `libecpg` library, like so:

```
cc -o myprog prog1.o prog2.o ... -lecpg
```

Again, you might have to add an option like `-L/usr/local/pgsql/lib` to that command line.

You can use `pg_config` or `pkg-config` with package name `libecpg` to get the paths for your installation.

If you manage the build process of a larger project using make, it might be convenient to include the following implicit rule to your makefiles:

```
ECPG = ecpg

%.c: %.pgc
        $(ECPG) $<
```

The complete syntax of the `ecpg` command is detailed in ecpg.

The ecpg library is thread-safe by default. However, you might need to use some threading command-line options to compile your client code.

33.11. Library Functions

The `libecpg` library primarily contains "hidden" functions that are used to implement the functionality expressed by the embedded SQL commands. But there are some functions that can usefully be called directly. Note that this makes your code unportable.

- `ECPGdebug(int on, FILE *stream)` turns on debug logging if called with the first argument non-zero. Debug logging is done on *stream*. The log contains all SQL statements with all the input variables inserted, and the results from the PostgreSQL server. This can be very useful when searching for errors in your SQL statements.

  Note: On Windows, if the ecpg libraries and an application are compiled with different flags, this function call will crash the application because the internal representation of the `FILE` pointers differ. Specifically, multithreaded/single-threaded, release/debug, and static/dynamic flags should be the same for the library and all applications using that library.

- `ECPGget_PGconn(const char *connection_name)` returns the library database connection handle identified by the given name. If *connection_name* is set to `NULL`, the current connection handle is returned. If no connection handle can be identified, the function returns `NULL`. The returned connection handle can be used to call any other functions from libpq, if necessary.

  Note: It is a bad idea to manipulate database connection handles made from ecpg directly with libpq routines.

- `ECPGtransactionStatus(const char *connection_name)` returns the current transaction status of the given connection identified by *connection_name*. See Section 31.2 and libpq's `PQtransactionStatus()` for details about the returned status codes.
- `ECPGstatus(int lineno, const char* connection_name)` returns true if you are connected to a database and false if not. *connection_name* can be `NULL` if a single connection is being used.

33.12. Large Objects

Large objects are not directly supported by ECPG, but ECPG application can manipulate large objects through the libpq large object functions, obtaining the necessary `PGconn` object by calling the

`ECPGget_PGconn()` function. (However, use of the `ECPGget_PGconn()` function and touching `PGconn` objects directly should be done very carefully and ideally not mixed with other ECPG database access calls.)

For more details about the `ECPGget_PGconn()`, see Section 33.11. For information about the large object function interface, see Chapter 32.

Large object functions have to be called in a transaction block, so when autocommit is off, `BEGIN` commands have to be issued explicitly.

Example 33-2 shows an example program that illustrates how to create, write, and read a large object in an ECPG application.

Example 33-2. ECPG Program Accessing Large Objects

```
#include <stdio.h>
#include <stdlib.h>
#include <libpq-fe.h>
#include <libpq/libpq-fs.h>

EXEC SQL WHENEVER SQLERROR STOP;

int
main(void)
{
    PGconn     *conn;
    Oid         loid;
    int         fd;
    char        buf[256];
    int         buflen = 256;
    char        buf2[256];
    int         rc;

    memset(buf, 1, buflen);

    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    conn = ECPGget_PGconn("con1");
    printf("conn = %p\n", conn);

    /* create */
    loid = lo_create(conn, 0);
    if (loid &lt; 0)
        printf("lo_create() failed: %s", PQerrorMessage(conn));

    printf("loid = %d\n", loid);

    /* write test */
    fd = lo_open(conn, loid, INV_READ|INV_WRITE);
    if (fd &lt; 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);
```

```
    rc = lo_write(conn, fd, buf, buflen);
    if (rc &lt; 0)
        printf("lo_write() failed\n");

    rc = lo_close(conn, fd);
    if (rc &lt; 0)
        printf("lo_close() failed: %s", PQerrorMessage(conn));

    /* read test */
    fd = lo_open(conn, loid, INV_READ);
    if (fd &lt; 0)
        printf("lo_open() failed: %s", PQerrorMessage(conn));

    printf("fd = %d\n", fd);

    rc = lo_read(conn, fd, buf2, buflen);
    if (rc &lt; 0)
        printf("lo_read() failed\n");

    rc = lo_close(conn, fd);
    if (rc &lt; 0)
        printf("lo_close() failed: %s", PQerrorMessage(conn));

    /* check */
    rc = memcmp(buf, buf2, buflen);
    printf("memcmp() = %d\n", rc);

    /* cleanup */
    rc = lo_unlink(conn, loid);
    if (rc &lt; 0)
        printf("lo_unlink() failed: %s", PQerrorMessage(conn));

    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT ALL;
    return 0;
}
```

33.13. C++ Applications

ECPG has some limited support for C++ applications. This section describes some caveats.

The `ecpg` preprocessor takes an input file written in C (or something like C) and embedded SQL commands, converts the embedded SQL commands into C language chunks, and finally generates a `.c` file. The header file declarations of the library functions used by the C language chunks that `ecpg` generates are wrapped in `extern "C" { ... }` blocks when used under C++, so they should work seamlessly in C++.

In general, however, the `ecpg` preprocessor only understands C; it does not handle the special syntax and reserved words of the C++ language. So, some embedded SQL code written in C++ application code that

uses complicated features specific to C++ might fail to be preprocessed correctly or might not work as expected.

A safe way to use the embedded SQL code in a C++ application is hiding the ECPG calls in a C module, which the C++ application code calls into to access the database, and linking that together with the rest of the C++ code. See Section 33.13.2 about that.

33.13.1. Scope for Host Variables

The `ecpg` preprocessor understands the scope of variables in C. In the C language, this is rather simple because the scopes of variables is based on their code blocks. In C++, however, the class member variables are referenced in a different code block from the declared position, so the `ecpg` preprocessor will not understand the scope of the class member variables.

For example, in the following case, the `ecpg` preprocessor cannot find any declaration for the variable `dbname` in the `test` method, so an error will occur.

```
class TestCpp
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

  public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    EXEC SQL CONNECT TO testdb1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
}

void Test::test()
{
    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
}

TestCpp::~TestCpp()
{
    EXEC SQL DISCONNECT ALL;
}
```

This code will result in an error like this:

```
ecpg test_cpp.pgc
test_cpp.pgc:28: ERROR: variable "dbname" is not declared
```

To avoid this scope issue, the `test` method could be modified to use a local variable as intermediate storage. But this approach is only a poor workaround, because it uglifies the code and reduces performance.

```
void TestCpp::test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char tmp[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :tmp;
    strlcpy(dbname, tmp, sizeof(tmp));

    printf("current_database = %s\n", dbname);
}
```

33.13.2. C++ Application Development with External C Module

If you understand these technical limitations of the `ecpg` preprocessor in C++, you might come to the conclusion that linking C objects and C++ objects at the link stage to enable C++ applications to use ECPG features could be better than writing some embedded SQL commands in C++ code directly. This section describes a way to separate some embedded SQL commands from C++ application code with a simple example. In this example, the application is implemented in C++, while C and ECPG is used to connect to the PostgreSQL server.

Three kinds of files have to be created: a C file (`*.pgc`), a header file, and a C++ file:

`test_mod.pgc`

A sub-routine module to execute SQL commands embedded in C. It is going to be converted into `test_mod.c` by the preprocessor.

```
#include "test_mod.h"
#include <stdio.h>

void
db_connect()
{
    EXEC SQL CONNECT TO testdb1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
}

void
db_test()
{
    EXEC SQL BEGIN DECLARE SECTION;
    char dbname[1024];
    EXEC SQL END DECLARE SECTION;

    EXEC SQL SELECT current_database() INTO :dbname;
    printf("current_database = %s\n", dbname);
```

```
}

void
db_disconnect()
{
    EXEC SQL DISCONNECT ALL;
}
```

`test_mod.h`

A header file with declarations of the functions in the C module (`test_mod.pgc`). It is included by `test_cpp.cpp`. This file has to have an `extern "C"` block around the declarations, because it will be linked from the C++ module.

```
#ifdef __cplusplus
extern "C" {
#endif

void db_connect();
void db_test();
void db_disconnect();

#ifdef __cplusplus
}
#endif
```

`test_cpp.cpp`

The main code for the application, including the `main` routine, and in this example a C++ class.

```
#include "test_mod.h"

class TestCpp
{
  public:
    TestCpp();
    void test();
    ~TestCpp();
};

TestCpp::TestCpp()
{
    db_connect();
}

void
TestCpp::test()
{
    db_test();
}

TestCpp::~TestCpp()
{
    db_disconnect();
}
```

```
int
main(void)
{
    TestCpp *t = new TestCpp();

    t->test();
    return 0;
}
```

To build the application, proceed as follows. Convert `test_mod.pgc` into `test_mod.c` by running `ecpg`, and generate `test_mod.o` by compiling `test_mod.c` with the C compiler:

```
ecpg -o test_mod.c test_mod.pgc
cc -c test_mod.c -o test_mod.o
```

Next, generate `test_cpp.o` by compiling `test_cpp.cpp` with the C++ compiler:

```
c++ -c test_cpp.cpp -o test_cpp.o
```

Finally, link these object files, `test_cpp.o` and `test_mod.o`, into one executable, using the C++ compiler driver:

```
c++ test_cpp.o test_mod.o -lecpg -o test_cpp
```

33.14. Embedded SQL Commands

This section describes all SQL commands that are specific to embedded SQL. Also refer to the SQL commands listed in Reference I, *SQL Commands*, which can also be used in embedded SQL, unless stated otherwise.

ALLOCATE DESCRIPTOR

Name

`ALLOCATE DESCRIPTOR` — allocate an SQL descriptor area

Synopsis

```
ALLOCATE DESCRIPTOR name
```

Description

`ALLOCATE DESCRIPTOR` allocates a new named SQL descriptor area, which can be used to exchange data between the PostgreSQL server and the host program.

Descriptor areas should be freed after use using the `DEALLOCATE DESCRIPTOR` command.

Parameters

`name`

A name of SQL descriptor, case sensitive. This can be an SQL identifier or a host variable.

Examples

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
```

Compatibility

`ALLOCATE DESCRIPTOR` is specified in the SQL standard.

See Also

DEALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

CONNECT

Name

CONNECT — establish a database connection

Synopsis

```
CONNECT TO connection_target [ AS connection_name ] [ USER connection_user ]
CONNECT TO DEFAULT
CONNECT connection_user
DATABASE connection_target
```

Description

The CONNECT command establishes a connection between the client and the PostgreSQL server.

Parameters

connection_target

connection_target specifies the target server of the connection on one of several forms.

[*database_name*] [@*host*] [:*port*]

Connect over TCP/IP

unix:postgresql://*host* [:*port*] / [*database_name*] [?*connection_option*]

Connect over Unix-domain sockets

tcp:postgresql://*host* [:*port*] / [*database_name*] [?*connection_option*]

Connect over TCP/IP

SQL string constant

containing a value in one of the above forms

host variable

host variable of type char[] or VARCHAR[] containing a value in one of the above forms

connection_name

An optional identifier for the connection, so that it can be referred to in other commands. This can be an SQL identifier or a host variable.

connection_user

The user name for the database connection.

This parameter can also specify user name and password, using one the forms *user_name*/*password*, *user_name* IDENTIFIED BY *password*, or *user_name* USING *password*.

User name and password can be SQL identifiers, string constants, or host variables.

DEFAULT

Use all default connection parameters, as defined by libpq.

Examples

Here a several variants for specifying connection parameters:

```
EXEC SQL CONNECT TO "connectdb" AS main;
EXEC SQL CONNECT TO "connectdb" AS second;
EXEC SQL CONNECT TO "unix:postgresql://200.46.204.71/connectdb" AS main USER connectuser;
EXEC SQL CONNECT TO "unix:postgresql://localhost/connectdb" AS main USER connectuser;
EXEC SQL CONNECT TO 'connectdb' AS main;
EXEC SQL CONNECT TO 'unix:postgresql://localhost/connectdb' AS main USER :user;
EXEC SQL CONNECT TO :db AS :id;
EXEC SQL CONNECT TO :db USER connectuser USING :pw;
EXEC SQL CONNECT TO @localhost AS main USER connectdb;
EXEC SQL CONNECT TO REGRESSDB1 as main;
EXEC SQL CONNECT TO AS main USER connectdb;
EXEC SQL CONNECT TO connectdb AS :id;
EXEC SQL CONNECT TO connectdb AS main USER connectuser/connectdb;
EXEC SQL CONNECT TO connectdb AS main;
EXEC SQL CONNECT TO connectdb@localhost AS main;
EXEC SQL CONNECT TO tcp:postgresql://localhost/ USER connectdb;
EXEC SQL CONNECT TO tcp:postgresql://localhost/connectdb USER connectuser IDENTIFIED BY con
EXEC SQL CONNECT TO tcp:postgresql://localhost:20/connectdb USER connectuser IDENTIFIED BY
EXEC SQL CONNECT TO unix:postgresql://localhost/ AS main USER connectdb;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb AS main USER connectuser;
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser IDENTIFIED BY "c
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser USING "connectpw
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb?connect_timeout=14 USER connectus
```

Here is an example program that illustrates the use of host variables to specify connection parameters:

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    char *dbname     = "testdb";    /* database name */
    char *user       = "testuser";  /* connection user name */
    char *connection = "tcp:postgresql://localhost:5432/testdb";
                                    /* connection string */
    char ver[256];                  /* buffer to store the version string */
```

```
EXEC SQL END DECLARE SECTION;

    ECPGdebug(1, stderr);

    EXEC SQL CONNECT TO :dbname USER :user;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL SELECT version() INTO :ver;
    EXEC SQL DISCONNECT;

    printf("version: %s\n", ver);

    EXEC SQL CONNECT TO :connection USER :user;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL SELECT version() INTO :ver;
    EXEC SQL DISCONNECT;

    printf("version: %s\n", ver);

    return 0;
}
```

Compatibility

`CONNECT` is specified in the SQL standard, but the format of the connection parameters is implementation-specific.

See Also

DISCONNECT, SET CONNECTION

DEALLOCATE DESCRIPTOR

Name

`DEALLOCATE DESCRIPTOR` — deallocate an SQL descriptor area

Synopsis

```
DEALLOCATE DESCRIPTOR name
```

Description

`DEALLOCATE DESCRIPTOR` deallocates a named SQL descriptor area.

Parameters

`name`

The name of the descriptor which is going to be deallocated. It is case sensitive. This can be an SQL identifier or a host variable.

Examples

```
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

Compatibility

`DEALLOCATE DESCRIPTOR` is specified in the SQL standard.

See Also

ALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

DECLARE

Name

DECLARE — define a cursor

Synopsis

```
DECLARE cursor_name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH | WITHOUT }
DECLARE cursor_name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH | WITHOUT }
```

Description

DECLARE declares a cursor for iterating over the result set of a prepared statement. This command has slightly different semantics from the direct SQL command DECLARE: Whereas the latter executes a query and prepares the result set for retrieval, this embedded SQL command merely declares a name as a "loop variable" for iterating over the result set of a query; the actual execution happens when the cursor is opened with the OPEN command.

Parameters

cursor_name

A cursor name, case sensitive. This can be an SQL identifier or a host variable.

prepared_name

The name of a prepared query, either as an SQL identifier or a host variable.

query

A SELECT or VALUES command which will provide the rows to be returned by the cursor.

For the meaning of the cursor options, see DECLARE.

Examples

Examples declaring a cursor for a query:

```
EXEC SQL DECLARE C CURSOR FOR SELECT * FROM My_Table;
EXEC SQL DECLARE C CURSOR FOR SELECT Item1 FROM T;
EXEC SQL DECLARE cur1 CURSOR FOR SELECT version();
```

An example declaring a cursor for a prepared statement:

```
EXEC SQL PREPARE stmt1 AS SELECT version();
```

```
EXEC SQL DECLARE cur1 CURSOR FOR stmt1;
```

Compatibility

`DECLARE` is specified in the SQL standard.

See Also

OPEN, CLOSE, DECLARE

DESCRIBE

Name

DESCRIBE — obtain information about a prepared statement or result set

Synopsis

```
DESCRIBE [ OUTPUT ] prepared_name USING [ SQL ] DESCRIPTOR descriptor_name
DESCRIBE [ OUTPUT ] prepared_name INTO [ SQL ] DESCRIPTOR descriptor_name
DESCRIBE [ OUTPUT ] prepared_name INTO sqlda_name
```

Description

DESCRIBE retrieves metadata information about the result columns contained in a prepared statement, without actually fetching a row.

Parameters

prepared_name

The name of a prepared statement. This can be an SQL identifier or a host variable.

descriptor_name

A descriptor name. It is case sensitive. It can be an SQL identifier or a host variable.

sqlda_name

The name of an SQLDA variable.

Examples

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc;
EXEC SQL PREPARE stmt1 FROM :sql_stmt;
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc;
EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :charvar = NAME;
EXEC SQL DEALLOCATE DESCRIPTOR mydesc;
```

Compatibility

DESCRIBE is specified in the SQL standard.

See Also

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

DISCONNECT

Name

DISCONNECT — terminate a database connection

Synopsis

```
DISCONNECT connection_name
DISCONNECT [ CURRENT ]
DISCONNECT DEFAULT
DISCONNECT ALL
```

Description

DISCONNECT closes a connection (or all connections) to the database.

Parameters

connection_name

A database connection name established by the CONNECT command.

CURRENT

Close the "current" connection, which is either the most recently opened connection, or the connection set by the SET CONNECTION command. This is also the default if no argument is given to the DISCONNECT command.

DEFAULT

Close the default connection.

ALL

Close all open connections.

Examples

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS DEFAULT USER testuser;
    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL CONNECT TO testdb AS con2 USER testuser;
    EXEC SQL CONNECT TO testdb AS con3 USER testuser;

    EXEC SQL DISCONNECT CURRENT;  /* close con3          */
```

```
    EXEC SQL DISCONNECT DEFAULT;  /* close DEFAULT       */
    EXEC SQL DISCONNECT ALL;      /* close con2 and con1 */

    return 0;
}
```

Compatibility

`DISCONNECT` is specified in the SQL standard.

See Also

CONNECT, SET CONNECTION

EXECUTE IMMEDIATE

Name

`EXECUTE IMMEDIATE` — dynamically prepare and execute a statement

Synopsis

```
EXECUTE IMMEDIATE string
```

Description

`EXECUTE IMMEDIATE` immediately prepares and executes a dynamically specified SQL statement, without retrieving result rows.

Parameters

`string`

A literal C string or a host variable containing the SQL statement to be executed.

Examples

Here is an example that executes an `INSERT` statement using `EXECUTE IMMEDIATE` and a host variable named `command`:

```
sprintf(command, "INSERT INTO test (name, amount, letter) VALUES ('db: ''r1''', 1, 'f')");
EXEC SQL EXECUTE IMMEDIATE :command;
```

Compatibility

`EXECUTE IMMEDIATE` is specified in the SQL standard.

GET DESCRIPTOR

Name

`GET DESCRIPTOR` — get information from an SQL descriptor area

Synopsis

```
GET DESCRIPTOR descriptor_name :cvariable = descriptor_header_item [, ... ]
GET DESCRIPTOR descriptor_name VALUE column_number :cvariable = descriptor_item [, ... ]
```

Description

`GET DESCRIPTOR` retrieves information about a query result set from an SQL descriptor area and stores it into host variables. A descriptor area is typically populated using `FETCH` or `SELECT` before using this command to transfer the information into host language variables.

This command has two forms: The first form retrieves descriptor "header" items, which apply to the result set in its entirety. One example is the row count. The second form, which requires the column number as additional parameter, retrieves information about a particular column. Examples are the column name and the actual column value.

Parameters

`descriptor_name`

A descriptor name.

`descriptor_header_item`

A token identifying which header information item to retrieve. Only `COUNT`, to get the number of columns in the result set, is currently supported.

`column_number`

The number of the column about which information is to be retrieved. The count starts at 1.

`descriptor_item`

A token identifying which item of information about a column to retrieve. See Section 33.7.1 for a list of supported items.

`cvariable`

A host variable that will receive the data retrieved from the descriptor area.

Examples

An example to retrieve the number of columns in a result set:

```
EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
```

An example to retrieve a data length in the first column:

```
EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length = RETURNED_OCTET_LENGTH;
```

An example to retrieve the data body of the second column as a string:

```
EXEC SQL GET DESCRIPTOR d VALUE 2 :d_data = DATA;
```

Here is an example for a whole procedure of executing `SELECT current_database();` and showing the number of columns, the column data length, and the column data:

```
int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    int  d_count;
    char d_data[1024];
    int  d_returned_octet_length;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1 USER testuser;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL ALLOCATE DESCRIPTOR d;

    /* Declare, open a cursor, and assign a descriptor to the cursor  */
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database();
    EXEC SQL OPEN cur;
    EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;

    /* Get a number of total columns */
    EXEC SQL GET DESCRIPTOR d :d_count = COUNT;
    printf("d_count                 = %d\n", d_count);

    /* Get length of a returned column */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length = RETURNED_OCTET_LENGTH;
    printf("d_returned_octet_length = %d\n", d_returned_octet_length);

    /* Fetch the returned column as a string */
    EXEC SQL GET DESCRIPTOR d VALUE 1 :d_data = DATA;
    printf("d_data                  = %s\n", d_data);

    /* Closing */
    EXEC SQL CLOSE cur;
```

```
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

When the example is executed, the result will look like this:

```
d_count                 = 1
d_returned_octet_length = 6
d_data                  = testdb
```

Compatibility

`GET DESCRIPTOR` is specified in the SQL standard.

See Also

ALLOCATE DESCRIPTOR, SET DESCRIPTOR

OPEN

Name

OPEN — open a dynamic cursor

Synopsis

```
OPEN cursor_name
OPEN cursor_name USING value [, ... ]
OPEN cursor_name USING SQL DESCRIPTOR descriptor_name
```

Description

OPEN opens a cursor and optionally binds actual values to the placeholders in the cursor's declaration. The cursor must previously have been declared with the DECLARE command. The execution of OPEN causes the query to start executing on the server.

Parameters

cursor_name

The name of the cursor to be opened. This can be an SQL identifier or a host variable.

value

A value to be bound to a placeholder in the cursor. This can be an SQL constant, a host variable, or a host variable with indicator.

descriptor_name

The name of a descriptor containing values to be bound to the placeholders in the cursor. This can be an SQL identifier or a host variable.

Examples

```
EXEC SQL OPEN a;
EXEC SQL OPEN d USING 1, 'test';
EXEC SQL OPEN c1 USING SQL DESCRIPTOR mydesc;
EXEC SQL OPEN :curname1;
```

Compatibility

OPEN is specified in the SQL standard.

See Also

DECLARE, CLOSE

PREPARE

Name

PREPARE — prepare a statement for execution

Synopsis

```
PREPARE name FROM string
```

Description

PREPARE prepares a statement dynamically specified as a string for execution. This is different from the direct SQL statement PREPARE, which can also be used in embedded programs. The EXECUTE command is used to execute either kind of prepared statement.

Parameters

prepared_name

An identifier for the prepared query.

string

A literal C string or a host variable containing a preparable statement, one of the SELECT, INSERT, UPDATE, or DELETE.

Examples

```
char *stmt = "SELECT * FROM test1 WHERE a = ? AND b = ?";

EXEC SQL ALLOCATE DESCRIPTOR outdesc;
EXEC SQL PREPARE foo FROM :stmt;

EXEC SQL EXECUTE foo USING SQL DESCRIPTOR indesc INTO SQL DESCRIPTOR outdesc;
```

Compatibility

PREPARE is specified in the SQL standard.

See Also

EXECUTE

SET AUTOCOMMIT

Name

`SET AUTOCOMMIT` — set the autocommit behavior of the current session

Synopsis

```
SET AUTOCOMMIT { = | TO } { ON | OFF }
```

Description

`SET AUTOCOMMIT` sets the autocommit behavior of the current database session. By default, embedded SQL programs are *not* in autocommit mode, so `COMMIT` needs to be issued explicitly when desired. This command can change the session to autocommit mode, where each individual statement is committed implicitly.

Compatibility

`SET AUTOCOMMIT` is an extension of PostgreSQL ECPG.

SET CONNECTION

Name

`SET CONNECTION` — select a database connection

Synopsis

```
SET CONNECTION [ TO | = ] connection_name
```

Description

`SET CONNECTION` sets the "current" database connection, which is the one that all commands use unless overridden.

Parameters

`connection_name`

A database connection name established by the `CONNECT` command.

`DEFAULT`

Set the connection to the default connection.

Examples

```
EXEC SQL SET CONNECTION TO con2;
EXEC SQL SET CONNECTION = con1;
```

Compatibility

`SET CONNECTION` is specified in the SQL standard.

See Also

CONNECT, DISCONNECT

SET DESCRIPTOR

Name

`SET DESCRIPTOR` — set information in an SQL descriptor area

Synopsis

```
SET DESCRIPTOR descriptor_name descriptor_header_item = value [, ... ]
SET DESCRIPTOR descriptor_name VALUE number descriptor_item = value [, ...]
```

Description

`SET DESCRIPTOR` populates an SQL descriptor area with values. The descriptor area is then typically used to bind parameters in a prepared query execution.

This command has two forms: The first form applies to the descriptor "header", which is independent of a particular datum. The second form assigns values to particular datums, identified by number.

Parameters

`descriptor_name`

A descriptor name.

`descriptor_header_item`

A token identifying which header information item to set. Only `COUNT`, to set the number of descriptor items, is currently supported.

`number`

The number of the descriptor item to set. The count starts at 1.

`descriptor_item`

A token identifying which item of information to set in the descriptor. See Section 33.7.1 for a list of supported items.

`value`

A value to store into the descriptor item. This can be an SQL constant or a host variable.

Examples

```
EXEC SQL SET DESCRIPTOR indesc COUNT = 1;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = 2;
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = :val1;
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val1, DATA = 'some string';
```

```
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val2null, DATA = :val2;
```

Compatibility

`SET DESCRIPTOR` is specified in the SQL standard.

See Also

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

TYPE

Name

`TYPE` — define a new data type

Synopsis

```
TYPE type_name IS ctype
```

Description

The `TYPE` command defines a new C type. It is equivalent to putting a `typedef` into a declare section.

This command is only recognized when `ecpg` is run with the `-c` option.

Parameters

type_name

The name for the new type. It must be a valid C type name.

ctype

A C type specification.

Examples

```
EXEC SQL TYPE customer IS
    struct
    {
        varchar name[50];
        int     phone;
    };

EXEC SQL TYPE cust_ind IS
    struct ind
    {
        short   name_ind;
        short   phone_ind;
    };

EXEC SQL TYPE c IS char reference;
EXEC SQL TYPE ind IS union { int integer; short smallint; };
EXEC SQL TYPE intarray IS int[AMOUNT];
EXEC SQL TYPE str IS varchar[BUFFERSIZ];
EXEC SQL TYPE string IS char[11];
```

Here is an example program that uses `EXEC SQL TYPE`:

```
EXEC SQL WHENEVER SQLERROR SQLPRINT;

EXEC SQL TYPE tt IS
    struct
    {
        varchar v[256];
        int     i;
    };

EXEC SQL TYPE tt_ind IS
    struct ind {
        short   v_ind;
        short   i_ind;
    };

int
main(void)
{
EXEC SQL BEGIN DECLARE SECTION;
    tt t;
    tt_ind t_ind;
EXEC SQL END DECLARE SECTION;

    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;

    EXEC SQL SELECT current_database(), 256 INTO :t:t_ind LIMIT 1;

    printf("t.v = %s\n", t.v.arr);
    printf("t.i = %d\n", t.i);

    printf("t_ind.v_ind = %d\n", t_ind.v_ind);
    printf("t_ind.i_ind = %d\n", t_ind.i_ind);

    EXEC SQL DISCONNECT con1;

    return 0;
}
```

The output from this program looks like this:

```
t.v = testdb
t.i = 256
t_ind.v_ind = 0
t_ind.i_ind = 0
```

Compatibility

The `TYPE` command is a PostgreSQL extension.

VAR

Name

VAR — define a variable

Synopsis

```
VAR varname IS ctype
```

Description

The VAR command assigns a new C data type to a host variable. The host variable must be previously declared in a declare section.

Parameters

varname

A C variable name.

ctype

A C type specification.

Examples

```
Exec sql begin declare section;
short a;
exec sql end declare section;
EXEC SQL VAR a IS int;
```

Compatibility

The VAR command is a PostgreSQL extension.

WHENEVER

Name

WHENEVER — specify the action to be taken when an SQL statement causes a specific class condition to be raised

Synopsis

```
WHENEVER { NOT FOUND | SQLERROR | SQLWARNING } action
```

Description

Define a behavior which is called on the special cases (Rows not found, SQL warnings or errors) in the result of SQL execution.

Parameters

See Section 33.8.1 for a description of the parameters.

Examples

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
EXEC SQL WHENEVER NOT FOUND DO BREAK;
EXEC SQL WHENEVER SQLWARNING SQLPRINT;
EXEC SQL WHENEVER SQLWARNING DO warn();
EXEC SQL WHENEVER SQLERROR sqlprint;
EXEC SQL WHENEVER SQLERROR CALL print2();
EXEC SQL WHENEVER SQLERROR DO handle_error("select");
EXEC SQL WHENEVER SQLERROR DO sqlnotice(NULL, NONO);
EXEC SQL WHENEVER SQLERROR DO sqlprint();
EXEC SQL WHENEVER SQLERROR GOTO error_label;
EXEC SQL WHENEVER SQLERROR STOP;
```

A typical application is the use of `WHENEVER NOT FOUND BREAK` to handle looping through result sets:

```
int
main(void)
{
    EXEC SQL CONNECT TO testdb AS con1;
    EXEC SQL SELECT pg_catalog.set_config('search_path', '', false); EXEC SQL COMMIT;
    EXEC SQL ALLOCATE DESCRIPTOR d;
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database(), 'hoge', 256;
    EXEC SQL OPEN cur;
```

```
    /* when end of result set reached, break out of while loop */
    EXEC SQL WHENEVER NOT FOUND DO BREAK;

    while (1)
    {
        EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d;
        ...
    }

    EXEC SQL CLOSE cur;
    EXEC SQL COMMIT;

    EXEC SQL DEALLOCATE DESCRIPTOR d;
    EXEC SQL DISCONNECT ALL;

    return 0;
}
```

Compatibility

`WHENEVER` is specified in the SQL standard, but most of the actions are PostgreSQL extensions.

33.15. Informix Compatibility Mode

`ecpg` can be run in a so-called *Informix compatibility mode*. If this mode is active, it tries to behave as if it were the Informix precompiler for Informix E/SQL. Generally spoken this will allow you to use the dollar sign instead of the `EXEC SQL` primitive to introduce embedded SQL commands:

```
$int j = 3;
$CONNECT TO :dbname;
$CREATE TABLE test(i INT PRIMARY KEY, j INT);
$INSERT INTO test(i, j) VALUES (7, :j);
$COMMIT;
```

> **Note:** There must not be any white space between the `$` and a following preprocessor directive, that is, `include`, `define`, `ifdef`, etc. Otherwise, the preprocessor will parse the token as a host variable.

There are two compatibility modes: `INFORMIX`, `INFORMIX_SE`

When linking programs that use this compatibility mode, remember to link against `libcompat` that is shipped with ECPG.

Besides the previously explained syntactic sugar, the Informix compatibility mode ports some functions for input, output and transformation of data as well as embedded SQL statements known from E/SQL to ECPG.

Informix compatibility mode is closely connected to the pgtypeslib library of ECPG. pgtypeslib maps SQL data types to data types within the C host program and most of the additional functions of the Informix compatibility mode allow you to operate on those C host program types. Note however that the extent of the compatibility is limited. It does not try to copy Informix behavior; it allows you to do more or less the same operations and gives you functions that have the same name and the same basic behavior but it is no drop-in replacement if you are using Informix at the moment. Moreover, some of the data types are different. For example, PostgreSQL's datetime and interval types do not know about ranges like for example `YEAR TO MINUTE` so you won't find support in ECPG for that either.

33.15.1. Additional Types

The Informix-special "string" pseudo-type for storing right-trimmed character string data is now supported in Informix-mode without using `typedef`. In fact, in Informix-mode, ECPG refuses to process source files that contain `typedef sometype string;`

```
EXEC SQL BEGIN DECLARE SECTION;
string userid; /* this variable will contain trimmed data */
EXEC SQL END DECLARE SECTION;

EXEC SQL FETCH MYCUR INTO :userid;
```

33.15.2. Additional/Missing Embedded SQL Statements

`CLOSE DATABASE`

This statement closes the current connection. In fact, this is a synonym for ECPG's `DISCONNECT CURRENT`:

```
$CLOSE DATABASE;                /* close the current connection */
EXEC SQL CLOSE DATABASE;
```

`FREE cursor_name`

Due to the differences how ECPG works compared to Informix's ESQL/C (i.e., which steps are purely grammar transformations and which steps rely on the underlying run-time library) there is no `FREE cursor_name` statement in ECPG. This is because in ECPG, `DECLARE CURSOR` doesn't translate to a function call into the run-time library that uses to the cursor name. This means that there's no run-time bookkeeping of SQL cursors in the ECPG run-time library, only in the PostgreSQL server.

`FREE statement_name`

`FREE statement_name` is a synonym for `DEALLOCATE PREPARE statement_name`.

33.15.3. Informix-compatible SQLDA Descriptor Areas

Informix-compatible mode supports a different structure than the one described in Section 33.7.2. See below:

```
struct sqlvar_compat
{
    short   sqltype;
    int     sqllen;
    char   *sqldata;
    short  *sqlind;
    char   *sqlname;
    char   *sqlformat;
    short   sqlitype;
    short   sqlilen;
    char   *sqlidata;
    int     sqlxid;
    char   *sqltypename;
    short   sqltypelen;
    short   sqlownerlen;
    short   sqlsourcetype;
    char   *sqlownername;
    int     sqlsourceid;
    char   *sqlilongdata;
    int     sqlflags;
    void   *sqlreserved;
};
```

```
struct sqlda_compat
{
    short  sqld;
    struct sqlvar_compat *sqlvar;
    char   desc_name[19];
    short  desc_occ;
    struct sqlda_compat *desc_next;
    void  *reserved;
};

typedef struct sqlvar_compat    sqlvar_t;
typedef struct sqlda_compat     sqlda_t;
```

The global properties are:

`sqld`

The number of fields in the `SQLDA` descriptor.

`sqlvar`

Pointer to the per-field properties.

`desc_name`

Unused, filled with zero-bytes.

`desc_occ`

Size of the allocated structure.

`desc_next`

Pointer to the next SQLDA structure if the result set contains more than one record.

`reserved`

Unused pointer, contains NULL. Kept for Informix-compatibility.

The per-field properties are below, they are stored in the `sqlvar` array:

`sqltype`

Type of the field. Constants are in `sqltypes.h`

`sqllen`

Length of the field data.

`sqldata`

Pointer to the field data. The pointer is of `char *` type, the data pointed by it is in a binary format. Example:

```
int intval;

switch (sqldata->sqlvar[i].sqltype)
{
    case SQLINTEGER:
        intval = *(int *)sqldata->sqlvar[i].sqldata;
```

```
            break;
      ...
    }
```

`sqlind`

Pointer to the NULL indicator. If returned by DESCRIBE or FETCH then it's always a valid pointer. If used as input for `EXECUTE ... USING sqlda;` then NULL-pointer value means that the value for this field is non-NULL. Otherwise a valid pointer and `sqlitype` has to be properly set. Example:

```
if (*(int2 *)sqldata->sqlvar[i].sqlind != 0)
    printf("value is NULL\n");
```

`sqlname`

Name of the field. 0-terminated string.

`sqlformat`

Reserved in Informix, value of `PQfformat()` for the field.

`sqlitype`

Type of the NULL indicator data. It's always SQLSMINT when returning data from the server. When the `SQLDA` is used for a parametrized query, the data is treated according to the set type.

`sqlilen`

Length of the NULL indicator data.

`sqlxid`

Extended type of the field, result of `PQftype()`.

`sqltypename`
`sqltypelen`
`sqlownerlen`
`sqlsourcetype`
`sqlownername`
`sqlsourceid`
`sqlflags`
`sqlreserved`

Unused.

`sqlilongdata`

It equals to `sqldata` if `sqllen` is larger than 32KB.

Example:

```
EXEC SQL INCLUDE sqlda.h;

    sqlda_t        *sqlda; /* This doesn't need to be under embedded DECLARE SECTION */

    EXEC SQL BEGIN DECLARE SECTION;
    char *prep_stmt = "select * from table1";
    int i;
    EXEC SQL END DECLARE SECTION;
```

```
...

EXEC SQL PREPARE mystmt FROM :prep_stmt;

EXEC SQL DESCRIBE mystmt INTO sqlda;

printf("# of fields: %d\n", sqlda->sqld);
for (i = 0; i < sqlda->sqld; i++)
  printf("field %d: \"%s\"\n", sqlda->sqlvar[i]->sqlname);

EXEC SQL DECLARE mycursor CURSOR FOR mystmt;
EXEC SQL OPEN mycursor;
EXEC SQL WHENEVER NOT FOUND GOTO out;

while (1)
{
  EXEC SQL FETCH mycursor USING sqlda;
}

EXEC SQL CLOSE mycursor;

free(sqlda); /* The main structure is all to be free(),
              * sqlda and sqlda->sqlvar is in one allocated area */
```

For more information, see the `sqlda.h` header and the `src/interfaces/ecpg/test/compat_informix/sqlda.pgc` regression test.

33.15.4. Additional Functions

`decadd`

Add two decimal type values.

```
int decadd(decimal *arg1, decimal *arg2, decimal *sum);
```

The function receives a pointer to the first operand of type decimal (`arg1`), a pointer to the second operand of type decimal (`arg2`) and a pointer to a value of type decimal that will contain the sum (`sum`). On success, the function returns 0. `ECPG_INFORMIX_NUM_OVERFLOW` is returned in case of overflow and `ECPG_INFORMIX_NUM_UNDERFLOW` in case of underflow. -1 is returned for other failures and `errno` is set to the respective `errno` number of the pgtypeslib.

`deccmp`

Compare two variables of type decimal.

```
int deccmp(decimal *arg1, decimal *arg2);
```

The function receives a pointer to the first decimal value (`arg1`), a pointer to the second decimal value (`arg2`) and returns an integer value that indicates which is the bigger value.

- 1, if the value that `arg1` points to is bigger than the value that `var2` points to
- -1, if the value that `arg1` points to is smaller than the value that `arg2` points to
- 0, if the value that `arg1` points to and the value that `arg2` points to are equal

`deccopy`

Copy a decimal value.

```
void deccopy(decimal *src, decimal *target);
```

The function receives a pointer to the decimal value that should be copied as the first argument (`src`) and a pointer to the target structure of type decimal (`target`) as the second argument.

`deccvasc`

Convert a value from its ASCII representation into a decimal type.

```
int deccvasc(char *cp, int len, decimal *np);
```

The function receives a pointer to string that contains the string representation of the number to be converted (`cp`) as well as its length `len`. `np` is a pointer to the decimal value that saves the result of the operation.

Valid formats are for example: `-2`, `.794`, `+3.44`, `592.49E07` or `-32.84e-4`.

The function returns 0 on success. If overflow or underflow occurred, `ECPG_INFORMIX_NUM_OVERFLOW` or `ECPG_INFORMIX_NUM_UNDERFLOW` is returned. If the ASCII representation could not be parsed, `ECPG_INFORMIX_BAD_NUMERIC` is returned or `ECPG_INFORMIX_BAD_EXPONENT` if this problem occurred while parsing the exponent.

`deccvdbl`

Convert a value of type double to a value of type decimal.

```
int deccvdbl(double dbl, decimal *np);
```

The function receives the variable of type double that should be converted as its first argument (`dbl`). As the second argument (`np`), the function receives a pointer to the decimal variable that should hold the result of the operation.

The function returns 0 on success and a negative value if the conversion failed.

`deccvint`

Convert a value of type int to a value of type decimal.

```
int deccvint(int in, decimal *np);
```

The function receives the variable of type int that should be converted as its first argument (`in`). As the second argument (`np`), the function receives a pointer to the decimal variable that should hold the result of the operation.

The function returns 0 on success and a negative value if the conversion failed.

`deccvlong`

Convert a value of type long to a value of type decimal.

```
int deccvlong(long lng, decimal *np);
```

The function receives the variable of type long that should be converted as its first argument (`lng`). As the second argument (`np`), the function receives a pointer to the decimal variable that should hold the result of the operation.

The function returns 0 on success and a negative value if the conversion failed.

`decdiv`

Divide two variables of type decimal.

```
int decdiv(decimal *n1, decimal *n2, decimal *result);
```

The function receives pointers to the variables that are the first (`n1`) and the second (`n2`) operands and calculates `n1`/`n2`. `result` is a pointer to the variable that should hold the result of the operation.

On success, 0 is returned and a negative value if the division fails. If overflow or underflow occurred, the function returns `ECPG_INFORMIX_NUM_OVERFLOW` or `ECPG_INFORMIX_NUM_UNDERFLOW` respectively. If an attempt to divide by zero is observed, the function returns `ECPG_INFORMIX_DIVIDE_ZERO`.

`decmul`

Multiply two decimal values.

```
int decmul(decimal *n1, decimal *n2, decimal *result);
```

The function receives pointers to the variables that are the first (`n1`) and the second (`n2`) operands and calculates `n1`*`n2`. `result` is a pointer to the variable that should hold the result of the operation.

On success, 0 is returned and a negative value if the multiplication fails. If overflow or underflow occurred, the function returns `ECPG_INFORMIX_NUM_OVERFLOW` or `ECPG_INFORMIX_NUM_UNDERFLOW` respectively.

`decsub`

Subtract one decimal value from another.

```
int decsub(decimal *n1, decimal *n2, decimal *result);
```

The function receives pointers to the variables that are the first (`n1`) and the second (`n2`) operands and calculates `n1`-`n2`. `result` is a pointer to the variable that should hold the result of the operation.

On success, 0 is returned and a negative value if the subtraction fails. If overflow or underflow occurred, the function returns `ECPG_INFORMIX_NUM_OVERFLOW` or `ECPG_INFORMIX_NUM_UNDERFLOW` respectively.

`dectoasc`

Convert a variable of type decimal to its ASCII representation in a C char* string.

```
int dectoasc(decimal *np, char *cp, int len, int right)
```

The function receives a pointer to a variable of type decimal (`np`) that it converts to its textual representation. `cp` is the buffer that should hold the result of the operation. The parameter `right` specifies, how many digits right of the decimal point should be included in the output. The result will be rounded to this number of decimal digits. Setting `right` to -1 indicates that all available decimal digits should be included in the output. If the length of the output buffer, which is indicated by `len` is not sufficient to hold the textual representation including the trailing zero byte, only a single `*` character is stored in the result and -1 is returned.

The function returns either -1 if the buffer `cp` was too small or `ECPG_INFORMIX_OUT_OF_MEMORY` if memory was exhausted.

`dectodbl`

Convert a variable of type decimal to a double.

```
int dectodbl(decimal *np, double *dblp);
```

The function receives a pointer to the decimal value to convert (`np`) and a pointer to the double variable that should hold the result of the operation (`dblp`).

On success, 0 is returned and a negative value if the conversion failed.

`dectoint`

Convert a variable to type decimal to an integer.

```
int dectoint(decimal *np, int *ip);
```

The function receives a pointer to the decimal value to convert (`np`) and a pointer to the integer variable that should hold the result of the operation (`ip`).

On success, 0 is returned and a negative value if the conversion failed. If an overflow occurred, `ECPG_INFORMIX_NUM_OVERFLOW` is returned.

Note that the ECPG implementation differs from the Informix implementation. Informix limits an integer to the range from -32767 to 32767, while the limits in the ECPG implementation depend on the architecture (`-INT_MAX .. INT_MAX`).

`dectolong`

Convert a variable to type decimal to a long integer.

```
int dectolong(decimal *np, long *lngp);
```

The function receives a pointer to the decimal value to convert (`np`) and a pointer to the long variable that should hold the result of the operation (`lngp`).

On success, 0 is returned and a negative value if the conversion failed. If an overflow occurred, `ECPG_INFORMIX_NUM_OVERFLOW` is returned.

Note that the ECPG implementation differs from the Informix implementation. Informix limits a long integer to the range from -2,147,483,647 to 2,147,483,647, while the limits in the ECPG implementation depend on the architecture (`-LONG_MAX .. LONG_MAX`).

`rdatestr`

Converts a date to a C char* string.

```
int rdatestr(date d, char *str);
```

The function receives two arguments, the first one is the date to convert (`d`) and the second one is a pointer to the target string. The output format is always `yyyy-mm-dd`, so you need to allocate at least 11 bytes (including the zero-byte terminator) for the string.

The function returns 0 on success and a negative value in case of error.

Note that ECPG's implementation differs from the Informix implementation. In Informix the format can be influenced by setting environment variables. In ECPG however, you cannot change the output format.

`rstrdate`

Parse the textual representation of a date.

```
int rstrdate(char *str, date *d);
```

The function receives the textual representation of the date to convert (`str`) and a pointer to a variable of type date (`d`). This function does not allow you to specify a format mask. It uses the default format mask of Informix which is `mm/dd/yyyy`. Internally, this function is implemented by means of `rdefmtdate`. Therefore, `rstrdate` is not faster and if you have the choice you should opt for `rdefmtdate` which allows you to specify the format mask explicitly.

The function returns the same values as `rdefmtdate`.

`rtoday`

Get the current date.

```
void rtoday(date *d);
```

The function receives a pointer to a date variable (`d`) that it sets to the current date.

Internally this function uses the *`PGTYPESdate_today`* function.

`rjulmdy`

Extract the values for the day, the month and the year from a variable of type date.

```
int rjulmdy(date d, short mdy[3]);
```

The function receives the date `d` and a pointer to an array of 3 short integer values `mdy`. The variable name indicates the sequential order: `mdy[0]` will be set to contain the number of the month, `mdy[1]` will be set to the value of the day and `mdy[2]` will contain the year.

The function always returns 0 at the moment.

Internally the function uses the *`PGTYPESdate_julmdy`* function.

`rdefmtdate`

Use a format mask to convert a character string to a value of type date.

```
int rdefmtdate(date *d, char *fmt, char *str);
```

The function receives a pointer to the date value that should hold the result of the operation (`d`), the format mask to use for parsing the date (`fmt`) and the C char* string containing the textual representation of the date (`str`). The textual representation is expected to match the format mask. However you do not need to have a 1:1 mapping of the string to the format mask. The function only analyzes the sequential order and looks for the literals `yy` or `yyyy` that indicate the position of the year, `mm` to indicate the position of the month and `dd` to indicate the position of the day.

The function returns the following values:

- 0 - The function terminated successfully.
- `ECPG_INFORMIX_ENOSHORTDATE` - The date does not contain delimiters between day, month and year. In this case the input string must be exactly 6 or 8 bytes long but isn't.
- `ECPG_INFORMIX_ENOTDMY` - The format string did not correctly indicate the sequential order of year, month and day.
- `ECPG_INFORMIX_BAD_DAY` - The input string does not contain a valid day.
- `ECPG_INFORMIX_BAD_MONTH` - The input string does not contain a valid month.
- `ECPG_INFORMIX_BAD_YEAR` - The input string does not contain a valid year.

Internally this function is implemented to use the *`PGTYPESdate_defmt_asc`* function. See the reference there for a table of example input.

`rfmtdate`

Convert a variable of type date to its textual representation using a format mask.

```
int rfmtdate(date d, char *fmt, char *str);
```

The function receives the date to convert (`d`), the format mask (`fmt`) and the string that will hold the textual representation of the date (`str`).

On success, 0 is returned and a negative value if an error occurred.

Internally this function uses the *`PGTYPESdate_fmt_asc`* function, see the reference there for examples.

`rmdyjul`

Create a date value from an array of 3 short integers that specify the day, the month and the year of the date.

```
int rmdyjul(short mdy[3], date *d);
```

The function receives the array of the 3 short integers (`mdy`) and a pointer to a variable of type date that should hold the result of the operation.

Currently the function returns always 0.

Internally the function is implemented to use the function *`PGTYPESdate_mdyjul`*.

`rdayofweek`

Return a number representing the day of the week for a date value.

```
int rdayofweek(date d);
```

The function receives the date variable `d` as its only argument and returns an integer that indicates the day of the week for this date.

- 0 - Sunday
- 1 - Monday
- 2 - Tuesday
- 3 - Wednesday
- 4 - Thursday
- 5 - Friday
- 6 - Saturday

Internally the function is implemented to use the function *`PGTYPESdate_dayofweek`*.

`dtcurrent`

Retrieve the current timestamp.

```
void dtcurrent(timestamp *ts);
```

The function retrieves the current timestamp and saves it into the timestamp variable that `ts` points to.

`dtcvasc`

Parses a timestamp from its textual representation into a timestamp variable.

```
int dtcvasc(char *str, timestamp *ts);
```

The function receives the string to parse (`str`) and a pointer to the timestamp variable that should hold the result of the operation (`ts`).

The function returns 0 on success and a negative value in case of error.

Internally this function uses the *PGTYPEStimestamp_from_asc* function. See the reference there for a table with example inputs.

dtcvfmtasc

Parses a timestamp from its textual representation using a format mask into a timestamp variable.

```
dtcvfmtasc(char *inbuf, char *fmtstr, timestamp *dtvalue)
```

The function receives the string to parse (inbuf), the format mask to use (fmtstr) and a pointer to the timestamp variable that should hold the result of the operation (dtvalue).

This function is implemented by means of the *PGTYPEStimestamp_defmt_asc* function. See the documentation there for a list of format specifiers that can be used.

The function returns 0 on success and a negative value in case of error.

dtsub

Subtract one timestamp from another and return a variable of type interval.

```
int dtsub(timestamp *ts1, timestamp *ts2, interval *iv);
```

The function will subtract the timestamp variable that ts2 points to from the timestamp variable that ts1 points to and will store the result in the interval variable that iv points to.

Upon success, the function returns 0 and a negative value if an error occurred.

dttoasc

Convert a timestamp variable to a C char* string.

```
int dttoasc(timestamp *ts, char *output);
```

The function receives a pointer to the timestamp variable to convert (ts) and the string that should hold the result of the operation (output). It converts ts to its textual representation according to the SQL standard, which is be YYYY-MM-DD HH:MM:SS.

Upon success, the function returns 0 and a negative value if an error occurred.

dttofmtasc

Convert a timestamp variable to a C char* using a format mask.

```
int dttofmtasc(timestamp *ts, char *output, int str_len, char *fmtstr);
```

The function receives a pointer to the timestamp to convert as its first argument (ts), a pointer to the output buffer (output), the maximal length that has been allocated for the output buffer (str_len) and the format mask to use for the conversion (fmtstr).

Upon success, the function returns 0 and a negative value if an error occurred.

Internally, this function uses the *PGTYPEStimestamp_fmt_asc* function. See the reference there for information on what format mask specifiers can be used.

intoasc

Convert an interval variable to a C char* string.

```
int intoasc(interval *i, char *str);
```

The function receives a pointer to the interval variable to convert (i) and the string that should hold the result of the operation (str). It converts i to its textual representation according to the SQL standard, which is be YYYY-MM-DD HH:MM:SS.

Upon success, the function returns 0 and a negative value if an error occurred.

`rfmtlong`

Convert a long integer value to its textual representation using a format mask.

```
int rfmtlong(long lng_val, char *fmt, char *outbuf);
```

The function receives the long value `lng_val`, the format mask `fmt` and a pointer to the output buffer `outbuf`. It converts the long value according to the format mask to its textual representation.

The format mask can be composed of the following format specifying characters:

- `*` (asterisk) - if this position would be blank otherwise, fill it with an asterisk.
- `&` (ampersand) - if this position would be blank otherwise, fill it with a zero.
- `#` - turn leading zeroes into blanks.
- `<` - left-justify the number in the string.
- `,` (comma) - group numbers of four or more digits into groups of three digits separated by a comma.
- `.` (period) - this character separates the whole-number part of the number from the fractional part.
- `-` (minus) - the minus sign appears if the number is a negative value.
- `+` (plus) - the plus sign appears if the number is a positive value.
- `(` - this replaces the minus sign in front of the negative number. The minus sign will not appear.
- `)` - this character replaces the minus and is printed behind the negative value.
- `$` - the currency symbol.

`rupshift`

Convert a string to upper case.

```
void rupshift(char *str);
```

The function receives a pointer to the string and transforms every lower case character to upper case.

`byleng`

Return the number of characters in a string without counting trailing blanks.

```
int byleng(char *str, int len);
```

The function expects a fixed-length string as its first argument (`str`) and its length as its second argument (`len`). It returns the number of significant characters, that is the length of the string without trailing blanks.

`ldchar`

Copy a fixed-length string into a null-terminated string.

```
void ldchar(char *src, int len, char *dest);
```

The function receives the fixed-length string to copy (`src`), its length (`len`) and a pointer to the destination memory (`dest`). Note that you need to reserve at least `len+1` bytes for the string that `dest` points to. The function copies at most `len` bytes to the new location (less if the source string has trailing blanks) and adds the null-terminator.

`rgetmsg`

```
int rgetmsg(int msgnum, char *s, int maxsize);
```

This function exists but is not implemented at the moment!

`rtypalign`

```
int rtypalign(int offset, int type);
```

This function exists but is not implemented at the moment!

`rtypmsize`

```
int rtypmsize(int type, int len);
```

This function exists but is not implemented at the moment!

`rtypwidth`

```
int rtypwidth(int sqltype, int sqllen);
```

This function exists but is not implemented at the moment!

`rsetnull`

Set a variable to NULL.

```
int rsetnull(int t, char *ptr);
```

The function receives an integer that indicates the type of the variable and a pointer to the variable itself that is cast to a C char* pointer.

The following types exist:

- `CCHARTYPE` - For a variable of type `char` or `char*`
- `CSHORTTYPE` - For a variable of type `short int`
- `CINTTYPE` - For a variable of type `int`
- `CBOOLTYPE` - For a variable of type `boolean`
- `CFLOATTYPE` - For a variable of type `float`
- `CLONGTYPE` - For a variable of type `long`
- `CDOUBLETYPE` - For a variable of type `double`
- `CDECIMALTYPE` - For a variable of type `decimal`
- `CDATETYPE` - For a variable of type `date`
- `CDTIMETYPE` - For a variable of type `timestamp`

Here is an example of a call to this function:

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

rsetnull(CCHARTYPE, (char *) c);
rsetnull(CSHORTTYPE, (char *) &s);
rsetnull(CINTTYPE, (char *) &i);
```

`risnull`

Test if a variable is NULL.

```
int risnull(int t, char *ptr);
```

The function receives the type of the variable to test (`t`) as well a pointer to this variable (`ptr`). Note that the latter needs to be cast to a char*. See the function *`rsetnull`* for a list of possible variable types.

Here is an example of how to use this function:

```
$char c[] = "abc       ";
$short s = 17;
$int i = -74874;

risnull(CCHARTYPE, (char *) c);
risnull(CSHORTTYPE, (char *) &s);
risnull(CINTTYPE, (char *) &i);
```

33.15.5. Additional Constants

Note that all constants here describe errors and all of them are defined to represent negative values. In the descriptions of the different constants you can also find the value that the constants represent in the current implementation. However you should not rely on this number. You can however rely on the fact all of them are defined to represent negative values.

`ECPG_INFORMIX_NUM_OVERFLOW`

Functions return this value if an overflow occurred in a calculation. Internally it is defined as -1200 (the Informix definition).

`ECPG_INFORMIX_NUM_UNDERFLOW`

Functions return this value if an underflow occurred in a calculation. Internally it is defined as -1201 (the Informix definition).

`ECPG_INFORMIX_DIVIDE_ZERO`

Functions return this value if an attempt to divide by zero is observed. Internally it is defined as -1202 (the Informix definition).

`ECPG_INFORMIX_BAD_YEAR`

Functions return this value if a bad value for a year was found while parsing a date. Internally it is defined as -1204 (the Informix definition).

`ECPG_INFORMIX_BAD_MONTH`

Functions return this value if a bad value for a month was found while parsing a date. Internally it is defined as -1205 (the Informix definition).

`ECPG_INFORMIX_BAD_DAY`

Functions return this value if a bad value for a day was found while parsing a date. Internally it is defined as -1206 (the Informix definition).

ECPG_INFORMIX_ENOSHORTDATE

Functions return this value if a parsing routine needs a short date representation but did not get the date string in the right length. Internally it is defined as -1209 (the Informix definition).

ECPG_INFORMIX_DATE_CONVERT

Functions return this value if an error occurred during date formatting. Internally it is defined as -1210 (the Informix definition).

ECPG_INFORMIX_OUT_OF_MEMORY

Functions return this value if memory was exhausted during their operation. Internally it is defined as -1211 (the Informix definition).

ECPG_INFORMIX_ENOTDMY

Functions return this value if a parsing routine was supposed to get a format mask (like `mmddyy`) but not all fields were listed correctly. Internally it is defined as -1212 (the Informix definition).

ECPG_INFORMIX_BAD_NUMERIC

Functions return this value either if a parsing routine cannot parse the textual representation for a numeric value because it contains errors or if a routine cannot complete a calculation involving numeric variables because at least one of the numeric variables is invalid. Internally it is defined as -1213 (the Informix definition).

ECPG_INFORMIX_BAD_EXPONENT

Functions return this value if a parsing routine cannot parse an exponent. Internally it is defined as -1216 (the Informix definition).

ECPG_INFORMIX_BAD_DATE

Functions return this value if a parsing routine cannot parse a date. Internally it is defined as -1218 (the Informix definition).

ECPG_INFORMIX_EXTRA_CHARS

Functions return this value if a parsing routine is passed extra characters it cannot parse. Internally it is defined as -1264 (the Informix definition).

33.16. Internals

This section explains how ECPG works internally. This information can occasionally be useful to help users understand how to use ECPG.

The first four lines written by `ecpg` to the output are fixed lines. Two are comments and two are include lines necessary to interface to the library. Then the preprocessor reads through the file and writes output. Normally it just echoes everything to the output.

When it sees an `EXEC SQL` statement, it intervenes and changes it. The command starts with `EXEC SQL` and ends with `;`. Everything in between is treated as an SQL statement and parsed for variable substitution.

Variable substitution occurs when a symbol starts with a colon (`:`). The variable with that name is looked up among the variables that were previously declared within a `EXEC SQL DECLARE` section.

The most important function in the library is `ECPGdo`, which takes care of executing most commands. It takes a variable number of arguments. This can easily add up to 50 or so arguments, and we hope this will not be a problem on any platform.

The arguments are:

A line number

This is the line number of the original line; used in error messages only.

A string

This is the SQL command that is to be issued. It is modified by the input variables, i.e., the variables that where not known at compile time but are to be entered in the command. Where the variables should go the string contains `?`.

Input variables

Every input variable causes ten arguments to be created. (See below.)

`ECPGt_EOIT`

An `enum` telling that there are no more input variables.

Output variables

Every output variable causes ten arguments to be created. (See below.) These variables are filled by the function.

`ECPGt_EORT`

An `enum` telling that there are no more variables.

For every variable that is part of the SQL command, the function gets ten arguments:

1. The type as a special symbol.
2. A pointer to the value or a pointer to the pointer.
3. The size of the variable if it is a `char` or `varchar`.
4. The number of elements in the array (for array fetches).
5. The offset to the next element in the array (for array fetches).
6. The type of the indicator variable as a special symbol.
7. A pointer to the indicator variable.
8. 0
9. The number of elements in the indicator array (for array fetches).
10. The offset to the next element in the indicator array (for array fetches).

Note that not all SQL commands are treated in this way. For instance, an open cursor statement like:

```
EXEC SQL OPEN cursor;
```

is not copied to the output. Instead, the cursor's `DECLARE` command is used at the position of the `OPEN` command because it indeed opens the cursor.

Here is a complete example describing the output of the preprocessor of a file `foo.pgc` (details might change with each particular version of the preprocessor):

```
EXEC SQL BEGIN DECLARE SECTION;
int index;
int result;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT res INTO :result FROM mytable WHERE index = :index;
```

is translated into:

```
/* Processed by ecpg (2.6.0) */
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* exec sql begin declare section */

#line 1 "foo.pgc"

 int index;
 int result;
/* exec sql end declare section */
...
ECPGdo(__LINE__, NULL, "SELECT res FROM mytable WHERE index = ?     ",
        ECPGt_int,&(index),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(result),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 147 "foo.pgc"
```

(The indentation here is added for readability and not something the preprocessor does.)

Chapter 34. The Information Schema

The information schema consists of a set of views that contain information about the objects defined in the current database. The information schema is defined in the SQL standard and can therefore be expected to be portable and remain stable — unlike the system catalogs, which are specific to PostgreSQL and are modeled after implementation concerns. The information schema views do not, however, contain information about PostgreSQL-specific features; to inquire about those you need to query the system catalogs or other PostgreSQL-specific views.

> **Note:** When querying the database for constraint information, it is possible for a standard-compliant query that expects to return one row to return several. This is because the SQL standard requires constraint names to be unique within a schema, but PostgreSQL does not enforce this restriction. PostgreSQL automatically-generated constraint names avoid duplicates in the same schema, but users can specify such duplicate names.
>
> This problem can appear when querying information schema views such as `check_constraint_routine_usage`, `check_constraints`, `domain_constraints`, and `referential_constraints`. Some other views have similar issues but contain the table name to help distinguish duplicate rows, e.g., `constraint_column_usage`, `constraint_table_usage`, `table_constraints`.

34.1. The Schema

The information schema itself is a schema named `information_schema`. This schema automatically exists in all databases. The owner of this schema is the initial database user in the cluster, and that user naturally has all the privileges on this schema, including the ability to drop it (but the space savings achieved by that are minuscule).

By default, the information schema is not in the schema search path, so you need to access all objects in it through qualified names. Since the names of some of the objects in the information schema are generic names that might occur in user applications, you should be careful if you want to put the information schema in the path.

34.2. Data Types

The columns of the information schema views use special data types that are defined in the information schema. These are defined as simple domains over ordinary built-in types. You should not use these types for work outside the information schema, but your applications must be prepared for them if they select from the information schema.

These types are:

`cardinal_number`

A nonnegative integer.

`character_data`

A character string (without specific maximum length).

`sql_identifier`

A character string. This type is used for SQL identifiers, the type `character_data` is used for any other kind of text data.

`time_stamp`

A domain over the type `timestamp with time zone`

`yes_or_no`

A character string domain that contains either `YES` or `NO`. This is used to represent Boolean (true/false) data in the information schema. (The information schema was invented before the type `boolean` was added to the SQL standard, so this convention is necessary to keep the information schema backward compatible.)

Every column in the information schema has one of these five types.

34.3. `information_schema_catalog_name`

`information_schema_catalog_name` is a table that always contains one row and one column containing the name of the current database (current catalog, in SQL terminology).

Table 34-1. `information_schema_catalog_name` Columns

Name	Data Type	Description
`catalog_name`	`sql_identifier`	Name of the database that contains this information schema

34.4. `administrable_role_authorizations`

The view `administrable_role_authorizations` identifies all roles that the current user has the admin option for.

Table 34-2. `administrable_role_authorizations` Columns

Name	Data Type	Description
`grantee`	`sql_identifier`	Name of the role to which this role membership was granted (can be the current user, or a different role in case of nested role memberships)
`role_name`	`sql_identifier`	Name of a role

Name	Data Type	Description
is_grantable	yes_or_no	Always YES

34.5. applicable_roles

The view applicable_roles identifies all roles whose privileges the current user can use. This means there is some chain of role grants from the current user to the role in question. The current user itself is also an applicable role. The set of applicable roles is generally used for permission checking.

Table 34-3. applicable_roles Columns

Name	Data Type	Description
grantee	sql_identifier	Name of the role to which this role membership was granted (can be the current user, or a different role in case of nested role memberships)
role_name	sql_identifier	Name of a role
is_grantable	yes_or_no	YES if the grantee has the admin option on the role, NO if not

34.6. attributes

The view attributes contains information about the attributes of composite data types defined in the database. (Note that the view does not give information about table columns, which are sometimes called attributes in PostgreSQL contexts.) Only those attributes are shown that the current user has access to (by way of being the owner of or having some privilege on the type).

Table 34-4. attributes Columns

Name	Data Type	Description
udt_catalog	sql_identifier	Name of the database containing the data type (always the current database)
udt_schema	sql_identifier	Name of the schema containing the data type
udt_name	sql_identifier	Name of the data type
attribute_name	sql_identifier	Name of the attribute
ordinal_position	cardinal_number	Ordinal position of the attribute within the data type (count starts at 1)

Name	Data Type	Description
attribute_default	character_data	Default expression of the attribute
is_nullable	yes_or_no	YES if the attribute is possibly nullable, NO if it is known not nullable.
data_type	character_data	Data type of the attribute, if it is a built-in type, or ARRAY if it is some array (in that case, see the view element_types), else USER-DEFINED (in that case, the type is identified in attribute_udt_name and associated columns).
character_maximum_length	cardinal_number	If data_type identifies a character or bit string type, the declared maximum length; null for all other data types or if no maximum length was declared.
character_octet_length	cardinal_number	If data_type identifies a character type, the maximum possible length in octets (bytes) of a datum; null for all other data types. The maximum octet length depends on the declared character maximum length (see above) and the server encoding.
character_set_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_schema	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_name	sql_identifier	Applies to a feature not available in PostgreSQL
collation_catalog	sql_identifier	Name of the database containing the collation of the attribute (always the current database), null if default or the data type of the attribute is not collatable
collation_schema	sql_identifier	Name of the schema containing the collation of the attribute, null if default or the data type of the attribute is not collatable

Name	Data Type	Description
collation_name	sql_identifier	Name of the collation of the attribute, null if default or the data type of the attribute is not collatable
numeric_precision	cardinal_number	If data_type identifies a numeric type, this column contains the (declared or implicit) precision of the type for this attribute. The precision indicates the number of significant digits. It can be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix. For all other data types, this column is null.
numeric_precision_radix	cardinal_number	If data_type identifies a numeric type, this column indicates in which base the values in the columns numeric_precision and numeric_scale are expressed. The value is either 2 or 10. For all other data types, this column is null.
numeric_scale	cardinal_number	If data_type identifies an exact numeric type, this column contains the (declared or implicit) scale of the type for this attribute. The scale indicates the number of significant digits to the right of the decimal point. It can be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix. For all other data types, this column is null.

Name	Data Type	Description
datetime_precision	cardinal_number	If data_type identifies a date, time, timestamp, or interval type, this column contains the (declared or implicit) fractional seconds precision of the type for this attribute, that is, the number of decimal digits maintained following the decimal point in the seconds value. For all other data types, this column is null.
interval_type	character_data	If data_type identifies an interval type, this column contains the specification which fields the intervals include for this attribute, e.g., YEAR TO MONTH, DAY TO SECOND, etc. If no field restrictions were specified (that is, the interval accepts all fields), and for all other data types, this field is null.
interval_precision	cardinal_number	Applies to a feature not available in PostgreSQL (see datetime_precision for the fractional seconds precision of interval type attributes)
attribute_udt_catalog	sql_identifier	Name of the database that the attribute data type is defined in (always the current database)
attribute_udt_schema	sql_identifier	Name of the schema that the attribute data type is defined in
attribute_udt_name	sql_identifier	Name of the attribute data type
scope_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
scope_schema	sql_identifier	Applies to a feature not available in PostgreSQL
scope_name	sql_identifier	Applies to a feature not available in PostgreSQL
maximum_cardinality	cardinal_number	Always null, because arrays always have unlimited maximum cardinality in PostgreSQL

Name	Data Type	Description
dtd_identifier	sql_identifier	An identifier of the data type descriptor of the column, unique among the data type descriptors pertaining to the table. This is mainly useful for joining with other instances of such identifiers. (The specific format of the identifier is not defined and not guaranteed to remain the same in future versions.)
is_derived_reference_attribute	yes_or_no	Applies to a feature not available in PostgreSQL

See also under Section 34.16, a similarly structured view, for further information on some of the columns.

34.7. character_sets

The view character_sets identifies the character sets available in the current database. Since PostgreSQL does not support multiple character sets within one database, this view only shows one, which is the database encoding.

Take note of how the following terms are used in the SQL standard:

character repertoire

An abstract collection of characters, for example UNICODE, UCS, or LATIN1. Not exposed as an SQL object, but visible in this view.

character encoding form

An encoding of some character repertoire. Most older character repertoires only use one encoding form, and so there are no separate names for them (e.g., LATIN1 is an encoding form applicable to the LATIN1 repertoire). But for example Unicode has the encoding forms UTF8, UTF16, etc. (not all supported by PostgreSQL). Encoding forms are not exposed as an SQL object, but are visible in this view.

character set

A named SQL object that identifies a character repertoire, a character encoding, and a default collation. A predefined character set would typically have the same name as an encoding form, but users could define other names. For example, the character set UTF8 would typically identify the character repertoire UCS, encoding form UTF8, and some default collation.

You can think of an "encoding" in PostgreSQL either as a character set or a character encoding form. They will have the same name, and there can only be one in one database.

Table 34-5. character_sets Columns

Name	Data Type	Description

Name	Data Type	Description
character_set_catalog	sql_identifier	Character sets are currently not implemented as schema objects, so this column is null.
character_set_schema	sql_identifier	Character sets are currently not implemented as schema objects, so this column is null.
character_set_name	sql_identifier	Name of the character set, currently implemented as showing the name of the database encoding
character_repertoire	sql_identifier	Character repertoire, showing UCS if the encoding is UTF8, else just the encoding name
form_of_use	sql_identifier	Character encoding form, same as the database encoding
default_collate_catalog	sql_identifier	Name of the database containing the default collation (always the current database, if any collation is identified)
default_collate_schema	sql_identifier	Name of the schema containing the default collation
default_collate_name	sql_identifier	Name of the default collation. The default collation is identified as the collation that matches the COLLATE and CTYPE settings of the current database. If there is no such collation, then this column and the associated schema and catalog columns are null.

34.8. check_constraint_routine_usage

The view check_constraint_routine_usage identifies routines (functions and procedures) that are used by a check constraint. Only those routines are shown that are owned by a currently enabled role.

Table 34-6. check_constraint_routine_usage Columns

Name	Data Type	Description
constraint_catalog	sql_identifier	Name of the database containing the constraint (always the current database)

Name	Data Type	Description
constraint_schema	sql_identifier	Name of the schema containing the constraint
constraint_name	sql_identifier	Name of the constraint
specific_catalog	sql_identifier	Name of the database containing the function (always the current database)
specific_schema	sql_identifier	Name of the schema containing the function
specific_name	sql_identifier	The "specific name" of the function. See Section 34.40 for more information.

34.9. check_constraints

The view check_constraints contains all check constraints, either defined on a table or on a domain, that are owned by a currently enabled role. (The owner of the table or domain is the owner of the constraint.)

Table 34-7. check_constraints Columns

Name	Data Type	Description
constraint_catalog	sql_identifier	Name of the database containing the constraint (always the current database)
constraint_schema	sql_identifier	Name of the schema containing the constraint
constraint_name	sql_identifier	Name of the constraint
check_clause	character_data	The check expression of the check constraint

34.10. collations

The view collations contains the collations available in the current database.

Table 34-8. collations Columns

Name	Data Type	Description
collation_catalog	sql_identifier	Name of the database containing the collation (always the current database)

Name	Data Type	Description
collation_schema	sql_identifier	Name of the schema containing the collation
collation_name	sql_identifier	Name of the default collation
pad_attribute	character_data	Always NO PAD (The alternative PAD SPACE is not supported by PostgreSQL.)

34.11. collation_character_set_applicability

The view collation_character_set_applicability identifies which character set the available collations are applicable to. In PostgreSQL, there is only one character set per database (see explanation in Section 34.7), so this view does not provide much useful information.

Table 34-9. collation_character_set_applicability Columns

Name	Data Type	Description
collation_catalog	sql_identifier	Name of the database containing the collation (always the current database)
collation_schema	sql_identifier	Name of the schema containing the collation
collation_name	sql_identifier	Name of the default collation
character_set_catalog	sql_identifier	Character sets are currently not implemented as schema objects, so this column is null
character_set_schema	sql_identifier	Character sets are currently not implemented as schema objects, so this column is null
character_set_name	sql_identifier	Name of the character set

34.12. column_domain_usage

The view column_domain_usage identifies all columns (of a table or a view) that make use of some domain defined in the current database and owned by a currently enabled role.

Table 34-10. column_domain_usage Columns

Name	Data Type	Description

Name	Data Type	Description
domain_catalog	sql_identifier	Name of the database containing the domain (always the current database)
domain_schema	sql_identifier	Name of the schema containing the domain
domain_name	sql_identifier	Name of the domain
table_catalog	sql_identifier	Name of the database containing the table (always the current database)
table_schema	sql_identifier	Name of the schema containing the table
table_name	sql_identifier	Name of the table
column_name	sql_identifier	Name of the column

34.13. column_options

The view column_options contains all the options defined for foreign table columns in the current database. Only those foreign table columns are shown that the current user has access to (by way of being the owner or having some privilege).

Table 34-11. column_options Columns

Name	Data Type	Description
table_catalog	sql_identifier	Name of the database that contains the foreign table (always the current database)
table_schema	sql_identifier	Name of the schema that contains the foreign table
table_name	sql_identifier	Name of the foreign table
column_name	sql_identifier	Name of the column
option_name	sql_identifier	Name of an option
option_value	character_data	Value of the option

34.14. column_privileges

The view column_privileges identifies all privileges granted on columns to a currently enabled role or by a currently enabled role. There is one row for each combination of column, grantor, and grantee.

If a privilege has been granted on an entire table, it will show up in this view as a grant for each column, but only for the privilege types where column granularity is possible: SELECT, INSERT, UPDATE, REFERENCES.

Table 34-12. `column_privileges` Columns

Name	Data Type	Description
`grantor`	`sql_identifier`	Name of the role that granted the privilege
`grantee`	`sql_identifier`	Name of the role that the privilege was granted to
`table_catalog`	`sql_identifier`	Name of the database that contains the table that contains the column (always the current database)
`table_schema`	`sql_identifier`	Name of the schema that contains the table that contains the column
`table_name`	`sql_identifier`	Name of the table that contains the column
`column_name`	`sql_identifier`	Name of the column
`privilege_type`	`character_data`	Type of the privilege: `SELECT`, `INSERT`, `UPDATE`, or `REFERENCES`
`is_grantable`	`yes_or_no`	`YES` if the privilege is grantable, `NO` if not

34.15. `column_udt_usage`

The view `column_udt_usage` identifies all columns that use data types owned by a currently enabled role. Note that in PostgreSQL, built-in data types behave like user-defined types, so they are included here as well. See also Section 34.16 for details.

Table 34-13. `column_udt_usage` Columns

Name	Data Type	Description
`udt_catalog`	`sql_identifier`	Name of the database that the column data type (the underlying type of the domain, if applicable) is defined in (always the current database)
`udt_schema`	`sql_identifier`	Name of the schema that the column data type (the underlying type of the domain, if applicable) is defined in

Name	Data Type	Description
udt_name	sql_identifier	Name of the column data type (the underlying type of the domain, if applicable)
table_catalog	sql_identifier	Name of the database containing the table (always the current database)
table_schema	sql_identifier	Name of the schema containing the table
table_name	sql_identifier	Name of the table
column_name	sql_identifier	Name of the column

34.16. columns

The view columns contains information about all table columns (or view columns) in the database. System columns (oid, etc.) are not included. Only those columns are shown that the current user has access to (by way of being the owner or having some privilege).

Table 34-14. columns Columns

Name	Data Type	Description
table_catalog	sql_identifier	Name of the database containing the table (always the current database)
table_schema	sql_identifier	Name of the schema containing the table
table_name	sql_identifier	Name of the table
column_name	sql_identifier	Name of the column
ordinal_position	cardinal_number	Ordinal position of the column within the table (count starts at 1)
column_default	character_data	Default expression of the column
is_nullable	yes_or_no	YES if the column is possibly nullable, NO if it is known not nullable. A not-null constraint is one way a column can be known not nullable, but there can be others.

Name	Data Type	Description
data_type	character_data	Data type of the column, if it is a built-in type, or ARRAY if it is some array (in that case, see the view element_types), else USER-DEFINED (in that case, the type is identified in udt_name and associated columns). If the column is based on a domain, this column refers to the type underlying the domain (and the domain is identified in domain_name and associated columns).
character_maximum_length	cardinal_number	If data_type identifies a character or bit string type, the declared maximum length; null for all other data types or if no maximum length was declared.
character_octet_length	cardinal_number	If data_type identifies a character type, the maximum possible length in octets (bytes) of a datum; null for all other data types. The maximum octet length depends on the declared character maximum length (see above) and the server encoding.
numeric_precision	cardinal_number	If data_type identifies a numeric type, this column contains the (declared or implicit) precision of the type for this column. The precision indicates the number of significant digits. It can be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix. For all other data types, this column is null.

Name	Data Type	Description
numeric_precision_radix	cardinal_number	If data_type identifies a numeric type, this column indicates in which base the values in the columns numeric_precision and numeric_scale are expressed. The value is either 2 or 10. For all other data types, this column is null.
numeric_scale	cardinal_number	If data_type identifies an exact numeric type, this column contains the (declared or implicit) scale of the type for this column. The scale indicates the number of significant digits to the right of the decimal point. It can be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix. For all other data types, this column is null.
datetime_precision	cardinal_number	If data_type identifies a date, time, timestamp, or interval type, this column contains the (declared or implicit) fractional seconds precision of the type for this column, that is, the number of decimal digits maintained following the decimal point in the seconds value. For all other data types, this column is null.
interval_type	character_data	If data_type identifies an interval type, this column contains the specification which fields the intervals include for this column, e.g., YEAR TO MONTH, DAY TO SECOND, etc. If no field restrictions were specified (that is, the interval accepts all fields), and for all other data types, this field is null.

Name	Data Type	Description
interval_precision	cardinal_number	Applies to a feature not available in PostgreSQL (see datetime_precision for the fractional seconds precision of interval type columns)
character_set_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_schema	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_name	sql_identifier	Applies to a feature not available in PostgreSQL
collation_catalog	sql_identifier	Name of the database containing the collation of the column (always the current database), null if default or the data type of the column is not collatable
collation_schema	sql_identifier	Name of the schema containing the collation of the column, null if default or the data type of the column is not collatable
collation_name	sql_identifier	Name of the collation of the column, null if default or the data type of the column is not collatable
domain_catalog	sql_identifier	If the column has a domain type, the name of the database that the domain is defined in (always the current database), else null.
domain_schema	sql_identifier	If the column has a domain type, the name of the schema that the domain is defined in, else null.
domain_name	sql_identifier	If the column has a domain type, the name of the domain, else null.
udt_catalog	sql_identifier	Name of the database that the column data type (the underlying type of the domain, if applicable) is defined in (always the current database)
udt_schema	sql_identifier	Name of the schema that the column data type (the underlying type of the domain, if applicable) is defined in

Name	Data Type	Description
udt_name	sql_identifier	Name of the column data type (the underlying type of the domain, if applicable)
scope_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
scope_schema	sql_identifier	Applies to a feature not available in PostgreSQL
scope_name	sql_identifier	Applies to a feature not available in PostgreSQL
maximum_cardinality	cardinal_number	Always null, because arrays always have unlimited maximum cardinality in PostgreSQL
dtd_identifier	sql_identifier	An identifier of the data type descriptor of the column, unique among the data type descriptors pertaining to the table. This is mainly useful for joining with other instances of such identifiers. (The specific format of the identifier is not defined and not guaranteed to remain the same in future versions.)
is_self_referencing	yes_or_no	Applies to a feature not available in PostgreSQL
is_identity	yes_or_no	Applies to a feature not available in PostgreSQL
identity_generation	character_data	Applies to a feature not available in PostgreSQL
identity_start	character_data	Applies to a feature not available in PostgreSQL
identity_increment	character_data	Applies to a feature not available in PostgreSQL
identity_maximum	character_data	Applies to a feature not available in PostgreSQL
identity_minimum	character_data	Applies to a feature not available in PostgreSQL
identity_cycle	yes_or_no	Applies to a feature not available in PostgreSQL
is_generated	character_data	Applies to a feature not available in PostgreSQL
generation_expression	character_data	Applies to a feature not available in PostgreSQL

Name	Data Type	Description
is_updatable	yes_or_no	YES if the column is updatable, NO if not (Columns in base tables are always updatable, columns in views not necessarily)

Since data types can be defined in a variety of ways in SQL, and PostgreSQL contains additional ways to define data types, their representation in the information schema can be somewhat difficult. The column data_type is supposed to identify the underlying built-in type of the column. In PostgreSQL, this means that the type is defined in the system catalog schema pg_catalog. This column might be useful if the application can handle the well-known built-in types specially (for example, format the numeric types differently or use the data in the precision columns). The columns udt_name, udt_schema, and udt_catalog always identify the underlying data type of the column, even if the column is based on a domain. (Since PostgreSQL treats built-in types like user-defined types, built-in types appear here as well. This is an extension of the SQL standard.) These columns should be used if an application wants to process data differently according to the type, because in that case it wouldn't matter if the column is really based on a domain. If the column is based on a domain, the identity of the domain is stored in the columns domain_name, domain_schema, and domain_catalog. If you want to pair up columns with their associated data types and treat domains as separate types, you could write coalesce(domain_name, udt_name), etc.

34.17. constraint_column_usage

The view constraint_column_usage identifies all columns in the current database that are used by some constraint. Only those columns are shown that are contained in a table owned by a currently enabled role. For a check constraint, this view identifies the columns that are used in the check expression. For a foreign key constraint, this view identifies the columns that the foreign key references. For a unique or primary key constraint, this view identifies the constrained columns.

Table 34-15. constraint_column_usage Columns

Name	Data Type	Description
table_catalog	sql_identifier	Name of the database that contains the table that contains the column that is used by some constraint (always the current database)
table_schema	sql_identifier	Name of the schema that contains the table that contains the column that is used by some constraint
table_name	sql_identifier	Name of the table that contains the column that is used by some constraint

Name	Data Type	Description
column_name	sql_identifier	Name of the column that is used by some constraint
constraint_catalog	sql_identifier	Name of the database that contains the constraint (always the current database)
constraint_schema	sql_identifier	Name of the schema that contains the constraint
constraint_name	sql_identifier	Name of the constraint

34.18. constraint_table_usage

The view constraint_table_usage identifies all tables in the current database that are used by some constraint and are owned by a currently enabled role. (This is different from the view table_constraints, which identifies all table constraints along with the table they are defined on.) For a foreign key constraint, this view identifies the table that the foreign key references. For a unique or primary key constraint, this view simply identifies the table the constraint belongs to. Check constraints and not-null constraints are not included in this view.

Table 34-16. constraint_table_usage Columns

Name	Data Type	Description
table_catalog	sql_identifier	Name of the database that contains the table that is used by some constraint (always the current database)
table_schema	sql_identifier	Name of the schema that contains the table that is used by some constraint
table_name	sql_identifier	Name of the table that is used by some constraint
constraint_catalog	sql_identifier	Name of the database that contains the constraint (always the current database)
constraint_schema	sql_identifier	Name of the schema that contains the constraint
constraint_name	sql_identifier	Name of the constraint

34.19. data_type_privileges

The view data_type_privileges identifies all data type descriptors that the current user has access to, by way of being the owner of the described object or having some privilege for it. A data type descriptor

is generated whenever a data type is used in the definition of a table column, a domain, or a function (as parameter or return type) and stores some information about how the data type is used in that instance (for example, the declared maximum length, if applicable). Each data type descriptor is assigned an arbitrary identifier that is unique among the data type descriptor identifiers assigned for one object (table, domain, function). This view is probably not useful for applications, but it is used to define some other views in the information schema.

Table 34-17. `data_type_privileges` Columns

Name	Data Type	Description
`object_catalog`	`sql_identifier`	Name of the database that contains the described object (always the current database)
`object_schema`	`sql_identifier`	Name of the schema that contains the described object
`object_name`	`sql_identifier`	Name of the described object
`object_type`	`character_data`	The type of the described object: one of `TABLE` (the data type descriptor pertains to a column of that table), `DOMAIN` (the data type descriptors pertains to that domain), `ROUTINE` (the data type descriptor pertains to a parameter or the return data type of that function).
`dtd_identifier`	`sql_identifier`	The identifier of the data type descriptor, which is unique among the data type descriptors for that same object.

34.20. `domain_constraints`

The view `domain_constraints` contains all constraints belonging to domains defined in the current database. Only those domains are shown that the current user has access to (by way of being the owner or having some privilege).

Table 34-18. `domain_constraints` Columns

Name	Data Type	Description
`constraint_catalog`	`sql_identifier`	Name of the database that contains the constraint (always the current database)
`constraint_schema`	`sql_identifier`	Name of the schema that contains the constraint

Name	Data Type	Description
constraint_name	sql_identifier	Name of the constraint
domain_catalog	sql_identifier	Name of the database that contains the domain (always the current database)
domain_schema	sql_identifier	Name of the schema that contains the domain
domain_name	sql_identifier	Name of the domain
is_deferrable	yes_or_no	YES if the constraint is deferrable, NO if not
initially_deferred	yes_or_no	YES if the constraint is deferrable and initially deferred, NO if not

34.21. domain_udt_usage

The view domain_udt_usage identifies all domains that are based on data types owned by a currently enabled role. Note that in PostgreSQL, built-in data types behave like user-defined types, so they are included here as well.

Table 34-19. domain_udt_usage Columns

Name	Data Type	Description
udt_catalog	sql_identifier	Name of the database that the domain data type is defined in (always the current database)
udt_schema	sql_identifier	Name of the schema that the domain data type is defined in
udt_name	sql_identifier	Name of the domain data type
domain_catalog	sql_identifier	Name of the database that contains the domain (always the current database)
domain_schema	sql_identifier	Name of the schema that contains the domain
domain_name	sql_identifier	Name of the domain

34.22. domains

The view domains contains all domains defined in the current database. Only those domains are shown that the current user has access to (by way of being the owner or having some privilege).

Table 34-20. `domains` Columns

Name	Data Type	Description
`domain_catalog`	`sql_identifier`	Name of the database that contains the domain (always the current database)
`domain_schema`	`sql_identifier`	Name of the schema that contains the domain
`domain_name`	`sql_identifier`	Name of the domain
`data_type`	`character_data`	Data type of the domain, if it is a built-in type, or `ARRAY` if it is some array (in that case, see the view `element_types`), else `USER-DEFINED` (in that case, the type is identified in `udt_name` and associated columns).
`character_maximum_length`	`cardinal_number`	If the domain has a character or bit string type, the declared maximum length; null for all other data types or if no maximum length was declared.
`character_octet_length`	`cardinal_number`	If the domain has a character type, the maximum possible length in octets (bytes) of a datum; null for all other data types. The maximum octet length depends on the declared character maximum length (see above) and the server encoding.
`character_set_catalog`	`sql_identifier`	Applies to a feature not available in PostgreSQL
`character_set_schema`	`sql_identifier`	Applies to a feature not available in PostgreSQL
`character_set_name`	`sql_identifier`	Applies to a feature not available in PostgreSQL
`collation_catalog`	`sql_identifier`	Name of the database containing the collation of the domain (always the current database), null if default or the data type of the domain is not collatable
`collation_schema`	`sql_identifier`	Name of the schema containing the collation of the domain, null if default or the data type of the domain is not collatable

Name	Data Type	Description
collation_name	sql_identifier	Name of the collation of the domain, null if default or the data type of the domain is not collatable
numeric_precision	cardinal_number	If the domain has a numeric type, this column contains the (declared or implicit) precision of the type for this domain. The precision indicates the number of significant digits. It can be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix. For all other data types, this column is null.
numeric_precision_radix	cardinal_number	If the domain has a numeric type, this column indicates in which base the values in the columns numeric_precision and numeric_scale are expressed. The value is either 2 or 10. For all other data types, this column is null.
numeric_scale	cardinal_number	If the domain has an exact numeric type, this column contains the (declared or implicit) scale of the type for this domain. The scale indicates the number of significant digits to the right of the decimal point. It can be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix. For all other data types, this column is null.

Name	Data Type	Description
datetime_precision	cardinal_number	If data_type identifies a date, time, timestamp, or interval type, this column contains the (declared or implicit) fractional seconds precision of the type for this domain, that is, the number of decimal digits maintained following the decimal point in the seconds value. For all other data types, this column is null.
interval_type	character_data	If data_type identifies an interval type, this column contains the specification which fields the intervals include for this domain, e.g., YEAR TO MONTH, DAY TO SECOND, etc. If no field restrictions were specified (that is, the interval accepts all fields), and for all other data types, this field is null.
interval_precision	cardinal_number	Applies to a feature not available in PostgreSQL (see datetime_precision for the fractional seconds precision of interval type domains)
domain_default	character_data	Default expression of the domain
udt_catalog	sql_identifier	Name of the database that the domain data type is defined in (always the current database)
udt_schema	sql_identifier	Name of the schema that the domain data type is defined in
udt_name	sql_identifier	Name of the domain data type
scope_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
scope_schema	sql_identifier	Applies to a feature not available in PostgreSQL
scope_name	sql_identifier	Applies to a feature not available in PostgreSQL
maximum_cardinality	cardinal_number	Always null, because arrays always have unlimited maximum cardinality in PostgreSQL

Name	Data Type	Description
dtd_identifier	sql_identifier	An identifier of the data type descriptor of the domain, unique among the data type descriptors pertaining to the domain (which is trivial, because a domain only contains one data type descriptor). This is mainly useful for joining with other instances of such identifiers. (The specific format of the identifier is not defined and not guaranteed to remain the same in future versions.)

34.23. element_types

The view element_types contains the data type descriptors of the elements of arrays. When a table column, composite-type attribute, domain, function parameter, or function return value is defined to be of an array type, the respective information schema view only contains ARRAY in the column data_type. To obtain information on the element type of the array, you can join the respective view with this view. For example, to show the columns of a table with data types and array element types, if applicable, you could do:

```
SELECT c.column_name, c.data_type, e.data_type AS element_type
FROM information_schema.columns c LEFT JOIN information_schema.element_types e
     ON ((c.table_catalog, c.table_schema, c.table_name, 'TABLE', c.dtd_identifier)
       = (e.object_catalog, e.object_schema, e.object_name, e.object_type, e.collection_typ
WHERE c.table_schema = '...' AND c.table_name = '...'
ORDER BY c.ordinal_position;
```

This view only includes objects that the current user has access to, by way of being the owner or having some privilege.

Table 34-21. element_types Columns

Name	Data Type	Description
object_catalog	sql_identifier	Name of the database that contains the object that uses the array being described (always the current database)
object_schema	sql_identifier	Name of the schema that contains the object that uses the array being described
object_name	sql_identifier	Name of the object that uses the array being described

Name	Data Type	Description
object_type	character_data	The type of the object that uses the array being described: one of TABLE (the array is used by a column of that table), USER-DEFINED TYPE (the array is used by an attribute of that composite type), DOMAIN (the array is used by that domain), ROUTINE (the array is used by a parameter or the return data type of that function).
collection_type_identifier	sql_identifier	The identifier of the data type descriptor of the array being described. Use this to join with the dtd_identifier columns of other information schema views.
data_type	character_data	Data type of the array elements, if it is a built-in type, else USER-DEFINED (in that case, the type is identified in udt_name and associated columns).
character_maximum_length	cardinal_number	Always null, since this information is not applied to array element data types in PostgreSQL
character_octet_length	cardinal_number	Always null, since this information is not applied to array element data types in PostgreSQL
character_set_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_schema	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_name	sql_identifier	Applies to a feature not available in PostgreSQL
collation_catalog	sql_identifier	Name of the database containing the collation of the element type (always the current database), null if default or the data type of the element is not collatable

Name	Data Type	Description
collation_schema	sql_identifier	Name of the schema containing the collation of the element type, null if default or the data type of the element is not collatable
collation_name	sql_identifier	Name of the collation of the element type, null if default or the data type of the element is not collatable
numeric_precision	cardinal_number	Always null, since this information is not applied to array element data types in PostgreSQL
numeric_precision_radix	cardinal_number	Always null, since this information is not applied to array element data types in PostgreSQL
numeric_scale	cardinal_number	Always null, since this information is not applied to array element data types in PostgreSQL
datetime_precision	cardinal_number	Always null, since this information is not applied to array element data types in PostgreSQL
interval_type	character_data	Always null, since this information is not applied to array element data types in PostgreSQL
interval_precision	cardinal_number	Always null, since this information is not applied to array element data types in PostgreSQL
domain_default	character_data	Not yet implemented
udt_catalog	sql_identifier	Name of the database that the data type of the elements is defined in (always the current database)
udt_schema	sql_identifier	Name of the schema that the data type of the elements is defined in
udt_name	sql_identifier	Name of the data type of the elements
scope_catalog	sql_identifier	Applies to a feature not available in PostgreSQL

Name	Data Type	Description
scope_schema	sql_identifier	Applies to a feature not available in PostgreSQL
scope_name	sql_identifier	Applies to a feature not available in PostgreSQL
maximum_cardinality	cardinal_number	Always null, because arrays always have unlimited maximum cardinality in PostgreSQL
dtd_identifier	sql_identifier	An identifier of the data type descriptor of the element. This is currently not useful.

34.24. enabled_roles

The view enabled_roles identifies the currently "enabled roles". The enabled roles are recursively defined as the current user together with all roles that have been granted to the enabled roles with automatic inheritance. In other words, these are all roles that the current user has direct or indirect, automatically inheriting membership in.

For permission checking, the set of "applicable roles" is applied, which can be broader than the set of enabled roles. So generally, it is better to use the view applicable_roles instead of this one; See Section 34.5 for details on applicable_roles view.

Table 34-22. enabled_roles Columns

Name	Data Type	Description
role_name	sql_identifier	Name of a role

34.25. foreign_data_wrapper_options

The view foreign_data_wrapper_options contains all the options defined for foreign-data wrappers in the current database. Only those foreign-data wrappers are shown that the current user has access to (by way of being the owner or having some privilege).

Table 34-23. foreign_data_wrapper_options Columns

Name	Data Type	Description
foreign_data_wrapper_catalog	sql_identifier	Name of the database that the foreign-data wrapper is defined in (always the current database)
foreign_data_wrapper_name	sql_identifier	Name of the foreign-data wrapper

Name	Data Type	Description
option_name	sql_identifier	Name of an option
option_value	character_data	Value of the option

34.26. foreign_data_wrappers

The view foreign_data_wrappers contains all foreign-data wrappers defined in the current database. Only those foreign-data wrappers are shown that the current user has access to (by way of being the owner or having some privilege).

Table 34-24. foreign_data_wrappers Columns

Name	Data Type	Description
foreign_data_wrapper_catalog	sql_identifier	Name of the database that contains the foreign-data wrapper (always the current database)
foreign_data_wrapper_name	sql_identifier	Name of the foreign-data wrapper
authorization_identifier	sql_identifier	Name of the owner of the foreign server
library_name	character_data	File name of the library that implementing this foreign-data wrapper
foreign_data_wrapper_language	character_data	Language used to implement this foreign-data wrapper

34.27. foreign_server_options

The view foreign_server_options contains all the options defined for foreign servers in the current database. Only those foreign servers are shown that the current user has access to (by way of being the owner or having some privilege).

Table 34-25. foreign_server_options Columns

Name	Data Type	Description
foreign_server_catalog	sql_identifier	Name of the database that the foreign server is defined in (always the current database)
foreign_server_name	sql_identifier	Name of the foreign server
option_name	sql_identifier	Name of an option

Name	Data Type	Description
option_value	character_data	Value of the option

34.28. foreign_servers

The view foreign_servers contains all foreign servers defined in the current database. Only those foreign servers are shown that the current user has access to (by way of being the owner or having some privilege).

Table 34-26. foreign_servers Columns

Name	Data Type	Description
foreign_server_catalog	sql_identifier	Name of the database that the foreign server is defined in (always the current database)
foreign_server_name	sql_identifier	Name of the foreign server
foreign_data_wrapper_catalog	sql_identifier	Name of the database that contains the foreign-data wrapper used by the foreign server (always the current database)
foreign_data_wrapper_name	sql_identifier	Name of the foreign-data wrapper used by the foreign server
foreign_server_type	character_data	Foreign server type information, if specified upon creation
foreign_server_version	character_data	Foreign server version information, if specified upon creation
authorization_identifier	sql_identifier	Name of the owner of the foreign server

34.29. foreign_table_options

The view foreign_table_options contains all the options defined for foreign tables in the current database. Only those foreign tables are shown that the current user has access to (by way of being the owner or having some privilege).

Table 34-27. foreign_table_options Columns

Name	Data Type	Description

Name	Data Type	Description
foreign_table_catalog	sql_identifier	Name of the database that contains the foreign table (always the current database)
foreign_table_schema	sql_identifier	Name of the schema that contains the foreign table
foreign_table_name	sql_identifier	Name of the foreign table
option_name	sql_identifier	Name of an option
option_value	character_data	Value of the option

34.30. foreign_tables

The view foreign_tables contains all foreign tables defined in the current database. Only those foreign tables are shown that the current user has access to (by way of being the owner or having some privilege).

Table 34-28. foreign_tables Columns

Name	Data Type	Description
foreign_table_catalog	sql_identifier	Name of the database that the foreign table is defined in (always the current database)
foreign_table_schema	sql_identifier	Name of the schema that contains the foreign table
foreign_table_name	sql_identifier	Name of the foreign table
foreign_server_catalog	sql_identifier	Name of the database that the foreign server is defined in (always the current database)
foreign_server_name	sql_identifier	Name of the foreign server

34.31. key_column_usage

The view key_column_usage identifies all columns in the current database that are restricted by some unique, primary key, or foreign key constraint. Check constraints are not included in this view. Only those columns are shown that the current user has access to, by way of being the owner or having some privilege.

Table 34-29. key_column_usage Columns

Name	Data Type	Description
constraint_catalog	sql_identifier	Name of the database that contains the constraint (always the current database)

Name	Data Type	Description
constraint_schema	sql_identifier	Name of the schema that contains the constraint
constraint_name	sql_identifier	Name of the constraint
table_catalog	sql_identifier	Name of the database that contains the table that contains the column that is restricted by this constraint (always the current database)
table_schema	sql_identifier	Name of the schema that contains the table that contains the column that is restricted by this constraint
table_name	sql_identifier	Name of the table that contains the column that is restricted by this constraint
column_name	sql_identifier	Name of the column that is restricted by this constraint
ordinal_position	cardinal_number	Ordinal position of the column within the constraint key (count starts at 1)
position_in_unique_constraint	cardinal_number	For a foreign-key constraint, ordinal position of the referenced column within its unique constraint (count starts at 1); otherwise null

34.32. parameters

The view parameters contains information about the parameters (arguments) of all functions in the current database. Only those functions are shown that the current user has access to (by way of being the owner or having some privilege).

Table 34-30. parameters Columns

Name	Data Type	Description
specific_catalog	sql_identifier	Name of the database containing the function (always the current database)
specific_schema	sql_identifier	Name of the schema containing the function

Name	Data Type	Description
specific_name	sql_identifier	The "specific name" of the function. See Section 34.40 for more information.
ordinal_position	cardinal_number	Ordinal position of the parameter in the argument list of the function (count starts at 1)
parameter_mode	character_data	IN for input parameter, OUT for output parameter, and INOUT for input/output parameter.
is_result	yes_or_no	Applies to a feature not available in PostgreSQL
as_locator	yes_or_no	Applies to a feature not available in PostgreSQL
parameter_name	sql_identifier	Name of the parameter, or null if the parameter has no name
data_type	character_data	Data type of the parameter, if it is a built-in type, or ARRAY if it is some array (in that case, see the view element_types), else USER-DEFINED (in that case, the type is identified in udt_name and associated columns).
character_maximum_length	cardinal_number	Always null, since this information is not applied to parameter data types in PostgreSQL
character_octet_length	cardinal_number	Always null, since this information is not applied to parameter data types in PostgreSQL
character_set_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_schema	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_name	sql_identifier	Applies to a feature not available in PostgreSQL
collation_catalog	sql_identifier	Always null, since this information is not applied to parameter data types in PostgreSQL

Name	Data Type	Description
collation_schema	sql_identifier	Always null, since this information is not applied to parameter data types in PostgreSQL
collation_name	sql_identifier	Always null, since this information is not applied to parameter data types in PostgreSQL
numeric_precision	cardinal_number	Always null, since this information is not applied to parameter data types in PostgreSQL
numeric_precision_radix	cardinal_number	Always null, since this information is not applied to parameter data types in PostgreSQL
numeric_scale	cardinal_number	Always null, since this information is not applied to parameter data types in PostgreSQL
datetime_precision	cardinal_number	Always null, since this information is not applied to parameter data types in PostgreSQL
interval_type	character_data	Always null, since this information is not applied to parameter data types in PostgreSQL
interval_precision	cardinal_number	Always null, since this information is not applied to parameter data types in PostgreSQL
udt_catalog	sql_identifier	Name of the database that the data type of the parameter is defined in (always the current database)
udt_schema	sql_identifier	Name of the schema that the data type of the parameter is defined in
udt_name	sql_identifier	Name of the data type of the parameter
scope_catalog	sql_identifier	Applies to a feature not available in PostgreSQL

Name	Data Type	Description
scope_schema	sql_identifier	Applies to a feature not available in PostgreSQL
scope_name	sql_identifier	Applies to a feature not available in PostgreSQL
maximum_cardinality	cardinal_number	Always null, because arrays always have unlimited maximum cardinality in PostgreSQL
dtd_identifier	sql_identifier	An identifier of the data type descriptor of the parameter, unique among the data type descriptors pertaining to the function. This is mainly useful for joining with other instances of such identifiers. (The specific format of the identifier is not defined and not guaranteed to remain the same in future versions.)
parameter_default	character_data	The default expression of the parameter, or null if none or if the function is not owned by a currently enabled role.

34.33. referential_constraints

The view referential_constraints contains all referential (foreign key) constraints in the current database. Only those constraints are shown for which the current user has write access to the referencing table (by way of being the owner or having some privilege other than SELECT).

Table 34-31. referential_constraints Columns

Name	Data Type	Description
constraint_catalog	sql_identifier	Name of the database containing the constraint (always the current database)
constraint_schema	sql_identifier	Name of the schema containing the constraint
constraint_name	sql_identifier	Name of the constraint
unique_constraint_catalog	sql_identifier	Name of the database that contains the unique or primary key constraint that the foreign key constraint references (always the current database)

Name	Data Type	Description
unique_constraint_schema	sql_identifier	Name of the schema that contains the unique or primary key constraint that the foreign key constraint references
unique_constraint_name	sql_identifier	Name of the unique or primary key constraint that the foreign key constraint references
match_option	character_data	Match option of the foreign key constraint: FULL, PARTIAL, or NONE.
update_rule	character_data	Update rule of the foreign key constraint: CASCADE, SET NULL, SET DEFAULT, RESTRICT, or NO ACTION.
delete_rule	character_data	Delete rule of the foreign key constraint: CASCADE, SET NULL, SET DEFAULT, RESTRICT, or NO ACTION.

34.34. role_column_grants

The view role_column_grants identifies all privileges granted on columns where the grantor or grantee is a currently enabled role. Further information can be found under column_privileges. The only effective difference between this view and column_privileges is that this view omits columns that have been made accessible to the current user by way of a grant to PUBLIC.

Table 34-32. role_column_grants Columns

Name	Data Type	Description
grantor	sql_identifier	Name of the role that granted the privilege
grantee	sql_identifier	Name of the role that the privilege was granted to
table_catalog	sql_identifier	Name of the database that contains the table that contains the column (always the current database)
table_schema	sql_identifier	Name of the schema that contains the table that contains the column
table_name	sql_identifier	Name of the table that contains the column

Name	Data Type	Description
column_name	sql_identifier	Name of the column
privilege_type	character_data	Type of the privilege: SELECT, INSERT, UPDATE, or REFERENCES
is_grantable	yes_or_no	YES if the privilege is grantable, NO if not

34.35. role_routine_grants

The view role_routine_grants identifies all privileges granted on functions where the grantor or grantee is a currently enabled role. Further information can be found under routine_privileges. The only effective difference between this view and routine_privileges is that this view omits functions that have been made accessible to the current user by way of a grant to PUBLIC.

Table 34-33. role_routine_grants Columns

Name	Data Type	Description
grantor	sql_identifier	Name of the role that granted the privilege
grantee	sql_identifier	Name of the role that the privilege was granted to
specific_catalog	sql_identifier	Name of the database containing the function (always the current database)
specific_schema	sql_identifier	Name of the schema containing the function
specific_name	sql_identifier	The "specific name" of the function. See Section 34.40 for more information.
routine_catalog	sql_identifier	Name of the database containing the function (always the current database)
routine_schema	sql_identifier	Name of the schema containing the function
routine_name	sql_identifier	Name of the function (might be duplicated in case of overloading)
privilege_type	character_data	Always EXECUTE (the only privilege type for functions)
is_grantable	yes_or_no	YES if the privilege is grantable, NO if not

34.36. role_table_grants

The view role_table_grants identifies all privileges granted on tables or views where the grantor or grantee is a currently enabled role. Further information can be found under table_privileges. The only effective difference between this view and table_privileges is that this view omits tables that have been made accessible to the current user by way of a grant to PUBLIC.

Table 34-34. role_table_grants Columns

Name	Data Type	Description
grantor	sql_identifier	Name of the role that granted the privilege
grantee	sql_identifier	Name of the role that the privilege was granted to
table_catalog	sql_identifier	Name of the database that contains the table (always the current database)
table_schema	sql_identifier	Name of the schema that contains the table
table_name	sql_identifier	Name of the table
privilege_type	character_data	Type of the privilege: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, or TRIGGER
is_grantable	yes_or_no	YES if the privilege is grantable, NO if not
with_hierarchy	yes_or_no	In the SQL standard, WITH HIERARCHY OPTION is a separate (sub-)privilege allowing certain operations on table inheritance hierarchies. In PostgreSQL, this is included in the SELECT privilege, so this column shows YES if the privilege is SELECT, else NO.

34.37. role_udt_grants

The view role_udt_grants is intended to identify USAGE privileges granted on user-defined types where the grantor or grantee is a currently enabled role. Further information can be found under udt_privileges. The only effective difference between this view and udt_privileges is that this view omits objects that have been made accessible to the current user by way of a grant to PUBLIC. Since data types do not have real privileges in PostgreSQL, but only an implicit grant to PUBLIC, this view is empty.

Table 34-35. `role_udt_grants` Columns

Name	Data Type	Description
`grantor`	`sql_identifier`	The name of the role that granted the privilege
`grantee`	`sql_identifier`	The name of the role that the privilege was granted to
`udt_catalog`	`sql_identifier`	Name of the database containing the type (always the current database)
`udt_schema`	`sql_identifier`	Name of the schema containing the type
`udt_name`	`sql_identifier`	Name of the type
`privilege_type`	`character_data`	Always `TYPE USAGE`
`is_grantable`	`yes_or_no`	`YES` if the privilege is grantable, `NO` if not

34.38. `role_usage_grants`

The view `role_usage_grants` identifies `USAGE` privileges granted on various kinds of objects where the grantor or grantee is a currently enabled role. Further information can be found under `usage_privileges`. The only effective difference between this view and `usage_privileges` is that this view omits objects that have been made accessible to the current user by way of a grant to `PUBLIC`.

Table 34-36. `role_usage_grants` Columns

Name	Data Type	Description
`grantor`	`sql_identifier`	The name of the role that granted the privilege
`grantee`	`sql_identifier`	The name of the role that the privilege was granted to
`object_catalog`	`sql_identifier`	Name of the database containing the object (always the current database)
`object_schema`	`sql_identifier`	Name of the schema containing the object, if applicable, else an empty string
`object_name`	`sql_identifier`	Name of the object
`object_type`	`character_data`	`COLLATION` or `DOMAIN` or `FOREIGN DATA WRAPPER` or `FOREIGN SERVER` or `SEQUENCE`
`privilege_type`	`character_data`	Always `USAGE`

Name	Data Type	Description
is_grantable	yes_or_no	YES if the privilege is grantable, NO if not

34.39. routine_privileges

The view routine_privileges identifies all privileges granted on functions to a currently enabled role or by a currently enabled role. There is one row for each combination of function, grantor, and grantee.

Table 34-37. routine_privileges Columns

Name	Data Type	Description
grantor	sql_identifier	Name of the role that granted the privilege
grantee	sql_identifier	Name of the role that the privilege was granted to
specific_catalog	sql_identifier	Name of the database containing the function (always the current database)
specific_schema	sql_identifier	Name of the schema containing the function
specific_name	sql_identifier	The "specific name" of the function. See Section 34.40 for more information.
routine_catalog	sql_identifier	Name of the database containing the function (always the current database)
routine_schema	sql_identifier	Name of the schema containing the function
routine_name	sql_identifier	Name of the function (might be duplicated in case of overloading)
privilege_type	character_data	Always EXECUTE (the only privilege type for functions)
is_grantable	yes_or_no	YES if the privilege is grantable, NO if not

34.40. routines

The view routines contains all functions in the current database. Only those functions are shown that the current user has access to (by way of being the owner or having some privilege).

Table 34-38. `routines` Columns

Name	Data Type	Description
`specific_catalog`	`sql_identifier`	Name of the database containing the function (always the current database)
`specific_schema`	`sql_identifier`	Name of the schema containing the function
`specific_name`	`sql_identifier`	The "specific name" of the function. This is a name that uniquely identifies the function in the schema, even if the real name of the function is overloaded. The format of the specific name is not defined, it should only be used to compare it to other instances of specific routine names.
`routine_catalog`	`sql_identifier`	Name of the database containing the function (always the current database)
`routine_schema`	`sql_identifier`	Name of the schema containing the function
`routine_name`	`sql_identifier`	Name of the function (might be duplicated in case of overloading)
`routine_type`	`character_data`	Always `FUNCTION` (In the future there might be other types of routines.)
`module_catalog`	`sql_identifier`	Applies to a feature not available in PostgreSQL
`module_schema`	`sql_identifier`	Applies to a feature not available in PostgreSQL
`module_name`	`sql_identifier`	Applies to a feature not available in PostgreSQL
`udt_catalog`	`sql_identifier`	Applies to a feature not available in PostgreSQL
`udt_schema`	`sql_identifier`	Applies to a feature not available in PostgreSQL
`udt_name`	`sql_identifier`	Applies to a feature not available in PostgreSQL

Name	Data Type	Description
data_type	character_data	Return data type of the function, if it is a built-in type, or ARRAY if it is some array (in that case, see the view element_types), else USER-DEFINED (in that case, the type is identified in type_udt_name and associated columns).
character_maximum_length	cardinal_number	Always null, since this information is not applied to return data types in PostgreSQL
character_octet_length	cardinal_number	Always null, since this information is not applied to return data types in PostgreSQL
character_set_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_schema	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_name	sql_identifier	Applies to a feature not available in PostgreSQL
collation_catalog	sql_identifier	Always null, since this information is not applied to return data types in PostgreSQL
collation_schema	sql_identifier	Always null, since this information is not applied to return data types in PostgreSQL
collation_name	sql_identifier	Always null, since this information is not applied to return data types in PostgreSQL
numeric_precision	cardinal_number	Always null, since this information is not applied to return data types in PostgreSQL
numeric_precision_radix	cardinal_number	Always null, since this information is not applied to return data types in PostgreSQL
numeric_scale	cardinal_number	Always null, since this information is not applied to return data types in PostgreSQL
datetime_precision	cardinal_number	Always null, since this information is not applied to return data types in PostgreSQL

Name	Data Type	Description
interval_type	character_data	Always null, since this information is not applied to return data types in PostgreSQL
interval_precision	cardinal_number	Always null, since this information is not applied to return data types in PostgreSQL
type_udt_catalog	sql_identifier	Name of the database that the return data type of the function is defined in (always the current database)
type_udt_schema	sql_identifier	Name of the schema that the return data type of the function is defined in
type_udt_name	sql_identifier	Name of the return data type of the function
scope_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
scope_schema	sql_identifier	Applies to a feature not available in PostgreSQL
scope_name	sql_identifier	Applies to a feature not available in PostgreSQL
maximum_cardinality	cardinal_number	Always null, because arrays always have unlimited maximum cardinality in PostgreSQL
dtd_identifier	sql_identifier	An identifier of the data type descriptor of the return data type of this function, unique among the data type descriptors pertaining to the function. This is mainly useful for joining with other instances of such identifiers. (The specific format of the identifier is not defined and not guaranteed to remain the same in future versions.)
routine_body	character_data	If the function is an SQL function, then SQL, else EXTERNAL.

Name	Data Type	Description
routine_definition	character_data	The source text of the function (null if the function is not owned by a currently enabled role). (According to the SQL standard, this column is only applicable if routine_body is SQL, but in PostgreSQL it will contain whatever source text was specified when the function was created.)
external_name	character_data	If this function is a C function, then the external name (link symbol) of the function; else null. (This works out to be the same value that is shown in routine_definition.)
external_language	character_data	The language the function is written in
parameter_style	character_data	Always GENERAL (The SQL standard defines other parameter styles, which are not available in PostgreSQL.)
is_deterministic	yes_or_no	If the function is declared immutable (called deterministic in the SQL standard), then YES, else NO. (You cannot query the other volatility levels available in PostgreSQL through the information schema.)
sql_data_access	character_data	Always MODIFIES, meaning that the function possibly modifies SQL data. This information is not useful for PostgreSQL.
is_null_call	yes_or_no	If the function automatically returns null if any of its arguments are null, then YES, else NO.
sql_path	character_data	Applies to a feature not available in PostgreSQL

Name	Data Type	Description
schema_level_routine	yes_or_no	Always YES (The opposite would be a method of a user-defined type, which is a feature not available in PostgreSQL.)
max_dynamic_result_sets	cardinal_number	Applies to a feature not available in PostgreSQL
is_user_defined_cast	yes_or_no	Applies to a feature not available in PostgreSQL
is_implicitly_invocable	yes_or_no	Applies to a feature not available in PostgreSQL
security_type	character_data	If the function runs with the privileges of the current user, then INVOKER, if the function runs with the privileges of the user who defined it, then DEFINER.
to_sql_specific_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
to_sql_specific_schema	sql_identifier	Applies to a feature not available in PostgreSQL
to_sql_specific_name	sql_identifier	Applies to a feature not available in PostgreSQL
as_locator	yes_or_no	Applies to a feature not available in PostgreSQL
created	time_stamp	Applies to a feature not available in PostgreSQL
last_altered	time_stamp	Applies to a feature not available in PostgreSQL
new_savepoint_level	yes_or_no	Applies to a feature not available in PostgreSQL
is_udt_dependent	yes_or_no	Currently always NO. The alternative YES applies to a feature not available in PostgreSQL.
result_cast_from_data_type	character_data	Applies to a feature not available in PostgreSQL
result_cast_as_locator	yes_or_no	Applies to a feature not available in PostgreSQL
result_cast_char_max_length	cardinal_number	Applies to a feature not available in PostgreSQL
result_cast_char_octet_length	character_data	Applies to a feature not available in PostgreSQL

Name	Data Type	Description
result_cast_char_set_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_char_set_schema	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_char_set_name	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_collation_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_collation_schema	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_collation_name	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_numeric_precision	cardinal_number	Applies to a feature not available in PostgreSQL
result_cast_numeric_precision_radix	cardinal_number	Applies to a feature not available in PostgreSQL
result_cast_numeric_scale	cardinal_number	Applies to a feature not available in PostgreSQL
result_cast_datetime_precision	character_data	Applies to a feature not available in PostgreSQL
result_cast_interval_type	character_data	Applies to a feature not available in PostgreSQL
result_cast_interval_precision	cardinal_number	Applies to a feature not available in PostgreSQL
result_cast_type_udt_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_type_udt_schema	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_type_udt_name	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_scope_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_scope_schema	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_scope_name	sql_identifier	Applies to a feature not available in PostgreSQL
result_cast_maximum_cardinality	cardinal_number	Applies to a feature not available in PostgreSQL
result_cast_dtd_identifier	sql_identifier	Applies to a feature not available in PostgreSQL

34.41. schemata

The view schemata contains all schemas in the current database that the current user has access to (by way of being the owner or having some privilege).

Table 34-39. schemata Columns

Name	Data Type	Description
catalog_name	sql_identifier	Name of the database that the schema is contained in (always the current database)
schema_name	sql_identifier	Name of the schema
schema_owner	sql_identifier	Name of the owner of the schema
default_character_set_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
default_character_set_schema	sql_identifier	Applies to a feature not available in PostgreSQL
default_character_set_name	sql_identifier	Applies to a feature not available in PostgreSQL
sql_path	character_data	Applies to a feature not available in PostgreSQL

34.42. sequences

The view sequences contains all sequences defined in the current database. Only those sequences are shown that the current user has access to (by way of being the owner or having some privilege).

Table 34-40. sequences Columns

Name	Data Type	Description
sequence_catalog	sql_identifier	Name of the database that contains the sequence (always the current database)
sequence_schema	sql_identifier	Name of the schema that contains the sequence
sequence_name	sql_identifier	Name of the sequence
data_type	character_data	The data type of the sequence. In PostgreSQL, this is currently always bigint.

Name	Data Type	Description
numeric_precision	cardinal_number	This column contains the (declared or implicit) precision of the sequence data type (see above). The precision indicates the number of significant digits. It can be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix.
numeric_precision_radix	cardinal_number	This column indicates in which base the values in the columns numeric_precision and numeric_scale are expressed. The value is either 2 or 10.
numeric_scale	cardinal_number	This column contains the (declared or implicit) scale of the sequence data type (see above). The scale indicates the number of significant digits to the right of the decimal point. It can be expressed in decimal (base 10) or binary (base 2) terms, as specified in the column numeric_precision_radix.
start_value	character_data	The start value of the sequence
minimum_value	character_data	The minimum value of the sequence
maximum_value	character_data	The maximum value of the sequence
increment	character_data	The increment of the sequence
cycle_option	yes_or_no	YES if the sequence cycles, else NO

Note that in accordance with the SQL standard, the start, minimum, maximum, and increment values are returned as character strings.

34.43. sql_features

The table sql_features contains information about which formal features defined in the SQL standard are supported by PostgreSQL. This is the same information that is presented in Appendix D. There you can also find some additional background information.

Table 34-41. sql_features Columns

Name	Data Type	Description
feature_id	character_data	Identifier string of the feature
feature_name	character_data	Descriptive name of the feature
sub_feature_id	character_data	Identifier string of the subfeature, or a zero-length string if not a subfeature
sub_feature_name	character_data	Descriptive name of the subfeature, or a zero-length string if not a subfeature
is_supported	yes_or_no	YES if the feature is fully supported by the current version of PostgreSQL, NO if not
is_verified_by	character_data	Always null, since the PostgreSQL development group does not perform formal testing of feature conformance
comments	character_data	Possibly a comment about the supported status of the feature

34.44. sql_implementation_info

The table sql_implementation_info contains information about various aspects that are left implementation-defined by the SQL standard. This information is primarily intended for use in the context of the ODBC interface; users of other interfaces will probably find this information to be of little use. For this reason, the individual implementation information items are not described here; you will find them in the description of the ODBC interface.

Table 34-42. sql_implementation_info Columns

Name	Data Type	Description
implementation_info_id	character_data	Identifier string of the implementation information item
implementation_info_name	character_data	Descriptive name of the implementation information item
integer_value	cardinal_number	Value of the implementation information item, or null if the value is contained in the column character_value

Name	Data Type	Description
character_value	character_data	Value of the implementation information item, or null if the value is contained in the column integer_value
comments	character_data	Possibly a comment pertaining to the implementation information item

34.45. sql_languages

The table sql_languages contains one row for each SQL language binding that is supported by PostgreSQL. PostgreSQL supports direct SQL and embedded SQL in C; that is all you will learn from this table.

This table was removed from the SQL standard in SQL:2008, so there are no entries referring to standards later than SQL:2003.

Table 34-43. sql_languages Columns

Name	Data Type	Description
sql_language_source	character_data	The name of the source of the language definition; always ISO 9075, that is, the SQL standard
sql_language_year	character_data	The year the standard referenced in sql_language_source was approved.
sql_language_conformance	character_data	The standard conformance level for the language binding. For ISO 9075:2003 this is always CORE.
sql_language_integrity	character_data	Always null (This value is relevant to an earlier version of the SQL standard.)
sql_language_implementation	character_data	Always null
sql_language_binding_style	character_data	The language binding style, either DIRECT or EMBEDDED
sql_language_programming_language	character_data	The programming language, if the binding style is EMBEDDED, else null. PostgreSQL only supports the language C.

34.46. sql_packages

The table sql_packages contains information about which feature packages defined in the SQL standard are supported by PostgreSQL. Refer to Appendix D for background information on feature packages.

Table 34-44. sql_packages Columns

Name	Data Type	Description
feature_id	character_data	Identifier string of the package
feature_name	character_data	Descriptive name of the package
is_supported	yes_or_no	YES if the package is fully supported by the current version of PostgreSQL, NO if not
is_verified_by	character_data	Always null, since the PostgreSQL development group does not perform formal testing of feature conformance
comments	character_data	Possibly a comment about the supported status of the package

34.47. sql_parts

The table sql_parts contains information about which of the several parts of the SQL standard are supported by PostgreSQL.

Table 34-45. sql_parts Columns

Name	Data Type	Description
feature_id	character_data	An identifier string containing the number of the part
feature_name	character_data	Descriptive name of the part
is_supported	yes_or_no	YES if the part is fully supported by the current version of PostgreSQL, NO if not
is_verified_by	character_data	Always null, since the PostgreSQL development group does not perform formal testing of feature conformance
comments	character_data	Possibly a comment about the supported status of the part

34.48. sql_sizing

The table sql_sizing contains information about various size limits and maximum values in PostgreSQL. This information is primarily intended for use in the context of the ODBC interface; users of other interfaces will probably find this information to be of little use. For this reason, the individual sizing items are not described here; you will find them in the description of the ODBC interface.

Table 34-46. sql_sizing Columns

Name	Data Type	Description
sizing_id	cardinal_number	Identifier of the sizing item
sizing_name	character_data	Descriptive name of the sizing item
supported_value	cardinal_number	Value of the sizing item, or 0 if the size is unlimited or cannot be determined, or null if the features for which the sizing item is applicable are not supported
comments	character_data	Possibly a comment pertaining to the sizing item

34.49. sql_sizing_profiles

The table sql_sizing_profiles contains information about the sql_sizing values that are required by various profiles of the SQL standard. PostgreSQL does not track any SQL profiles, so this table is empty.

Table 34-47. sql_sizing_profiles Columns

Name	Data Type	Description
sizing_id	cardinal_number	Identifier of the sizing item
sizing_name	character_data	Descriptive name of the sizing item
profile_id	character_data	Identifier string of a profile
required_value	cardinal_number	The value required by the SQL profile for the sizing item, or 0 if the profile places no limit on the sizing item, or null if the profile does not require any of the features for which the sizing item is applicable
comments	character_data	Possibly a comment pertaining to the sizing item within the profile

34.50. table_constraints

The view table_constraints contains all constraints belonging to tables that the current user owns or has some privilege other than SELECT on.

Table 34-48. table_constraints Columns

Name	Data Type	Description
constraint_catalog	sql_identifier	Name of the database that contains the constraint (always the current database)
constraint_schema	sql_identifier	Name of the schema that contains the constraint
constraint_name	sql_identifier	Name of the constraint
table_catalog	sql_identifier	Name of the database that contains the table (always the current database)
table_schema	sql_identifier	Name of the schema that contains the table
table_name	sql_identifier	Name of the table
constraint_type	character_data	Type of the constraint: CHECK, FOREIGN KEY, PRIMARY KEY, or UNIQUE
is_deferrable	yes_or_no	YES if the constraint is deferrable, NO if not
initially_deferred	yes_or_no	YES if the constraint is deferrable and initially deferred, NO if not

34.51. table_privileges

The view table_privileges identifies all privileges granted on tables or views to a currently enabled role or by a currently enabled role. There is one row for each combination of table, grantor, and grantee.

Table 34-49. table_privileges Columns

Name	Data Type	Description
grantor	sql_identifier	Name of the role that granted the privilege
grantee	sql_identifier	Name of the role that the privilege was granted to
table_catalog	sql_identifier	Name of the database that contains the table (always the current database)

Name	Data Type	Description
table_schema	sql_identifier	Name of the schema that contains the table
table_name	sql_identifier	Name of the table
privilege_type	character_data	Type of the privilege: SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, or TRIGGER
is_grantable	yes_or_no	YES if the privilege is grantable, NO if not
with_hierarchy	yes_or_no	In the SQL standard, WITH HIERARCHY OPTION is a separate (sub-)privilege allowing certain operations on table inheritance hierarchies. In PostgreSQL, this is included in the SELECT privilege, so this column shows YES if the privilege is SELECT, else NO.

34.52. tables

The view tables contains all tables and views defined in the current database. Only those tables and views are shown that the current user has access to (by way of being the owner or having some privilege).

Table 34-50. tables Columns

Name	Data Type	Description
table_catalog	sql_identifier	Name of the database that contains the table (always the current database)
table_schema	sql_identifier	Name of the schema that contains the table
table_name	sql_identifier	Name of the table
table_type	character_data	Type of the table: BASE TABLE for a persistent base table (the normal table type), VIEW for a view, FOREIGN TABLE for a foreign table, or LOCAL TEMPORARY for a temporary table
self_referencing_column_name	sql_identifier	Applies to a feature not available in PostgreSQL

Name	Data Type	Description
reference_generation	character_data	Applies to a feature not available in PostgreSQL
user_defined_type_catalog	sql_identifier	If the table is a typed table, the name of the database that contains the underlying data type (always the current database), else null.
user_defined_type_schema	sql_identifier	If the table is a typed table, the name of the schema that contains the underlying data type, else null.
user_defined_type_name	sql_identifier	If the table is a typed table, the name of the underlying data type, else null.
is_insertable_into	yes_or_no	YES if the table is insertable into, NO if not (Base tables are always insertable into, views not necessarily.)
is_typed	yes_or_no	YES if the table is a typed table, NO if not
commit_action	character_data	Not yet implemented

34.53. transforms

The view transforms contains information about the transforms defined in the current database. More precisely, it contains a row for each function contained in a transform (the "from SQL" or "to SQL" function).

Table 34-51. transforms Columns

Name	Data Type	Description
udt_catalog	sql_identifier	Name of the database that contains the type the transform is for (always the current database)
udt_schema	sql_identifier	Name of the schema that contains the type the transform is for
udt_name	sql_identifier	Name of the type the transform is for
specific_catalog	sql_identifier	Name of the database containing the function (always the current database)

Name	Data Type	Description
specific_schema	sql_identifier	Name of the schema containing the function
specific_name	sql_identifier	The "specific name" of the function. See Section 34.40 for more information.
group_name	sql_identifier	The SQL standard allows defining transforms in "groups", and selecting a group at run time. PostgreSQL does not support this. Instead, transforms are specific to a language. As a compromise, this field contains the language the transform is for.
transform_type	character_data	FROM SQL or TO SQL

34.54. triggered_update_columns

For triggers in the current database that specify a column list (like UPDATE OF column1, column2), the view triggered_update_columns identifies these columns. Triggers that do not specify a column list are not included in this view. Only those columns are shown that the current user owns or has some privilege other than SELECT on.

Table 34-52. triggered_update_columns Columns

Name	Data Type	Description
trigger_catalog	sql_identifier	Name of the database that contains the trigger (always the current database)
trigger_schema	sql_identifier	Name of the schema that contains the trigger
trigger_name	sql_identifier	Name of the trigger
event_object_catalog	sql_identifier	Name of the database that contains the table that the trigger is defined on (always the current database)
event_object_schema	sql_identifier	Name of the schema that contains the table that the trigger is defined on
event_object_table	sql_identifier	Name of the table that the trigger is defined on

Name	Data Type	Description
event_object_column	sql_identifier	Name of the column that the trigger is defined on

34.55. triggers

The view triggers contains all triggers defined in the current database on tables and views that the current user owns or has some privilege other than SELECT on.

Table 34-53. triggers Columns

Name	Data Type	Description
trigger_catalog	sql_identifier	Name of the database that contains the trigger (always the current database)
trigger_schema	sql_identifier	Name of the schema that contains the trigger
trigger_name	sql_identifier	Name of the trigger
event_manipulation	character_data	Event that fires the trigger (INSERT, UPDATE, or DELETE)
event_object_catalog	sql_identifier	Name of the database that contains the table that the trigger is defined on (always the current database)
event_object_schema	sql_identifier	Name of the schema that contains the table that the trigger is defined on
event_object_table	sql_identifier	Name of the table that the trigger is defined on
action_order	cardinal_number	Not yet implemented
action_condition	character_data	WHEN condition of the trigger, null if none (also null if the table is not owned by a currently enabled role)
action_statement	character_data	Statement that is executed by the trigger (currently always EXECUTE PROCEDURE *function*(...))
action_orientation	character_data	Identifies whether the trigger fires once for each processed row or once for each statement (ROW or STATEMENT)

Name	Data Type	Description
action_timing	character_data	Time at which the trigger fires (BEFORE, AFTER, or INSTEAD OF)
action_reference_old_table	sql_identifier	Applies to a feature not available in PostgreSQL
action_reference_new_table	sql_identifier	Applies to a feature not available in PostgreSQL
action_reference_old_row	sql_identifier	Applies to a feature not available in PostgreSQL
action_reference_new_row	sql_identifier	Applies to a feature not available in PostgreSQL
created	time_stamp	Applies to a feature not available in PostgreSQL

Triggers in PostgreSQL have two incompatibilities with the SQL standard that affect the representation in the information schema. First, trigger names are local to each table in PostgreSQL, rather than being independent schema objects. Therefore there can be duplicate trigger names defined in one schema, so long as they belong to different tables. (trigger_catalog and trigger_schema are really the values pertaining to the table that the trigger is defined on.) Second, triggers can be defined to fire on multiple events in PostgreSQL (e.g., ON INSERT OR UPDATE), whereas the SQL standard only allows one. If a trigger is defined to fire on multiple events, it is represented as multiple rows in the information schema, one for each type of event. As a consequence of these two issues, the primary key of the view triggers is really (trigger_catalog, trigger_schema, event_object_table, trigger_name, event_manipulation) instead of (trigger_catalog, trigger_schema, trigger_name), which is what the SQL standard specifies. Nonetheless, if you define your triggers in a manner that conforms with the SQL standard (trigger names unique in the schema and only one event type per trigger), this will not affect you.

> **Note:** Prior to PostgreSQL 9.1, this view's columns action_timing, action_reference_old_table, action_reference_new_table, action_reference_old_row, and action_reference_new_row were named condition_timing, condition_reference_old_table, condition_reference_new_table, condition_reference_old_row, and condition_reference_new_row respectively. That was how they were named in the SQL:1999 standard. The new naming conforms to SQL:2003 and later.

34.56. udt_privileges

The view udt_privileges identifies USAGE privileges granted on user-defined types to a currently enabled role or by a currently enabled role. There is one row for each combination of type, grantor, and grantee. This view shows only composite types (see under Section 34.58 for why); see Section 34.57 for domain privileges.

Table 34-54. `udt_privileges` Columns

Name	Data Type	Description
`grantor`	`sql_identifier`	Name of the role that granted the privilege
`grantee`	`sql_identifier`	Name of the role that the privilege was granted to
`udt_catalog`	`sql_identifier`	Name of the database containing the type (always the current database)
`udt_schema`	`sql_identifier`	Name of the schema containing the type
`udt_name`	`sql_identifier`	Name of the type
`privilege_type`	`character_data`	Always `TYPE USAGE`
`is_grantable`	`yes_or_no`	`YES` if the privilege is grantable, `NO` if not

34.57. `usage_privileges`

The view `usage_privileges` identifies `USAGE` privileges granted on various kinds of objects to a currently enabled role or by a currently enabled role. In PostgreSQL, this currently applies to collations, domains, foreign-data wrappers, foreign servers, and sequences. There is one row for each combination of object, grantor, and grantee.

Since collations do not have real privileges in PostgreSQL, this view shows implicit non-grantable `USAGE` privileges granted by the owner to `PUBLIC` for all collations. The other object types, however, show real privileges.

In PostgreSQL, sequences also support `SELECT` and `UPDATE` privileges in addition to the `USAGE` privilege. These are nonstandard and therefore not visible in the information schema.

Table 34-55. `usage_privileges` Columns

Name	Data Type	Description
`grantor`	`sql_identifier`	Name of the role that granted the privilege
`grantee`	`sql_identifier`	Name of the role that the privilege was granted to
`object_catalog`	`sql_identifier`	Name of the database containing the object (always the current database)
`object_schema`	`sql_identifier`	Name of the schema containing the object, if applicable, else an empty string
`object_name`	`sql_identifier`	Name of the object

Name	Data Type	Description
object_type	character_data	COLLATION or DOMAIN or FOREIGN DATA WRAPPER or FOREIGN SERVER or SEQUENCE
privilege_type	character_data	Always USAGE
is_grantable	yes_or_no	YES if the privilege is grantable, NO if not

34.58. user_defined_types

The view user_defined_types currently contains all composite types defined in the current database. Only those types are shown that the current user has access to (by way of being the owner or having some privilege).

SQL knows about two kinds of user-defined types: structured types (also known as composite types in PostgreSQL) and distinct types (not implemented in PostgreSQL). To be future-proof, use the column user_defined_type_category to differentiate between these. Other user-defined types such as base types and enums, which are PostgreSQL extensions, are not shown here. For domains, see Section 34.22 instead.

Table 34-56. user_defined_types Columns

Name	Data Type	Description
user_defined_type_catalog	sql_identifier	Name of the database that contains the type (always the current database)
user_defined_type_schema	sql_identifier	Name of the schema that contains the type
user_defined_type_name	sql_identifier	Name of the type
user_defined_type_category	character_data	Currently always STRUCTURED
is_instantiable	yes_or_no	Applies to a feature not available in PostgreSQL
is_final	yes_or_no	Applies to a feature not available in PostgreSQL
ordering_form	character_data	Applies to a feature not available in PostgreSQL
ordering_category	character_data	Applies to a feature not available in PostgreSQL
ordering_routine_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
ordering_routine_schema	sql_identifier	Applies to a feature not available in PostgreSQL

Name	Data Type	Description
ordering_routine_name	sql_identifier	Applies to a feature not available in PostgreSQL
reference_type	character_data	Applies to a feature not available in PostgreSQL
data_type	character_data	Applies to a feature not available in PostgreSQL
character_maximum_length	cardinal_number	Applies to a feature not available in PostgreSQL
character_octet_length	cardinal_number	Applies to a feature not available in PostgreSQL
character_set_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_schema	sql_identifier	Applies to a feature not available in PostgreSQL
character_set_name	sql_identifier	Applies to a feature not available in PostgreSQL
collation_catalog	sql_identifier	Applies to a feature not available in PostgreSQL
collation_schema	sql_identifier	Applies to a feature not available in PostgreSQL
collation_name	sql_identifier	Applies to a feature not available in PostgreSQL
numeric_precision	cardinal_number	Applies to a feature not available in PostgreSQL
numeric_precision_radix	cardinal_number	Applies to a feature not available in PostgreSQL
numeric_scale	cardinal_number	Applies to a feature not available in PostgreSQL
datetime_precision	cardinal_number	Applies to a feature not available in PostgreSQL
interval_type	character_data	Applies to a feature not available in PostgreSQL
interval_precision	cardinal_number	Applies to a feature not available in PostgreSQL
source_dtd_identifier	sql_identifier	Applies to a feature not available in PostgreSQL
ref_dtd_identifier	sql_identifier	Applies to a feature not available in PostgreSQL

34.59. user_mapping_options

The view user_mapping_options contains all the options defined for user mappings in the current database. Only those user mappings are shown where the current user has access to the corresponding foreign server (by way of being the owner or having some privilege).

Table 34-57. user_mapping_options Columns

Name	Data Type	Description
authorization_identifier	sql_identifier	Name of the user being mapped, or PUBLIC if the mapping is public
foreign_server_catalog	sql_identifier	Name of the database that the foreign server used by this mapping is defined in (always the current database)
foreign_server_name	sql_identifier	Name of the foreign server used by this mapping
option_name	sql_identifier	Name of an option
option_value	character_data	Value of the option. This column will show as null unless the current user is the user being mapped, or the mapping is for PUBLIC and the current user is the server owner, or the current user is a superuser. The intent is to protect password information stored as user mapping option.

34.60. user_mappings

The view user_mappings contains all user mappings defined in the current database. Only those user mappings are shown where the current user has access to the corresponding foreign server (by way of being the owner or having some privilege).

Table 34-58. user_mappings Columns

Name	Data Type	Description
authorization_identifier	sql_identifier	Name of the user being mapped, or PUBLIC if the mapping is public

Name	Data Type	Description
foreign_server_catalog	sql_identifier	Name of the database that the foreign server used by this mapping is defined in (always the current database)
foreign_server_name	sql_identifier	Name of the foreign server used by this mapping

34.61. view_column_usage

The view view_column_usage identifies all columns that are used in the query expression of a view (the SELECT statement that defines the view). A column is only included if the table that contains the column is owned by a currently enabled role.

Note: Columns of system tables are not included. This should be fixed sometime.

Table 34-59. view_column_usage Columns

Name	Data Type	Description
view_catalog	sql_identifier	Name of the database that contains the view (always the current database)
view_schema	sql_identifier	Name of the schema that contains the view
view_name	sql_identifier	Name of the view
table_catalog	sql_identifier	Name of the database that contains the table that contains the column that is used by the view (always the current database)
table_schema	sql_identifier	Name of the schema that contains the table that contains the column that is used by the view
table_name	sql_identifier	Name of the table that contains the column that is used by the view
column_name	sql_identifier	Name of the column that is used by the view

34.62. view_routine_usage

The view view_routine_usage identifies all routines (functions and procedures) that are used in the query expression of a view (the SELECT statement that defines the view). A routine is only included if that routine is owned by a currently enabled role.

Table 34-60. view_routine_usage Columns

Name	Data Type	Description
table_catalog	sql_identifier	Name of the database containing the view (always the current database)
table_schema	sql_identifier	Name of the schema containing the view
table_name	sql_identifier	Name of the view
specific_catalog	sql_identifier	Name of the database containing the function (always the current database)
specific_schema	sql_identifier	Name of the schema containing the function
specific_name	sql_identifier	The "specific name" of the function. See Section 34.40 for more information.

34.63. view_table_usage

The view view_table_usage identifies all tables that are used in the query expression of a view (the SELECT statement that defines the view). A table is only included if that table is owned by a currently enabled role.

Note: System tables are not included. This should be fixed sometime.

Table 34-61. view_table_usage Columns

Name	Data Type	Description
view_catalog	sql_identifier	Name of the database that contains the view (always the current database)
view_schema	sql_identifier	Name of the schema that contains the view
view_name	sql_identifier	Name of the view

Name	Data Type	Description
table_catalog	sql_identifier	Name of the database that contains the table that is used by the view (always the current database)
table_schema	sql_identifier	Name of the schema that contains the table that is used by the view
table_name	sql_identifier	Name of the table that is used by the view

34.64. views

The view views contains all views defined in the current database. Only those views are shown that the current user has access to (by way of being the owner or having some privilege).

Table 34-62. views Columns

Name	Data Type	Description
table_catalog	sql_identifier	Name of the database that contains the view (always the current database)
table_schema	sql_identifier	Name of the schema that contains the view
table_name	sql_identifier	Name of the view
view_definition	character_data	Query expression defining the view (null if the view is not owned by a currently enabled role)
check_option	character_data	CASCADED or LOCAL if the view has a CHECK OPTION defined on it, NONE if not
is_updatable	yes_or_no	YES if the view is updatable (allows UPDATE and DELETE), NO if not
is_insertable_into	yes_or_no	YES if the view is insertable into (allows INSERT), NO if not
is_trigger_updatable	yes_or_no	YES if the view has an INSTEAD OF UPDATE trigger defined on it, NO if not
is_trigger_deletable	yes_or_no	YES if the view has an INSTEAD OF DELETE trigger defined on it, NO if not

Name	Data Type	Description
is_trigger_insertable_into	yes_or_no	YES if the view has an INSTEAD OF INSERT trigger defined on it, NO if not

V. Server Programming

This part is about extending the server functionality with user-defined functions, data types, triggers, etc. These are advanced topics which should probably be approached only after all the other user documentation about PostgreSQL has been understood. Later chapters in this part describe the server-side programming languages available in the PostgreSQL distribution as well as general issues concerning server-side programming languages. It is essential to read at least the earlier sections of Chapter 35 (covering functions) before diving into the material about server-side programming languages.

Chapter 35. Extending SQL

In the sections that follow, we will discuss how you can extend the PostgreSQL SQL query language by adding:

- functions (starting in Section 35.3)
- aggregates (starting in Section 35.10)
- data types (starting in Section 35.11)
- operators (starting in Section 35.12)
- operator classes for indexes (starting in Section 35.14)
- packages of related objects (starting in Section 35.15)

35.1. How Extensibility Works

PostgreSQL is extensible because its operation is catalog-driven. If you are familiar with standard relational database systems, you know that they store information about databases, tables, columns, etc., in what are commonly known as system catalogs. (Some systems call this the data dictionary.) The catalogs appear to the user as tables like any other, but the DBMS stores its internal bookkeeping in them. One key difference between PostgreSQL and standard relational database systems is that PostgreSQL stores much more information in its catalogs: not only information about tables and columns, but also information about data types, functions, access methods, and so on. These tables can be modified by the user, and since PostgreSQL bases its operation on these tables, this means that PostgreSQL can be extended by users. By comparison, conventional database systems can only be extended by changing hardcoded procedures in the source code or by loading modules specially written by the DBMS vendor.

The PostgreSQL server can moreover incorporate user-written code into itself through dynamic loading. That is, the user can specify an object code file (e.g., a shared library) that implements a new type or function, and PostgreSQL will load it as required. Code written in SQL is even more trivial to add to the server. This ability to modify its operation "on the fly" makes PostgreSQL uniquely suited for rapid prototyping of new applications and storage structures.

35.2. The PostgreSQL Type System

PostgreSQL data types are divided into base types, composite types, domains, and pseudo-types.

35.2.1. Base Types

Base types are those, like `int4`, that are implemented below the level of the SQL language (typically in a low-level language such as C). They generally correspond to what are often known as abstract data types. PostgreSQL can only operate on such types through functions provided by the user and only understands the behavior of such types to the extent that the user describes them. Base types are further subdivided

into scalar and array types. For each scalar type, a corresponding array type is automatically created that can hold variable-size arrays of that scalar type.

35.2.2. Composite Types

Composite types, or row types, are created whenever the user creates a table. It is also possible to use CREATE TYPE to define a "stand-alone" composite type with no associated table. A composite type is simply a list of types with associated field names. A value of a composite type is a row or record of field values. The user can access the component fields from SQL queries. Refer to Section 8.16 for more information on composite types.

35.2.3. Domains

A domain is based on a particular base type and for many purposes is interchangeable with its base type. However, a domain can have constraints that restrict its valid values to a subset of what the underlying base type would allow.

Domains can be created using the SQL command CREATE DOMAIN. Their creation and use is not discussed in this chapter.

35.2.4. Pseudo-Types

There are a few "pseudo-types" for special purposes. Pseudo-types cannot appear as columns of tables or attributes of composite types, but they can be used to declare the argument and result types of functions. This provides a mechanism within the type system to identify special classes of functions. Table 8-25 lists the existing pseudo-types.

35.2.5. Polymorphic Types

Five pseudo-types of special interest are `anyelement`, `anyarray`, `anynonarray`, `anyenum`, and `anyrange`, which are collectively called *polymorphic types*. Any function declared using these types is said to be a *polymorphic function*. A polymorphic function can operate on many different data types, with the specific data type(s) being determined by the data types actually passed to it in a particular call.

Polymorphic arguments and results are tied to each other and are resolved to a specific data type when a query calling a polymorphic function is parsed. Each position (either argument or return value) declared as `anyelement` is allowed to have any specific actual data type, but in any given call they must all be the *same* actual type. Each position declared as `anyarray` can have any array data type, but similarly they must all be the same type. And similarly, positions declared as `anyrange` must all be the same range type. Furthermore, if there are positions declared `anyarray` and others declared `anyelement`, the actual array type in the `anyarray` positions must be an array whose elements are the same type appearing in the `anyelement` positions. Similarly, if there are positions declared `anyrange` and others declared `anyelement` or `anyarray`, the actual range type in the `anyrange` positions must be a range whose subtype is the same type appearing in the `anyelement` positions and the same as the element type of the `anyarray` positions. `anynonarray` is treated exactly the same as `anyelement`, but adds the

additional constraint that the actual type must not be an array type. `anyenum` is treated exactly the same as `anyelement`, but adds the additional constraint that the actual type must be an enum type.

Thus, when more than one argument position is declared with a polymorphic type, the net effect is that only certain combinations of actual argument types are allowed. For example, a function declared as `equal(anyelement, anyelement)` will take any two input values, so long as they are of the same data type.

When the return value of a function is declared as a polymorphic type, there must be at least one argument position that is also polymorphic, and the actual data type supplied as the argument determines the actual result type for that call. For example, if there were not already an array subscripting mechanism, one could define a function that implements subscripting as `subscript(anyarray, integer) returns anyelement`. This declaration constrains the actual first argument to be an array type, and allows the parser to infer the correct result type from the actual first argument's type. Another example is that a function declared as `f(anyarray) returns anyenum` will only accept arrays of enum types.

In most cases, the parser can infer the actual data type for a polymorphic result type from arguments that are of a different polymorphic type; for example `anyarray` can be deduced from `anyelement` or vice versa. The exception is that a polymorphic result of type `anyrange` requires an argument of type `anyrange`; it cannot be deduced from `anyarray` or `anyelement` arguments. This is because there could be multiple range types with the same subtype.

Note that `anynonarray` and `anyenum` do not represent separate type variables; they are the same type as `anyelement`, just with an additional constraint. For example, declaring a function as `f(anyelement, anyenum)` is equivalent to declaring it as `f(anyenum, anyenum)`: both actual arguments have to be the same enum type.

A variadic function (one taking a variable number of arguments, as in Section 35.4.5) can be polymorphic: this is accomplished by declaring its last parameter as `VARIADIC anyarray`. For purposes of argument matching and determining the actual result type, such a function behaves the same as if you had written the appropriate number of `anynonarray` parameters.

35.3. User-defined Functions

PostgreSQL provides four kinds of functions:

- query language functions (functions written in SQL) (Section 35.4)
- procedural language functions (functions written in, for example, PL/pgSQL or PL/Tcl) (Section 35.7)
- internal functions (Section 35.8)
- C-language functions (Section 35.9)

Every kind of function can take base types, composite types, or combinations of these as arguments (parameters). In addition, every kind of function can return a base type or a composite type. Functions can also be defined to return sets of base or composite values.

Many kinds of functions can take or return certain pseudo-types (such as polymorphic types), but the available facilities vary. Consult the description of each kind of function for more details.

It's easiest to define SQL functions, so we'll start by discussing those. Most of the concepts presented for SQL functions will carry over to the other types of functions.

Throughout this chapter, it can be useful to look at the reference page of the CREATE FUNCTION command to understand the examples better. Some examples from this chapter can be found in `funcs.sql` and `funcs.c` in the `src/tutorial` directory in the PostgreSQL source distribution.

35.4. Query Language (SQL) Functions

SQL functions execute an arbitrary list of SQL statements, returning the result of the last query in the list. In the simple (non-set) case, the first row of the last query's result will be returned. (Bear in mind that "the first row" of a multirow result is not well-defined unless you use `ORDER BY`.) If the last query happens to return no rows at all, the null value will be returned.

Alternatively, an SQL function can be declared to return a set (that is, multiple rows) by specifying the function's return type as `SETOF` *sometype*, or equivalently by declaring it as `RETURNS TABLE(`*columns*`)`. In this case all rows of the last query's result are returned. Further details appear below.

The body of an SQL function must be a list of SQL statements separated by semicolons. A semicolon after the last statement is optional. Unless the function is declared to return `void`, the last statement must be a `SELECT`, or an `INSERT`, `UPDATE`, or `DELETE` that has a `RETURNING` clause.

Any collection of commands in the SQL language can be packaged together and defined as a function. Besides `SELECT` queries, the commands can include data modification queries (`INSERT`, `UPDATE`, and `DELETE`), as well as other SQL commands. (You cannot use transaction control commands, e.g., `COMMIT`, `SAVEPOINT`, and some utility commands, e.g., `VACUUM`, in SQL functions.) However, the final command must be a `SELECT` or have a `RETURNING` clause that returns whatever is specified as the function's return type. Alternatively, if you want to define a SQL function that performs actions but has no useful value to return, you can define it as returning `void`. For example, this function removes rows with negative salaries from the `emp` table:

```
CREATE FUNCTION clean_emp() RETURNS void AS '
    DELETE FROM emp
        WHERE salary < 0;
' LANGUAGE SQL;

SELECT clean_emp();

 clean_emp
-----------

(1 row)
```

> **Note:** The entire body of a SQL function is parsed before any of it is executed. While a SQL function can contain commands that alter the system catalogs (e.g., `CREATE TABLE`), the effects of such commands will not be visible during parse analysis of later commands in the function. Thus, for example,

`CREATE TABLE foo (...); INSERT INTO foo VALUES(...);` will not work as desired if packaged up into a single SQL function, since `foo` won't exist yet when the `INSERT` command is parsed. It's recommended to use PL/PgSQL instead of a SQL function in this type of situation.

The syntax of the `CREATE FUNCTION` command requires the function body to be written as a string constant. It is usually most convenient to use dollar quoting (see Section 4.1.2.4) for the string constant. If you choose to use regular single-quoted string constant syntax, you must double single quote marks (`'`) and backslashes (`\`) (assuming escape string syntax) in the body of the function (see Section 4.1.2.1).

35.4.1. Arguments for SQL Functions

Arguments of a SQL function can be referenced in the function body using either names or numbers. Examples of both methods appear below.

To use a name, declare the function argument as having a name, and then just write that name in the function body. If the argument name is the same as any column name in the current SQL command within the function, the column name will take precedence. To override this, qualify the argument name with the name of the function itself, that is *`function_name`*`.`*`argument_name`*. (If this would conflict with a qualified column name, again the column name wins. You can avoid the ambiguity by choosing a different alias for the table within the SQL command.)

In the older numeric approach, arguments are referenced using the syntax `$`*`n`*: `$1` refers to the first input argument, `$2` to the second, and so on. This will work whether or not the particular argument was declared with a name.

If an argument is of a composite type, then the dot notation, e.g., `argname.fieldname` or `$1.fieldname`, can be used to access attributes of the argument. Again, you might need to qualify the argument's name with the function name to make the form with an argument name unambiguous.

SQL function arguments can only be used as data values, not as identifiers. Thus for example this is reasonable:

```
INSERT INTO mytable VALUES ($1);
```

but this will not work:

```
INSERT INTO $1 VALUES (42);
```

Note: The ability to use names to reference SQL function arguments was added in PostgreSQL 9.2. Functions to be used in older servers must use the `$`*`n`* notation.

35.4.2. SQL Functions on Base Types

The simplest possible SQL function has no arguments and simply returns a base type, such as `integer`:

```
CREATE FUNCTION one() RETURNS integer AS $$
    SELECT 1 AS result;
$$ LANGUAGE SQL;

-- Alternative syntax for string literal:
CREATE FUNCTION one() RETURNS integer AS '
    SELECT 1 AS result;
' LANGUAGE SQL;

SELECT one();

 one
-----
   1
```

Notice that we defined a column alias within the function body for the result of the function (with the name `result`), but this column alias is not visible outside the function. Hence, the result is labeled `one` instead of `result`.

It is almost as easy to define SQL functions that take base types as arguments:

```
CREATE FUNCTION add_em(x integer, y integer) RETURNS integer AS $$
    SELECT x + y;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

Alternatively, we could dispense with names for the arguments and use numbers:

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;

SELECT add_em(1, 2) AS answer;

 answer
--------
      3
```

Here is a more useful function, which might be used to debit a bank account:

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS integer AS $$
```

```
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT 1;
$$ LANGUAGE SQL;
```

A user could execute this function to debit account 17 by $100.00 as follows:

```
SELECT tf1(17, 100.0);
```

In this example, we chose the name `accountno` for the first argument, but this is the same as the name of a column in the `bank` table. Within the `UPDATE` command, `accountno` refers to the column `bank.accountno`, so `tf1.accountno` must be used to refer to the argument. We could of course avoid this by using a different name for the argument.

In practice one would probably like a more useful result from the function than a constant 1, so a more likely definition is:

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS integer AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno;
    SELECT balance FROM bank WHERE accountno = tf1.accountno;
$$ LANGUAGE SQL;
```

which adjusts the balance and returns the new balance. The same thing could be done in one command using `RETURNING`:

```
CREATE FUNCTION tf1 (accountno integer, debit numeric) RETURNS integer AS $$
    UPDATE bank
        SET balance = balance - debit
        WHERE accountno = tf1.accountno
    RETURNING balance;
$$ LANGUAGE SQL;
```

35.4.3. SQL Functions on Composite Types

When writing functions with arguments of composite types, we must not only specify which argument we want but also the desired attribute (field) of that argument. For example, suppose that `emp` is a table containing employee data, and therefore also the name of the composite type of each row of the table. Here is a function `double_salary` that computes what someone's salary would be if it were doubled:

```
CREATE TABLE emp (
    name        text,
    salary      numeric,
    age         integer,
    cubicle     point
);
```

```
INSERT INTO emp VALUES ('Bill', 4200, 45, '(2,1)');

CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;

SELECT name, double_salary(emp.*) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';

 name | dream
------+-------
 Bill |  8400
```

Notice the use of the syntax `$1.salary` to select one field of the argument row value. Also notice how the calling `SELECT` command uses *`table_name`*`.*` to select the entire current row of a table as a composite value. The table row can alternatively be referenced using just the table name, like this:

```
SELECT name, double_salary(emp) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

but this usage is deprecated since it's easy to get confused. (See Section 8.16.5 for details about these two notations for the composite value of a table row.)

Sometimes it is handy to construct a composite argument value on-the-fly. This can be done with the `ROW` construct. For example, we could adjust the data being passed to the function:

```
SELECT name, double_salary(ROW(name, salary*1.1, age, cubicle)) AS dream
    FROM emp;
```

It is also possible to build a function that returns a composite type. This is an example of a function that returns a single `emp` row:

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT text 'None' AS name,
        1000.0 AS salary,
        25 AS age,
        point '(2,2)' AS cubicle;
$$ LANGUAGE SQL;
```

In this example we have specified each of the attributes with a constant value, but any computation could have been substituted for these constants.

Note two important things about defining the function:

- The select list order in the query must be exactly the same as that in which the columns appear in the table associated with the composite type. (Naming the columns, as we did above, is irrelevant to the system.)

- You must typecast the expressions to match the definition of the composite type, or you will get errors like this:

```
ERROR:  function declared to return emp returns varchar instead of text at column 1
```

A different way to define the same function is:

```
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
```

Here we wrote a `SELECT` that returns just a single column of the correct composite type. This isn't really better in this situation, but it is a handy alternative in some cases — for example, if we need to compute the result by calling another function that returns the desired composite value.

We could call this function directly either by using it in a value expression:

```
SELECT new_emp();

         new_emp
-------------------------
 (None,1000.0,25,"(2,2)")
```

or by calling it as a table function:

```
SELECT * FROM new_emp();

 name | salary | age | cubicle
------+--------+-----+---------
 None | 1000.0 |  25 | (2,2)
```

The second way is described more fully in Section 35.4.7.

When you use a function that returns a composite type, you might want only one field (attribute) from its result. You can do that with syntax like this:

```
SELECT (new_emp()).name;

 name
------
 None
```

The extra parentheses are needed to keep the parser from getting confused. If you try to do it without them, you get something like this:

```
SELECT new_emp().name;
ERROR:  syntax error at or near "."
LINE 1: SELECT new_emp().name;
                        ^
```

Another option is to use functional notation for extracting an attribute:

```
SELECT name(new_emp());

 name
------
 None
```

As explained in Section 8.16.5, the field notation and functional notation are equivalent.

Another way to use a function returning a composite type is to pass the result to another function that accepts the correct row type as input:

```
CREATE FUNCTION getname(emp) RETURNS text AS $$
    SELECT $1.name;
$$ LANGUAGE SQL;

SELECT getname(new_emp());
 getname
---------
 None
(1 row)
```

35.4.4. SQL Functions with Output Parameters

An alternative way of describing a function's results is to define it with *output parameters*, as in this example:

```
CREATE FUNCTION add_em (IN x int, IN y int, OUT sum int)
AS 'SELECT x + y'
LANGUAGE SQL;

SELECT add_em(3,7);
 add_em
--------
     10
(1 row)
```

This is not essentially different from the version of `add_em` shown in Section 35.4.2. The real value of output parameters is that they provide a convenient way of defining functions that return several columns. For example,

```
CREATE FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int)
AS 'SELECT x + y, x * y'
LANGUAGE SQL;

 SELECT * FROM sum_n_product(11,42);
 sum | product
-----+---------
  53 |     462
(1 row)
```

What has essentially happened here is that we have created an anonymous composite type for the result of the function. The above example has the same end result as

```
CREATE TYPE sum_prod AS (sum int, product int);

CREATE FUNCTION sum_n_product (int, int) RETURNS sum_prod
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

but not having to bother with the separate composite type definition is often handy. Notice that the names attached to the output parameters are not just decoration, but determine the column names of the anonymous composite type. (If you omit a name for an output parameter, the system will choose a name on its own.)

Notice that output parameters are not included in the calling argument list when invoking such a function from SQL. This is because PostgreSQL considers only the input parameters to define the function's calling signature. That means also that only the input parameters matter when referencing the function for purposes such as dropping it. We could drop the above function with either of

```
DROP FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int);
DROP FUNCTION sum_n_product (int, int);
```

Parameters can be marked as `IN` (the default), `OUT`, `INOUT`, or `VARIADIC`. An `INOUT` parameter serves as both an input parameter (part of the calling argument list) and an output parameter (part of the result record type). `VARIADIC` parameters are input parameters, but are treated specially as described next.

35.4.5. SQL Functions with Variable Numbers of Arguments

SQL functions can be declared to accept variable numbers of arguments, so long as all the "optional" arguments are of the same data type. The optional arguments will be passed to the function as an array. The function is declared by marking the last parameter as `VARIADIC`; this parameter must be declared as being of an array type. For example:

```
CREATE FUNCTION mleast(VARIADIC arr numeric[]) RETURNS numeric AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT mleast(10, -1, 5, 4.4);
 mleast
--------
     -1
(1 row)
```

Effectively, all the actual arguments at or beyond the `VARIADIC` position are gathered up into a one-dimensional array, as if you had written

```
SELECT mleast(ARRAY[10, -1, 5, 4.4]);    -- doesn't work
```

You can't actually write that, though — or at least, it will not match this function definition. A parameter marked `VARIADIC` matches one or more occurrences of its element type, not of its own type.

Sometimes it is useful to be able to pass an already-constructed array to a variadic function; this is particularly handy when one variadic function wants to pass on its array parameter to another one. Also, this is the only secure way to call a variadic function found in a schema that permits untrusted users to create objects; see Section 10.3. You can do this by specifying `VARIADIC` in the call:

```
SELECT mleast(VARIADIC ARRAY[10, -1, 5, 4.4]);
```

This prevents expansion of the function's variadic parameter into its element type, thereby allowing the array argument value to match normally. `VARIADIC` can only be attached to the last actual argument of a function call.

Specifying `VARIADIC` in the call is also the only way to pass an empty array to a variadic function, for example:

```
SELECT mleast(VARIADIC ARRAY[]::numeric[]);
```

Simply writing `SELECT mleast()` does not work because a variadic parameter must match at least one actual argument. (You could define a second function also named `mleast`, with no parameters, if you wanted to allow such calls.)

The array element parameters generated from a variadic parameter are treated as not having any names of their own. This means it is not possible to call a variadic function using named arguments (Section 4.3), except when you specify `VARIADIC`. For example, this will work:

```
SELECT mleast(VARIADIC arr => ARRAY[10, -1, 5, 4.4]);
```

but not these:

```
SELECT mleast(arr => 10);
SELECT mleast(arr => ARRAY[10, -1, 5, 4.4]);
```

35.4.6. SQL Functions with Default Values for Arguments

Functions can be declared with default values for some or all input arguments. The default values are inserted whenever the function is called with insufficiently many actual arguments. Since arguments can only be omitted from the end of the actual argument list, all parameters after a parameter with a default value have to have default values as well. (Although the use of named argument notation could allow this restriction to be relaxed, it's still enforced so that positional argument notation works sensibly.) Whether or not you use it, this capability creates a need for precautions when calling functions in databases where some users mistrust other users; see Section 10.3.

For example:

```
CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3)
RETURNS int
LANGUAGE SQL
AS $$
    SELECT $1 + $2 + $3;
$$;
```

```
SELECT foo(10, 20, 30);
 foo
-----
  60
(1 row)

SELECT foo(10, 20);
 foo
-----
  33
(1 row)

SELECT foo(10);
 foo
-----
  15
(1 row)

SELECT foo();  -- fails since there is no default for the first argument
ERROR:  function foo() does not exist
```

The = sign can also be used in place of the key word `DEFAULT`.

35.4.7. SQL Functions as Table Sources

All SQL functions can be used in the `FROM` clause of a query, but it is particularly useful for functions returning composite types. If the function is defined to return a base type, the table function produces a one-column table. If the function is defined to return a composite type, the table function produces a column for each attribute of the composite type.

Here is an example:

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT *, upper(fooname) FROM getfoo(1) AS t1;

 fooid | foosubid | fooname | upper
-------+----------+---------+-------
     1 |        1 | Joe     | JOE
(1 row)
```

As the example shows, we can work with the columns of the function's result just the same as if they were columns of a regular table.

Note that we only got one row out of the function. This is because we did not use SETOF. That is described in the next section.

35.4.8. SQL Functions Returning Sets

When an SQL function is declared as returning SETOF *sometype*, the function's final query is executed to completion, and each row it outputs is returned as an element of the result set.

This feature is normally used when calling the function in the FROM clause. In this case each row returned by the function becomes a row of the table seen by the query. For example, assume that table foo has the same contents as above, and we say:

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;
```

Then we would get:

```
 fooid | foosubid | fooname
-------+----------+---------
     1 |        1 | Joe
     1 |        2 | Ed
(2 rows)
```

It is also possible to return multiple rows with the columns defined by output parameters, like this:

```
CREATE TABLE tab (y int, z int);
INSERT INTO tab VALUES (1, 2), (3, 4), (5, 6), (7, 8);

CREATE FUNCTION sum_n_product_with_tab (x int, OUT sum int, OUT product int)
RETURNS SETOF record
AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;

SELECT * FROM sum_n_product_with_tab(10);
 sum | product
-----+---------
  11 |      10
  13 |      30
  15 |      50
  17 |      70
(4 rows)
```

The key point here is that you must write RETURNS SETOF record to indicate that the function returns multiple rows instead of just one. If there is only one output parameter, write that parameter's type instead of record.

It is frequently useful to construct a query's result by invoking a set-returning function multiple times, with the parameters for each invocation coming from successive rows of a table or subquery. The preferred way to do this is to use the `LATERAL` key word, which is described in Section 7.2.1.5. Here is an example using a set-returning function to enumerate elements of a tree structure:

```
SELECT * FROM nodes;
   name    | parent
-----------+--------
 Top       |
 Child1    | Top
 Child2    | Top
 Child3    | Top
 SubChild1 | Child1
 SubChild2 | Child1
(6 rows)

CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
    SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL STABLE;

SELECT * FROM listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)

SELECT name, child FROM nodes, LATERAL listchildren(name) AS child;
  name  |   child
--------+-----------
 Top    | Child1
 Top    | Child2
 Top    | Child3
 Child1 | SubChild1
 Child1 | SubChild2
(5 rows)
```

This example does not do anything that we couldn't have done with a simple join, but in more complex calculations the option to put some of the work into a function can be quite convenient.

Currently, functions returning sets can also be called in the select list of a query. For each row that the query generates by itself, the function returning set is invoked, and an output row is generated for each element of the function's result set. Note, however, that this capability is deprecated and might be removed in future releases. The previous example could also be done with queries like these:

```
SELECT listchildren('Top');
 listchildren
--------------
 Child1
 Child2
 Child3
(3 rows)
```

```
SELECT name, listchildren(name) FROM nodes;
  name  | listchildren
--------+--------------
 Top    | Child1
 Top    | Child2
 Top    | Child3
 Child1 | SubChild1
 Child1 | SubChild2
(5 rows)
```

In the last SELECT, notice that no output row appears for Child2, Child3, etc. This happens because listchildren returns an empty set for those arguments, so no result rows are generated. This is the same behavior as we got from an inner join to the function result when using the LATERAL syntax.

> **Note:** If a function's last command is INSERT, UPDATE, or DELETE with RETURNING, that command will always be executed to completion, even if the function is not declared with SETOF or the calling query does not fetch all the result rows. Any extra rows produced by the RETURNING clause are silently dropped, but the commanded table modifications still happen (and are all completed before returning from the function).

> **Note:** The key problem with using set-returning functions in the select list, rather than the FROM clause, is that putting more than one set-returning function in the same select list does not behave very sensibly. (What you actually get if you do so is a number of output rows equal to the least common multiple of the numbers of rows produced by each set-returning function.) The LATERAL syntax produces less surprising results when calling multiple set-returning functions, and should usually be used instead.

35.4.9. SQL Functions Returning TABLE

There is another way to declare a function as returning a set, which is to use the syntax RETURNS TABLE(*columns*). This is equivalent to using one or more OUT parameters plus marking the function as returning SETOF record (or SETOF a single output parameter's type, as appropriate). This notation is specified in recent versions of the SQL standard, and thus may be more portable than using SETOF.

For example, the preceding sum-and-product example could also be done this way:

```
CREATE FUNCTION sum_n_product_with_tab (x int)
RETURNS TABLE(sum int, product int) AS $$
    SELECT $1 + tab.y, $1 * tab.y FROM tab;
$$ LANGUAGE SQL;
```

It is not allowed to use explicit OUT or INOUT parameters with the RETURNS TABLE notation — you must put all the output columns in the TABLE list.

35.4.10. Polymorphic SQL Functions

SQL functions can be declared to accept and return the polymorphic types `anyelement`, `anyarray`, `anynonarray`, `anyenum`, and `anyrange`. See Section 35.2.5 for a more detailed explanation of polymorphic functions. Here is a polymorphic function `make_array` that builds up an array from two arbitrary data type elements:

```
CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;
 intarray | textarray
----------+-----------
 {1,2}    | {a,b}
(1 row)
```

Notice the use of the typecast `'a'::text` to specify that the argument is of type `text`. This is required if the argument is just a string literal, since otherwise it would be treated as type `unknown`, and array of `unknown` is not a valid type. Without the typecast, you will get errors like this:

```
ERROR:  could not determine polymorphic type because input has type "unknown"
```

It is permitted to have polymorphic arguments with a fixed return type, but the converse is not. For example:

```
CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);
 is_greater
------------
 f
(1 row)

CREATE FUNCTION invalid_func() RETURNS anyelement AS $$
    SELECT 1;
$$ LANGUAGE SQL;
ERROR:  cannot determine result data type
DETAIL:  A function returning a polymorphic type must have at least one polymorphic argumen
```

Polymorphism can be used with functions that have output arguments. For example:

```
CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE SQL;

SELECT * FROM dup(22);
 f2 |   f3
```

```
----+---------
 22 | {22,22}
(1 row)
```

Polymorphism can also be used with variadic functions. For example:

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT anyleast(10, -1, 5, 4);
 anyleast
----------
       -1
(1 row)

SELECT anyleast('abc'::text, 'def');
 anyleast
----------
 abc
(1 row)

CREATE FUNCTION concat_values(text, VARIADIC anyarray) RETURNS text AS $$
    SELECT array_to_string($2, $1);
$$ LANGUAGE SQL;

SELECT concat_values('|', 1, 4, 2);
 concat_values
---------------
 1|4|2
(1 row)
```

35.4.11. SQL Functions with Collations

When a SQL function has one or more parameters of collatable data types, a collation is identified for each function call depending on the collations assigned to the actual arguments, as described in Section 22.2. If a collation is successfully identified (i.e., there are no conflicts of implicit collations among the arguments) then all the collatable parameters are treated as having that collation implicitly. This will affect the behavior of collation-sensitive operations within the function. For example, using the `anyleast` function described above, the result of

```
SELECT anyleast('abc'::text, 'ABC');
```

will depend on the database's default collation. In `C` locale the result will be `ABC`, but in many other locales it will be `abc`. The collation to use can be forced by adding a `COLLATE` clause to any of the arguments, for example

```
SELECT anyleast('abc'::text, 'ABC' COLLATE "C");
```

Alternatively, if you wish a function to operate with a particular collation regardless of what it is called with, insert COLLATE clauses as needed in the function definition. This version of anyleast would always use en_US locale to compare strings:

```
CREATE FUNCTION anyleast (VARIADIC anyarray) RETURNS anyelement AS $$
    SELECT min($1[i] COLLATE "en_US") FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;
```

But note that this will throw an error if applied to a non-collatable data type.

If no common collation can be identified among the actual arguments, then a SQL function treats its parameters as having their data types' default collation (which is usually the database's default collation, but could be different for parameters of domain types).

The behavior of collatable parameters can be thought of as a limited form of polymorphism, applicable only to textual data types.

35.5. Function Overloading

More than one function can be defined with the same SQL name, so long as the arguments they take are different. In other words, function names can be *overloaded*. Whether or not you use it, this capability entails security precautions when calling functions in databases where some users mistrust other users; see Section 10.3. When a query is executed, the server will determine which function to call from the data types and the number of the provided arguments. Overloading can also be used to simulate functions with a variable number of arguments, up to a finite maximum number.

When creating a family of overloaded functions, one should be careful not to create ambiguities. For instance, given the functions:

```
CREATE FUNCTION test(int, real) RETURNS ...
CREATE FUNCTION test(smallint, double precision) RETURNS ...
```

it is not immediately clear which function would be called with some trivial input like test(1, 1.5). The currently implemented resolution rules are described in Chapter 10, but it is unwise to design a system that subtly relies on this behavior.

A function that takes a single argument of a composite type should generally not have the same name as any attribute (field) of that type. Recall that attribute(table) is considered equivalent to table.attribute. In the case that there is an ambiguity between a function on a composite type and an attribute of the composite type, the attribute will always be used. It is possible to override that choice by schema-qualifying the function name (that is, schema.func(table)) but it's better to avoid the problem by not choosing conflicting names.

Another possible conflict is between variadic and non-variadic functions. For instance, it is possible to create both foo(numeric) and foo(VARIADIC numeric[]). In this case it is unclear which one should be matched to a call providing a single numeric argument, such as foo(10.1). The rule is that the function appearing earlier in the search path is used, or if the two functions are in the same schema, the non-variadic one is preferred.

When overloading C-language functions, there is an additional constraint: The C name of each function in the family of overloaded functions must be different from the C names of all other functions, either

internal or dynamically loaded. If this rule is violated, the behavior is not portable. You might get a run-time linker error, or one of the functions will get called (usually the internal one). The alternative form of the AS clause for the SQL CREATE FUNCTION command decouples the SQL function name from the function name in the C source code. For instance:

```
CREATE FUNCTION test(int) RETURNS int
    AS 'filename', 'test_1arg'
    LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
    AS 'filename', 'test_2arg'
    LANGUAGE C;
```

The names of the C functions here reflect one of many possible conventions.

35.6. Function Volatility Categories

Every function has a *volatility* classification, with the possibilities being VOLATILE, STABLE, or IMMUTABLE. VOLATILE is the default if the CREATE FUNCTION command does not specify a category. The volatility category is a promise to the optimizer about the behavior of the function:

- A VOLATILE function can do anything, including modifying the database. It can return different results on successive calls with the same arguments. The optimizer makes no assumptions about the behavior of such functions. A query using a volatile function will re-evaluate the function at every row where its value is needed.
- A STABLE function cannot modify the database and is guaranteed to return the same results given the same arguments for all rows within a single statement. This category allows the optimizer to optimize multiple calls of the function to a single call. In particular, it is safe to use an expression containing such a function in an index scan condition. (Since an index scan will evaluate the comparison value only once, not once at each row, it is not valid to use a VOLATILE function in an index scan condition.)
- An IMMUTABLE function cannot modify the database and is guaranteed to return the same results given the same arguments forever. This category allows the optimizer to pre-evaluate the function when a query calls it with constant arguments. For example, a query like SELECT ... WHERE x = 2 + 2 can be simplified on sight to SELECT ... WHERE x = 4, because the function underlying the integer addition operator is marked IMMUTABLE.

For best optimization results, you should label your functions with the strictest volatility category that is valid for them.

Any function with side-effects *must* be labeled VOLATILE, so that calls to it cannot be optimized away. Even a function with no side-effects needs to be labeled VOLATILE if its value can change within a single query; some examples are random(), currval(), timeofday().

Another important example is that the current_timestamp family of functions qualify as STABLE, since their values do not change within a transaction.

There is relatively little difference between STABLE and IMMUTABLE categories when considering simple interactive queries that are planned and immediately executed: it doesn't matter a lot whether a function

is executed once during planning or once during query execution startup. But there is a big difference if the plan is saved and reused later. Labeling a function IMMUTABLE when it really isn't might allow it to be prematurely folded to a constant during planning, resulting in a stale value being re-used during subsequent uses of the plan. This is a hazard when using prepared statements or when using function languages that cache plans (such as PL/pgSQL).

For functions written in SQL or in any of the standard procedural languages, there is a second important property determined by the volatility category, namely the visibility of any data changes that have been made by the SQL command that is calling the function. A VOLATILE function will see such changes, a STABLE or IMMUTABLE function will not. This behavior is implemented using the snapshotting behavior of MVCC (see Chapter 13): STABLE and IMMUTABLE functions use a snapshot established as of the start of the calling query, whereas VOLATILE functions obtain a fresh snapshot at the start of each query they execute.

> **Note:** Functions written in C can manage snapshots however they want, but it's usually a good idea to make C functions work this way too.

Because of this snapshotting behavior, a function containing only SELECT commands can safely be marked STABLE, even if it selects from tables that might be undergoing modifications by concurrent queries. PostgreSQL will execute all commands of a STABLE function using the snapshot established for the calling query, and so it will see a fixed view of the database throughout that query.

The same snapshotting behavior is used for SELECT commands within IMMUTABLE functions. It is generally unwise to select from database tables within an IMMUTABLE function at all, since the immutability will be broken if the table contents ever change. However, PostgreSQL does not enforce that you do not do that.

A common error is to label a function IMMUTABLE when its results depend on a configuration parameter. For example, a function that manipulates timestamps might well have results that depend on the TimeZone setting. For safety, such functions should be labeled STABLE instead.

> **Note:** PostgreSQL requires that STABLE and IMMUTABLE functions contain no SQL commands other than SELECT to prevent data modification. (This is not a completely bulletproof test, since such functions could still call VOLATILE functions that modify the database. If you do that, you will find that the STABLE or IMMUTABLE function does not notice the database changes applied by the called function, since they are hidden from its snapshot.)

35.7. Procedural Language Functions

PostgreSQL allows user-defined functions to be written in other languages besides SQL and C. These other languages are generically called *procedural languages* (PLs). Procedural languages aren't built into the PostgreSQL server; they are offered by loadable modules. See Chapter 39 and following chapters for more information.

35.8. Internal Functions

Internal functions are functions written in C that have been statically linked into the PostgreSQL server. The "body" of the function definition specifies the C-language name of the function, which need not be the same as the name being declared for SQL use. (For reasons of backward compatibility, an empty body is accepted as meaning that the C-language function name is the same as the SQL name.)

Normally, all internal functions present in the server are declared during the initialization of the database cluster (see Section 17.2), but a user could use `CREATE FUNCTION` to create additional alias names for an internal function. Internal functions are declared in `CREATE FUNCTION` with language name `internal`. For instance, to create an alias for the `sqrt` function:

```
CREATE FUNCTION square_root(double precision) RETURNS double precision
    AS 'dsqrt'
    LANGUAGE internal
    STRICT;
```

(Most internal functions expect to be declared "strict".)

> **Note:** Not all "predefined" functions are "internal" in the above sense. Some predefined functions are written in SQL.

35.9. C-Language Functions

User-defined functions can be written in C (or a language that can be made compatible with C, such as C++). Such functions are compiled into dynamically loadable objects (also called shared libraries) and are loaded by the server on demand. The dynamic loading feature is what distinguishes "C language" functions from "internal" functions — the actual coding conventions are essentially the same for both. (Hence, the standard internal function library is a rich source of coding examples for user-defined C functions.)

Two different calling conventions are currently used for C functions. The newer "version 1" calling convention is indicated by writing a `PG_FUNCTION_INFO_V1()` macro call for the function, as illustrated below. Lack of such a macro indicates an old-style ("version 0") function. The language name specified in `CREATE FUNCTION` is `C` in either case. Old-style functions are now deprecated because of portability problems and lack of functionality, but they are still supported for compatibility reasons.

35.9.1. Dynamic Loading

The first time a user-defined function in a particular loadable object file is called in a session, the dynamic loader loads that object file into memory so that the function can be called. The `CREATE FUNCTION` for a user-defined C function must therefore specify two pieces of information for the function: the name of the loadable object file, and the C name (link symbol) of the specific function to call within that object file. If the C name is not explicitly specified then it is assumed to be the same as the SQL function name.

The following algorithm is used to locate the shared object file based on the name given in the `CREATE FUNCTION` command:

1. If the name is an absolute path, the given file is loaded.
2. If the name starts with the string `$libdir`, that part is replaced by the PostgreSQL package library directory name, which is determined at build time.
3. If the name does not contain a directory part, the file is searched for in the path specified by the configuration variable dynamic_library_path.
4. Otherwise (the file was not found in the path, or it contains a non-absolute directory part), the dynamic loader will try to take the name as given, which will most likely fail. (It is unreliable to depend on the current working directory.)

If this sequence does not work, the platform-specific shared library file name extension (often `.so`) is appended to the given name and this sequence is tried again. If that fails as well, the load will fail.

It is recommended to locate shared libraries either relative to `$libdir` or through the dynamic library path. This simplifies version upgrades if the new installation is at a different location. The actual directory that `$libdir` stands for can be found out with the command `pg_config --pkglibdir`.

The user ID the PostgreSQL server runs as must be able to traverse the path to the file you intend to load. Making the file or a higher-level directory not readable and/or not executable by the postgres user is a common mistake.

In any case, the file name that is given in the `CREATE FUNCTION` command is recorded literally in the system catalogs, so if the file needs to be loaded again the same procedure is applied.

> **Note:** PostgreSQL will not compile a C function automatically. The object file must be compiled before it is referenced in a `CREATE FUNCTION` command. See Section 35.9.6 for additional information.

To ensure that a dynamically loaded object file is not loaded into an incompatible server, PostgreSQL checks that the file contains a "magic block" with the appropriate contents. This allows the server to detect obvious incompatibilities, such as code compiled for a different major version of PostgreSQL. A magic block is required as of PostgreSQL 8.2. To include a magic block, write this in one (and only one) of the module source files, after having included the header `fmgr.h`:

```
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif
```

The `#ifdef` test can be omitted if the code doesn't need to compile against pre-8.2 PostgreSQL releases.

After it is used for the first time, a dynamically loaded object file is retained in memory. Future calls in the same session to the function(s) in that file will only incur the small overhead of a symbol table lookup. If you need to force a reload of an object file, for example after recompiling it, begin a fresh session.

Optionally, a dynamically loaded file can contain initialization and finalization functions. If the file includes a function named `_PG_init`, that function will be called immediately after loading the file. The function receives no parameters and should return void. If the file includes a function named `_PG_fini`, that function will be called immediately before unloading the file. Likewise, the function receives no parameters and should return void. Note that `_PG_fini` will only be called during an unload of the file, not during process termination. (Presently, unloads are disabled and will never occur, but this may change in the future.)

35.9.2. Base Types in C-Language Functions

To know how to write C-language functions, you need to know how PostgreSQL internally represents base data types and how they can be passed to and from functions. Internally, PostgreSQL regards a base type as a "blob of memory". The user-defined functions that you define over a type in turn define the way that PostgreSQL can operate on it. That is, PostgreSQL will only store and retrieve the data from disk and use your user-defined functions to input, process, and output the data.

Base types can have one of three internal formats:

- pass by value, fixed-length
- pass by reference, fixed-length
- pass by reference, variable-length

By-value types can only be 1, 2, or 4 bytes in length (also 8 bytes, if `sizeof(Datum)` is 8 on your machine). You should be careful to define your types such that they will be the same size (in bytes) on all architectures. For example, the `long` type is dangerous because it is 4 bytes on some machines and 8 bytes on others, whereas `int` type is 4 bytes on most Unix machines. A reasonable implementation of the `int4` type on Unix machines might be:

```
/* 4-byte integer, passed by value */
typedef int int4;
```

(The actual PostgreSQL C code calls this type `int32`, because it is a convention in C that `int`*XX* means *XX bits*. Note therefore also that the C type `int8` is 1 byte in size. The SQL type `int8` is called `int64` in C. See also Table 35-1.)

On the other hand, fixed-length types of any size can be passed by-reference. For example, here is a sample implementation of a PostgreSQL type:

```
/* 16-byte structure, passed by reference */
typedef struct
{
    double  x, y;
} Point;
```

Only pointers to such types can be used when passing them in and out of PostgreSQL functions. To return a value of such a type, allocate the right amount of memory with `palloc`, fill in the allocated memory, and return a pointer to it. (Also, if you just want to return the same value as one of your input arguments that's of the same data type, you can skip the extra `palloc` and just return the pointer to the input value.)

Finally, all variable-length types must also be passed by reference. All variable-length types must begin with an opaque length field of exactly 4 bytes, which will be set by `SET_VARSIZE`; never set this field directly! All data to be stored within that type must be located in the memory immediately following that length field. The length field contains the total length of the structure, that is, it includes the size of the length field itself.

Another important point is to avoid leaving any uninitialized bits within data type values; for example, take care to zero out any alignment padding bytes that might be present in structs. Without this, logically-equivalent constants of your data type might be seen as unequal by the planner, leading to inefficient (though not incorrect) plans.

> **Warning**
>
> *Never* modify the contents of a pass-by-reference input value. If you do so you are likely to corrupt on-disk data, since the pointer you are given might point directly into a disk buffer. The sole exception to this rule is explained in Section 35.10.

As an example, we can define the type `text` as follows:

```
typedef struct {
    int32 length;
    char data[FLEXIBLE_ARRAY_MEMBER];
} text;
```

The `[FLEXIBLE_ARRAY_MEMBER]` notation means that the actual length of the data part is not specified by this declaration.

When manipulating variable-length types, we must be careful to allocate the correct amount of memory and set the length field correctly. For example, if we wanted to store 40 bytes in a `text` structure, we might use a code fragment like this:

```
#include "postgres.h"
...
char buffer[40]; /* our source data */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
SET_VARSIZE(destination, VARHDRSZ + 40);
memcpy(destination->data, buffer, 40);
...
```

`VARHDRSZ` is the same as `sizeof(int32)`, but it's considered good style to use the macro `VARHDRSZ` to refer to the size of the overhead for a variable-length type. Also, the length field *must* be set using the `SET_VARSIZE` macro, not by simple assignment.

Table 35-1 specifies which C type corresponds to which SQL type when writing a C-language function that uses a built-in type of PostgreSQL. The "Defined In" column gives the header file that needs to be included to get the type definition. (The actual definition might be in a different file that is included by the listed file. It is recommended that users stick to the defined interface.) Note that you should always include `postgres.h` first in any source file, because it declares a number of things that you will need anyway.

Table 35-1. Equivalent C Types for Built-in SQL Types

SQL Type	C Type	Defined In
`abstime`	`AbsoluteTime`	`utils/nabstime.h`
`bigint` (`int8`)	`int64`	`postgres.h`
`boolean`	`bool`	`postgres.h` (maybe compiler built-in)
`box`	`BOX*`	`utils/geo_decls.h`
`bytea`	`bytea*`	`postgres.h`

SQL Type	C Type	Defined In
"char"	char	(compiler built-in)
character	BpChar*	postgres.h
cid	CommandId	postgres.h
date	DateADT	utils/date.h
smallint (int2)	int16	postgres.h
int2vector	int2vector*	postgres.h
integer (int4)	int32	postgres.h
real (float4)	float4*	postgres.h
double precision (float8)	float8*	postgres.h
interval	Interval*	datatype/timestamp.h
lseg	LSEG*	utils/geo_decls.h
name	Name	postgres.h
oid	Oid	postgres.h
oidvector	oidvector*	postgres.h
path	PATH*	utils/geo_decls.h
point	POINT*	utils/geo_decls.h
regproc	regproc	postgres.h
reltime	RelativeTime	utils/nabstime.h
text	text*	postgres.h
tid	ItemPointer	storage/itemptr.h
time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
timestamp	Timestamp	datatype/timestamp.h
tinterval	TimeInterval	utils/nabstime.h
varchar	VarChar*	postgres.h
xid	TransactionId	postgres.h

Now that we've gone over all of the possible structures for base types, we can show some examples of real functions.

35.9.3. Version 0 Calling Conventions

We present the "old style" calling convention first — although this approach is now deprecated, it's easier to get a handle on initially. In the version-0 method, the arguments and result of the C function are just declared in normal C style, but being careful to use the C representation of each SQL data type as shown above.

Here are some examples:

```
#include "postgres.h"
#include <string.h>
#include "utils/geo_decls.h"
```

```
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

/* by value */

int
add_one(int arg)
{
    return arg + 1;
}

/* by reference, fixed length */

float8 *
add_one_float8(float8 *arg)
{
    float8    *result = (float8 *) palloc(sizeof(float8));

    *result = *arg + 1.0;

    return result;
}

Point *
makepoint(Point *pointx, Point *pointy)
{
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    return new_point;
}

/* by reference, variable length */

text *
copytext(text *t)
{
    /*
     * VARSIZE is the total size of the struct in bytes.
     */
    text *new_t = (text *) palloc(VARSIZE(t));
    SET_VARSIZE(new_t, VARSIZE(t));
    /*
     * VARDATA is a pointer to the data region of the struct.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARSIZE(t) - VARHDRSZ);  /* how many bytes */
    return new_t;
```

```
}

text *
concat_text(text *arg1, text *arg2)
{
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    return new_text;
}
```

Supposing that the above code has been prepared in file `funcs.c` and compiled into a shared object, we could define the functions to PostgreSQL with commands like this:

```
CREATE FUNCTION add_one(integer) RETURNS integer
     AS 'DIRECTORY/funcs', 'add_one'
     LANGUAGE C STRICT;

-- note overloading of SQL function name "add_one"
CREATE FUNCTION add_one(double precision) RETURNS double precision
     AS 'DIRECTORY/funcs', 'add_one_float8'
     LANGUAGE C STRICT;

CREATE FUNCTION makepoint(point, point) RETURNS point
     AS 'DIRECTORY/funcs', 'makepoint'
     LANGUAGE C STRICT;

CREATE FUNCTION copytext(text) RETURNS text
     AS 'DIRECTORY/funcs', 'copytext'
     LANGUAGE C STRICT;

CREATE FUNCTION concat_text(text, text) RETURNS text
     AS 'DIRECTORY/funcs', 'concat_text'
     LANGUAGE C STRICT;
```

Here, *DIRECTORY* stands for the directory of the shared library file (for instance the PostgreSQL tutorial directory, which contains the code for the examples used in this section). (Better style would be to use just `'funcs'` in the `AS` clause, after having added *DIRECTORY* to the search path. In any case, we can omit the system-specific extension for a shared library, commonly `.so` or `.sl`.)

Notice that we have specified the functions as "strict", meaning that the system should automatically assume a null result if any input value is null. By doing this, we avoid having to check for null inputs in the function code. Without this, we'd have to check for null values explicitly, by checking for a null pointer for each pass-by-reference argument. (For pass-by-value arguments, we don't even have a way to check!)

Although this calling convention is simple to use, it is not very portable; on some architectures there are problems with passing data types that are smaller than `int` this way. Also, there is no simple way to return a null result, nor to cope with null arguments in any way other than making the function strict. The version-1 convention, presented next, overcomes these objections.

35.9.4. Version 1 Calling Conventions

The version-1 calling convention relies on macros to suppress most of the complexity of passing arguments and results. The C declaration of a version-1 function is always:

```
Datum funcname(PG_FUNCTION_ARGS)
```

In addition, the macro call:

```
PG_FUNCTION_INFO_V1(funcname);
```

must appear in the same source file. (Conventionally, it's written just before the function itself.) This macro call is not needed for `internal`-language functions, since PostgreSQL assumes that all internal functions use the version-1 convention. It is, however, required for dynamically-loaded functions.

In a version-1 function, each actual argument is fetched using a `PG_GETARG_`*xxx*`()` macro that corresponds to the argument's data type, and the result is returned using a `PG_RETURN_`*xxx*`()` macro for the return type. `PG_GETARG_`*xxx*`()` takes as its argument the number of the function argument to fetch, where the count starts at 0. `PG_RETURN_`*xxx*`()` takes as its argument the actual value to return.

Here we show the same functions as above, coded in version-1 style:

```
#include "postgres.h"
#include <string.h>
#include "fmgr.h"
#include "utils/geo_decls.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

/* by value */

PG_FUNCTION_INFO_V1(add_one);

Datum
add_one(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);

    PG_RETURN_INT32(arg + 1);
}

/* by reference, fixed length */

PG_FUNCTION_INFO_V1(add_one_float8);
```

```
Datum
add_one_float8(PG_FUNCTION_ARGS)
{
    /* The macros for FLOAT8 hide its pass-by-reference nature. */
    float8   arg = PG_GETARG_FLOAT8(0);

    PG_RETURN_FLOAT8(arg + 1.0);
}

PG_FUNCTION_INFO_V1(makepoint);

Datum
makepoint(PG_FUNCTION_ARGS)
{
    /* Here, the pass-by-reference nature of Point is not hidden. */
    Point     *pointx = PG_GETARG_POINT_P(0);
    Point     *pointy = PG_GETARG_POINT_P(1);
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    PG_RETURN_POINT_P(new_point);
}

/* by reference, variable length */

PG_FUNCTION_INFO_V1(copytext);

Datum
copytext(PG_FUNCTION_ARGS)
{
    text     *t = PG_GETARG_TEXT_P(0);
    /*
     * VARSIZE is the total size of the struct in bytes.
     */
    text     *new_t = (text *) palloc(VARSIZE(t));
    SET_VARSIZE(new_t, VARSIZE(t));
    /*
     * VARDATA is a pointer to the data region of the struct.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARSIZE(t) - VARHDRSZ);  /* how many bytes */
    PG_RETURN_TEXT_P(new_t);
}

PG_FUNCTION_INFO_V1(concat_text);

Datum
concat_text(PG_FUNCTION_ARGS)
{
    text  *arg1 = PG_GETARG_TEXT_P(0);
```

```
    text  *arg2 = PG_GETARG_TEXT_P(1);
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    SET_VARSIZE(new_text, new_text_size);
    memcpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1) - VARHDRSZ);
    memcpy(VARDATA(new_text) + (VARSIZE(arg1) - VARHDRSZ),
           VARDATA(arg2), VARSIZE(arg2) - VARHDRSZ);
    PG_RETURN_TEXT_P(new_text);
}
```

The `CREATE FUNCTION` commands are the same as for the version-0 equivalents.

At first glance, the version-1 coding conventions might appear to be just pointless obscurantism. They do, however, offer a number of improvements, because the macros can hide unnecessary detail. An example is that in coding `add_one_float8`, we no longer need to be aware that `float8` is a pass-by-reference type. Another example is that the `GETARG` macros for variable-length types allow for more efficient fetching of "toasted" (compressed or out-of-line) values.

One big improvement in version-1 functions is better handling of null inputs and results. The macro `PG_ARGISNULL(`*`n`*`)` allows a function to test whether each input is null. (Of course, doing this is only necessary in functions not declared "strict".) As with the `PG_GETARG_`*`xxx`*`()` macros, the input arguments are counted beginning at zero. Note that one should refrain from executing `PG_GETARG_`*`xxx`*`()` until one has verified that the argument isn't null. To return a null result, execute `PG_RETURN_NULL()`; this works in both strict and nonstrict functions.

Other options provided in the new-style interface are two variants of the `PG_GETARG_`*`xxx`*`()` macros. The first of these, `PG_GETARG_`*`xxx`*`_COPY()`, guarantees to return a copy of the specified argument that is safe for writing into. (The normal macros will sometimes return a pointer to a value that is physically stored in a table, which must not be written to. Using the `PG_GETARG_`*`xxx`*`_COPY()` macros guarantees a writable result.) The second variant consists of the `PG_GETARG_`*`xxx`*`_SLICE()` macros which take three arguments. The first is the number of the function argument (as above). The second and third are the offset and length of the segment to be returned. Offsets are counted from zero, and a negative length requests that the remainder of the value be returned. These macros provide more efficient access to parts of large values in the case where they have storage type "external". (The storage type of a column can be specified using `ALTER TABLE` *`tablename`* `ALTER COLUMN` *`colname`* `SET STORAGE` *`storagetype`*. *`storagetype`* is one of `plain`, `external`, `extended`, or `main`.)

Finally, the version-1 function call conventions make it possible to return set results (Section 35.9.9) and implement trigger functions (Chapter 36) and procedural-language call handlers (Chapter 53). Version-1 code is also more portable than version-0, because it does not break restrictions on function call protocol in the C standard. For more details see `src/backend/utils/fmgr/README` in the source distribution.

35.9.5. Writing Code

Before we turn to the more advanced topics, we should discuss some coding rules for PostgreSQL C-language functions. While it might be possible to load functions written in languages other than C into PostgreSQL, this is usually difficult (when it is possible at all) because other languages, such as C++, FORTRAN, or Pascal often do not follow the same calling convention as C. That is, other languages do

not pass argument and return values between functions in the same way. For this reason, we will assume that your C-language functions are actually written in C.

The basic rules for writing and building C functions are as follows:

- Use `pg_config --includedir-server` to find out where the PostgreSQL server header files are installed on your system (or the system that your users will be running on).
- Compiling and linking your code so that it can be dynamically loaded into PostgreSQL always requires special flags. See Section 35.9.6 for a detailed explanation of how to do it for your particular operating system.
- Remember to define a "magic block" for your shared library, as described in Section 35.9.1.
- When allocating memory, use the PostgreSQL functions `palloc` and `pfree` instead of the corresponding C library functions `malloc` and `free`. The memory allocated by `palloc` will be freed automatically at the end of each transaction, preventing memory leaks.
- Always zero the bytes of your structures using `memset` (or allocate them with `palloc0` in the first place). Even if you assign to each field of your structure, there might be alignment padding (holes in the structure) that contain garbage values. Without this, it's difficult to support hash indexes or hash joins, as you must pick out only the significant bits of your data structure to compute a hash. The planner also sometimes relies on comparing constants via bitwise equality, so you can get undesirable planning results if logically-equivalent values aren't bitwise equal.
- Most of the internal PostgreSQL types are declared in `postgres.h`, while the function manager interfaces (`PG_FUNCTION_ARGS`, etc.) are in `fmgr.h`, so you will need to include at least these two files. For portability reasons it's best to include `postgres.h` *first*, before any other system or user header files. Including `postgres.h` will also include `elog.h` and `palloc.h` for you.
- Symbol names defined within object files must not conflict with each other or with symbols defined in the PostgreSQL server executable. You will have to rename your functions or variables if you get error messages to this effect.

35.9.6. Compiling and Linking Dynamically-loaded Functions

Before you are able to use your PostgreSQL extension functions written in C, they must be compiled and linked in a special way to produce a file that can be dynamically loaded by the server. To be precise, a *shared library* needs to be created.

For information beyond what is contained in this section you should read the documentation of your operating system, in particular the manual pages for the C compiler, `cc`, and the link editor, `ld`. In addition, the PostgreSQL source code contains several working examples in the `contrib` directory. If you rely on these examples you will make your modules dependent on the availability of the PostgreSQL source code, however.

Creating shared libraries is generally analogous to linking executables: first the source files are compiled into object files, then the object files are linked together. The object files need to be created as *position-independent code* (PIC), which conceptually means that they can be placed at an arbitrary location in memory when they are loaded by the executable. (Object files intended for executables are usually not

compiled that way.) The command to link a shared library contains special flags to distinguish it from linking an executable (at least in theory — on some systems the practice is much uglier).

In the following examples we assume that your source code is in a file `foo.c` and we will create a shared library `foo.so`. The intermediate object file will be called `foo.o` unless otherwise noted. A shared library can contain more than one object file, but we only use one here.

FreeBSD

The compiler flag to create PIC is `-fPIC`. To create shared libraries the compiler flag is `-shared`.

```
gcc -fPIC -c foo.c
gcc -shared -o foo.so foo.o
```

This is applicable as of version 3.0 of FreeBSD.

HP-UX

The compiler flag of the system compiler to create PIC is `+z`. When using GCC it's `-fPIC`. The linker flag for shared libraries is `-b`. So:

```
cc +z -c foo.c
```

or:

```
gcc -fPIC -c foo.c
```

and then:

```
ld -b -o foo.sl foo.o
```

HP-UX uses the extension `.sl` for shared libraries, unlike most other systems.

Linux

The compiler flag to create PIC is `-fPIC`. The compiler flag to create a shared library is `-shared`. A complete example looks like this:

```
cc -fPIC -c foo.c
cc -shared -o foo.so foo.o
```

OS X

Here is an example. It assumes the developer tools are installed.

```
cc -c foo.c
cc -bundle -flat_namespace -undefined suppress -o foo.so foo.o
```

NetBSD

The compiler flag to create PIC is `-fPIC`. For ELF systems, the compiler with the flag `-shared` is used to link shared libraries. On the older non-ELF systems, `ld -Bshareable` is used.

```
gcc -fPIC -c foo.c
gcc -shared -o foo.so foo.o
```

OpenBSD

The compiler flag to create PIC is `-fPIC`. `ld -Bshareable` is used to link shared libraries.

```
gcc -fPIC -c foo.c
ld -Bshareable -o foo.so foo.o
```

Solaris

The compiler flag to create PIC is `-KPIC` with the Sun compiler and `-fPIC` with GCC. To link shared libraries, the compiler option is `-G` with either compiler or alternatively `-shared` with GCC.

```
cc -KPIC -c foo.c
cc -G -o foo.so foo.o
```

or

```
gcc -fPIC -c foo.c
gcc -G -o foo.so foo.o
```

UnixWare

The compiler flag to create PIC is `-K PIC` with the SCO compiler and `-fpic` with GCC. To link shared libraries, the compiler option is `-G` with the SCO compiler and `-shared` with GCC.

```
cc -K PIC -c foo.c
cc -G -o foo.so foo.o
```

or

```
gcc -fpic -c foo.c
gcc -shared -o foo.so foo.o
```

Tip: If this is too complicated for you, you should consider using GNU Libtool[1], which hides the platform differences behind a uniform interface.

The resulting shared library file can then be loaded into PostgreSQL. When specifying the file name to the `CREATE FUNCTION` command, one must give it the name of the shared library file, not the intermediate object file. Note that the system's standard shared-library extension (usually `.so` or `.sl`) can be omitted from the `CREATE FUNCTION` command, and normally should be omitted for best portability.

Refer back to Section 35.9.1 about where the server expects to find the shared library files.

35.9.7. Composite-type Arguments

Composite types do not have a fixed layout like C structures. Instances of a composite type can contain null fields. In addition, composite types that are part of an inheritance hierarchy can have different fields than other members of the same inheritance hierarchy. Therefore, PostgreSQL provides a function interface for accessing fields of composite types from C.

Suppose we want to write a function to answer the query:

```
SELECT name, c_overpaid(emp, 1500) AS overpaid
    FROM emp
    WHERE name = 'Bill' OR name = 'Sam';
```

Using call conventions version 0, we can define `c_overpaid` as:

```
#include "postgres.h"
#include "executor/executor.h"  /* for GetAttributeByName() */
```

1. https://www.gnu.org/software/libtool/

```
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

bool
c_overpaid(HeapTupleHeader t, /* the current row of emp */
           int32 limit)
{
    bool isnull;
    int32 salary;

    salary = DatumGetInt32(GetAttributeByName(t, "salary", &isnull));
    if (isnull)
        return false;
    return salary > limit;
}
```

In version-1 coding, the above would look like this:

```
#include "postgres.h"
#include "executor/executor.h"  /* for GetAttributeByName() */

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(c_overpaid);

Datum
c_overpaid(PG_FUNCTION_ARGS)
{
    HeapTupleHeader  t = PG_GETARG_HEAPTUPLEHEADER(0);
    int32            limit = PG_GETARG_INT32(1);
    bool isnull;
    Datum salary;

    salary = GetAttributeByName(t, "salary", &isnull);
    if (isnull)
        PG_RETURN_BOOL(false);
    /* Alternatively, we might prefer to do PG_RETURN_NULL() for null salary. */

    PG_RETURN_BOOL(DatumGetInt32(salary) > limit);
}
```

`GetAttributeByName` is the PostgreSQL system function that returns attributes out of the specified row. It has three arguments: the argument of type `HeapTupleHeader` passed into the function, the name of the desired attribute, and a return parameter that tells whether the attribute is null. `GetAttributeByName` returns a `Datum` value that you can convert to the proper data type by using the appropriate `DatumGet`*`XXX`*`()` macro. Note that the return value is meaningless if the null flag is set; always check the null flag before trying to do anything with the result.

There is also `GetAttributeByNum`, which selects the target attribute by column number instead of name.

The following command declares the function `c_overpaid` in SQL:

```
CREATE FUNCTION c_overpaid(emp, integer) RETURNS boolean
    AS 'DIRECTORY/funcs', 'c_overpaid'
    LANGUAGE C STRICT;
```

Notice we have used `STRICT` so that we did not have to check whether the input arguments were NULL.

35.9.8. Returning Rows (Composite Types)

To return a row or composite-type value from a C-language function, you can use a special API that provides macros and functions to hide most of the complexity of building composite data types. To use this API, the source file must include:

```
#include "funcapi.h"
```

There are two ways you can build a composite data value (henceforth a "tuple"): you can build it from an array of Datum values, or from an array of C strings that can be passed to the input conversion functions of the tuple's column data types. In either case, you first need to obtain or construct a `TupleDesc` descriptor for the tuple structure. When working with Datums, you pass the `TupleDesc` to `BlessTupleDesc`, and then call `heap_form_tuple` for each row. When working with C strings, you pass the `TupleDesc` to `TupleDescGetAttInMetadata`, and then call `BuildTupleFromCStrings` for each row. In the case of a function returning a set of tuples, the setup steps can all be done once during the first call of the function.

Several helper functions are available for setting up the needed `TupleDesc`. The recommended way to do this in most functions returning composite values is to call:

```
TypeFuncClass get_call_result_type(FunctionCallInfo fcinfo,
                                   Oid *resultTypeId,
                                   TupleDesc *resultTupleDesc)
```

passing the same `fcinfo` struct passed to the calling function itself. (This of course requires that you use the version-1 calling conventions.) `resultTypeId` can be specified as `NULL` or as the address of a local variable to receive the function's result type OID. `resultTupleDesc` should be the address of a local `TupleDesc` variable. Check that the result is `TYPEFUNC_COMPOSITE`; if so, `resultTupleDesc` has been filled with the needed `TupleDesc`. (If it is not, you can report an error along the lines of "function returning record called in context that cannot accept type record".)

> **Tip:** `get_call_result_type` can resolve the actual type of a polymorphic function result; so it is useful in functions that return scalar polymorphic results, not only functions that return composites. The `resultTypeId` output is primarily useful for functions returning polymorphic scalars.

> **Note:** `get_call_result_type` has a sibling `get_expr_result_type`, which can be used to resolve the expected output type for a function call represented by an expression tree. This can be used when trying to determine the result type from outside the function itself. There is also `get_func_result_type`, which can be used when only the function's OID is available. However these functions are not able to deal with functions declared to return `record`, and

`get_func_result_type` cannot resolve polymorphic types, so you should preferentially use `get_call_result_type`.

Older, now-deprecated functions for obtaining `TupleDesc`s are:

```
TupleDesc RelationNameGetTupleDesc(const char *relname)
```

to get a `TupleDesc` for the row type of a named relation, and:

```
TupleDesc TypeGetTupleDesc(Oid typeoid, List *colaliases)
```

to get a `TupleDesc` based on a type OID. This can be used to get a `TupleDesc` for a base or composite type. It will not work for a function that returns `record`, however, and it cannot resolve polymorphic types.

Once you have a `TupleDesc`, call:

```
TupleDesc BlessTupleDesc(TupleDesc tupdesc)
```

if you plan to work with Datums, or:

```
AttInMetadata *TupleDescGetAttInMetadata(TupleDesc tupdesc)
```

if you plan to work with C strings. If you are writing a function returning set, you can save the results of these functions in the `FuncCallContext` structure — use the `tuple_desc` or `attinmeta` field respectively.

When working with Datums, use:

```
HeapTuple heap_form_tuple(TupleDesc tupdesc, Datum *values, bool *isnull)
```

to build a `HeapTuple` given user data in Datum form.

When working with C strings, use:

```
HeapTuple BuildTupleFromCStrings(AttInMetadata *attinmeta, char **values)
```

to build a `HeapTuple` given user data in C string form. `values` is an array of C strings, one for each attribute of the return row. Each C string should be in the form expected by the input function of the attribute data type. In order to return a null value for one of the attributes, the corresponding pointer in the `values` array should be set to `NULL`. This function will need to be called again for each row you return.

Once you have built a tuple to return from your function, it must be converted into a `Datum`. Use:

```
HeapTupleGetDatum(HeapTuple tuple)
```

to convert a `HeapTuple` into a valid Datum. This `Datum` can be returned directly if you intend to return just a single row, or it can be used as the current return value in a set-returning function.

An example appears in the next section.

35.9.9. Returning Sets

C-language functions have two options for returning sets (multiple rows). In one method, called *ValuePerCall* mode, a set-returning function is called repeatedly (passing the same arguments each time) and it returns one new row on each call, until it has no more rows to return and signals that by returning NULL. The set-returning function (SRF) must therefore save enough state across calls to remember what it was doing and return the correct next item on each call. In the other method, called *Materialize* mode, a SRF fills and returns a tuplestore object containing its entire result; then only one call occurs for the whole result, and no inter-call state is needed.

When using ValuePerCall mode, it is important to remember that the query is not guaranteed to be run to completion; that is, due to options such as `LIMIT`, the executor might stop making calls to the set-returning function before all rows have been fetched. This means it is not safe to perform cleanup activities in the last call, because that might not ever happen. It's recommended to use Materialize mode for functions that need access to external resources, such as file descriptors.

The remainder of this section documents a set of helper macros that are commonly used (though not required to be used) for SRFs using ValuePerCall mode. Additional details about Materialize mode can be found in `src/backend/utils/fmgr/README`. Also, the `contrib` modules in the PostgreSQL source distribution contain many examples of SRFs using both ValuePerCall and Materialize mode.

To use the ValuePerCall support macros described here, include `funcapi.h`. These macros work with a structure `FuncCallContext` that contains the state that needs to be saved across calls. Within the calling SRF, `fcinfo->flinfo->fn_extra` is used to hold a pointer to `FuncCallContext` across calls. The macros automatically fill that field on first use, and expect to find the same pointer there on subsequent uses.

```
typedef struct
{
    /*
     * Number of times we've been called before
     *
     * call_cntr is initialized to 0 for you by SRF_FIRSTCALL_INIT(), and
     * incremented for you every time SRF_RETURN_NEXT() is called.
     */
    uint32 call_cntr;

    /*
     * OPTIONAL maximum number of calls
     *
     * max_calls is here for convenience only and setting it is optional.
     * If not set, you must provide alternative means to know when the
     * function is done.
     */
    uint32 max_calls;

    /*
     * OPTIONAL pointer to result slot
     *
     * This is obsolete and only present for backward compatibility, viz,
     * user-defined SRFs that use the deprecated TupleDescGetSlot().
     */
    TupleTableSlot *slot;
```

```
    /*
     * OPTIONAL pointer to miscellaneous user-provided context information
     *
     * user_fctx is for use as a pointer to your own data to retain
     * arbitrary context information between calls of your function.
     */
    void *user_fctx;

    /*
     * OPTIONAL pointer to struct containing attribute type input metadata
     *
     * attinmeta is for use when returning tuples (i.e., composite data types)
     * and is not used when returning base data types. It is only needed
     * if you intend to use BuildTupleFromCStrings() to create the return
     * tuple.
     */
    AttInMetadata *attinmeta;

    /*
     * memory context used for structures that must live for multiple calls
     *
     * multi_call_memory_ctx is set by SRF_FIRSTCALL_INIT() for you, and used
     * by SRF_RETURN_DONE() for cleanup. It is the most appropriate memory
     * context for any memory that is to be reused across multiple calls
     * of the SRF.
     */
    MemoryContext multi_call_memory_ctx;

    /*
     * OPTIONAL pointer to struct containing tuple description
     *
     * tuple_desc is for use when returning tuples (i.e., composite data types)
     * and is only needed if you are going to build the tuples with
     * heap_form_tuple() rather than with BuildTupleFromCStrings().  Note that
     * the TupleDesc pointer stored here should usually have been run through
     * BlessTupleDesc() first.
     */
    TupleDesc tuple_desc;

} FuncCallContext;
```

The macros to be used by an SRF using this infrastructure are:

`SRF_IS_FIRSTCALL()`

Use this to determine if your function is being called for the first or a subsequent time. On the first call (only), call:

`SRF_FIRSTCALL_INIT()`

to initialize the `FuncCallContext`. On every function call, including the first, call:

```
SRF_PERCALL_SETUP()
```

to set up for using the `FuncCallContext`.

If your function has data to return in the current call, use:

```
SRF_RETURN_NEXT(funcctx, result)
```

to return it to the caller. (`result` must be of type `Datum`, either a single value or a tuple prepared as described above.) Finally, when your function is finished returning data, use:

```
SRF_RETURN_DONE(funcctx)
```

to clean up and end the SRF.

The memory context that is current when the SRF is called is a transient context that will be cleared between calls. This means that you do not need to call `pfree` on everything you allocated using `palloc`; it will go away anyway. However, if you want to allocate any data structures to live across calls, you need to put them somewhere else. The memory context referenced by `multi_call_memory_ctx` is a suitable location for any data that needs to survive until the SRF is finished running. In most cases, this means that you should switch into `multi_call_memory_ctx` while doing the first-call setup. Use `funcctx->user_fctx` to hold a pointer to any such cross-call data structures. (Data you allocate in `multi_call_memory_ctx` will go away automatically when the query ends, so it is not necessary to free that data manually, either.)

> **Warning**
>
> While the actual arguments to the function remain unchanged between calls, if you detoast the argument values (which is normally done transparently by the `PG_GETARG_xxx` macro) in the transient context then the detoasted copies will be freed on each cycle. Accordingly, if you keep references to such values in your `user_fctx`, you must either copy them into the `multi_call_memory_ctx` after detoasting, or ensure that you detoast the values only in that context.

A complete pseudo-code example looks like the following:

```
Datum
my_set_returning_function(PG_FUNCTION_ARGS)
{
    FuncCallContext  *funcctx;
    Datum             result;
    further declarations as needed

    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext oldcontext;

        funcctx = SRF_FIRSTCALL_INIT();
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);
        /* One-time setup code appears here: */
        user code
        if returning composite
            build TupleDesc, and perhaps AttInMetadata
```

```
        endif returning composite
        user code
        MemoryContextSwitchTo(oldcontext);
    }

    /* Each-time setup code appears here: */
    user code
    funcctx = SRF_PERCALL_SETUP();
    user code

    /* this is just one way we might test whether we are done: */
    if (funcctx->call_cntr < funcctx->max_calls)
    {
        /* Here we want to return another item: */
        user code
        obtain result Datum
        SRF_RETURN_NEXT(funcctx, result);
    }
    else
    {
        /* Here we are done returning items, so just report that fact. */
        /* (Resist the temptation to put cleanup code here.) */
        SRF_RETURN_DONE(funcctx);
    }
}
```

A complete example of a simple SRF returning a composite type looks like:

```
PG_FUNCTION_INFO_V1(retcomposite);

Datum
retcomposite(PG_FUNCTION_ARGS)
{
    FuncCallContext     *funcctx;
    int                  call_cntr;
    int                  max_calls;
    TupleDesc            tupdesc;
    AttInMetadata       *attinmeta;

    /* stuff done only on the first call of the function */
    if (SRF_IS_FIRSTCALL())
    {
        MemoryContext   oldcontext;

        /* create a function context for cross-call persistence */
        funcctx = SRF_FIRSTCALL_INIT();

        /* switch to memory context appropriate for multiple function calls */
        oldcontext = MemoryContextSwitchTo(funcctx->multi_call_memory_ctx);

        /* total number of tuples to be returned */
        funcctx->max_calls = PG_GETARG_UINT32(0);
```

```
    /* Build a tuple descriptor for our result type */
    if (get_call_result_type(fcinfo, NULL, &tupdesc) != TYPEFUNC_COMPOSITE)
        ereport(ERROR,
                (errcode(ERRCODE_FEATURE_NOT_SUPPORTED),
                 errmsg("function returning record called in context "
                        "that cannot accept type record")));

    /*
     * generate attribute metadata needed later to produce tuples from raw
     * C strings
     */
    attinmeta = TupleDescGetAttInMetadata(tupdesc);
    funcctx->attinmeta = attinmeta;

    MemoryContextSwitchTo(oldcontext);
}

/* stuff done on every call of the function */
funcctx = SRF_PERCALL_SETUP();

call_cntr = funcctx->call_cntr;
max_calls = funcctx->max_calls;
attinmeta = funcctx->attinmeta;

if (call_cntr < max_calls)    /* do when there is more left to send */
{
    char       **values;
    HeapTuple    tuple;
    Datum        result;

    /*
     * Prepare a values array for building the returned tuple.
     * This should be an array of C strings which will
     * be processed later by the type input functions.
     */
    values = (char **) palloc(3 * sizeof(char *));
    values[0] = (char *) palloc(16 * sizeof(char));
    values[1] = (char *) palloc(16 * sizeof(char));
    values[2] = (char *) palloc(16 * sizeof(char));

    snprintf(values[0], 16, "%d", 1 * PG_GETARG_INT32(1));
    snprintf(values[1], 16, "%d", 2 * PG_GETARG_INT32(1));
    snprintf(values[2], 16, "%d", 3 * PG_GETARG_INT32(1));

    /* build a tuple */
    tuple = BuildTupleFromCStrings(attinmeta, values);

    /* make the tuple into a datum */
    result = HeapTupleGetDatum(tuple);

    /* clean up (this is not really necessary) */
    pfree(values[0]);
```

```
        pfree(values[1]);
        pfree(values[2]);
        pfree(values);

        SRF_RETURN_NEXT(funcctx, result);
    }
    else    /* do when there is no more left */
    {
        SRF_RETURN_DONE(funcctx);
    }
}
```

One way to declare this function in SQL is:

```
CREATE TYPE __retcomposite AS (f1 integer, f2 integer, f3 integer);

CREATE OR REPLACE FUNCTION retcomposite(integer, integer)
    RETURNS SETOF __retcomposite
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

A different way is to use OUT parameters:

```
CREATE OR REPLACE FUNCTION retcomposite(IN integer, IN integer,
    OUT f1 integer, OUT f2 integer, OUT f3 integer)
    RETURNS SETOF record
    AS 'filename', 'retcomposite'
    LANGUAGE C IMMUTABLE STRICT;
```

Notice that in this method the output type of the function is formally an anonymous `record` type.

35.9.10. Polymorphic Arguments and Return Types

C-language functions can be declared to accept and return the polymorphic types `anyelement`, `anyarray`, `anynonarray`, `anyenum`, and `anyrange`. See Section 35.2.5 for a more detailed explanation of polymorphic functions. When function arguments or return types are defined as polymorphic types, the function author cannot know in advance what data type it will be called with, or need to return. There are two routines provided in `fmgr.h` to allow a version-1 C function to discover the actual data types of its arguments and the type it is expected to return. The routines are called `get_fn_expr_rettype(FmgrInfo *flinfo)` and `get_fn_expr_argtype(FmgrInfo *flinfo, int argnum)`. They return the result or argument type OID, or `InvalidOid` if the information is not available. The structure `flinfo` is normally accessed as `fcinfo->flinfo`. The parameter `argnum` is zero based. `get_call_result_type` can also be used as an alternative to `get_fn_expr_rettype`. There is also `get_fn_expr_variadic`, which can be used to find out whether variadic arguments have been merged into an array. This is primarily useful for `VARIADIC "any"` functions, since such merging will always have occurred for variadic functions taking ordinary array types.

For example, suppose we want to write a function to accept a single element of any type, and return a one-dimensional array of that type:

```
PG_FUNCTION_INFO_V1(make_array);
```

```
Datum
make_array(PG_FUNCTION_ARGS)
{
    ArrayType  *result;
    Oid         element_type = get_fn_expr_argtype(fcinfo->flinfo, 0);
    Datum       element;
    bool        isnull;
    int16       typlen;
    bool        typbyval;
    char        typalign;
    int         ndims;
    int         dims[MAXDIM];
    int         lbs[MAXDIM];

    if (!OidIsValid(element_type))
        elog(ERROR, "could not determine data type of input");

    /* get the provided element, being careful in case it's NULL */
    isnull = PG_ARGISNULL(0);
    if (isnull)
        element = (Datum) 0;
    else
        element = PG_GETARG_DATUM(0);

    /* we have one dimension */
    ndims = 1;
    /* and one element */
    dims[0] = 1;
    /* and lower bound is 1 */
    lbs[0] = 1;

    /* get required info about the element type */
    get_typlenbyvalalign(element_type, &typlen, &typbyval, &typalign);

    /* now build the array */
    result = construct_md_array(&element, &isnull, ndims, dims, lbs,
                                element_type, typlen, typbyval, typalign);

    PG_RETURN_ARRAYTYPE_P(result);
}
```

The following command declares the function `make_array` in SQL:

```
CREATE FUNCTION make_array(anyelement) RETURNS anyarray
    AS 'DIRECTORY/funcs', 'make_array'
    LANGUAGE C IMMUTABLE;
```

There is a variant of polymorphism that is only available to C-language functions: they can be declared to take parameters of type `"any"`. (Note that this type name must be double-quoted, since it's also a SQL reserved word.) This works like `anyelement` except that it does not constrain different `"any"` arguments

to be the same type, nor do they help determine the function's result type. A C-language function can also declare its final parameter to be `VARIADIC "any"`. This will match one or more actual arguments of any type (not necessarily the same type). These arguments will *not* be gathered into an array as happens with normal variadic functions; they will just be passed to the function separately. The `PG_NARGS()` macro and the methods described above must be used to determine the number of actual arguments and their types when using this feature. Also, users of such a function might wish to use the `VARIADIC` keyword in their function call, with the expectation that the function would treat the array elements as separate arguments. The function itself must implement that behavior if wanted, after using `get_fn_expr_variadic` to detect that the actual argument was marked with `VARIADIC`.

35.9.11. Transform Functions

Some function calls can be simplified during planning based on properties specific to the function. For example, `int4mul(n, 1)` could be simplified to just `n`. To define such function-specific optimizations, write a *transform function* and place its OID in the `protransform` field of the primary function's `pg_proc` entry. The transform function must have the SQL signature `protransform(internal) RETURNS internal`. The argument, actually `FuncExpr *`, is a dummy node representing a call to the primary function. If the transform function's study of the expression tree proves that a simplified expression tree can substitute for all possible concrete calls represented thereby, build and return that simplified expression. Otherwise, return a `NULL` pointer (*not* a SQL null).

We make no guarantee that PostgreSQL will never call the primary function in cases that the transform function could simplify. Ensure rigorous equivalence between the simplified expression and an actual call to the primary function.

Currently, this facility is not exposed to users at the SQL level because of security concerns, so it is only practical to use for optimizing built-in functions.

35.9.12. Shared Memory and LWLocks

Add-ins can reserve LWLocks and an allocation of shared memory on server startup. The add-in's shared library must be preloaded by specifying it in shared_preload_libraries. Shared memory is reserved by calling:

```
void RequestAddinShmemSpace(int size)
```

from your `_PG_init` function.

LWLocks are reserved by calling:

```
void RequestAddinLWLocks(int n)
```

from `_PG_init`.

To avoid possible race-conditions, each backend should use the LWLock `AddinShmemInitLock` when connecting to and initializing its allocation of shared memory, as shown here:

```
static mystruct *ptr = NULL;

if (!ptr)
```

```
{
        bool    found;

        LWLockAcquire(AddinShmemInitLock, LW_EXCLUSIVE);
        ptr = ShmemInitStruct("my struct name", size, &found);
        if (!found)
        {
                initialize contents of shmem area;
                acquire any requested LWLocks using:
                ptr->mylockid = LWLockAssign();
        }
        LWLockRelease(AddinShmemInitLock);
}
```

35.9.13. Using C++ for Extensibility

Although the PostgreSQL backend is written in C, it is possible to write extensions in C++ if these guidelines are followed:

- All functions accessed by the backend must present a C interface to the backend; these C functions can then call C++ functions. For example, `extern C` linkage is required for backend-accessed functions. This is also necessary for any functions that are passed as pointers between the backend and C++ code.
- Free memory using the appropriate deallocation method. For example, most backend memory is allocated using `palloc()`, so use `pfree()` to free it. Using C++ `delete` in such cases will fail.
- Prevent exceptions from propagating into the C code (use a catch-all block at the top level of all `extern C` functions). This is necessary even if the C++ code does not explicitly throw any exceptions, because events like out-of-memory can still throw exceptions. Any exceptions must be caught and appropriate errors passed back to the C interface. If possible, compile C++ with `-fno-exceptions` to eliminate exceptions entirely; in such cases, you must check for failures in your C++ code, e.g., check for NULL returned by `new()`.
- If calling backend functions from C++ code, be sure that the C++ call stack contains only plain old data structures (POD). This is necessary because backend errors generate a distant `longjmp()` that does not properly unroll a C++ call stack with non-POD objects.

In summary, it is best to place C++ code behind a wall of `extern C` functions that interface to the backend, and avoid exception, memory, and call stack leakage.

35.10. User-defined Aggregates

Aggregate functions in PostgreSQL are defined in terms of *state values* and *state transition functions*. That is, an aggregate operates using a state value that is updated as each successive input row is processed. To

define a new aggregate function, one selects a data type for the state value, an initial value for the state, and a state transition function. The state transition function takes the previous state value and the aggregate's input value(s) for the current row, and returns a new state value. A *final function* can also be specified, in case the desired result of the aggregate is different from the data that needs to be kept in the running state value. The final function takes the last state value and returns whatever is wanted as the aggregate result. In principle, the transition and final functions are just ordinary functions that could also be used outside the context of the aggregate. (In practice, it's often helpful for performance reasons to create specialized transition functions that can only work when called as part of an aggregate.)

Thus, in addition to the argument and result data types seen by a user of the aggregate, there is an internal state-value data type that might be different from both the argument and result types.

If we define an aggregate that does not use a final function, we have an aggregate that computes a running function of the column values from each row. `sum` is an example of this kind of aggregate. `sum` starts at zero and always adds the current row's value to its running total. For example, if we want to make a `sum` aggregate to work on a data type for complex numbers, we only need the addition function for that data type. The aggregate definition would be:

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)'
);
```

which we might use like this:

```
SELECT sum(a) FROM test_complex;

   sum
-----------
 (34,53.9)
```

(Notice that we are relying on function overloading: there is more than one aggregate named `sum`, but PostgreSQL can figure out which kind of sum applies to a column of type `complex`.)

The above definition of `sum` will return zero (the initial state value) if there are no nonnull input values. Perhaps we want to return null in that case instead — the SQL standard expects `sum` to behave that way. We can do this simply by omitting the `initcond` phrase, so that the initial state value is null. Ordinarily this would mean that the `sfunc` would need to check for a null state-value input. But for `sum` and some other simple aggregates like `max` and `min`, it is sufficient to insert the first nonnull input value into the state variable and then start applying the transition function at the second nonnull input value. PostgreSQL will do that automatically if the initial state value is null and the transition function is marked "strict" (i.e., not to be called for null inputs).

Another bit of default behavior for a "strict" transition function is that the previous state value is retained unchanged whenever a null input value is encountered. Thus, null values are ignored. If you need some other behavior for null inputs, do not declare your transition function as strict; instead code it to test for null inputs and do whatever is needed.

`avg` (average) is a more complex example of an aggregate. It requires two pieces of running state: the sum of the inputs and the count of the number of inputs. The final result is obtained by dividing these

quantities. Average is typically implemented by using an array as the state value. For example, the built-in implementation of `avg(float8)` looks like:

```
CREATE AGGREGATE avg (float8)
(
    sfunc = float8_accum,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0,0}'
);
```

> **Note:** `float8_accum` requires a three-element array, not just two elements, because it accumulates the sum of squares as well as the sum and count of the inputs. This is so that it can be used for some other aggregates as well as `avg`.

Aggregate function calls in SQL allow `DISTINCT` and `ORDER BY` options that control which rows are fed to the aggregate's transition function and in what order. These options are implemented behind the scenes and are not the concern of the aggregate's support functions.

For further details see the CREATE AGGREGATE command.

35.10.1. Moving-Aggregate Mode

Aggregate functions can optionally support *moving-aggregate mode*, which allows substantially faster execution of aggregate functions within windows with moving frame starting points. (See Section 3.5 and Section 4.2.8 for information about use of aggregate functions as window functions.) The basic idea is that in addition to a normal "forward" transition function, the aggregate provides an *inverse transition function*, which allows rows to be removed from the aggregate's running state value when they exit the window frame. For example a `sum` aggregate, which uses addition as the forward transition function, would use subtraction as the inverse transition function. Without an inverse transition function, the window function mechanism must recalculate the aggregate from scratch each time the frame starting point moves, resulting in run time proportional to the number of input rows times the average frame length. With an inverse transition function, the run time is only proportional to the number of input rows.

The inverse transition function is passed the current state value and the aggregate input value(s) for the earliest row included in the current state. It must reconstruct what the state value would have been if the given input row had never been aggregated, but only the rows following it. This sometimes requires that the forward transition function keep more state than is needed for plain aggregation mode. Therefore, the moving-aggregate mode uses a completely separate implementation from the plain mode: it has its own state data type, its own forward transition function, and its own final function if needed. These can be the same as the plain mode's data type and functions, if there is no need for extra state.

As an example, we could extend the `sum` aggregate given above to support moving-aggregate mode like this:

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
```

```
    stype = complex,
    initcond = '(0,0)',
    msfunc = complex_add,
    minvfunc = complex_sub,
    mstype = complex,
    minitcond = '(0,0)'
);
```

The parameters whose names begin with `m` define the moving-aggregate implementation. Except for the inverse transition function `minvfunc`, they correspond to the plain-aggregate parameters without `m`.

The forward transition function for moving-aggregate mode is not allowed to return null as the new state value. If the inverse transition function returns null, this is taken as an indication that the inverse function cannot reverse the state calculation for this particular input, and so the aggregate calculation will be redone from scratch for the current frame starting position. This convention allows moving-aggregate mode to be used in situations where there are some infrequent cases that are impractical to reverse out of the running state value. The inverse transition function can "punt" on these cases, and yet still come out ahead so long as it can work for most cases. As an example, an aggregate working with floating-point numbers might choose to punt when a `NaN` (not a number) input has to be removed from the running state value.

When writing moving-aggregate support functions, it is important to be sure that the inverse transition function can reconstruct the correct state value exactly. Otherwise there might be user-visible differences in results depending on whether the moving-aggregate mode is used. An example of an aggregate for which adding an inverse transition function seems easy at first, yet where this requirement cannot be met is `sum` over `float4` or `float8` inputs. A naive declaration of `sum(float8)` could be

```
CREATE AGGREGATE unsafe_sum (float8)
(
    stype = float8,
    sfunc = float8pl,
    mstype = float8,
    msfunc = float8pl,
    minvfunc = float8mi
);
```

This aggregate, however, can give wildly different results than it would have without the inverse transition function. For example, consider

```
SELECT
  unsafe_sum(x) OVER (ORDER BY n ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING)
FROM (VALUES (1, 1.0e20::float8),
             (2, 1.0::float8)) AS v (n,x);
```

This query returns `0` as its second result, rather than the expected answer of `1`. The cause is the limited precision of floating-point values: adding `1` to `1e20` results in `1e20` again, and so subtracting `1e20` from that yields `0`, not `1`. Note that this is a limitation of floating-point arithmetic in general, not a limitation of PostgreSQL.

35.10.2. Polymorphic and Variadic Aggregates

Aggregate functions can use polymorphic state transition functions or final functions, so that the same functions can be used to implement multiple aggregates. See Section 35.2.5 for an explanation of polymorphic functions. Going a step further, the aggregate function itself can be specified with polymorphic input type(s) and state type, allowing a single aggregate definition to serve for multiple input data types. Here is an example of a polymorphic aggregate:

```
CREATE AGGREGATE array_accum (anyelement)
(
    sfunc = array_append,
    stype = anyarray,
    initcond = '{}'
);
```

Here, the actual state type for any given aggregate call is the array type having the actual input type as elements. The behavior of the aggregate is to concatenate all the inputs into an array of that type. (Note: the built-in aggregate `array_agg` provides similar functionality, with better performance than this definition would have.)

Here's the output using two different actual data types as arguments:

```
SELECT attrelid::regclass, array_accum(attname)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;

   attrelid    |              array_accum
---------------+---------------------------------------
 pg_tablespace | {spcname,spcowner,spcacl,spcoptions}
(1 row)

SELECT attrelid::regclass, array_accum(atttypid::regtype)
    FROM pg_attribute
    WHERE attnum > 0 AND attrelid = 'pg_tablespace'::regclass
    GROUP BY attrelid;

   attrelid    |        array_accum
---------------+---------------------------
 pg_tablespace | {name,oid,aclitem[],text[]}
(1 row)
```

Ordinarily, an aggregate function with a polymorphic result type has a polymorphic state type, as in the above example. This is necessary because otherwise the final function cannot be declared sensibly: it would need to have a polymorphic result type but no polymorphic argument type, which `CREATE FUNCTION` will reject on the grounds that the result type cannot be deduced from a call. But sometimes it is inconvenient to use a polymorphic state type. The most common case is where the aggregate support functions are to be written in C and the state type should be declared as `internal` because there is no SQL-level equivalent for it. To address this case, it is possible to declare the final function as taking extra "dummy" arguments that match the input arguments of the aggregate. Such dummy arguments are always passed as null values since no specific value is available when the final function is called. Their only use

is to allow a polymorphic final function's result type to be connected to the aggregate's input type(s). For example, the definition of the built-in aggregate `array_agg` is equivalent to

```
CREATE FUNCTION array_agg_transfn(internal, anynonarray)
  RETURNS internal ...;
CREATE FUNCTION array_agg_finalfn(internal, anynonarray)
  RETURNS anyarray ...;

CREATE AGGREGATE array_agg (anynonarray)
(
    sfunc = array_agg_transfn,
    stype = internal,
    finalfunc = array_agg_finalfn,
    finalfunc_extra
);
```

Here, the `finalfunc_extra` option specifies that the final function receives, in addition to the state value, extra dummy argument(s) corresponding to the aggregate's input argument(s). The extra `anynonarray` argument allows the declaration of `array_agg_finalfn` to be valid.

An aggregate function can be made to accept a varying number of arguments by declaring its last argument as a `VARIADIC` array, in much the same fashion as for regular functions; see Section 35.4.5. The aggregate's transition function(s) must have the same array type as their last argument. The transition function(s) typically would also be marked `VARIADIC`, but this is not strictly required.

> **Note:** Variadic aggregates are easily misused in connection with the `ORDER BY` option (see Section 4.2.7), since the parser cannot tell whether the wrong number of actual arguments have been given in such a combination. Keep in mind that everything to the right of `ORDER BY` is a sort key, not an argument to the aggregate. For example, in
>
> ```
> SELECT myaggregate(a ORDER BY a, b, c) FROM ...
> ```
>
> the parser will see this as a single aggregate function argument and three sort keys. However, the user might have intended
>
> ```
> SELECT myaggregate(a, b, c ORDER BY a) FROM ...
> ```
>
> If `myaggregate` is variadic, both these calls could be perfectly valid.
>
> For the same reason, it's wise to think twice before creating aggregate functions with the same names and different numbers of regular arguments.

35.10.3. Ordered-Set Aggregates

The aggregates we have been describing so far are "normal" aggregates. PostgreSQL also supports *ordered-set aggregates*, which differ from normal aggregates in two key ways. First, in addition to ordinary aggregated arguments that are evaluated once per input row, an ordered-set aggregate can have "direct" arguments that are evaluated only once per aggregation operation. Second, the syntax for the ordinary aggregated arguments specifies a sort ordering for them explicitly. An ordered-set aggregate is usually used to implement a computation that depends on a specific row ordering, for instance rank or

percentile, so that the sort ordering is a required aspect of any call. For example, the built-in definition of `percentile_disc` is equivalent to:

```
CREATE FUNCTION ordered_set_transition(internal, anyelement)
  RETURNS internal ...;
CREATE FUNCTION percentile_disc_final(internal, float8, anyelement)
  RETURNS anyelement ...;

CREATE AGGREGATE percentile_disc (float8 ORDER BY anyelement)
(
    sfunc = ordered_set_transition,
    stype = internal,
    finalfunc = percentile_disc_final,
    finalfunc_extra
);
```

This aggregate takes a `float8` direct argument (the percentile fraction) and an aggregated input that can be of any sortable data type. It could be used to obtain a median household income like this:

```
SELECT percentile_disc(0.5) WITHIN GROUP (ORDER BY income) FROM households;
 percentile_disc
-----------------
           50489
```

Here, `0.5` is a direct argument; it would make no sense for the percentile fraction to be a value varying across rows.

Unlike the case for normal aggregates, the sorting of input rows for an ordered-set aggregate is *not* done behind the scenes, but is the responsibility of the aggregate's support functions. The typical implementation approach is to keep a reference to a "tuplesort" object in the aggregate's state value, feed the incoming rows into that object, and then complete the sorting and read out the data in the final function. This design allows the final function to perform special operations such as injecting additional "hypothetical" rows into the data to be sorted. While normal aggregates can often be implemented with support functions written in PL/pgSQL or another PL language, ordered-set aggregates generally have to be written in C, since their state values aren't definable as any SQL data type. (In the above example, notice that the state value is declared as type `internal` — this is typical.)

The state transition function for an ordered-set aggregate receives the current state value plus the aggregated input values for each row, and returns the updated state value. This is the same definition as for normal aggregates, but note that the direct arguments (if any) are not provided. The final function receives the last state value, the values of the direct arguments if any, and (if `finalfunc_extra` is specified) null values corresponding to the aggregated input(s). As with normal aggregates, `finalfunc_extra` is only really useful if the aggregate is polymorphic; then the extra dummy argument(s) are needed to connect the final function's result type to the aggregate's input type(s).

Currently, ordered-set aggregates cannot be used as window functions, and therefore there is no need for them to support moving-aggregate mode.

35.10.4. Support Functions for Aggregates

A function written in C can detect that it is being called as an aggregate transition or final function by calling `AggCheckCallContext`, for example:

```
if (AggCheckCallContext(fcinfo, NULL))
```

One reason for checking this is that when it is true for a transition function, the first input must be a temporary state value and can therefore safely be modified in-place rather than allocating a new copy. See `int8inc()` for an example. (This is the *only* case where it is safe for a function to modify a pass-by-reference input. In particular, final functions for normal aggregates must not modify their inputs in any case, because in some cases they will be re-executed on the same final state value.)

The second argument of `AggCheckCallContext` can be used to retrieve the memory context in which aggregate state values are being kept. This is useful for transition functions that wish to use "expanded" objects (see Section 35.11.1) as their state values. On first call, the transition function should return an expanded object whose memory context is a child of the aggregate state context, and then keep returning the same expanded object on subsequent calls. See `array_append()` for an example. (`array_append()` is not the transition function of any built-in aggregate, but it is written to behave efficiently when used as transition function of a custom aggregate.)

Another support routine available to aggregate functions written in C is `AggGetAggref`, which returns the `Aggref` parse node that defines the aggregate call. This is mainly useful for ordered-set aggregates, which can inspect the substructure of the `Aggref` node to find out what sort ordering they are supposed to implement. Examples can be found in `orderedsetaggs.c` in the PostgreSQL source code.

35.11. User-defined Types

As described in Section 35.2, PostgreSQL can be extended to support new data types. This section describes how to define new base types, which are data types defined below the level of the SQL language. Creating a new base type requires implementing functions to operate on the type in a low-level language, usually C.

The examples in this section can be found in `complex.sql` and `complex.c` in the `src/tutorial` directory of the source distribution. See the `README` file in that directory for instructions about running the examples.

A user-defined type must always have input and output functions. These functions determine how the type appears in strings (for input by the user and output to the user) and how the type is organized in memory. The input function takes a null-terminated character string as its argument and returns the internal (in memory) representation of the type. The output function takes the internal representation of the type as argument and returns a null-terminated character string. If we want to do anything more with the type than merely store it, we must provide additional functions to implement whatever operations we'd like to have for the type.

Suppose we want to define a type `complex` that represents complex numbers. A natural way to represent a complex number in memory would be the following C structure:

```
typedef struct Complex {
    double      x;
```

```
    double      y;
} Complex;
```

We will need to make this a pass-by-reference type, since it's too large to fit into a single `Datum` value.

As the external string representation of the type, we choose a string of the form `(x,y)`.

The input and output functions are usually not hard to write, especially the output function. But when defining the external string representation of the type, remember that you must eventually write a complete and robust parser for that representation as your input function. For instance:

```
PG_FUNCTION_INFO_V1(complex_in);

Datum
complex_in(PG_FUNCTION_ARGS)
{
    char       *str = PG_GETARG_CSTRING(0);
    double      x,
                y;
    Complex    *result;

    if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        ereport(ERROR,
                (errcode(ERRCODE_INVALID_TEXT_REPRESENTATION),
                 errmsg("invalid input syntax for complex: \"%s\"",
                        str)));

    result = (Complex *) palloc(sizeof(Complex));
    result->x = x;
    result->y = y;
    PG_RETURN_POINTER(result);
}
```

The output function can simply be:

```
PG_FUNCTION_INFO_V1(complex_out);

Datum
complex_out(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    char       *result;

    result = psprintf("(%g,%g)", complex->x, complex->y);
    PG_RETURN_CSTRING(result);
}
```

You should be careful to make the input and output functions inverses of each other. If you do not, you will have severe problems when you need to dump your data into a file and then read it back in. This is a particularly common problem when floating-point numbers are involved.

Optionally, a user-defined type can provide binary input and output routines. Binary I/O is normally faster but less portable than textual I/O. As with textual I/O, it is up to you to define exactly what the external

binary representation is. Most of the built-in data types try to provide a machine-independent binary representation. For `complex`, we will piggy-back on the binary I/O converters for type `float8`:

```
PG_FUNCTION_INFO_V1(complex_recv);

Datum
complex_recv(PG_FUNCTION_ARGS)
{
    StringInfo  buf = (StringInfo) PG_GETARG_POINTER(0);
    Complex    *result;

    result = (Complex *) palloc(sizeof(Complex));
    result->x = pq_getmsgfloat8(buf);
    result->y = pq_getmsgfloat8(buf);
    PG_RETURN_POINTER(result);
}

PG_FUNCTION_INFO_V1(complex_send);

Datum
complex_send(PG_FUNCTION_ARGS)
{
    Complex    *complex = (Complex *) PG_GETARG_POINTER(0);
    StringInfoData buf;

    pq_begintypsend(&buf);
    pq_sendfloat8(&buf, complex->x);
    pq_sendfloat8(&buf, complex->y);
    PG_RETURN_BYTEA_P(pq_endtypsend(&buf));
}
```

Once we have written the I/O functions and compiled them into a shared library, we can define the `complex` type in SQL. First we declare it as a shell type:

```
CREATE TYPE complex;
```

This serves as a placeholder that allows us to reference the type while defining its I/O functions. Now we can define the I/O functions:

```
CREATE FUNCTION complex_in(cstring)
    RETURNS complex
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_out(complex)
    RETURNS cstring
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_recv(internal)
   RETURNS complex
   AS 'filename'
```

```
    LANGUAGE C IMMUTABLE STRICT;

CREATE FUNCTION complex_send(complex)
    RETURNS bytea
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;
```

Finally, we can provide the full definition of the data type:

```
CREATE TYPE complex (
    internallength = 16,
    input = complex_in,
    output = complex_out,
    receive = complex_recv,
    send = complex_send,
    alignment = double
);
```

When you define a new base type, PostgreSQL automatically provides support for arrays of that type. The array type typically has the same name as the base type with the underscore character (_) prepended.

Once the data type exists, we can declare additional functions to provide useful operations on the data type. Operators can then be defined atop the functions, and if needed, operator classes can be created to support indexing of the data type. These additional layers are discussed in following sections.

If the internal representation of the data type is variable-length, the internal representation must follow the standard layout for variable-length data: the first four bytes must be a `char[4]` field which is never accessed directly (customarily named `vl_len_`). You must use the `SET_VARSIZE()` macro to store the total size of the datum (including the length field itself) in this field and `VARSIZE()` to retrieve it. (These macros exist because the length field may be encoded depending on platform.)

For further details see the description of the CREATE TYPE command.

35.11.1. TOAST Considerations

If the values of your data type vary in size (in internal form), it's usually desirable to make the data type TOAST-able (see Section 63.2). You should do this even if the values are always too small to be compressed or stored externally, because TOAST can save space on small data too, by reducing header overhead.

To support TOAST storage, the C functions operating on the data type must always be careful to unpack any toasted values they are handed by using `PG_DETOAST_DATUM`. (This detail is customarily hidden by defining type-specific `GETARG_DATATYPE_P` macros.) Then, when running the `CREATE TYPE` command, specify the internal length as `variable` and select some appropriate storage option other than `plain`.

If data alignment is unimportant (either just for a specific function or because the data type specifies byte alignment anyway) then it's possible to avoid some of the overhead of `PG_DETOAST_DATUM`. You can use `PG_DETOAST_DATUM_PACKED` instead (customarily hidden by defining a `GETARG_DATATYPE_PP` macro) and using the macros `VARSIZE_ANY_EXHDR` and `VARDATA_ANY` to access a potentially-packed

datum. Again, the data returned by these macros is not aligned even if the data type definition specifies an alignment. If the alignment is important you must go through the regular `PG_DETOAST_DATUM` interface.

> **Note:** Older code frequently declares `vl_len_` as an `int32` field instead of `char[4]`. This is OK as long as the struct definition has other fields that have at least `int32` alignment. But it is dangerous to use such a struct definition when working with a potentially unaligned datum; the compiler may take it as license to assume the datum actually is aligned, leading to core dumps on architectures that are strict about alignment.

Another feature that's enabled by TOAST support is the possibility of having an *expanded* in-memory data representation that is more convenient to work with than the format that is stored on disk. The regular or "flat" varlena storage format is ultimately just a blob of bytes; it cannot for example contain pointers, since it may get copied to other locations in memory. For complex data types, the flat format may be quite expensive to work with, so PostgreSQL provides a way to "expand" the flat format into a representation that is more suited to computation, and then pass that format in-memory between functions of the data type.

To use expanded storage, a data type must define an expanded format that follows the rules given in `src/include/utils/expandeddatum.h`, and provide functions to "expand" a flat varlena value into expanded format and "flatten" the expanded format back to the regular varlena representation. Then ensure that all C functions for the data type can accept either representation, possibly by converting one into the other immediately upon receipt. This does not require fixing all existing functions for the data type at once, because the standard `PG_DETOAST_DATUM` macro is defined to convert expanded inputs into regular flat format. Therefore, existing functions that work with the flat varlena format will continue to work, though slightly inefficiently, with expanded inputs; they need not be converted until and unless better performance is important.

C functions that know how to work with an expanded representation typically fall into two categories: those that can only handle expanded format, and those that can handle either expanded or flat varlena inputs. The former are easier to write but may be less efficient overall, because converting a flat input to expanded form for use by a single function may cost more than is saved by operating on the expanded format. When only expanded format need be handled, conversion of flat inputs to expanded form can be hidden inside an argument-fetching macro, so that the function appears no more complex than one working with traditional varlena input. To handle both types of input, write an argument-fetching function that will detoast external, short-header, and compressed varlena inputs, but not expanded inputs. Such a function can be defined as returning a pointer to a union of the flat varlena format and the expanded format. Callers can use the `VARATT_IS_EXPANDED_HEADER()` macro to determine which format they received.

The TOAST infrastructure not only allows regular varlena values to be distinguished from expanded values, but also distinguishes "read-write" and "read-only" pointers to expanded values. C functions that only need to examine an expanded value, or will only change it in safe and non-semantically-visible ways, need not care which type of pointer they receive. C functions that produce a modified version of an input value are allowed to modify an expanded input value in-place if they receive a read-write pointer, but must not modify the input if they receive a read-only pointer; in that case they have to copy the value first, producing a new value to modify. A C function that has constructed a new expanded value should always return a read-write pointer to it. Also, a C function that is modifying a read-write expanded value in-place should take care to leave the value in a sane state if it fails partway through.

For examples of working with expanded values, see the standard array infrastructure, particularly `src/backend/utils/adt/array_expanded.c`.

35.12. User-defined Operators

Every operator is "syntactic sugar" for a call to an underlying function that does the real work; so you must first create the underlying function before you can create the operator. However, an operator is *not merely* syntactic sugar, because it carries additional information that helps the query planner optimize queries that use the operator. The next section will be devoted to explaining that additional information.

PostgreSQL supports left unary, right unary, and binary operators. Operators can be overloaded; that is, the same operator name can be used for different operators that have different numbers and types of operands. When a query is executed, the system determines the operator to call from the number and types of the provided operands.

Here is an example of creating an operator for adding two complex numbers. We assume we've already created the definition of type `complex` (see Section 35.11). First we need a function that does the work, then we can define the operator:

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'filename', 'complex_add'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
    commutator = +
);
```

Now we could execute a query like this:

```
SELECT (a + b) AS c FROM test_complex;

        c
-----------------
 (5.2,6.05)
 (133.42,144.95)
```

We've shown how to create a binary operator here. To create unary operators, just omit one of `leftarg` (for left unary) or `rightarg` (for right unary). The `procedure` clause and the argument clauses are the only required items in `CREATE OPERATOR`. The `commutator` clause shown in the example is an optional hint to the query optimizer. Further details about `commutator` and other optimizer hints appear in the next section.

35.13. Operator Optimization Information

A PostgreSQL operator definition can include several optional clauses that tell the system useful things about how the operator behaves. These clauses should be provided whenever appropriate, because they can make for considerable speedups in execution of queries that use the operator. But if you provide them, you must be sure that they are right! Incorrect use of an optimization clause can result in slow queries, subtly wrong output, or other Bad Things. You can always leave out an optimization clause if you are not sure about it; the only consequence is that queries might run slower than they need to.

Additional optimization clauses might be added in future versions of PostgreSQL. The ones described here are all the ones that release 9.5.24 understands.

35.13.1. COMMUTATOR

The COMMUTATOR clause, if provided, names an operator that is the commutator of the operator being defined. We say that operator A is the commutator of operator B if (x A y) equals (y B x) for all possible input values x, y. Notice that B is also the commutator of A. For example, operators < and > for a particular data type are usually each others' commutators, and operator + is usually commutative with itself. But operator – is usually not commutative with anything.

The left operand type of a commutable operator is the same as the right operand type of its commutator, and vice versa. So the name of the commutator operator is all that PostgreSQL needs to be given to look up the commutator, and that's all that needs to be provided in the COMMUTATOR clause.

It's critical to provide commutator information for operators that will be used in indexes and join clauses, because this allows the query optimizer to "flip around" such a clause to the forms needed for different plan types. For example, consider a query with a WHERE clause like `tab1.x = tab2.y`, where `tab1.x` and `tab2.y` are of a user-defined type, and suppose that `tab2.y` is indexed. The optimizer cannot generate an index scan unless it can determine how to flip the clause around to `tab2.y = tab1.x`, because the index-scan machinery expects to see the indexed column on the left of the operator it is given. PostgreSQL will *not* simply assume that this is a valid transformation — the creator of the = operator must specify that it is valid, by marking the operator with commutator information.

When you are defining a self-commutative operator, you just do it. When you are defining a pair of commutative operators, things are a little trickier: how can the first one to be defined refer to the other one, which you haven't defined yet? There are two solutions to this problem:

- One way is to omit the COMMUTATOR clause in the first operator that you define, and then provide one in the second operator's definition. Since PostgreSQL knows that commutative operators come in pairs, when it sees the second definition it will automatically go back and fill in the missing COMMUTATOR clause in the first definition.
- The other, more straightforward way is just to include COMMUTATOR clauses in both definitions. When PostgreSQL processes the first definition and realizes that COMMUTATOR refers to a nonexistent operator, the system will make a dummy entry for that operator in the system catalog. This dummy entry will have valid data only for the operator name, left and right operand types, and result type, since that's all that PostgreSQL can deduce at this point. The first operator's catalog entry will link to this dummy entry. Later, when you define the second operator, the system updates the dummy entry with the additional information from the second definition. If you try to use the dummy operator before it's been filled in, you'll just get an error message.

35.13.2. NEGATOR

The NEGATOR clause, if provided, names an operator that is the negator of the operator being defined. We say that operator A is the negator of operator B if both return Boolean results and (x A y) equals NOT (x B y) for all possible inputs x, y. Notice that B is also the negator of A. For example, < and >= are a negator pair for most data types. An operator can never validly be its own negator.

Unlike commutators, a pair of unary operators could validly be marked as each other's negators; that would mean (A x) equals NOT (B x) for all x, or the equivalent for right unary operators.

An operator's negator must have the same left and/or right operand types as the operator to be defined, so just as with COMMUTATOR, only the operator name need be given in the NEGATOR clause.

Providing a negator is very helpful to the query optimizer since it allows expressions like NOT (x = y) to be simplified into x <> y. This comes up more often than you might think, because NOT operations can be inserted as a consequence of other rearrangements.

Pairs of negator operators can be defined using the same methods explained above for commutator pairs.

35.13.3. RESTRICT

The RESTRICT clause, if provided, names a restriction selectivity estimation function for the operator. (Note that this is a function name, not an operator name.) RESTRICT clauses only make sense for binary operators that return boolean. The idea behind a restriction selectivity estimator is to guess what fraction of the rows in a table will satisfy a WHERE-clause condition of the form:

```
column OP constant
```

for the current operator and a particular constant value. This assists the optimizer by giving it some idea of how many rows will be eliminated by WHERE clauses that have this form. (What happens if the constant is on the left, you might be wondering? Well, that's one of the things that COMMUTATOR is for...)

Writing new restriction selectivity estimation functions is far beyond the scope of this chapter, but fortunately you can usually just use one of the system's standard estimators for many of your own operators. These are the standard restriction estimators:

eqsel for =
neqsel for <>
scalarltsel for < or <=
scalargtsel for > or >=

It might seem a little odd that these are the categories, but they make sense if you think about it. = will typically accept only a small fraction of the rows in a table; <> will typically reject only a small fraction. < will accept a fraction that depends on where the given constant falls in the range of values for that table column (which, it just so happens, is information collected by ANALYZE and made available to the selectivity estimator). <= will accept a slightly larger fraction than < for the same comparison constant, but they're close enough to not be worth distinguishing, especially since we're not likely to do better than a rough guess anyhow. Similar remarks apply to > and >=.

You can frequently get away with using either eqsel or neqsel for operators that have very high or very low selectivity, even if they aren't really equality or inequality. For example, the approximate-equality geometric operators use eqsel on the assumption that they'll usually only match a small fraction of the entries in a table.

You can use scalarltsel and scalargtsel for comparisons on data types that have some sensible means of being converted into numeric scalars for range comparisons. If possible, add the data type to those understood by the function convert_to_scalar() in src/backend/utils/adt/selfuncs.c. (Eventually, this function should be replaced by per-data-type functions identified through a column of the pg_type system catalog; but that hasn't happened yet.) If you do not do this, things will still work, but the optimizer's estimates won't be as good as they could be.

There are additional selectivity estimation functions designed for geometric operators in src/backend/utils/adt/geo_selfuncs.c: areasel, positionsel, and contsel. At this writing these are just stubs, but you might want to use them (or even better, improve them) anyway.

35.13.4. JOIN

The JOIN clause, if provided, names a join selectivity estimation function for the operator. (Note that this is a function name, not an operator name.) JOIN clauses only make sense for binary operators that return boolean. The idea behind a join selectivity estimator is to guess what fraction of the rows in a pair of tables will satisfy a WHERE-clause condition of the form:

```
table1.column1 OP table2.column2
```

for the current operator. As with the RESTRICT clause, this helps the optimizer very substantially by letting it figure out which of several possible join sequences is likely to take the least work.

As before, this chapter will make no attempt to explain how to write a join selectivity estimator function, but will just suggest that you use one of the standard estimators if one is applicable:

eqjoinsel for =
neqjoinsel for <>
scalarltjoinsel for < or <=
scalargtjoinsel for > or >=
areajoinsel for 2D area-based comparisons
positionjoinsel for 2D position-based comparisons
contjoinsel for 2D containment-based comparisons

35.13.5. HASHES

The HASHES clause, if present, tells the system that it is permissible to use the hash join method for a join based on this operator. HASHES only makes sense for a binary operator that returns boolean, and in practice the operator must represent equality for some data type or pair of data types.

The assumption underlying hash join is that the join operator can only return true for pairs of left and right values that hash to the same hash code. If two values get put in different hash buckets, the join

will never compare them at all, implicitly assuming that the result of the join operator must be false. So it never makes sense to specify HASHES for operators that do not represent some form of equality. In most cases it is only practical to support hashing for operators that take the same data type on both sides. However, sometimes it is possible to design compatible hash functions for two or more data types; that is, functions that will generate the same hash codes for "equal" values, even though the values have different representations. For example, it's fairly simple to arrange this property when hashing integers of different widths.

To be marked HASHES, the join operator must appear in a hash index operator family. This is not enforced when you create the operator, since of course the referencing operator family couldn't exist yet. But attempts to use the operator in hash joins will fail at run time if no such operator family exists. The system needs the operator family to find the data-type-specific hash function(s) for the operator's input data type(s). Of course, you must also create suitable hash functions before you can create the operator family.

Care should be exercised when preparing a hash function, because there are machine-dependent ways in which it might fail to do the right thing. For example, if your data type is a structure in which there might be uninteresting pad bits, you cannot simply pass the whole structure to hash_any. (Unless you write your other operators and functions to ensure that the unused bits are always zero, which is the recommended strategy.) Another example is that on machines that meet the IEEE floating-point standard, negative zero and positive zero are different values (different bit patterns) but they are defined to compare equal. If a float value might contain negative zero then extra steps are needed to ensure it generates the same hash value as positive zero.

A hash-joinable operator must have a commutator (itself if the two operand data types are the same, or a related equality operator if they are different) that appears in the same operator family. If this is not the case, planner errors might occur when the operator is used. Also, it is a good idea (but not strictly required) for a hash operator family that supports multiple data types to provide equality operators for every combination of the data types; this allows better optimization.

> **Note:** The function underlying a hash-joinable operator must be marked immutable or stable. If it is volatile, the system will never attempt to use the operator for a hash join.

> **Note:** If a hash-joinable operator has an underlying function that is marked strict, the function must also be complete: that is, it should return true or false, never null, for any two nonnull inputs. If this rule is not followed, hash-optimization of IN operations might generate wrong results. (Specifically, IN might return false where the correct answer according to the standard would be null; or it might yield an error complaining that it wasn't prepared for a null result.)

35.13.6. MERGES

The MERGES clause, if present, tells the system that it is permissible to use the merge-join method for a join based on this operator. MERGES only makes sense for a binary operator that returns boolean, and in practice the operator must represent equality for some data type or pair of data types.

Merge join is based on the idea of sorting the left- and right-hand tables into order and then scanning them in parallel. So, both data types must be capable of being fully ordered, and the join operator must be one that can only succeed for pairs of values that fall at the "same place" in the sort order. In practice this means that the join operator must behave like equality. But it is possible to merge-join two distinct data types so long as they are logically compatible. For example, the `smallint`-versus-`integer` equality operator is merge-joinable. We only need sorting operators that will bring both data types into a logically compatible sequence.

To be marked `MERGES`, the join operator must appear as an equality member of a `btree` index operator family. This is not enforced when you create the operator, since of course the referencing operator family couldn't exist yet. But the operator will not actually be used for merge joins unless a matching operator family can be found. The `MERGES` flag thus acts as a hint to the planner that it's worth looking for a matching operator family.

A merge-joinable operator must have a commutator (itself if the two operand data types are the same, or a related equality operator if they are different) that appears in the same operator family. If this is not the case, planner errors might occur when the operator is used. Also, it is a good idea (but not strictly required) for a `btree` operator family that supports multiple data types to provide equality operators for every combination of the data types; this allows better optimization.

> **Note:** The function underlying a merge-joinable operator must be marked immutable or stable. If it is volatile, the system will never attempt to use the operator for a merge join.

35.14. Interfacing Extensions To Indexes

The procedures described thus far let you define new types, new functions, and new operators. However, we cannot yet define an index on a column of a new data type. To do this, we must define an *operator class* for the new data type. Later in this section, we will illustrate this concept in an example: a new operator class for the B-tree index method that stores and sorts complex numbers in ascending absolute value order.

Operator classes can be grouped into *operator families* to show the relationships between semantically compatible classes. When only a single data type is involved, an operator class is sufficient, so we'll focus on that case first and then return to operator families.

35.14.1. Index Methods and Operator Classes

The `pg_am` table contains one row for every index method (internally known as access method). Support for regular access to tables is built into PostgreSQL, but all index methods are described in `pg_am`. It is possible to add a new index method by defining the required interface routines and then creating a row in `pg_am` — but that is beyond the scope of this chapter (see Chapter 58).

The routines for an index method do not directly know anything about the data types that the index method will operate on. Instead, an *operator class* identifies the set of operations that the index method needs to use to work with a particular data type. Operator classes are so called because one thing they specify is the set of `WHERE`-clause operators that can be used with an index (i.e., can be converted into an index-scan

qualification). An operator class can also specify some *support procedures* that are needed by the internal operations of the index method, but do not directly correspond to any `WHERE`-clause operator that can be used with the index.

It is possible to define multiple operator classes for the same data type and index method. By doing this, multiple sets of indexing semantics can be defined for a single data type. For example, a B-tree index requires a sort ordering to be defined for each data type it works on. It might be useful for a complex-number data type to have one B-tree operator class that sorts the data by complex absolute value, another that sorts by real part, and so on. Typically, one of the operator classes will be deemed most commonly useful and will be marked as the default operator class for that data type and index method.

The same operator class name can be used for several different index methods (for example, both B-tree and hash index methods have operator classes named `int4_ops`), but each such class is an independent entity and must be defined separately.

35.14.2. Index Method Strategies

The operators associated with an operator class are identified by "strategy numbers", which serve to identify the semantics of each operator within the context of its operator class. For example, B-trees impose a strict ordering on keys, lesser to greater, and so operators like "less than" and "greater than or equal to" are interesting with respect to a B-tree. Because PostgreSQL allows the user to define operators, PostgreSQL cannot look at the name of an operator (e.g., < or >=) and tell what kind of comparison it is. Instead, the index method defines a set of "strategies", which can be thought of as generalized operators. Each operator class specifies which actual operator corresponds to each strategy for a particular data type and interpretation of the index semantics.

The B-tree index method defines five strategies, shown in Table 35-2.

Table 35-2. B-tree Strategies

Operation	Strategy Number
less than	1
less than or equal	2
equal	3
greater than or equal	4
greater than	5

Hash indexes support only equality comparisons, and so they use only one strategy, shown in Table 35-3.

Table 35-3. Hash Strategies

Operation	Strategy Number
equal	1

GiST indexes are more flexible: they do not have a fixed set of strategies at all. Instead, the "consistency" support routine of each particular GiST operator class interprets the strategy numbers however it likes. As an example, several of the built-in GiST index operator classes index two-dimensional geometric objects, providing the "R-tree" strategies shown in Table 35-4. Four of these are true two-dimensional

tests (overlaps, same, contains, contained by); four of them consider only the X direction; and the other four provide the same tests in the Y direction.

Table 35-4. GiST Two-Dimensional "R-tree" Strategies

Operation	Strategy Number
strictly left of	1
does not extend to right of	2
overlaps	3
does not extend to left of	4
strictly right of	5
same	6
contains	7
contained by	8
does not extend above	9
strictly below	10
strictly above	11
does not extend below	12

SP-GiST indexes are similar to GiST indexes in flexibility: they don't have a fixed set of strategies. Instead the support routines of each operator class interpret the strategy numbers according to the operator class's definition. As an example, the strategy numbers used by the built-in operator classes for points are shown in Table 35-5.

Table 35-5. SP-GiST Point Strategies

Operation	Strategy Number
strictly left of	1
strictly right of	5
same	6
contained by	8
strictly below	10
strictly above	11

GIN indexes are similar to GiST and SP-GiST indexes, in that they don't have a fixed set of strategies either. Instead the support routines of each operator class interpret the strategy numbers according to the operator class's definition. As an example, the strategy numbers used by the built-in operator classes for arrays are shown in Table 35-6.

Table 35-6. GIN Array Strategies

Operation	Strategy Number
overlap	1
contains	2

Operation	Strategy Number
is contained by	3
equal	4

BRIN indexes are similar to GiST, SP-GiST and GIN indexes in that they don't have a fixed set of strategies either. Instead the support routines of each operator class interpret the strategy numbers according to the operator class's definition. As an example, the strategy numbers used by the built-in `Minmax` operator classes are shown in Table 35-7.

Table 35-7. BRIN Minmax Strategies

Operation	Strategy Number
less than	1
less than or equal	2
equal	3
greater than or equal	4
greater than	5

Notice that all the operators listed above return Boolean values. In practice, all operators defined as index method search operators must return type `boolean`, since they must appear at the top level of a `WHERE` clause to be used with an index. (Some index access methods also support *ordering operators*, which typically don't return Boolean values; that feature is discussed in Section 35.14.7.)

35.14.3. Index Method Support Routines

Strategies aren't usually enough information for the system to figure out how to use an index. In practice, the index methods require additional support routines in order to work. For example, the B-tree index method must be able to compare two keys and determine whether one is greater than, equal to, or less than the other. Similarly, the hash index method must be able to compute hash codes for key values. These operations do not correspond to operators used in qualifications in SQL commands; they are administrative routines used by the index methods, internally.

Just as with strategies, the operator class identifies which specific functions should play each of these roles for a given data type and semantic interpretation. The index method defines the set of functions it needs, and the operator class identifies the correct functions to use by assigning them to the "support function numbers" specified by the index method.

B-trees require a single support function, and allow a second one to be supplied at the operator class author's option, as shown in Table 35-8.

Table 35-8. B-tree Support Functions

Function	Support Number

Function	Support Number
Compare two keys and return an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second	1
Return the addresses of C-callable sort support function(s), as documented in `utils/sortsupport.h` (optional)	2

Hash indexes require one support function, shown in Table 35-9.

Table 35-9. Hash Support Functions

Function	Support Number
Compute the hash value for a key	1

GiST indexes have nine support functions, two of which are optional, as shown in Table 35-10. (For more information see Chapter 59.)

Table 35-10. GiST Support Functions

Function	Description	Support Number
`consistent`	determine whether key satisfies the query qualifier	1
`union`	compute union of a set of keys	2
`compress`	compute a compressed representation of a key or value to be indexed	3
`decompress`	compute a decompressed representation of a compressed key	4
`penalty`	compute penalty for inserting new key into subtree with given subtree's key	5
`picksplit`	determine which entries of a page are to be moved to the new page and compute the union keys for resulting pages	6
`equal`	compare two keys and return true if they are equal	7
`distance`	determine distance from key to query value (optional)	8
`fetch`	compute original representation of a compressed key for index-only scans (optional)	9

SP-GiST indexes require five support functions, as shown in Table 35-11. (For more information see Chapter 60.)

Table 35-11. SP-GiST Support Functions

Function	Description	Support Number
`config`	provide basic information about the operator class	1
`choose`	determine how to insert a new value into an inner tuple	2
`picksplit`	determine how to partition a set of values	3
`inner_consistent`	determine which sub-partitions need to be searched for a query	4
`leaf_consistent`	determine whether key satisfies the query qualifier	5

GIN indexes have six support functions, three of which are optional, as shown in Table 35-12. (For more information see Chapter 61.)

Table 35-12. GIN Support Functions

Function	Description	Support Number
`compare`	compare two keys and return an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second	1
`extractValue`	extract keys from a value to be indexed	2
`extractQuery`	extract keys from a query condition	3
`consistent`	determine whether value matches query condition (Boolean variant) (optional if support function 6 is present)	4
`comparePartial`	compare partial key from query and key from index, and return an integer less than zero, zero, or greater than zero, indicating whether GIN should ignore this index entry, treat the entry as a match, or stop the index scan (optional)	5

Function	Description	Support Number
triConsistent	determine whether value matches query condition (ternary variant) (optional if support function 4 is present)	6

BRIN indexes have four basic support functions, as shown in Table 35-13; those basic functions may require additional support functions to be provided. (For more information see Section 62.3.)

Table 35-13. BRIN Support Functions

Function	Description	Support Number
opcInfo	return internal information describing the indexed columns' summary data	1
add_value	add a new value to an existing summary index tuple	2
consistent	determine whether value matches query condition	3
union	compute union of two summary tuples	4

Unlike search operators, support functions return whichever data type the particular index method expects; for example in the case of the comparison function for B-trees, a signed integer. The number and types of the arguments to each support function are likewise dependent on the index method. For B-tree and hash the comparison and hashing support functions take the same input data types as do the operators included in the operator class, but this is not the case for most GiST, SP-GiST, GIN, and BRIN support functions.

35.14.4. An Example

Now that we have seen the ideas, here is the promised example of creating a new operator class. (You can find a working copy of this example in src/tutorial/complex.c and src/tutorial/complex.sql in the source distribution.) The operator class encapsulates operators that sort complex numbers in absolute value order, so we choose the name complex_abs_ops. First, we need a set of operators. The procedure for defining operators was discussed in Section 35.12. For an operator class on B-trees, the operators we require are:

- absolute-value less-than (strategy 1)
- absolute-value less-than-or-equal (strategy 2)
- absolute-value equal (strategy 3)
- absolute-value greater-than-or-equal (strategy 4)
- absolute-value greater-than (strategy 5)

The least error-prone way to define a related set of comparison operators is to write the B-tree comparison support function first, and then write the other functions as one-line wrappers around the support function. This reduces the odds of getting inconsistent results for corner cases. Following this approach, we first write:

```
#define Mag(c)  ((c)->x*(c)->x + (c)->y*(c)->y)

static int
complex_abs_cmp_internal(Complex *a, Complex *b)
{
    double      amag = Mag(a),
                bmag = Mag(b);

    if (amag < bmag)
        return -1;
    if (amag > bmag)
        return 1;
    return 0;
}
```

Now the less-than function looks like:

```
PG_FUNCTION_INFO_V1(complex_abs_lt);

Datum
complex_abs_lt(PG_FUNCTION_ARGS)
{
    Complex    *a = (Complex *) PG_GETARG_POINTER(0);
    Complex    *b = (Complex *) PG_GETARG_POINTER(1);

    PG_RETURN_BOOL(complex_abs_cmp_internal(a, b) < 0);
}
```

The other four functions differ only in how they compare the internal function's result to zero.

Next we declare the functions and the operators based on the functions to SQL:

```
CREATE FUNCTION complex_abs_lt(complex, complex) RETURNS bool
    AS 'filename', 'complex_abs_lt'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR < (
   leftarg = complex, rightarg = complex, procedure = complex_abs_lt,
   commutator = > , negator = >= ,
   restrict = scalarltsel, join = scalarltjoinsel
);
```

It is important to specify the correct commutator and negator operators, as well as suitable restriction and join selectivity functions, otherwise the optimizer will be unable to make effective use of the index. Note that the less-than, equal, and greater-than cases should use different selectivity functions.

Other things worth noting are happening here:

- There can only be one operator named, say, = and taking type `complex` for both operands. In this case we don't have any other operator = for `complex`, but if we were building a practical data type we'd probably want = to be the ordinary equality operation for complex numbers (and not the equality of the absolute values). In that case, we'd need to use some other operator name for `complex_abs_eq`.
- Although PostgreSQL can cope with functions having the same SQL name as long as they have different argument data types, C can only cope with one global function having a given name. So we shouldn't name the C function something simple like `abs_eq`. Usually it's a good practice to include the data type name in the C function name, so as not to conflict with functions for other data types.
- We could have made the SQL name of the function `abs_eq`, relying on PostgreSQL to distinguish it by argument data types from any other SQL function of the same name. To keep the example simple, we make the function have the same names at the C level and SQL level.

The next step is the registration of the support routine required by B-trees. The example C code that implements this is in the same file that contains the operator functions. This is how we declare the function:

```
CREATE FUNCTION complex_abs_cmp(complex, complex)
    RETURNS integer
    AS 'filename'
    LANGUAGE C IMMUTABLE STRICT;
```

Now that we have the required operators and support routine, we can finally create the operator class:

```
CREATE OPERATOR CLASS complex_abs_ops
    DEFAULT FOR TYPE complex USING btree AS
        OPERATOR        1       < ,
        OPERATOR        2       <= ,
        OPERATOR        3       = ,
        OPERATOR        4       >= ,
        OPERATOR        5       > ,
        FUNCTION        1       complex_abs_cmp(complex, complex);
```

And we're done! It should now be possible to create and use B-tree indexes on `complex` columns.

We could have written the operator entries more verbosely, as in:

```
        OPERATOR        1       < (complex, complex) ,
```

but there is no need to do so when the operators take the same data type we are defining the operator class for.

The above example assumes that you want to make this new operator class the default B-tree operator class for the `complex` data type. If you don't, just leave out the word `DEFAULT`.

35.14.5. Operator Classes and Operator Families

So far we have implicitly assumed that an operator class deals with only one data type. While there certainly can be only one data type in a particular index column, it is often useful to index operations that compare an indexed column to a value of a different data type. Also, if there is use for a cross-data-type operator in connection with an operator class, it is often the case that the other data type has a related operator class of its own. It is helpful to make the connections between related classes explicit, because this can aid the planner in optimizing SQL queries (particularly for B-tree operator classes, since the planner contains a great deal of knowledge about how to work with them).

To handle these needs, PostgreSQL uses the concept of an *operator family*. An operator family contains one or more operator classes, and can also contain indexable operators and corresponding support functions that belong to the family as a whole but not to any single class within the family. We say that such operators and functions are "loose" within the family, as opposed to being bound into a specific class. Typically each operator class contains single-data-type operators while cross-data-type operators are loose in the family.

All the operators and functions in an operator family must have compatible semantics, where the compatibility requirements are set by the index method. You might therefore wonder why bother to single out particular subsets of the family as operator classes; and indeed for many purposes the class divisions are irrelevant and the family is the only interesting grouping. The reason for defining operator classes is that they specify how much of the family is needed to support any particular index. If there is an index using an operator class, then that operator class cannot be dropped without dropping the index — but other parts of the operator family, namely other operator classes and loose operators, could be dropped. Thus, an operator class should be specified to contain the minimum set of operators and functions that are reasonably needed to work with an index on a specific data type, and then related but non-essential operators can be added as loose members of the operator family.

As an example, PostgreSQL has a built-in B-tree operator family `integer_ops`, which includes operator classes `int8_ops`, `int4_ops`, and `int2_ops` for indexes on `bigint` (`int8`), `integer` (`int4`), and `smallint` (`int2`) columns respectively. The family also contains cross-data-type comparison operators allowing any two of these types to be compared, so that an index on one of these types can be searched using a comparison value of another type. The family could be duplicated by these definitions:

```
CREATE OPERATOR FAMILY integer_ops USING btree;

CREATE OPERATOR CLASS int8_ops
DEFAULT FOR TYPE int8 USING btree FAMILY integer_ops AS
  -- standard int8 comparisons
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint8cmp(int8, int8) ,
  FUNCTION 2 btint8sortsupport(internal) ;

CREATE OPERATOR CLASS int4_ops
DEFAULT FOR TYPE int4 USING btree FAMILY integer_ops AS
  -- standard int4 comparisons
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
```

```
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint4cmp(int4, int4) ,
  FUNCTION 2 btint4sortsupport(internal) ;

CREATE OPERATOR CLASS int2_ops
DEFAULT FOR TYPE int2 USING btree FAMILY integer_ops AS
  -- standard int2 comparisons
  OPERATOR 1 < ,
  OPERATOR 2 <= ,
  OPERATOR 3 = ,
  OPERATOR 4 >= ,
  OPERATOR 5 > ,
  FUNCTION 1 btint2cmp(int2, int2) ,
  FUNCTION 2 btint2sortsupport(internal) ;

ALTER OPERATOR FAMILY integer_ops USING btree ADD
  -- cross-type comparisons int8 vs int2
  OPERATOR 1 < (int8, int2) ,
  OPERATOR 2 <= (int8, int2) ,
  OPERATOR 3 = (int8, int2) ,
  OPERATOR 4 >= (int8, int2) ,
  OPERATOR 5 > (int8, int2) ,
  FUNCTION 1 btint82cmp(int8, int2) ,

  -- cross-type comparisons int8 vs int4
  OPERATOR 1 < (int8, int4) ,
  OPERATOR 2 <= (int8, int4) ,
  OPERATOR 3 = (int8, int4) ,
  OPERATOR 4 >= (int8, int4) ,
  OPERATOR 5 > (int8, int4) ,
  FUNCTION 1 btint84cmp(int8, int4) ,

  -- cross-type comparisons int4 vs int2
  OPERATOR 1 < (int4, int2) ,
  OPERATOR 2 <= (int4, int2) ,
  OPERATOR 3 = (int4, int2) ,
  OPERATOR 4 >= (int4, int2) ,
  OPERATOR 5 > (int4, int2) ,
  FUNCTION 1 btint42cmp(int4, int2) ,

  -- cross-type comparisons int4 vs int8
  OPERATOR 1 < (int4, int8) ,
  OPERATOR 2 <= (int4, int8) ,
  OPERATOR 3 = (int4, int8) ,
  OPERATOR 4 >= (int4, int8) ,
  OPERATOR 5 > (int4, int8) ,
  FUNCTION 1 btint48cmp(int4, int8) ,

  -- cross-type comparisons int2 vs int8
  OPERATOR 1 < (int2, int8) ,
  OPERATOR 2 <= (int2, int8) ,
```

```
OPERATOR 3 = (int2, int8) ,
OPERATOR 4 >= (int2, int8) ,
OPERATOR 5 > (int2, int8) ,
FUNCTION 1 btint28cmp(int2, int8) ,

-- cross-type comparisons int2 vs int4
OPERATOR 1 < (int2, int4) ,
OPERATOR 2 <= (int2, int4) ,
OPERATOR 3 = (int2, int4) ,
OPERATOR 4 >= (int2, int4) ,
OPERATOR 5 > (int2, int4) ,
FUNCTION 1 btint24cmp(int2, int4) ;
```

Notice that this definition "overloads" the operator strategy and support function numbers: each number occurs multiple times within the family. This is allowed so long as each instance of a particular number has distinct input data types. The instances that have both input types equal to an operator class's input type are the primary operators and support functions for that operator class, and in most cases should be declared as part of the operator class rather than as loose members of the family.

In a B-tree operator family, all the operators in the family must sort compatibly, meaning that the transitive laws hold across all the data types supported by the family: "if A = B and B = C, then A = C", and "if A $<$ B and B $<$ C, then A $<$ C". Moreover, implicit or binary coercion casts between types represented in the operator family must not change the associated sort ordering. For each operator in the family there must be a support function having the same two input data types as the operator. It is recommended that a family be complete, i.e., for each combination of data types, all operators are included. Each operator class should include just the non-cross-type operators and support function for its data type.

To build a multiple-data-type hash operator family, compatible hash support functions must be created for each data type supported by the family. Here compatibility means that the functions are guaranteed to return the same hash code for any two values that are considered equal by the family's equality operators, even when the values are of different types. This is usually difficult to accomplish when the types have different physical representations, but it can be done in some cases. Furthermore, casting a value from one data type represented in the operator family to another data type also represented in the operator family via an implicit or binary coercion cast must not change the computed hash value. Notice that there is only one support function per data type, not one per equality operator. It is recommended that a family be complete, i.e., provide an equality operator for each combination of data types. Each operator class should include just the non-cross-type equality operator and the support function for its data type.

GiST, SP-GiST, and GIN indexes do not have any explicit notion of cross-data-type operations. The set of operators supported is just whatever the primary support functions for a given operator class can handle.

In BRIN, the requirements depends on the framework that provides the operator classes. For operator classes based on `minmax`, the behavior required is the same as for B-tree operator families: all the operators in the family must sort compatibly, and casts must not change the associated sort ordering.

> **Note:** Prior to PostgreSQL 8.3, there was no concept of operator families, and so any cross-data-type operators intended to be used with an index had to be bound directly into the index's operator class. While this approach still works, it is deprecated because it makes an index's dependencies too broad, and because the planner can handle cross-data-type comparisons more effectively when both data types have operators in the same operator family.

35.14.6. System Dependencies on Operator Classes

PostgreSQL uses operator classes to infer the properties of operators in more ways than just whether they can be used with indexes. Therefore, you might want to create operator classes even if you have no intention of indexing any columns of your data type.

In particular, there are SQL features such as `ORDER BY` and `DISTINCT` that require comparison and sorting of values. To implement these features on a user-defined data type, PostgreSQL looks for the default B-tree operator class for the data type. The "equals" member of this operator class defines the system's notion of equality of values for `GROUP BY` and `DISTINCT`, and the sort ordering imposed by the operator class defines the default `ORDER BY` ordering.

Comparison of arrays of user-defined types also relies on the semantics defined by the default B-tree operator class.

If there is no default B-tree operator class for a data type, the system will look for a default hash operator class. But since that kind of operator class only provides equality, in practice it is only enough to support array equality.

When there is no default operator class for a data type, you will get errors like "could not identify an ordering operator" if you try to use these SQL features with the data type.

> **Note:** In PostgreSQL versions before 7.4, sorting and grouping operations would implicitly use operators named =, <, and >. The new behavior of relying on default operator classes avoids having to make any assumption about the behavior of operators with particular names.

Another important point is that an operator that appears in a hash operator family is a candidate for hash joins, hash aggregation, and related optimizations. The hash operator family is essential here since it identifies the hash function(s) to use.

35.14.7. Ordering Operators

Some index access methods (currently, only GiST) support the concept of *ordering operators*. What we have been discussing so far are *search operators*. A search operator is one for which the index can be searched to find all rows satisfying `WHERE` *`indexed_column operator constant`*. Note that nothing is promised about the order in which the matching rows will be returned. In contrast, an ordering operator does not restrict the set of rows that can be returned, but instead determines their order. An ordering operator is one for which the index can be scanned to return rows in the order represented by `ORDER BY` *`indexed_column operator constant`*. The reason for defining ordering operators that way is that it supports nearest-neighbor searches, if the operator is one that measures distance. For example, a query like

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

finds the ten places closest to a given target point. A GiST index on the location column can do this efficiently because <-> is an ordering operator.

While search operators have to return Boolean results, ordering operators usually return some other type, such as float or numeric for distances. This type is normally not the same as the data type being indexed. To avoid hard-wiring assumptions about the behavior of different data types, the definition of an ordering

operator is required to name a B-tree operator family that specifies the sort ordering of the result data type. As was stated in the previous section, B-tree operator families define PostgreSQL's notion of ordering, so this is a natural representation. Since the point <-> operator returns `float8`, it could be specified in an operator class creation command like this:

```
OPERATOR 15    <-> (point, point) FOR ORDER BY float_ops
```

where `float_ops` is the built-in operator family that includes operations on `float8`. This declaration states that the index is able to return rows in order of increasing values of the <-> operator.

35.14.8. Special Features of Operator Classes

There are two special features of operator classes that we have not discussed yet, mainly because they are not useful with the most commonly used index methods.

Normally, declaring an operator as a member of an operator class (or family) means that the index method can retrieve exactly the set of rows that satisfy a `WHERE` condition using the operator. For example:

```
SELECT * FROM table WHERE integer_column < 4;
```

can be satisfied exactly by a B-tree index on the integer column. But there are cases where an index is useful as an inexact guide to the matching rows. For example, if a GiST index stores only bounding boxes for geometric objects, then it cannot exactly satisfy a `WHERE` condition that tests overlap between nonrectangular objects such as polygons. Yet we could use the index to find objects whose bounding box overlaps the bounding box of the target object, and then do the exact overlap test only on the objects found by the index. If this scenario applies, the index is said to be "lossy" for the operator. Lossy index searches are implemented by having the index method return a *recheck* flag when a row might or might not really satisfy the query condition. The core system will then test the original query condition on the retrieved row to see whether it should be returned as a valid match. This approach works if the index is guaranteed to return all the required rows, plus perhaps some additional rows, which can be eliminated by performing the original operator invocation. The index methods that support lossy searches (currently, GiST, SP-GiST and GIN) allow the support functions of individual operator classes to set the recheck flag, and so this is essentially an operator-class feature.

Consider again the situation where we are storing in the index only the bounding box of a complex object such as a polygon. In this case there's not much value in storing the whole polygon in the index entry — we might as well store just a simpler object of type `box`. This situation is expressed by the `STORAGE` option in `CREATE OPERATOR CLASS`: we'd write something like:

```
CREATE OPERATOR CLASS polygon_ops
    DEFAULT FOR TYPE polygon USING gist AS
        ...
        STORAGE box;
```

At present, only the GiST, GIN and BRIN index methods support a `STORAGE` type that's different from the column data type. The GiST `compress` and `decompress` support routines must deal with data-type conversion when `STORAGE` is used. In GIN, the `STORAGE` type identifies the type of the "key" values, which normally is different from the type of the indexed column — for example, an operator class for integer-array columns might have keys that are just integers. The GIN `extractValue` and `extractQuery` support routines are responsible for extracting keys from indexed values. BRIN is similar to GIN: the

STORAGE type identifies the type of the stored summary values, and operator classes' support procedures are responsible for interpreting the summary values correctly.

35.15. Packaging Related Objects into an Extension

A useful extension to PostgreSQL typically includes multiple SQL objects; for example, a new data type will require new functions, new operators, and probably new index operator classes. It is helpful to collect all these objects into a single package to simplify database management. PostgreSQL calls such a package an *extension*. To define an extension, you need at least a *script file* that contains the SQL commands to create the extension's objects, and a *control file* that specifies a few basic properties of the extension itself. If the extension includes C code, there will typically also be a shared library file into which the C code has been built. Once you have these files, a simple CREATE EXTENSION command loads the objects into your database.

The main advantage of using an extension, rather than just running the SQL script to load a bunch of "loose" objects into your database, is that PostgreSQL will then understand that the objects of the extension go together. You can drop all the objects with a single DROP EXTENSION command (no need to maintain a separate "uninstall" script). Even more useful, pg_dump knows that it should not dump the individual member objects of the extension — it will just include a CREATE EXTENSION command in dumps, instead. This vastly simplifies migration to a new version of the extension that might contain more or different objects than the old version. Note however that you must have the extension's control, script, and other files available when loading such a dump into a new database.

PostgreSQL will not let you drop an individual object contained in an extension, except by dropping the whole extension. Also, while you can change the definition of an extension member object (for example, via CREATE OR REPLACE FUNCTION for a function), bear in mind that the modified definition will not be dumped by pg_dump. Such a change is usually only sensible if you concurrently make the same change in the extension's script file. (But there are special provisions for tables containing configuration data; see below.)

The extension mechanism also has provisions for packaging modification scripts that adjust the definitions of the SQL objects contained in an extension. For example, if version 1.1 of an extension adds one function and changes the body of another function compared to 1.0, the extension author can provide an *update script* that makes just those two changes. The ALTER EXTENSION UPDATE command can then be used to apply these changes and track which version of the extension is actually installed in a given database.

The kinds of SQL objects that can be members of an extension are shown in the description of ALTER EXTENSION. Notably, objects that are database-cluster-wide, such as databases, roles, and tablespaces, cannot be extension members since an extension is only known within one database. (Although an extension script is not prohibited from creating such objects, if it does so they will not be tracked as part of the extension.) Also notice that while a table can be a member of an extension, its subsidiary objects such as indexes are not directly considered members of the extension. Another important point is that schemas can belong to extensions, but not vice versa: an extension as such has an unqualified name and does not exist "within" any schema. The extension's member objects, however, will belong to schemas whenever appropriate for their object types. It may or may not be appropriate for an extension to own the schema(s) its member objects are within.

35.15.1. Extension Files

The CREATE EXTENSION command relies on a control file for each extension, which must be named the same as the extension with a suffix of `.control`, and must be placed in the installation's `SHAREDIR/extension` directory. There must also be at least one SQL script file, which follows the naming pattern *`extension--version`*`.sql` (for example, `foo--1.0.sql` for version `1.0` of extension `foo`). By default, the script file(s) are also placed in the `SHAREDIR/extension` directory; but the control file can specify a different directory for the script file(s).

The file format for an extension control file is the same as for the `postgresql.conf` file, namely a list of *`parameter_name`* `=` *`value`* assignments, one per line. Blank lines and comments introduced by `#` are allowed. Be sure to quote any value that is not a single word or number.

A control file can set the following parameters:

`directory` (`string`)

The directory containing the extension's SQL script file(s). Unless an absolute path is given, the name is relative to the installation's `SHAREDIR` directory. The default behavior is equivalent to specifying `directory = 'extension'`.

`default_version` (`string`)

The default version of the extension (the one that will be installed if no version is specified in `CREATE EXTENSION`). Although this can be omitted, that will result in `CREATE EXTENSION` failing if no `VERSION` option appears, so you generally don't want to do that.

`comment` (`string`)

A comment (any string) about the extension. Alternatively, the comment can be set by means of the COMMENT command in the script file.

`encoding` (`string`)

The character set encoding used by the script file(s). This should be specified if the script files contain any non-ASCII characters. Otherwise the files will be assumed to be in the database encoding.

`module_pathname` (`string`)

The value of this parameter will be substituted for each occurrence of `MODULE_PATHNAME` in the script file(s). If it is not set, no substitution is made. Typically, this is set to `$libdir/`*`shared_library_name`* and then `MODULE_PATHNAME` is used in `CREATE FUNCTION` commands for C-language functions, so that the script files do not need to hard-wire the name of the shared library.

`requires` (`string`)

A list of names of extensions that this extension depends on, for example `requires = 'foo, bar'`. Those extensions must be installed before this one can be installed.

`superuser` (`boolean`)

If this parameter is `true` (which is the default), only superusers can create the extension or update it to a new version. If it is set to `false`, just the privileges required to execute the commands in the installation or update script are required.

`relocatable` (`boolean`)

An extension is *relocatable* if it is possible to move its contained objects into a different schema after initial creation of the extension. The default is `false`, i.e., the extension is not relocatable. See below for more information.

`schema` (`string`)

This parameter can only be set for non-relocatable extensions. It forces the extension to be loaded into exactly the named schema and not any other. See below for more information.

In addition to the primary control file *`extension`*`.control`, an extension can have secondary control files named in the style *`extension--version`*`.control`. If supplied, these must be located in the script file directory. Secondary control files follow the same format as the primary control file. Any parameters set in a secondary control file override the primary control file when installing or updating to that version of the extension. However, the parameters `directory` and `default_version` cannot be set in a secondary control file.

An extension's SQL script files can contain any SQL commands, except for transaction control commands (`BEGIN`, `COMMIT`, etc) and commands that cannot be executed inside a transaction block (such as `VACUUM`). This is because the script files are implicitly executed within a transaction block.

An extension's SQL script files can also contain lines beginning with `\echo`, which will be ignored (treated as comments) by the extension mechanism. This provision is commonly used to throw an error if the script file is fed to psql rather than being loaded via `CREATE EXTENSION` (see example script below). Without that, users might accidentally load the extension's contents as "loose" objects rather than as an extension, a state of affairs that's a bit tedious to recover from.

While the script files can contain any characters allowed by the specified encoding, control files should contain only plain ASCII, because there is no way for PostgreSQL to know what encoding a control file is in. In practice this is only an issue if you want to use non-ASCII characters in the extension's comment. Recommended practice in that case is to not use the control file `comment` parameter, but instead use `COMMENT ON EXTENSION` within a script file to set the comment.

35.15.2. Extension Relocatability

Users often wish to load the objects contained in an extension into a different schema than the extension's author had in mind. There are three supported levels of relocatability:

- A fully relocatable extension can be moved into another schema at any time, even after it's been loaded into a database. This is done with the `ALTER EXTENSION SET SCHEMA` command, which automatically renames all the member objects into the new schema. Normally, this is only possible if the extension contains no internal assumptions about what schema any of its objects are in. Also, the extension's objects must all be in one schema to begin with (ignoring objects that do not belong to any schema, such as procedural languages). Mark a fully relocatable extension by setting `relocatable = true` in its control file.
- An extension might be relocatable during installation but not afterwards. This is typically the case if the extension's script file needs to reference the target schema explicitly, for example in setting `search_path` properties for SQL functions. For such an extension, set `relocatable = false` in its control file, and use `@extschema@` to refer to the target schema in the script file. All occurrences of

this string will be replaced by the actual target schema's name before the script is executed. The user can set the target schema using the SCHEMA option of CREATE EXTENSION.

- If the extension does not support relocation at all, set relocatable = false in its control file, and also set schema to the name of the intended target schema. This will prevent use of the SCHEMA option of CREATE EXTENSION, unless it specifies the same schema named in the control file. This choice is typically necessary if the extension contains internal assumptions about schema names that can't be replaced by uses of @extschema@. The @extschema@ substitution mechanism is available in this case too, although it is of limited use since the schema name is determined by the control file.

In all cases, the script file will be executed with search_path initially set to point to the target schema; that is, CREATE EXTENSION does the equivalent of this:

```
SET LOCAL search_path TO @extschema@, pg_temp;
```

This allows the objects created by the script file to go into the target schema. The script file can change search_path if it wishes, but that is generally undesirable. search_path is restored to its previous setting upon completion of CREATE EXTENSION.

The target schema is determined by the schema parameter in the control file if that is given, otherwise by the SCHEMA option of CREATE EXTENSION if that is given, otherwise the current default object creation schema (the first one in the caller's search_path). When the control file schema parameter is used, the target schema will be created if it doesn't already exist, but in the other two cases it must already exist.

If any prerequisite extensions are listed in requires in the control file, their target schemas are added to the initial setting of search_path, following the new extension's target schema. This allows their objects to be visible to the new extension's script file.

For security, pg_temp is automatically appended to the end of search_path in all cases.

Although a non-relocatable extension can contain objects spread across multiple schemas, it is usually desirable to place all the objects meant for external use into a single schema, which is considered the extension's target schema. Such an arrangement works conveniently with the default setting of search_path during creation of dependent extensions.

35.15.3. Extension Configuration Tables

Some extensions include configuration tables, which contain data that might be added or changed by the user after installation of the extension. Ordinarily, if a table is part of an extension, neither the table's definition nor its content will be dumped by pg_dump. But that behavior is undesirable for a configuration table; any data changes made by the user need to be included in dumps, or the extension will behave differently after a dump and reload.

To solve this problem, an extension's script file can mark a table it has created as a configuration table, which will cause pg_dump to include the table's contents (not its definition) in dumps. To do that, call the function pg_extension_config_dump(regclass, text) after creating the table, for example

```
CREATE TABLE my_config (key text, value text);

SELECT pg_catalog.pg_extension_config_dump('my_config', '');
```

Any number of tables can be marked this way.

When the second argument of pg_extension_config_dump is an empty string, the entire contents of the table are dumped by pg_dump. This is usually only correct if the table is initially empty as created by the extension script. If there is a mixture of initial data and user-provided data in the table, the second argument of pg_extension_config_dump provides a WHERE condition that selects the data to be dumped. For example, you might do

```
CREATE TABLE my_config (key text, value text, standard_entry boolean);

SELECT pg_catalog.pg_extension_config_dump('my_config', 'WHERE NOT standard_entry');
```

and then make sure that standard_entry is true only in the rows created by the extension's script.

More complicated situations, such as initially-provided rows that might be modified by users, can be handled by creating triggers on the configuration table to ensure that modified rows are marked correctly.

You can alter the filter condition associated with a configuration table by calling pg_extension_config_dump again. (This would typically be useful in an extension update script.) The only way to mark a table as no longer a configuration table is to dissociate it from the extension with ALTER EXTENSION ... DROP TABLE.

Note that foreign key relationships between these tables will dictate the order in which the tables are dumped out by pg_dump. Specifically, pg_dump will attempt to dump the referenced-by table before the referencing table. As the foreign key relationships are set up at CREATE EXTENSION time (prior to data being loaded into the tables) circular dependencies are not supported. When circular dependencies exist, the data will still be dumped out but the dump will not be able to be restored directly and user intervention will be required.

35.15.4. Extension Updates

One advantage of the extension mechanism is that it provides convenient ways to manage updates to the SQL commands that define an extension's objects. This is done by associating a version name or number with each released version of the extension's installation script. In addition, if you want users to be able to update their databases dynamically from one version to the next, you should provide *update scripts* that make the necessary changes to go from one version to the next. Update scripts have names following the pattern *extension*--*old_version*--*target_version*.sql (for example, foo--1.0--1.1.sql contains the commands to modify version 1.0 of extension foo into version 1.1).

Given that a suitable update script is available, the command ALTER EXTENSION UPDATE will update an installed extension to the specified new version. The update script is run in the same environment that CREATE EXTENSION provides for installation scripts: in particular, search_path is set up in the same way, and any new objects created by the script are automatically added to the extension.

If an extension has secondary control files, the control parameters that are used for an update script are those associated with the script's target (new) version.

The update mechanism can be used to solve an important special case: converting a "loose" collection of objects into an extension. Before the extension mechanism was added to PostgreSQL (in 9.1), many people wrote extension modules that simply created assorted unpackaged objects. Given an existing database containing such objects, how can we convert the objects into a properly packaged extension? Dropping them and then doing a plain CREATE EXTENSION is one way, but it's not desirable if the objects have dependencies (for example, if there are table columns of a data type created by the extension). The way

to fix this situation is to create an empty extension, then use ALTER EXTENSION ADD to attach each pre-existing object to the extension, then finally create any new objects that are in the current extension version but were not in the unpackaged release. CREATE EXTENSION supports this case with its FROM *old_version* option, which causes it to not run the normal installation script for the target version, but instead the update script named *extension*--*old_version*--*target_version*.sql. The choice of the dummy version name to use as *old_version* is up to the extension author, though unpackaged is a common convention. If you have multiple prior versions you need to be able to update into extension style, use multiple dummy version names to identify them.

ALTER EXTENSION is able to execute sequences of update script files to achieve a requested update. For example, if only foo--1.0--1.1.sql and foo--1.1--2.0.sql are available, ALTER EXTENSION will apply them in sequence if an update to version 2.0 is requested when 1.0 is currently installed.

PostgreSQL doesn't assume anything about the properties of version names: for example, it does not know whether 1.1 follows 1.0. It just matches up the available version names and follows the path that requires applying the fewest update scripts. (A version name can actually be any string that doesn't contain -- or leading or trailing -.)

Sometimes it is useful to provide "downgrade" scripts, for example foo--1.1--1.0.sql to allow reverting the changes associated with version 1.1. If you do that, be careful of the possibility that a downgrade script might unexpectedly get applied because it yields a shorter path. The risky case is where there is a "fast path" update script that jumps ahead several versions as well as a downgrade script to the fast path's start point. It might take fewer steps to apply the downgrade and then the fast path than to move ahead one version at a time. If the downgrade script drops any irreplaceable objects, this will yield undesirable results.

To check for unexpected update paths, use this command:

```
SELECT * FROM pg_extension_update_paths('extension_name');
```

This shows each pair of distinct known version names for the specified extension, together with the update path sequence that would be taken to get from the source version to the target version, or NULL if there is no available update path. The path is shown in textual form with -- separators. You can use regexp_split_to_array(path,'--') if you prefer an array format.

35.15.5. Security Considerations for Extensions

Widely-distributed extensions should assume little about the database they occupy. Therefore, it's appropriate to write functions provided by an extension in a secure style that cannot be compromised by search-path-based attacks.

An extension that has the superuser property set to true must also consider security hazards for the actions taken within its installation and update scripts. It is not terribly difficult for a malicious user to create trojan-horse objects that will compromise later execution of a carelessly-written extension script, allowing that user to acquire superuser privileges.

Advice about writing functions securely is provided in Section 35.15.5.1 below, and advice about writing installation scripts securely is provided in Section 35.15.5.2.

35.15.5.1. Security Considerations for Extension Functions

SQL-language and PL-language functions provided by extensions are at risk of search-path-based attacks when they are executed, since parsing of these functions occurs at execution time not creation time.

The `CREATE FUNCTION` reference page contains advice about writing `SECURITY DEFINER` functions safely. It's good practice to apply those techniques for any function provided by an extension, since the function might be called by a high-privilege user.

If you cannot set the `search_path` to contain only secure schemas, assume that each unqualified name could resolve to an object that a malicious user has defined. Beware of constructs that depend on `search_path` implicitly; for example, `IN` and `CASE` *`expression`* `WHEN` always select an operator using the search path. In their place, use `OPERATOR(`*`schema`*`.=) ANY` and `CASE WHEN` *`expression`*.

A general-purpose extension usually should not assume that it's been installed into a secure schema, which means that even schema-qualified references to its own objects are not entirely risk-free. For example, if the extension has defined a function `myschema.myfunc(bigint)` then a call such as `myschema.myfunc(42)` could be captured by a hostile function `myschema.myfunc(integer)`. Be careful that the data types of function and operator parameters exactly match the declared argument types, using explicit casts where necessary.

35.15.5.2. Security Considerations for Extension Scripts

An extension installation or update script should be written to guard against search-path-based attacks occurring when the script executes. If an object reference in the script can be made to resolve to some other object than the script author intended, then a compromise might occur immediately, or later when the mis-defined extension object is used.

DDL commands such as `CREATE FUNCTION` and `CREATE OPERATOR CLASS` are generally secure, but beware of any command having a general-purpose expression as a component. For example, `CREATE VIEW` needs to be vetted, as does a `DEFAULT` expression in `CREATE FUNCTION`.

Sometimes an extension script might need to execute general-purpose SQL, for example to make catalog adjustments that aren't possible via DDL. Be careful to execute such commands with a secure `search_path`; do *not* trust the path provided by `CREATE/ALTER EXTENSION` to be secure. Best practice is to temporarily set `search_path` to `'pg_catalog, pg_temp'` and insert references to the extension's installation schema explicitly where needed. (This practice might also be helpful for creating views.) Examples can be found in the `contrib` modules in the PostgreSQL source code distribution.

Cross-extension references are extremely difficult to make fully secure, partially because of uncertainty about which schema the other extension is in. The hazards are reduced if both extensions are installed in the same schema, because then a hostile object cannot be placed ahead of the referenced extension in the installation-time `search_path`. However, no mechanism currently exists to require that.

Do *not* use `CREATE OR REPLACE FUNCTION`, except in an update script that must change the definition of a function that is known to be an extension member already. (Likewise for other `OR REPLACE` options.) Using `OR REPLACE` unnecessarily not only has a risk of accidentally overwriting someone else's function, but it creates a security hazard since the overwritten function would still be owned by its original owner, who could modify it.

35.15.6. Extension Example

Here is a complete example of an SQL-only extension, a two-element composite type that can store any type of value in its slots, which are named “k” and “v”. Non-text values are automatically coerced to text for storage.

The script file `pair--1.0.sql` looks like this:

```
-- complain if script is sourced in psql, rather than via CREATE EXTENSION
\echo Use "CREATE EXTENSION pair" to load this file. \quit

CREATE TYPE pair AS ( k text, v text );

CREATE FUNCTION pair(text, text)
RETURNS pair LANGUAGE SQL AS 'SELECT ROW($1, $2)::@extschema@.pair;';

CREATE OPERATOR ~> (LEFTARG = text, RIGHTARG = text, PROCEDURE = pair);

-- "SET search_path" is easy to get right, but qualified names perform better.
CREATE FUNCTION lower(pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW(lower($1.k), lower($1.v))::@extschema@.pair;'
SET search_path = pg_temp;

CREATE FUNCTION pair_concat(pair, pair)
RETURNS pair LANGUAGE SQL
AS 'SELECT ROW($1.k OPERATOR(pg_catalog.||) $2.k,
               $1.v OPERATOR(pg_catalog.||) $2.v)::@extschema@.pair;';
```

The control file `pair.control` looks like this:

```
# pair extension
comment = 'A key/value pair data type'
default_version = '1.0'
# cannot be relocatable because of use of @extschema@
relocatable = false
```

While you hardly need a makefile to install these two files into the correct directory, you could use a `Makefile` containing this:

```
EXTENSION = pair
DATA = pair--1.0.sql

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

This makefile relies on PGXS, which is described in Section 35.16. The command `make install` will install the control and script files into the correct directory as reported by pg_config.

Once the files are installed, use the CREATE EXTENSION command to load the objects into any particular database.

35.16. Extension Building Infrastructure

If you are thinking about distributing your PostgreSQL extension modules, setting up a portable build system for them can be fairly difficult. Therefore the PostgreSQL installation provides a build infrastructure for extensions, called PGXS, so that simple extension modules can be built simply against an already installed server. PGXS is mainly intended for extensions that include C code, although it can be used for pure-SQL extensions too. Note that PGXS is not intended to be a universal build system framework that can be used to build any software interfacing to PostgreSQL; it simply automates common build rules for simple server extension modules. For more complicated packages, you might need to write your own build system.

To use the PGXS infrastructure for your extension, you must write a simple makefile. In the makefile, you need to set some variables and include the global PGXS makefile. Here is an example that builds an extension module named `isbn_issn`, consisting of a shared library containing some C code, an extension control file, a SQL script, and a documentation text file:

```
MODULES = isbn_issn
EXTENSION = isbn_issn
DATA = isbn_issn--1.0.sql
DOCS = README.isbn_issn

PG_CONFIG = pg_config
PGXS := $(shell $(PG_CONFIG) --pgxs)
include $(PGXS)
```

The last three lines should always be the same. Earlier in the file, you assign variables or add custom make rules.

Set one of these three variables to specify what is built:

`MODULES`

list of shared-library objects to be built from source files with same stem (do not include library suffixes in this list)

`MODULE_big`

a shared library to build from multiple source files (list object files in `OBJS`)

`PROGRAM`

an executable program to build (list object files in `OBJS`)

The following variables can also be set:

`EXTENSION`

extension name(s); for each name you must provide an *`extension`*`.control` file, which will be installed into *`prefix`*`/share/extension`

MODULEDIR

subdirectory of *prefix*/share into which DATA and DOCS files should be installed (if not set, default is extension if EXTENSION is set, or contrib if not)

DATA

random files to install into *prefix*/share/$MODULEDIR

DATA_built

random files to install into *prefix*/share/$MODULEDIR, which need to be built first

DATA_TSEARCH

random files to install under *prefix*/share/tsearch_data

DOCS

random files to install under *prefix*/doc/$MODULEDIR

SCRIPTS

script files (not binaries) to install into *prefix*/bin

SCRIPTS_built

script files (not binaries) to install into *prefix*/bin, which need to be built first

REGRESS

list of regression test cases (without suffix), see below

REGRESS_OPTS

additional switches to pass to pg_regress

EXTRA_CLEAN

extra files to remove in make clean

PG_CPPFLAGS

will be prepended to CPPFLAGS

PG_CFLAGS

will be appended to CFLAGS

PG_CXXFLAGS

will be appended to CXXFLAGS

PG_LDFLAGS

will be prepended to LDFLAGS

PG_LIBS

will be added to PROGRAM link line

SHLIB_LINK

will be added to MODULE_big link line

PG_CONFIG

path to pg_config program for the PostgreSQL installation to build against (typically just pg_config to use the first one in your PATH)

Put this makefile as Makefile in the directory which holds your extension. Then you can do make to compile, and then make install to install your module. By default, the extension is compiled and installed for the PostgreSQL installation that corresponds to the first pg_config program found in your PATH. You can use a different installation by setting PG_CONFIG to point to its pg_config program, either within the makefile or on the make command line.

You can also run make in a directory outside the source tree of your extension, if you want to keep the build directory separate. This procedure is also called a *VPATH* build. Here's how:

```
mkdir build_dir
cd build_dir
make -f /path/to/extension/source/tree/Makefile
make -f /path/to/extension/source/tree/Makefile install
```

Alternatively, you can set up a directory for a VPATH build in a similar way to how it is done for the core code. One way to do this is using the core script config/prep_buildtree. Once this has been done you can build by setting the make variable VPATH like this:

```
make VPATH=/path/to/extension/source/tree
make VPATH=/path/to/extension/source/tree install
```

This procedure can work with a greater variety of directory layouts.

The scripts listed in the REGRESS variable are used for regression testing of your module, which can be invoked by make installcheck after doing make install. For this to work you must have a running PostgreSQL server. The script files listed in REGRESS must appear in a subdirectory named sql/ in your extension's directory. These files must have extension .sql, which must not be included in the REGRESS list in the makefile. For each test there should also be a file containing the expected output in a subdirectory named expected/, with the same stem and extension .out. make installcheck executes each test script with psql, and compares the resulting output to the matching expected file. Any differences will be written to the file regression.diffs in diff -c format. Note that trying to run a test that is missing its expected file will be reported as "trouble", so make sure you have all expected files.

Tip: The easiest way to create the expected files is to create empty files, then do a test run (which will of course report differences). Inspect the actual result files found in the results/ directory, then copy them to expected/ if they match what you expect from the test.

Chapter 36. Triggers

This chapter provides general information about writing trigger functions. Trigger functions can be written in most of the available procedural languages, including PL/pgSQL (Chapter 40), PL/Tcl (Chapter 41), PL/Perl (Chapter 42), and PL/Python (Chapter 43). After reading this chapter, you should consult the chapter for your favorite procedural language to find out the language-specific details of writing a trigger in it.

It is also possible to write a trigger function in C, although most people find it easier to use one of the procedural languages. It is not currently possible to write a trigger function in the plain SQL function language.

36.1. Overview of Trigger Behavior

A trigger is a specification that the database should automatically execute a particular function whenever a certain type of operation is performed. Triggers can be attached to tables, views, and foreign tables.

On tables and foreign tables, triggers can be defined to execute either before or after any `INSERT`, `UPDATE`, or `DELETE` operation, either once per modified row, or once per SQL statement. If an `INSERT` contains an `ON CONFLICT DO UPDATE` clause, it is possible that the effects of a BEFORE insert trigger and a BEFORE update trigger can both be applied together, if a reference to an `EXCLUDED` column appears. `UPDATE` triggers can moreover be set to fire only if certain columns are mentioned in the `SET` clause of the `UPDATE` statement. Triggers can also fire for `TRUNCATE` statements. If a trigger event occurs, the trigger's function is called at the appropriate time to handle the event. Foreign tables do not support the TRUNCATE statement at all.

On views, triggers can be defined to execute instead of `INSERT`, `UPDATE`, or `DELETE` operations. Such `INSTEAD OF` triggers are fired once for each row that needs to be modified in the view. It is the responsibility of the trigger's function to perform the necessary modifications to the view's underlying base table(s) and, where appropriate, return the modified row as it will appear in the view. Triggers on views can also be defined to execute once per SQL statement, before or after `INSERT`, `UPDATE`, or `DELETE` operations. However, such triggers are fired only if there is also an `INSTEAD OF` trigger on the view. Otherwise, any statement targeting the view must be rewritten into a statement affecting its underlying base table(s), and then the triggers that will be fired are the ones attached to the base table(s).

The trigger function must be defined before the trigger itself can be created. The trigger function must be declared as a function taking no arguments and returning type `trigger`. (The trigger function receives its input through a specially-passed `TriggerData` structure, not in the form of ordinary function arguments.)

Once a suitable trigger function has been created, the trigger is established with CREATE TRIGGER. The same trigger function can be used for multiple triggers.

PostgreSQL offers both *per-row* triggers and *per-statement* triggers. With a per-row trigger, the trigger function is invoked once for each row that is affected by the statement that fired the trigger. In contrast, a per-statement trigger is invoked only once when an appropriate statement is executed, regardless of the number of rows affected by that statement. In particular, a statement that affects zero rows will still result in the execution of any applicable per-statement triggers. These two types of triggers are sometimes called *row-level* triggers and *statement-level* triggers, respectively. Triggers on `TRUNCATE` may only be defined at statement level. On views, triggers that fire before or after may only be defined at statement level, while triggers that fire instead of an `INSERT`, `UPDATE`, or `DELETE` may only be defined at row level.

Triggers are also classified according to whether they fire *before*, *after*, or *instead of* the operation. These are referred to as BEFORE triggers, AFTER triggers, and INSTEAD OF triggers respectively. Statement-level BEFORE triggers naturally fire before the statement starts to do anything, while statement-level AFTER triggers fire at the very end of the statement. These types of triggers may be defined on tables or views. Row-level BEFORE triggers fire immediately before a particular row is operated on, while row-level AFTER triggers fire at the end of the statement (but before any statement-level AFTER triggers). These types of triggers may only be defined on tables and foreign tables. Row-level INSTEAD OF triggers may only be defined on views, and fire immediately as each row in the view is identified as needing to be operated on.

If an INSERT contains an ON CONFLICT DO UPDATE clause, it is possible that the effects of all row-level BEFORE INSERT triggers and all row-level BEFORE UPDATE triggers can both be applied in a way that is apparent from the final state of the updated row, if an EXCLUDED column is referenced. There need not be an EXCLUDED column reference for both sets of BEFORE row-level triggers to execute, though. The possibility of surprising outcomes should be considered when there are both BEFORE INSERT and BEFORE UPDATE row-level triggers that both affect a row being inserted/updated (this can still be problematic if the modifications are more or less equivalent if they're not also idempotent). Note that statement-level UPDATE triggers are executed when ON CONFLICT DO UPDATE is specified, regardless of whether or not any rows were affected by the UPDATE (and regardless of whether the alternative UPDATE path was ever taken). An INSERT with an ON CONFLICT DO UPDATE clause will execute statement-level BEFORE INSERT triggers first, then statement-level BEFORE UPDATE triggers, followed by statement-level AFTER UPDATE triggers and finally statement-level AFTER INSERT triggers.

Trigger functions invoked by per-statement triggers should always return NULL. Trigger functions invoked by per-row triggers can return a table row (a value of type HeapTuple) to the calling executor, if they choose. A row-level trigger fired before an operation has the following choices:

- It can return NULL to skip the operation for the current row. This instructs the executor to not perform the row-level operation that invoked the trigger (the insertion, modification, or deletion of a particular table row).
- For row-level INSERT and UPDATE triggers only, the returned row becomes the row that will be inserted or will replace the row being updated. This allows the trigger function to modify the row being inserted or updated.

A row-level BEFORE trigger that does not intend to cause either of these behaviors must be careful to return as its result the same row that was passed in (that is, the NEW row for INSERT and UPDATE triggers, the OLD row for DELETE triggers).

A row-level INSTEAD OF trigger should either return NULL to indicate that it did not modify any data from the view's underlying base tables, or it should return the view row that was passed in (the NEW row for INSERT and UPDATE operations, or the OLD row for DELETE operations). A nonnull return value is used to signal that the trigger performed the necessary data modifications in the view. This will cause the count of the number of rows affected by the command to be incremented. For INSERT and UPDATE operations only, the trigger may modify the NEW row before returning it. This will change the data returned by INSERT RETURNING or UPDATE RETURNING, and is useful when the view will not show exactly the same data that was provided.

The return value is ignored for row-level triggers fired after an operation, and so they can return NULL.

If more than one trigger is defined for the same event on the same relation, the triggers will be fired in alphabetical order by trigger name. In the case of BEFORE and INSTEAD OF triggers, the possibly-modified row returned by each trigger becomes the input to the next trigger. If any BEFORE or INSTEAD

OF trigger returns NULL, the operation is abandoned for that row and subsequent triggers are not fired (for that row).

A trigger definition can also specify a Boolean WHEN condition, which will be tested to see whether the trigger should be fired. In row-level triggers the WHEN condition can examine the old and/or new values of columns of the row. (Statement-level triggers can also have WHEN conditions, although the feature is not so useful for them.) In a BEFORE trigger, the WHEN condition is evaluated just before the function is or would be executed, so using WHEN is not materially different from testing the same condition at the beginning of the trigger function. However, in an AFTER trigger, the WHEN condition is evaluated just after the row update occurs, and it determines whether an event is queued to fire the trigger at the end of statement. So when an AFTER trigger's WHEN condition does not return true, it is not necessary to queue an event nor to re-fetch the row at end of statement. This can result in significant speedups in statements that modify many rows, if the trigger only needs to be fired for a few of the rows. INSTEAD OF triggers do not support WHEN conditions.

Typically, row-level BEFORE triggers are used for checking or modifying the data that will be inserted or updated. For example, a BEFORE trigger might be used to insert the current time into a timestamp column, or to check that two elements of the row are consistent. Row-level AFTER triggers are most sensibly used to propagate the updates to other tables, or make consistency checks against other tables. The reason for this division of labor is that an AFTER trigger can be certain it is seeing the final value of the row, while a BEFORE trigger cannot; there might be other BEFORE triggers firing after it. If you have no specific reason to make a trigger BEFORE or AFTER, the BEFORE case is more efficient, since the information about the operation doesn't have to be saved until end of statement.

If a trigger function executes SQL commands then these commands might fire triggers again. This is known as cascading triggers. There is no direct limitation on the number of cascade levels. It is possible for cascades to cause a recursive invocation of the same trigger; for example, an INSERT trigger might execute a command that inserts an additional row into the same table, causing the INSERT trigger to be fired again. It is the trigger programmer's responsibility to avoid infinite recursion in such scenarios.

When a trigger is being defined, arguments can be specified for it. The purpose of including arguments in the trigger definition is to allow different triggers with similar requirements to call the same function. As an example, there could be a generalized trigger function that takes as its arguments two column names and puts the current user in one and the current time stamp in the other. Properly written, this trigger function would be independent of the specific table it is triggering on. So the same function could be used for INSERT events on any table with suitable columns, to automatically track creation of records in a transaction table for example. It could also be used to track last-update events if defined as an UPDATE trigger.

Each programming language that supports triggers has its own method for making the trigger input data available to the trigger function. This input data includes the type of trigger event (e.g., INSERT or UPDATE) as well as any arguments that were listed in CREATE TRIGGER. For a row-level trigger, the input data also includes the NEW row for INSERT and UPDATE triggers, and/or the OLD row for UPDATE and DELETE triggers. Statement-level triggers do not currently have any way to examine the individual row(s) modified by the statement.

36.2. Visibility of Data Changes

If you execute SQL commands in your trigger function, and these commands access the table that the

trigger is for, then you need to be aware of the data visibility rules, because they determine whether these SQL commands will see the data change that the trigger is fired for. Briefly:

- Statement-level triggers follow simple visibility rules: none of the changes made by a statement are visible to statement-level triggers that are invoked before the statement, whereas all modifications are visible to statement-level `AFTER` triggers.
- The data change (insertion, update, or deletion) causing the trigger to fire is naturally *not* visible to SQL commands executed in a row-level `BEFORE` trigger, because it hasn't happened yet.
- However, SQL commands executed in a row-level `BEFORE` trigger *will* see the effects of data changes for rows previously processed in the same outer command. This requires caution, since the ordering of these change events is not in general predictable; a SQL command that affects multiple rows can visit the rows in any order.
- Similarly, a row-level `INSTEAD OF` trigger will see the effects of data changes made by previous firings of `INSTEAD OF` triggers in the same outer command.
- When a row-level `AFTER` trigger is fired, all data changes made by the outer command are already complete, and are visible to the invoked trigger function.

If your trigger function is written in any of the standard procedural languages, then the above statements apply only if the function is declared `VOLATILE`. Functions that are declared `STABLE` or `IMMUTABLE` will not see changes made by the calling command in any case.

Further information about data visibility rules can be found in Section 44.4. The example in Section 36.4 contains a demonstration of these rules.

36.3. Writing Trigger Functions in C

This section describes the low-level details of the interface to a trigger function. This information is only needed when writing trigger functions in C. If you are using a higher-level language then these details are handled for you. In most cases you should consider using a procedural language before writing your triggers in C. The documentation of each procedural language explains how to write a trigger in that language.

Trigger functions must use the "version 1" function manager interface.

When a function is called by the trigger manager, it is not passed any normal arguments, but it is passed a "context" pointer pointing to a `TriggerData` structure. C functions can check whether they were called from the trigger manager or not by executing the macro:

```
CALLED_AS_TRIGGER(fcinfo)
```

which expands to:

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, TriggerData))
```

If this returns true, then it is safe to cast `fcinfo->context` to type `TriggerData *` and make use of the pointed-to `TriggerData` structure. The function must *not* alter the `TriggerData` structure or any of the data it points to.

`struct TriggerData` is defined in `commands/trigger.h`:

```
typedef struct TriggerData
{
    NodeTag       type;
    TriggerEvent  tg_event;
    Relation      tg_relation;
    HeapTuple     tg_trigtuple;
    HeapTuple     tg_newtuple;
    Trigger      *tg_trigger;
    Buffer        tg_trigtuplebuf;
    Buffer        tg_newtuplebuf;
} TriggerData;
```

where the members are defined as follows:

`type`

Always `T_TriggerData`.

`tg_event`

Describes the event for which the function is called. You can use the following macros to examine `tg_event`:

`TRIGGER_FIRED_BEFORE(tg_event)`

Returns true if the trigger fired before the operation.

`TRIGGER_FIRED_AFTER(tg_event)`

Returns true if the trigger fired after the operation.

`TRIGGER_FIRED_INSTEAD(tg_event)`

Returns true if the trigger fired instead of the operation.

`TRIGGER_FIRED_FOR_ROW(tg_event)`

Returns true if the trigger fired for a row-level event.

`TRIGGER_FIRED_FOR_STATEMENT(tg_event)`

Returns true if the trigger fired for a statement-level event.

`TRIGGER_FIRED_BY_INSERT(tg_event)`

Returns true if the trigger was fired by an `INSERT` command.

`TRIGGER_FIRED_BY_UPDATE(tg_event)`

Returns true if the trigger was fired by an `UPDATE` command.

`TRIGGER_FIRED_BY_DELETE(tg_event)`

Returns true if the trigger was fired by a `DELETE` command.

`TRIGGER_FIRED_BY_TRUNCATE(tg_event)`

Returns true if the trigger was fired by a `TRUNCATE` command.

`tg_relation`

A pointer to a structure describing the relation that the trigger fired for. Look at `utils/rel.h` for details about this structure. The most interesting things are `tg_relation->rd_att` (descriptor of the relation tuples) and `tg_relation->rd_rel->relname` (relation name; the type is not `char*` but `NameData`; use `SPI_getrelname(tg_relation)` to get a `char*` if you need a copy of the name).

`tg_trigtuple`

A pointer to the row for which the trigger was fired. This is the row being inserted, updated, or deleted. If this trigger was fired for an `INSERT` or `DELETE` then this is what you should return from the function if you don't want to replace the row with a different one (in the case of `INSERT`) or skip the operation. For triggers on foreign tables, values of system columns herein are unspecified.

`tg_newtuple`

A pointer to the new version of the row, if the trigger was fired for an `UPDATE`, and `NULL` if it is for an `INSERT` or a `DELETE`. This is what you have to return from the function if the event is an `UPDATE` and you don't want to replace this row by a different one or skip the operation. For triggers on foreign tables, values of system columns herein are unspecified.

`tg_trigger`

A pointer to a structure of type `Trigger`, defined in `utils/reltrigger.h`:

```
typedef struct Trigger
{
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    int16       tgtype;
    char        tgenabled;
    bool        tgisinternal;
    Oid         tgconstrrelid;
    Oid         tgconstrindid;
    Oid         tgconstraint;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgnattr;
    int16      *tgattr;
    char      **tgargs;
    char       *tgqual;
} Trigger;
```

where `tgname` is the trigger's name, `tgnargs` is the number of arguments in `tgargs`, and `tgargs` is an array of pointers to the arguments specified in the `CREATE TRIGGER` statement. The other members are for internal use only.

`tg_trigtuplebuf`

The buffer containing `tg_trigtuple`, or `InvalidBuffer` if there is no such tuple or it is not stored in a disk buffer.

`tg_newtuplebuf`

The buffer containing `tg_newtuple`, or `InvalidBuffer` if there is no such tuple or it is not stored in a disk buffer.

A trigger function must return either a `HeapTuple` pointer or a `NULL` pointer (*not* an SQL null value, that is, do not set `isNull` true). Be careful to return either `tg_trigtuple` or `tg_newtuple`, as appropriate, if you don't want to modify the row being operated on.

36.4. A Complete Trigger Example

Here is a very simple example of a trigger function written in C. (Examples of triggers written in procedural languages can be found in the documentation of the procedural languages.)

The function `trigf` reports the number of rows in the table `ttest` and skips the actual operation if the command attempts to insert a null value into the column `x`. (So the trigger acts as a not-null constraint but doesn't abort the transaction.)

First, the table definition:

```
CREATE TABLE ttest (
    x integer
);
```

This is the source code of the trigger function:

```
#include "postgres.h"
#include "executor/spi.h"       /* this is what you need to work with SPI */
#include "commands/trigger.h"   /* ... triggers ... */
#include "utils/rel.h"          /* ... and relations */

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(trigf);

Datum
trigf(PG_FUNCTION_ARGS)
{
    TriggerData *trigdata = (TriggerData *) fcinfo->context;
    TupleDesc   tupdesc;
    HeapTuple   rettuple;
    char       *when;
    bool        checknull = false;
    bool        isnull;
    int         ret, i;
```

```
    /* make sure it's called as a trigger at all */
    if (!CALLED_AS_TRIGGER(fcinfo))
        elog(ERROR, "trigf: not called by trigger manager");

    /* tuple to return to executor */
    if (TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event))
        rettuple = trigdata->tg_newtuple;
    else
        rettuple = trigdata->tg_trigtuple;

    /* check for null values */
    if (!TRIGGER_FIRED_BY_DELETE(trigdata->tg_event)
        && TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        checknull = true;

    if (TRIGGER_FIRED_BEFORE(trigdata->tg_event))
        when = "before";
    else
        when = "after ";

    tupdesc = trigdata->tg_relation->rd_att;

    /* connect to SPI manager */
    if ((ret = SPI_connect()) < 0)
        elog(ERROR, "trigf (fired %s): SPI_connect returned %d", when, ret);

    /* get number of rows in table */
    ret = SPI_exec("SELECT count(*) FROM ttest", 0);

    if (ret < 0)
        elog(ERROR, "trigf (fired %s): SPI_exec returned %d", when, ret);

    /* count(*) returns int8, so be careful to convert */
    i = DatumGetInt64(SPI_getbinval(SPI_tuptable->vals[0],
                                    SPI_tuptable->tupdesc,
                                    1,
                                    &isnull));

    elog (INFO, "trigf (fired %s): there are %d rows in ttest", when, i);

    SPI_finish();

    if (checknull)
    {
        SPI_getbinval(rettuple, tupdesc, 1, &isnull);
        if (isnull)
            rettuple = NULL;
    }

    return PointerGetDatum(rettuple);
}
```

After you have compiled the source code (see Section 35.9.6), declare the function and the triggers:

```
CREATE FUNCTION trigf() RETURNS trigger
    AS 'filename'
    LANGUAGE C;

CREATE TRIGGER tbefore BEFORE INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();

CREATE TRIGGER tafter AFTER INSERT OR UPDATE OR DELETE ON ttest
    FOR EACH ROW EXECUTE PROCEDURE trigf();
```

Now you can test the operation of the trigger:

```
=> INSERT INTO ttest VALUES (NULL);
INFO:  trigf (fired before): there are 0 rows in ttest
INSERT 0 0

-- Insertion skipped and AFTER trigger is not fired

=> SELECT * FROM ttest;
 x
---
(0 rows)

=> INSERT INTO ttest VALUES (1);
INFO:  trigf (fired before): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 1 rows in ttest
                                       ^^^^^^^^
                             remember what we said about visibility.
INSERT 167793 1
vac=> SELECT * FROM ttest;
 x
---
 1
(1 row)

=> INSERT INTO ttest SELECT x * 2 FROM ttest;
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
                                       ^^^^^^
                             remember what we said about visibility.
INSERT 167794 1
=> SELECT * FROM ttest;
 x
---
 1
 2
(2 rows)
```

```
=> UPDATE ttest SET x = NULL WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
UPDATE 0
=> UPDATE ttest SET x = 4 WHERE x = 2;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired after ): there are 2 rows in ttest
UPDATE 1
vac=> SELECT * FROM ttest;
 x
---
 1
 4
(2 rows)

=> DELETE FROM ttest;
INFO:  trigf (fired before): there are 2 rows in ttest
INFO:  trigf (fired before): there are 1 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
INFO:  trigf (fired after ): there are 0 rows in ttest
                                       ^^^^^^
                             remember what we said about visibility.
DELETE 2
=> SELECT * FROM ttest;
 x
---
(0 rows)
```

There are more complex examples in `src/test/regress/regress.c` and in spi.

Chapter 37. Event Triggers

To supplement the trigger mechanism discussed in Chapter 36, PostgreSQL also provides event triggers. Unlike regular triggers, which are attached to a single table and capture only DML events, event triggers are global to a particular database and are capable of capturing DDL events.

Like regular triggers, event triggers can be written in any procedural language that includes event trigger support, or in C, but not in plain SQL.

37.1. Overview of Event Trigger Behavior

An event trigger fires whenever the event with which it is associated occurs in the database in which it is defined. Currently, the only supported events are `ddl_command_start`, `ddl_command_end`, `table_rewrite` and `sql_drop`. Support for additional events may be added in future releases.

The `ddl_command_start` event occurs just before the execution of a `CREATE`, `ALTER`, `DROP`, `SECURITY LABEL`, `COMMENT`, `GRANT` or `REVOKE` command. No check whether the affected object exists or doesn't exist is performed before the event trigger fires. As an exception, however, this event does not occur for DDL commands targeting shared objects — databases, roles, and tablespaces — or for commands targeting event triggers themselves. The event trigger mechanism does not support these object types. `ddl_command_start` also occurs just before the execution of a `SELECT INTO` command, since this is equivalent to `CREATE TABLE AS`.

The `ddl_command_end` event occurs just after the execution of this same set of commands. To obtain more details on the DDL operations that took place, use the set-returning function `pg_event_trigger_ddl_commands()` from the `ddl_command_end` event trigger code (see Section 9.28). Note that the trigger fires after the actions have taken place (but before the transaction commits), and thus the system catalogs can be read as already changed.

The `sql_drop` event occurs just before the `ddl_command_end` event trigger for any operation that drops database objects. To list the objects that have been dropped, use the set-returning function `pg_event_trigger_dropped_objects()` from the `sql_drop` event trigger code (see Section 9.28). Note that the trigger is executed after the objects have been deleted from the system catalogs, so it's not possible to look them up anymore.

The `table_rewrite` event occurs just before a table is rewritten by some actions of the commands `ALTER TABLE` and `ALTER TYPE`. While other control statements are available to rewrite a table, like `CLUSTER` and `VACUUM`, the `table_rewrite` event is not triggered by them.

Event triggers (like other functions) cannot be executed in an aborted transaction. Thus, if a DDL command fails with an error, any associated `ddl_command_end` triggers will not be executed. Conversely, if a `ddl_command_start` trigger fails with an error, no further event triggers will fire, and no attempt will be made to execute the command itself. Similarly, if a `ddl_command_end` trigger fails with an error, the effects of the DDL statement will be rolled back, just as they would be in any other case where the containing transaction aborts.

For a complete list of commands supported by the event trigger mechanism, see Section 37.2.

Event triggers are created using the command CREATE EVENT TRIGGER. In order to create an event trigger, you must first create a function with the special return type `event_trigger`. This function need

not (and may not) return a value; the return type serves merely as a signal that the function is to be invoked as an event trigger.

If more than one event trigger is defined for a particular event, they will fire in alphabetical order by trigger name.

A trigger definition can also specify a `WHEN` condition so that, for example, a `ddl_command_start` trigger can be fired only for particular commands which the user wishes to intercept. A common use of such triggers is to restrict the range of DDL operations which users may perform.

37.2. Event Trigger Firing Matrix

Table 37-1 lists all commands for which event triggers are supported.

Table 37-1. Event Trigger Support by Command Tag

Command Tag	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	Notes
ALTER AGGREGATE	X	X	-	-	
ALTER COLLATION	X	X	-	-	
ALTER CONVERSION	X	X	-	-	
ALTER DOMAIN	X	X	-	-	
ALTER DEFAULT PRIVILEGES	X	X	-	-	
ALTER EXTENSION	X	X	-	-	
ALTER FOREIGN DATA WRAPPER	X	X	-	-	
ALTER FOREIGN TABLE	X	X	X	-	
ALTER FUNCTION	X	X	-	-	
ALTER LANGUAGE	X	X	-	-	
ALTER LARGE OBJECT	X	X	-	-	

Command Tag	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	Notes
ALTER MATERIALIZED VIEW	X	X	-	-	
ALTER OPERATOR	X	X	-	-	
ALTER OPERATOR CLASS	X	X	-	-	
ALTER OPERATOR FAMILY	X	X	-	-	
ALTER POLICY	X	X	-	-	
ALTER SCHEMA	X	X	-	-	
ALTER SEQUENCE	X	X	-	-	
ALTER SERVER	X	X	-	-	
ALTER TABLE	X	X	X	X	
ALTER TEXT SEARCH CONFIGURATION	X	X	-	-	
ALTER TEXT SEARCH DICTIONARY	X	X	-	-	
ALTER TEXT SEARCH PARSER	X	X	-	-	
ALTER TEXT SEARCH TEMPLATE	X	X	-	-	
ALTER TRIGGER	X	X	-	-	
ALTER TYPE	X	X	-	X	
ALTER USER MAPPING	X	X	-	-	
ALTER VIEW	X	X	-	-	
COMMENT	X	X	-	-	Only for local objects

Command Tag	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	Notes
CREATE AGGREGATE	X	X	-	-	
CREATE CAST	X	X	-	-	
CREATE COLLATION	X	X	-	-	
CREATE CONVERSION	X	X	-	-	
CREATE DOMAIN	X	X	-	-	
CREATE EXTENSION	X	X	-	-	
CREATE FOREIGN DATA WRAPPER	X	X	-	-	
CREATE FOREIGN TABLE	X	X	-	-	
CREATE FUNCTION	X	X	-	-	
CREATE INDEX	X	X	-	-	
CREATE LANGUAGE	X	X	-	-	
CREATE MATERIALIZED VIEW	X	X	-	-	
CREATE OPERATOR	X	X	-	-	
CREATE OPERATOR CLASS	X	X	-	-	
CREATE OPERATOR FAMILY	X	X	-	-	
CREATE POLICY	X	X	-	-	
CREATE RULE	X	X	-	-	
CREATE SCHEMA	X	X	-	-	

Command Tag	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	Notes
CREATE SEQUENCE	X	X	–	–	
CREATE SERVER	X	X	–	–	
CREATE TABLE	X	X	–	–	
CREATE TABLE AS	X	X	–	–	
CREATE TEXT SEARCH CONFIGURATION	X	X	–	–	
CREATE TEXT SEARCH DICTIONARY	X	X	–	–	
CREATE TEXT SEARCH PARSER	X	X	–	–	
CREATE TEXT SEARCH TEMPLATE	X	X	–	–	
CREATE TRIGGER	X	X	–	–	
CREATE TYPE	X	X	–	–	
CREATE USER MAPPING	X	X	–	–	
CREATE VIEW	X	X	–	–	
DROP AGGREGATE	X	X	X	–	
DROP CAST	X	X	X	–	
DROP COLLATION	X	X	X	–	
DROP CONVERSION	X	X	X	–	
DROP DOMAIN	X	X	X	–	
DROP EXTENSION	X	X	X	–	
DROP FOREIGN DATA WRAPPER	X	X	X	–	

Command Tag	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	Notes
DROP FOREIGN TABLE	X	X	X	-	
DROP FUNCTION	X	X	X	-	
DROP INDEX	X	X	X	-	
DROP LANGUAGE	X	X	X	-	
DROP MATERIALIZED VIEW	X	X	X	-	
DROP OPERATOR	X	X	X	-	
DROP OPERATOR CLASS	X	X	X	-	
DROP OPERATOR FAMILY	X	X	X	-	
DROP OWNED	X	X	X	-	
DROP POLICY	X	X	X	-	
DROP RULE	X	X	X	-	
DROP SCHEMA	X	X	X	-	
DROP SEQUENCE	X	X	X	-	
DROP SERVER	X	X	X	-	
DROP TABLE	X	X	X	-	
DROP TEXT SEARCH CONFIGURATION	X	X	X	-	
DROP TEXT SEARCH DICTIONARY	X	X	X	-	
DROP TEXT SEARCH PARSER	X	X	X	-	
DROP TEXT SEARCH TEMPLATE	X	X	X	-	

Command Tag	ddl_command_start	ddl_command_end	sql_drop	table_rewrite	Notes
DROP TRIGGER	X	X	X	-	
DROP TYPE	X	X	X	-	
DROP USER MAPPING	X	X	X	-	
DROP VIEW	X	X	X	-	
GRANT	X	X	-	-	Only for local objects
IMPORT FOREIGN SCHEMA	X	X	-	-	
REFRESH MATERIALIZED VIEW	X	X	-	-	
REVOKE	X	X	-	-	Only for local objects
SECURITY LABEL	X	X	-	-	Only for local objects
SELECT INTO	X	X	-	-	

37.3. Writing Event Trigger Functions in C

This section describes the low-level details of the interface to an event trigger function. This information is only needed when writing event trigger functions in C. If you are using a higher-level language then these details are handled for you. In most cases you should consider using a procedural language before writing your event triggers in C. The documentation of each procedural language explains how to write an event trigger in that language.

Event trigger functions must use the "version 1" function manager interface.

When a function is called by the event trigger manager, it is not passed any normal arguments, but it is passed a "context" pointer pointing to a `EventTriggerData` structure. C functions can check whether they were called from the event trigger manager or not by executing the macro:

```
CALLED_AS_EVENT_TRIGGER(fcinfo)
```

which expands to:

```
((fcinfo)->context != NULL && IsA((fcinfo)->context, EventTriggerData))
```

If this returns true, then it is safe to cast `fcinfo->context` to type `EventTriggerData *` and make use of the pointed-to `EventTriggerData` structure. The function must *not* alter the `EventTriggerData` structure or any of the data it points to.

`struct EventTriggerData` is defined in `commands/event_trigger.h`:

```
typedef struct EventTriggerData
{
    NodeTag     type;
    const char *event;      /* event name */
    Node       *parsetree;  /* parse tree */
    const char *tag;        /* command tag */
} EventTriggerData;
```

where the members are defined as follows:

`type`

Always `T_EventTriggerData`.

`event`

Describes the event for which the function is called, one of `"ddl_command_start"`, `"ddl_command_end"`, `"sql_drop"`, `"table_rewrite"`. See Section 37.1 for the meaning of these events.

`parsetree`

A pointer to the parse tree of the command. Check the PostgreSQL source code for details. The parse tree structure is subject to change without notice.

`tag`

The command tag associated with the event for which the event trigger is run, for example `"CREATE FUNCTION"`.

An event trigger function must return a `NULL` pointer (*not* an SQL null value, that is, do not set `isNull` true).

37.4. A Complete Event Trigger Example

Here is a very simple example of an event trigger function written in C. (Examples of triggers written in procedural languages can be found in the documentation of the procedural languages.)

The function `noddl` raises an exception each time it is called. The event trigger definition associated the function with the `ddl_command_start` event. The effect is that all DDL commands (with the exceptions mentioned in Section 37.1) are prevented from running.

This is the source code of the trigger function:

```
#include "postgres.h"
#include "commands/event_trigger.h"


PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(noddl);
```

```
Datum
noddl(PG_FUNCTION_ARGS)
{
    EventTriggerData *trigdata;

    if (!CALLED_AS_EVENT_TRIGGER(fcinfo))  /* internal error */
        elog(ERROR, "not fired by event trigger manager");

    trigdata = (EventTriggerData *) fcinfo->context;

    ereport(ERROR,
        (errcode(ERRCODE_INSUFFICIENT_PRIVILEGE),
                 errmsg("command \"%s\" denied", trigdata->tag)));

    PG_RETURN_NULL();
}
```

After you have compiled the source code (see Section 35.9.6), declare the function and the triggers:

```
CREATE FUNCTION noddl() RETURNS event_trigger
    AS 'noddl' LANGUAGE C;

CREATE EVENT TRIGGER noddl ON ddl_command_start
    EXECUTE PROCEDURE noddl();
```

Now you can test the operation of the trigger:

```
=# \dy
                     List of event triggers
 Name  |       Event       | Owner | Enabled | Procedure | Tags
-------+-------------------+-------+---------+-----------+------
 noddl | ddl_command_start | dim   | enabled | noddl     |
(1 row)

=# CREATE TABLE foo(id serial);
ERROR:  command "CREATE TABLE" denied
```

In this situation, in order to be able to run some DDL commands when you need to do so, you have to either drop the event trigger or disable it. It can be convenient to disable the trigger for only the duration of a transaction:

```
BEGIN;
ALTER EVENT TRIGGER noddl DISABLE;
CREATE TABLE foo (id serial);
ALTER EVENT TRIGGER noddl ENABLE;
COMMIT;
```

(Recall that DDL commands on event triggers themselves are not affected by event triggers.)

37.5. A Table Rewrite Event Trigger Example

Thanks to the `table_rewrite` event, it is possible to implement a table rewriting policy only allowing the rewrite in maintenance windows.

Here's an example implementing such a policy.

```
CREATE OR REPLACE FUNCTION no_rewrite()
 RETURNS event_trigger
 LANGUAGE plpgsql AS
$$
---
--- Implement local Table Rewriting policy:
---   public.foo is not allowed rewriting, ever
---   other tables are only allowed rewriting between 1am and 6am
---   unless they have more than 100 blocks
---
DECLARE
  table_oid oid := pg_event_trigger_table_rewrite_oid();
  current_hour integer := extract('hour' from current_time);
  pages integer;
  max_pages integer := 100;
BEGIN
  IF pg_event_trigger_table_rewrite_oid() = 'public.foo'::regclass
  THEN
        RAISE EXCEPTION 'you"re not allowed to rewrite the table %',
                        table_oid::regclass;
  END IF;

  SELECT INTO pages relpages FROM pg_class WHERE oid = table_oid;
  IF pages > max_pages
  THEN
        RAISE EXCEPTION 'rewrites only allowed for table with less than % pages',
                        max_pages;
  END IF;

  IF current_hour NOT BETWEEN 1 AND 6
  THEN
        RAISE EXCEPTION 'rewrites only allowed between 1am and 6am';
  END IF;
END;
$$;

CREATE EVENT TRIGGER no_rewrite_allowed
                  ON table_rewrite
   EXECUTE PROCEDURE no_rewrite();
```

Chapter 38. The Rule System

This chapter discusses the rule system in PostgreSQL. Production rule systems are conceptually simple, but there are many subtle points involved in actually using them.

Some other database systems define active database rules, which are usually stored procedures and triggers. In PostgreSQL, these can be implemented using functions and triggers as well.

The rule system (more precisely speaking, the query rewrite rule system) is totally different from stored procedures and triggers. It modifies queries to take rules into consideration, and then passes the modified query to the query planner for planning and execution. It is very powerful, and can be used for many things such as query language procedures, views, and versions. The theoretical foundations and the power of this rule system are also discussed in *On Rules, Procedures, Caching and Views in Database Systems* and *A Unified Framework for Version Modeling Using Production Rules in a Database System*.

38.1. The Query Tree

To understand how the rule system works it is necessary to know when it is invoked and what its input and results are.

The rule system is located between the parser and the planner. It takes the output of the parser, one query tree, and the user-defined rewrite rules, which are also query trees with some extra information, and creates zero or more query trees as result. So its input and output are always things the parser itself could have produced and thus, anything it sees is basically representable as an SQL statement.

Now what is a query tree? It is an internal representation of an SQL statement where the single parts that it is built from are stored separately. These query trees can be shown in the server log if you set the configuration parameters `debug_print_parse`, `debug_print_rewritten`, or `debug_print_plan`. The rule actions are also stored as query trees, in the system catalog `pg_rewrite`. They are not formatted like the log output, but they contain exactly the same information.

Reading a raw query tree requires some experience. But since SQL representations of query trees are sufficient to understand the rule system, this chapter will not teach how to read them.

When reading the SQL representations of the query trees in this chapter it is necessary to be able to identify the parts the statement is broken into when it is in the query tree structure. The parts of a query tree are

the command type

This is a simple value telling which command (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) produced the query tree.

the range table

The range table is a list of relations that are used in the query. In a `SELECT` statement these are the relations given after the `FROM` key word.

Every range table entry identifies a table or view and tells by which name it is called in the other parts of the query. In the query tree, the range table entries are referenced by number rather than by name, so here it doesn't matter if there are duplicate names as it would in an SQL statement. This

can happen after the range tables of rules have been merged in. The examples in this chapter will not have this situation.

the result relation

This is an index into the range table that identifies the relation where the results of the query go.

SELECT queries don't have a result relation. (The special case of SELECT INTO is mostly identical to CREATE TABLE followed by INSERT ... SELECT, and is not discussed separately here.)

For INSERT, UPDATE, and DELETE commands, the result relation is the table (or view!) where the changes are to take effect.

the target list

The target list is a list of expressions that define the result of the query. In the case of a SELECT, these expressions are the ones that build the final output of the query. They correspond to the expressions between the key words SELECT and FROM. (* is just an abbreviation for all the column names of a relation. It is expanded by the parser into the individual columns, so the rule system never sees it.)

DELETE commands don't need a normal target list because they don't produce any result. Instead, the planner adds a special CTID entry to the empty target list, to allow the executor to find the row to be deleted. (CTID is added when the result relation is an ordinary table. If it is a view, a whole-row variable is added instead, by the rule system, as described in Section 38.2.4.)

For INSERT commands, the target list describes the new rows that should go into the result relation. It consists of the expressions in the VALUES clause or the ones from the SELECT clause in INSERT ... SELECT. The first step of the rewrite process adds target list entries for any columns that were not assigned to by the original command but have defaults. Any remaining columns (with neither a given value nor a default) will be filled in by the planner with a constant null expression.

For UPDATE commands, the target list describes the new rows that should replace the old ones. In the rule system, it contains just the expressions from the SET column = expression part of the command. The planner will handle missing columns by inserting expressions that copy the values from the old row into the new one. Just as for DELETE, a CTID or whole-row variable is added so that the executor can identify the old row to be updated.

Every entry in the target list contains an expression that can be a constant value, a variable pointing to a column of one of the relations in the range table, a parameter, or an expression tree made of function calls, constants, variables, operators, etc.

the qualification

The query's qualification is an expression much like one of those contained in the target list entries. The result value of this expression is a Boolean that tells whether the operation (INSERT, UPDATE, DELETE, or SELECT) for the final result row should be executed or not. It corresponds to the WHERE clause of an SQL statement.

the join tree

The query's join tree shows the structure of the FROM clause. For a simple query like SELECT ... FROM a, b, c, the join tree is just a list of the FROM items, because we are allowed to join them in any order. But when JOIN expressions, particularly outer joins, are used, we have to join in the order shown by the joins. In that case, the join tree shows the structure of the JOIN expressions. The restrictions associated with particular JOIN clauses (from ON or USING expressions) are stored as qualification expressions attached to those join-tree nodes. It turns out to be convenient to store the

top-level WHERE expression as a qualification attached to the top-level join-tree item, too. So really the join tree represents both the FROM and WHERE clauses of a SELECT.

the others

The other parts of the query tree like the ORDER BY clause aren't of interest here. The rule system substitutes some entries there while applying rules, but that doesn't have much to do with the fundamentals of the rule system.

38.2. Views and the Rule System

Views in PostgreSQL are implemented using the rule system. In fact, there is essentially no difference between:

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

compared against the two commands:

```
CREATE TABLE myview (same column list as mytab);
CREATE RULE "_RETURN" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

because this is exactly what the CREATE VIEW command does internally. This has some side effects. One of them is that the information about a view in the PostgreSQL system catalogs is exactly the same as it is for a table. So for the parser, there is absolutely no difference between a table and a view. They are the same thing: relations.

38.2.1. How SELECT Rules Work

Rules ON SELECT are applied to all queries as the last step, even if the command given is an INSERT, UPDATE or DELETE. And they have different semantics from rules on the other command types in that they modify the query tree in place instead of creating a new one. So SELECT rules are described first.

Currently, there can be only one action in an ON SELECT rule, and it must be an unconditional SELECT action that is INSTEAD. This restriction was required to make rules safe enough to open them for ordinary users, and it restricts ON SELECT rules to act like views.

The examples for this chapter are two join views that do some calculations and some more views using them in turn. One of the two first views is customized later by adding rules for INSERT, UPDATE, and DELETE operations so that the final result will be a view that behaves like a real table with some magic functionality. This is not such a simple example to start from and this makes things harder to get into. But it's better to have one example that covers all the points discussed step by step rather than having many different ones that might mix up in mind.

The real tables we need in the first two rule system descriptions are these:

```
CREATE TABLE shoe_data (
    shoename   text,          -- primary key
    sh_avail   integer,       -- available number of pairs
```

```
    slcolor    text,          -- preferred shoelace color
    slminlen   real,          -- minimum shoelace length
    slmaxlen   real,          -- maximum shoelace length
    slunit     text           -- length unit
);

CREATE TABLE shoelace_data (
    sl_name    text,          -- primary key
    sl_avail   integer,       -- available number of pairs
    sl_color   text,          -- shoelace color
    sl_len     real,          -- shoelace length
    sl_unit    text           -- length unit
);

CREATE TABLE unit (
    un_name    text,          -- primary key
    un_fact    real           -- factor to transform to cm
);
```

As you can see, they represent shoe-store data.

The views are created as:

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
      FROM shoe_data sh, unit un
     WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,
           s.sl_unit,
           s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
     WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           least(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
     WHERE rsl.sl_color = rsh.slcolor
       AND rsl.sl_len_cm >= rsh.slminlen_cm
```

```
        AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

The `CREATE VIEW` command for the `shoelace` view (which is the simplest one we have) will create a relation `shoelace` and an entry in `pg_rewrite` that tells that there is a rewrite rule that must be applied whenever the relation `shoelace` is referenced in a query's range table. The rule has no rule qualification (discussed later, with the non-`SELECT` rules, since `SELECT` rules currently cannot have them) and it is `INSTEAD`. Note that rule qualifications are not the same as query qualifications. The action of our rule has a query qualification. The action of the rule is one query tree that is a copy of the `SELECT` statement in the view creation command.

> **Note:** The two extra range table entries for `NEW` and `OLD` that you can see in the `pg_rewrite` entry aren't of interest for `SELECT` rules.

Now we populate `unit`, `shoe_data` and `shoelace_data` and run a simple query on a view:

```
INSERT INTO unit VALUES ('cm', 1.0);
INSERT INTO unit VALUES ('m', 100.0);
INSERT INTO unit VALUES ('inch', 2.54);

INSERT INTO shoe_data VALUES ('sh1', 2, 'black', 70.0, 90.0, 'cm');
INSERT INTO shoe_data VALUES ('sh2', 0, 'black', 30.0, 40.0, 'inch');
INSERT INTO shoe_data VALUES ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
INSERT INTO shoe_data VALUES ('sh4', 3, 'brown', 40.0, 50.0, 'inch');

INSERT INTO shoelace_data VALUES ('sl1', 5, 'black', 80.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl2', 6, 'black', 100.0, 'cm');
INSERT INTO shoelace_data VALUES ('sl3', 0, 'black', 35.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl4', 8, 'black', 40.0 , 'inch');
INSERT INTO shoelace_data VALUES ('sl5', 4, 'brown', 1.0 , 'm');
INSERT INTO shoelace_data VALUES ('sl6', 0, 'brown', 0.9 , 'm');
INSERT INTO shoelace_data VALUES ('sl7', 7, 'brown', 60 , 'cm');
INSERT INTO shoelace_data VALUES ('sl8', 1, 'brown', 40 , 'inch');

SELECT * FROM shoelace;

 sl_name   | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
-----------+----------+----------+--------+---------+-----------
 sl1       |        5 | black    |     80 | cm      |        80
 sl2       |        6 | black    |    100 | cm      |       100
 sl7       |        7 | brown    |     60 | cm      |        60
 sl3       |        0 | black    |     35 | inch    |      88.9
 sl4       |        8 | black    |     40 | inch    |     101.6
 sl8       |        1 | brown    |     40 | inch    |     101.6
 sl5       |        4 | brown    |      1 | m       |       100
 sl6       |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

This is the simplest `SELECT` you can do on our views, so we take this opportunity to explain the basics of view rules. The `SELECT * FROM shoelace` was interpreted by the parser and produced the query tree:

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace;
```

and this is given to the rule system. The rule system walks through the range table and checks if there are rules for any relation. When processing the range table entry for `shoelace` (the only one up to now) it finds the `_RETURN` rule with the query tree:

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       s.sl_len * u.un_fact AS sl_len_cm
  FROM shoelace old, shoelace new,
       shoelace_data s, unit u
 WHERE s.sl_unit = u.un_name;
```

To expand the view, the rewriter simply creates a subquery range-table entry containing the rule's action query tree, and substitutes this range table entry for the original one that referenced the view. The resulting rewritten query tree is almost the same as if you had typed:

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM (SELECT s.sl_name,
               s.sl_avail,
               s.sl_color,
               s.sl_len,
               s.sl_unit,
               s.sl_len * u.un_fact AS sl_len_cm
          FROM shoelace_data s, unit u
         WHERE s.sl_unit = u.un_name) shoelace;
```

There is one difference however: the subquery's range table has two extra entries `shoelace old` and `shoelace new`. These entries don't participate directly in the query, since they aren't referenced by the subquery's join tree or target list. The rewriter uses them to store the access privilege check information that was originally present in the range-table entry that referenced the view. In this way, the executor will still check that the user has proper privileges to access the view, even though there's no direct use of the view in the rewritten query.

That was the first rule applied. The rule system will continue checking the remaining range-table entries in the top query (in this example there are no more), and it will recursively check the range-table entries in the added subquery to see if any of them reference views. (But it won't expand `old` or `new` — otherwise we'd have infinite recursion!) In this example, there are no rewrite rules for `shoelace_data` or `unit`, so rewriting is complete and the above is the final result given to the planner.

Now we want to write a query that finds out for which shoes currently in the store we have the matching shoelaces (color and length) and where the total number of exactly matching pairs is greater or equal to two.

```
SELECT * FROM shoe_ready WHERE total_avail >= 2;
```

```
 shoename | sh_avail | sl_name | sl_avail | total_avail
----------+----------+---------+----------+-------------
 sh1      |        2 | sl1     |        5 |           2
 sh3      |        4 | sl7     |        7 |           4
(2 rows)
```

The output of the parser this time is the query tree:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

The first rule applied will be the one for the `shoe_ready` view and it results in the query tree:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               least(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM shoe rsh, shoelace rsl
         WHERE rsl.sl_color = rsh.slcolor
           AND rsl.sl_len_cm >= rsh.slminlen_cm
           AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail >= 2;
```

Similarly, the rules for `shoe` and `shoelace` are substituted into the range table of the subquery, leading to a three-level final query tree:

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM (SELECT rsh.shoename,
               rsh.sh_avail,
               rsl.sl_name,
               rsl.sl_avail,
               least(rsh.sh_avail, rsl.sl_avail) AS total_avail
          FROM (SELECT sh.shoename,
                       sh.sh_avail,
                       sh.slcolor,
                       sh.slminlen,
                       sh.slminlen * un.un_fact AS slminlen_cm,
                       sh.slmaxlen,
                       sh.slmaxlen * un.un_fact AS slmaxlen_cm,
                       sh.slunit
                  FROM shoe_data sh, unit un
                 WHERE sh.slunit = un.un_name) rsh,
               (SELECT s.sl_name,
```

```
                        s.sl_avail,
                        s.sl_color,
                        s.sl_len,
                        s.sl_unit,
                        s.sl_len * u.un_fact AS sl_len_cm
                    FROM shoelace_data s, unit u
                   WHERE s.sl_unit = u.un_name) rsl
          WHERE rsl.sl_color = rsh.slcolor
            AND rsl.sl_len_cm >= rsh.slminlen_cm
            AND rsl.sl_len_cm <= rsh.slmaxlen_cm) shoe_ready
 WHERE shoe_ready.total_avail > 2;
```

This might look inefficient, but the planner will collapse this into a single-level query tree by "pulling up" the subqueries, and then it will plan the joins just as if we'd written them out manually. So collapsing the query tree is an optimization that the rewrite system doesn't have to concern itself with.

38.2.2. View Rules in Non-SELECT Statements

Two details of the query tree aren't touched in the description of view rules above. These are the command type and the result relation. In fact, the command type is not needed by view rules, but the result relation may affect the way in which the query rewriter works, because special care needs to be taken if the result relation is a view.

There are only a few differences between a query tree for a `SELECT` and one for any other command. Obviously, they have a different command type and for a command other than a `SELECT`, the result relation points to the range-table entry where the result should go. Everything else is absolutely the same. So having two tables `t1` and `t2` with columns `a` and `b`, the query trees for the two statements:

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

```
UPDATE t1 SET b = t2.b FROM t2 WHERE t1.a = t2.a;
```

are nearly identical. In particular:

- The range tables contain entries for the tables `t1` and `t2`.
- The target lists contain one variable that points to column `b` of the range table entry for table `t2`.
- The qualification expressions compare the columns `a` of both range-table entries for equality.
- The join trees show a simple join between `t1` and `t2`.

The consequence is, that both query trees result in similar execution plans: They are both joins over the two tables. For the `UPDATE` the missing columns from `t1` are added to the target list by the planner and the final query tree will read as:

```
UPDATE t1 SET a = t1.a, b = t2.b FROM t2 WHERE t1.a = t2.a;
```

and thus the executor run over the join will produce exactly the same result set as:

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

But there is a little problem in UPDATE: the part of the executor plan that does the join does not care what the results from the join are meant for. It just produces a result set of rows. The fact that one is a SELECT command and the other is an UPDATE is handled higher up in the executor, where it knows that this is an UPDATE, and it knows that this result should go into table t1. But which of the rows that are there has to be replaced by the new row?

To resolve this problem, another entry is added to the target list in UPDATE (and also in DELETE) statements: the current tuple ID (CTID). This is a system column containing the file block number and position in the block for the row. Knowing the table, the CTID can be used to retrieve the original row of t1 to be updated. After adding the CTID to the target list, the query actually looks like:

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

Now another detail of PostgreSQL enters the stage. Old table rows aren't overwritten, and this is why ROLLBACK is fast. In an UPDATE, the new result row is inserted into the table (after stripping the CTID) and in the row header of the old row, which the CTID pointed to, the cmax and xmax entries are set to the current command counter and current transaction ID. Thus the old row is hidden, and after the transaction commits the vacuum cleaner can eventually remove the dead row.

Knowing all that, we can simply apply view rules in absolutely the same way to any command. There is no difference.

38.2.3. The Power of Views in PostgreSQL

The above demonstrates how the rule system incorporates view definitions into the original query tree. In the second example, a simple SELECT from one view created a final query tree that is a join of 4 tables (unit was used twice with different names).

The benefit of implementing views with the rule system is that the planner has all the information about which tables have to be scanned plus the relationships between these tables plus the restrictive qualifications from the views plus the qualifications from the original query in one single query tree. And this is still the situation when the original query is already a join over views. The planner has to decide which is the best path to execute the query, and the more information the planner has, the better this decision can be. And the rule system as implemented in PostgreSQL ensures that this is all information available about the query up to that point.

38.2.4. Updating a View

What happens if a view is named as the target relation for an INSERT, UPDATE, or DELETE? Doing the substitutions described above would give a query tree in which the result relation points at a subquery range-table entry, which will not work. There are several ways in which PostgreSQL can support the appearance of updating a view, however.

If the subquery selects from a single base relation and is simple enough, the rewriter can automatically replace the subquery with the underlying base relation so that the INSERT, UPDATE, or DELETE is applied

to the base relation in the appropriate way. Views that are "simple enough" for this are called *automatically updatable*. For detailed information on the kinds of view that can be automatically updated, see CREATE VIEW.

Alternatively, the operation may be handled by a user-provided INSTEAD OF trigger on the view. Rewriting works slightly differently in this case. For INSERT, the rewriter does nothing at all with the view, leaving it as the result relation for the query. For UPDATE and DELETE, it's still necessary to expand the view query to produce the "old" rows that the command will attempt to update or delete. So the view is expanded as normal, but another unexpanded range-table entry is added to the query to represent the view in its capacity as the result relation.

The problem that now arises is how to identify the rows to be updated in the view. Recall that when the result relation is a table, a special CTID entry is added to the target list to identify the physical locations of the rows to be updated. This does not work if the result relation is a view, because a view does not have any CTID, since its rows do not have actual physical locations. Instead, for an UPDATE or DELETE operation, a special wholerow entry is added to the target list, which expands to include all columns from the view. The executor uses this value to supply the "old" row to the INSTEAD OF trigger. It is up to the trigger to work out what to update based on the old and new row values.

Another possibility is for the user to define INSTEAD rules that specify substitute actions for INSERT, UPDATE, and DELETE commands on a view. These rules will rewrite the command, typically into a command that updates one or more tables, rather than views. That is the topic of Section 38.4.

Note that rules are evaluated first, rewriting the original query before it is planned and executed. Therefore, if a view has INSTEAD OF triggers as well as rules on INSERT, UPDATE, or DELETE, then the rules will be evaluated first, and depending on the result, the triggers may not be used at all.

Automatic rewriting of an INSERT, UPDATE, or DELETE query on a simple view is always tried last. Therefore, if a view has rules or triggers, they will override the default behavior of automatically updatable views.

If there are no INSTEAD rules or INSTEAD OF triggers for the view, and the rewriter cannot automatically rewrite the query as an update on the underlying base relation, an error will be thrown because the executor cannot update a view as such.

38.3. Materialized Views

Materialized views in PostgreSQL use the rule system like views do, but persist the results in a table-like form. The main differences between:

```
CREATE MATERIALIZED VIEW mymatview AS SELECT * FROM mytab;
```

and:

```
CREATE TABLE mymatview AS SELECT * FROM mytab;
```

are that the materialized view cannot subsequently be directly updated and that the query used to create the materialized view is stored in exactly the same way that a view's query is stored, so that fresh data can be generated for the materialized view with:

```
REFRESH MATERIALIZED VIEW mymatview;
```

The information about a materialized view in the PostgreSQL system catalogs is exactly the same as it is for a table or view. So for the parser, a materialized view is a relation, just like a table or a view. When a materialized view is referenced in a query, the data is returned directly from the materialized view, like from a table; the rule is only used for populating the materialized view.

While access to the data stored in a materialized view is often much faster than accessing the underlying tables directly or through a view, the data is not always current; yet sometimes current data is not needed. Consider a table which records sales:

```
CREATE TABLE invoice (
    invoice_no    integer        PRIMARY KEY,
    seller_no     integer,       -- ID of salesperson
    invoice_date  date,          -- date of sale
    invoice_amt   numeric(13,2)  -- amount of sale
);
```

If people want to be able to quickly graph historical sales data, they might want to summarize, and they may not care about the incomplete data for the current date:

```
CREATE MATERIALIZED VIEW sales_summary AS
  SELECT
      seller_no,
      invoice_date,
      sum(invoice_amt)::numeric(13,2) as sales_amt
    FROM invoice
    WHERE invoice_date < CURRENT_DATE
    GROUP BY
      seller_no,
      invoice_date
    ORDER BY
      seller_no,
      invoice_date;

CREATE UNIQUE INDEX sales_summary_seller
  ON sales_summary (seller_no, invoice_date);
```

This materialized view might be useful for displaying a graph in the dashboard created for salespeople. A job could be scheduled to update the statistics each night using this SQL statement:

```
REFRESH MATERIALIZED VIEW sales_summary;
```

Another use for a materialized view is to allow faster access to data brought across from a remote system through a foreign data wrapper. A simple example using `file_fdw` is below, with timings, but since this is using cache on the local system the performance difference compared to access to a remote system would usually be greater than shown here. Notice we are also exploiting the ability to put an index on the materialized view, whereas `file_fdw` does not support indexes; this advantage might not apply for other sorts of foreign data access.

Setup:

```
CREATE EXTENSION file_fdw;
CREATE SERVER local_file FOREIGN DATA WRAPPER file_fdw;
```

```
CREATE FOREIGN TABLE words (word text NOT NULL)
  SERVER local_file
  OPTIONS (filename '/usr/share/dict/words');
CREATE MATERIALIZED VIEW wrd AS SELECT * FROM words;
CREATE UNIQUE INDEX wrd_word ON wrd (word);
CREATE EXTENSION pg_trgm;
CREATE INDEX wrd_trgm ON wrd USING gist (word gist_trgm_ops);
VACUUM ANALYZE wrd;
```

Now let's spell-check a word. Using `file_fdw` directly:

```
SELECT count(*) FROM words WHERE word = 'caterpiler';

 count
-------
     0
(1 row)
```

With `EXPLAIN ANALYZE`, we see:

```
 Aggregate  (cost=21763.99..21764.00 rows=1 width=0) (actual time=188.180..188.181 rows=1 l
   ->  Foreign Scan on words  (cost=0.00..21761.41 rows=1032 width=0) (actual time=188.177.
         Filter: (word = 'caterpiler'::text)
         Rows Removed by Filter: 479829
         Foreign File: /usr/share/dict/words
         Foreign File Size: 4953699
 Planning time: 0.118 ms
 Execution time: 188.273 ms
```

If the materialized view is used instead, the query is much faster:

```
 Aggregate  (cost=4.44..4.45 rows=1 width=0) (actual time=0.042..0.042 rows=1 loops=1)
   ->  Index Only Scan using wrd_word on wrd  (cost=0.42..4.44 rows=1 width=0) (actual time
         Index Cond: (word = 'caterpiler'::text)
         Heap Fetches: 0
 Planning time: 0.164 ms
 Execution time: 0.117 ms
```

Either way, the word is spelled wrong, so let's look for what we might have wanted. Again using `file_fdw`:

```
SELECT word FROM words ORDER BY word <-> 'caterpiler' LIMIT 10;

     word
---------------
 cater
 caterpillar
 Caterpillar
 caterpillars
 caterpillar's
 Caterpillar's
 caterer
 caterer's
 caters
```

```
 catered
(10 rows)

 Limit  (cost=11583.61..11583.64 rows=10 width=32) (actual time=1431.591..1431.594 rows=10
   ->  Sort  (cost=11583.61..11804.76 rows=88459 width=32) (actual time=1431.589..1431.591
         Sort Key: ((word <-> 'caterpiler'::text))
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Foreign Scan on words  (cost=0.00..9672.05 rows=88459 width=32) (actual time=0
               Foreign File: /usr/share/dict/words
               Foreign File Size: 4953699
 Planning time: 0.128 ms
 Execution time: 1431.679 ms
```

Using the materialized view:

```
 Limit  (cost=0.29..1.06 rows=10 width=10) (actual time=187.222..188.257 rows=10 loops=1)
   ->  Index Scan using wrd_trgm on wrd  (cost=0.29..37020.87 rows=479829 width=10) (actual
         Order By: (word <-> 'caterpiler'::text)
 Planning time: 0.196 ms
 Execution time: 198.640 ms
```

If you can tolerate periodic update of the remote data to the local database, the performance benefit can be substantial.

38.4. Rules on INSERT, UPDATE, and DELETE

Rules that are defined on INSERT, UPDATE, and DELETE are significantly different from the view rules described in the previous section. First, their CREATE RULE command allows more:

- They are allowed to have no action.
- They can have multiple actions.
- They can be INSTEAD or ALSO (the default).
- The pseudorelations NEW and OLD become useful.
- They can have rule qualifications.

Second, they don't modify the query tree in place. Instead they create zero or more new query trees and can throw away the original one.

> **Caution**
>
> In many cases, tasks that could be performed by rules on INSERT/UPDATE/DELETE are better done with triggers. Triggers are notationally a bit more complicated, but their semantics are much simpler to understand. Rules tend to have surprising results when the original query contains volatile functions: volatile functions may get executed more times than expected in the process of carrying out the rules.
>
> Also, there are some cases that are not supported by these types of rules at all, notably including WITH clauses in the original query and multiple-assignment sub-SELECTs in the SET list of UPDATE queries. This is because copying these constructs into a rule query would result in multiple evaluations of the sub-query, contrary to the express intent of the query's author.

38.4.1. How Update Rules Work

Keep the syntax:

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

in mind. In the following, *update rules* means rules that are defined on INSERT, UPDATE, or DELETE.

Update rules get applied by the rule system when the result relation and the command type of a query tree are equal to the object and event given in the CREATE RULE command. For update rules, the rule system creates a list of query trees. Initially the query-tree list is empty. There can be zero (NOTHING key word), one, or multiple actions. To simplify, we will look at a rule with one action. This rule can have a qualification or not and it can be INSTEAD or ALSO (the default).

What is a rule qualification? It is a restriction that tells when the actions of the rule should be done and when not. This qualification can only reference the pseudorelations NEW and/or OLD, which basically represent the relation that was given as object (but with a special meaning).

So we have three cases that produce the following query trees for a one-action rule.

No qualification, with either ALSO or INSTEAD

> the query tree from the rule action with the original query tree's qualification added

Qualification given and ALSO

> the query tree from the rule action with the rule qualification and the original query tree's qualification added

Qualification given and INSTEAD

> the query tree from the rule action with the rule qualification and the original query tree's qualification; and the original query tree with the negated rule qualification added

Finally, if the rule is ALSO, the unchanged original query tree is added to the list. Since only qualified INSTEAD rules already add the original query tree, we end up with either one or two output query trees for a rule with one action.

For ON INSERT rules, the original query (if not suppressed by INSTEAD) is done before any actions added by rules. This allows the actions to see the inserted row(s). But for ON UPDATE and ON DELETE rules, the original query is done after the actions added by rules. This ensures that the actions can see the to-be-updated or to-be-deleted rows; otherwise, the actions might do nothing because they find no rows matching their qualifications.

The query trees generated from rule actions are thrown into the rewrite system again, and maybe more rules get applied resulting in more or less query trees. So a rule's actions must have either a different command type or a different result relation than the rule itself is on, otherwise this recursive process will end up in an infinite loop. (Recursive expansion of a rule will be detected and reported as an error.)

The query trees found in the actions of the pg_rewrite system catalog are only templates. Since they can reference the range-table entries for NEW and OLD, some substitutions have to be made before they can be used. For any reference to NEW, the target list of the original query is searched for a corresponding entry. If found, that entry's expression replaces the reference. Otherwise, NEW means the same as OLD (for an UPDATE) or is replaced by a null value (for an INSERT). Any reference to OLD is replaced by a reference to the range-table entry that is the result relation.

After the system is done applying update rules, it applies view rules to the produced query tree(s). Views cannot insert new update actions so there is no need to apply update rules to the output of view rewriting.

38.4.1.1. A First Rule Step by Step

Say we want to trace changes to the sl_avail column in the shoelace_data relation. So we set up a log table and a rule that conditionally writes a log entry when an UPDATE is performed on shoelace_data.

```
CREATE TABLE shoelace_log (
    sl_name    text,          -- shoelace changed
    sl_avail   integer,       -- new available value
    log_who    text,          -- who did it
    log_when   timestamp      -- when
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail <> OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
                                    NEW.sl_avail,
                                    current_user,
                                    current_timestamp
                                );
```

Now someone does:

```
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';
```

and we look at the log table:

```
SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who | log_when
```

```
---------+----------+---------+----------------------------------
 sl7     |        6 | Al      | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

That's what we expected. What happened in the background is the following. The parser created the query tree:

```
UPDATE shoelace_data SET sl_avail = 6
  FROM shoelace_data shoelace_data
 WHERE shoelace_data.sl_name = 'sl7';
```

There is a rule `log_shoelace` that is `ON UPDATE` with the rule qualification expression:

```
NEW.sl_avail <> OLD.sl_avail
```

and the action:

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old;
```

(This looks a little strange since you cannot normally write `INSERT ... VALUES ... FROM`. The `FROM` clause here is just to indicate that there are range-table entries in the query tree for `new` and `old`. These are needed so that they can be referenced by variables in the `INSERT` command's query tree.)

The rule is a qualified `ALSO` rule, so the rule system has to return two query trees: the modified rule action and the original query tree. In step 1, the range table of the original query is incorporated into the rule's action query tree. This results in:

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data;
```

In step 2, the rule qualification is added to it, so the result set is restricted to rows where `sl_avail` changes:

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE new.sl_avail <> old.sl_avail;
```

(This looks even stranger, since `INSERT ... VALUES` doesn't have a `WHERE` clause either, but the planner and executor will have no difficulty with it. They need to support this same functionality anyway for `INSERT ... SELECT`.)

In step 3, the original query tree's qualification is added, restricting the result set further to only the rows that would have been touched by the original query:

```
INSERT INTO shoelace_log VALUES (
       new.sl_name, new.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE new.sl_avail <> old.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

Step 4 replaces references to NEW by the target list entries from the original query tree or by the matching variable references from the result relation:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE 6 <> old.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

Step 5 changes OLD references into result relation references:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data new, shoelace_data old,
       shoelace_data shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

That's it. Since the rule is ALSO, we also output the original query tree. In short, the output from the rule system is a list of two query trees that correspond to these statements:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, 6,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE 6 <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
 WHERE sl_name = 'sl7';
```

These are executed in this order, and that is exactly what the rule was meant to do.

The substitutions and the added qualifications ensure that, if the original query would be, say:

```
UPDATE shoelace_data SET sl_color = 'green'
 WHERE sl_name = 'sl7';
```

no log entry would get written. In that case, the original query tree does not contain a target list entry for sl_avail, so NEW.sl_avail will get replaced by shoelace_data.sl_avail. Thus, the extra command generated by the rule is:

```
INSERT INTO shoelace_log VALUES (
       shoelace_data.sl_name, shoelace_data.sl_avail,
       current_user, current_timestamp )
  FROM shoelace_data
 WHERE shoelace_data.sl_avail <> shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';
```

and that qualification will never be true.

It will also work if the original query modifies multiple rows. So if someone issued the command:

```
UPDATE shoelace_data SET sl_avail = 0
 WHERE sl_color = 'black';
```

four rows in fact get updated (sl1, sl2, sl3, and sl4). But sl3 already has sl_avail = 0. In this case, the original query trees qualification is different and that results in the extra query tree:

```
INSERT INTO shoelace_log
SELECT shoelace_data.sl_name, 0,
       current_user, current_timestamp
  FROM shoelace_data
 WHERE 0 <> shoelace_data.sl_avail
   AND shoelace_data.sl_color = 'black';
```

being generated by the rule. This query tree will surely insert three new log entries. And that's absolutely correct.

Here we can see why it is important that the original query tree is executed last. If the UPDATE had been executed first, all the rows would have already been set to zero, so the logging INSERT would not find any row where 0 <> shoelace_data.sl_avail.

38.4.2. Cooperation with Views

A simple way to protect view relations from the mentioned possibility that someone can try to run INSERT, UPDATE, or DELETE on them is to let those query trees get thrown away. So we could create the rules:

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
    DO INSTEAD NOTHING;
```

If someone now tries to do any of these operations on the view relation shoe, the rule system will apply these rules. Since the rules have no actions and are INSTEAD, the resulting list of query trees will be empty and the whole query will become nothing because there is nothing left to be optimized or executed after the rule system is done with it.

A more sophisticated way to use the rule system is to create rules that rewrite the query tree into one that does the right operation on the real tables. To do that on the `shoelace` view, we create the following rules:

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    );

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
    DO INSTEAD
    UPDATE shoelace_data
       SET sl_name = NEW.sl_name,
           sl_avail = NEW.sl_avail,
           sl_color = NEW.sl_color,
           sl_len = NEW.sl_len,
           sl_unit = NEW.sl_unit
     WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
    DO INSTEAD
    DELETE FROM shoelace_data
     WHERE sl_name = OLD.sl_name;
```

If you want to support `RETURNING` queries on the view, you need to make the rules include `RETURNING` clauses that compute the view rows. This is usually pretty trivial for views on a single table, but it's a bit tedious for join views such as `shoelace`. An example for the insert case is:

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
           NEW.sl_avail,
           NEW.sl_color,
           NEW.sl_len,
           NEW.sl_unit
    )
    RETURNING
           shoelace_data.*,
           (SELECT shoelace_data.sl_len * u.un_fact
            FROM unit u WHERE shoelace_data.sl_unit = u.un_name);
```

Note that this one rule supports both `INSERT` and `INSERT RETURNING` queries on the view — the `RETURNING` clause is simply ignored for `INSERT`.

Now assume that once in a while, a pack of shoelaces arrives at the shop and a big parts list along with it. But you don't want to manually update the `shoelace` view every time. Instead we setup two little tables:

one where you can insert the items from the part list, and one with a special trick. The creation commands for these are:

```
CREATE TABLE shoelace_arrive (
    arr_name    text,
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     text,
    ok_quant    integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
    DO INSTEAD
    UPDATE shoelace
       SET sl_avail = sl_avail + NEW.ok_quant
     WHERE sl_name = NEW.ok_name;
```

Now you can fill the table `shoelace_arrive` with the data from the parts list:

```
SELECT * FROM shoelace_arrive;

 arr_name | arr_quant
----------+-----------
 sl3      |        10
 sl6      |        20
 sl8      |        20
(3 rows)
```

Take a quick look at the current data:

```
SELECT * FROM shoelace;

 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl3      |        0 | black    |     35 | inch    |      88.9
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl8      |        1 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |        0 | brown    |    0.9 | m       |        90
(8 rows)
```

Now move the arrived shoelaces in:

```
INSERT INTO shoelace_ok SELECT * FROM shoelace_arrive;
```

and check the results:

```
SELECT * FROM shoelace ORDER BY sl_name;
```

```
 sl_name  | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
----------+----------+----------+--------+---------+-----------
 sl1      |        5 | black    |     80 | cm      |        80
 sl2      |        6 | black    |    100 | cm      |       100
 sl7      |        6 | brown    |     60 | cm      |        60
 sl4      |        8 | black    |     40 | inch    |     101.6
 sl3      |       10 | black    |     35 | inch    |      88.9
 sl8      |       21 | brown    |     40 | inch    |     101.6
 sl5      |        4 | brown    |      1 | m       |       100
 sl6      |       20 | brown    |    0.9 | m       |        90
(8 rows)

SELECT * FROM shoelace_log;

 sl_name | sl_avail | log_who| log_when
---------+----------+--------+----------------------------------
 sl7     |        6 | Al     | Tue Oct 20 19:14:45 1998 MET DST
 sl3     |       10 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl6     |       20 | Al     | Tue Oct 20 19:25:16 1998 MET DST
 sl8     |       21 | Al     | Tue Oct 20 19:25:16 1998 MET DST
(4 rows)
```

It's a long way from the one `INSERT ... SELECT` to these results. And the description of the query-tree transformation will be the last in this chapter. First, there is the parser's output:

```
INSERT INTO shoelace_ok
SELECT shoelace_arrive.arr_name, shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok;
```

Now the first rule `shoelace_ok_ins` is applied and turns this into:

```
UPDATE shoelace
   SET sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace
 WHERE shoelace.sl_name = shoelace_arrive.arr_name;
```

and throws away the original `INSERT` on `shoelace_ok`. This rewritten query is passed to the rule system again, and the second applied rule `shoelace_upd` produces:

```
UPDATE shoelace_data
   SET sl_name = shoelace.sl_name,
       sl_avail = shoelace.sl_avail + shoelace_arrive.arr_quant,
       sl_color = shoelace.sl_color,
       sl_len = shoelace.sl_len,
       sl_unit = shoelace.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data
 WHERE shoelace.sl_name = shoelace_arrive.arr_name
```

```
   AND shoelace_data.sl_name = shoelace.sl_name;
```

Again it's an `INSTEAD` rule and the previous query tree is trashed. Note that this query still uses the view `shoelace`. But the rule system isn't finished with this step, so it continues and applies the `_RETURN` rule on it, and we get:

```
UPDATE shoelace_data
   SET sl_name = s.sl_name,
       sl_avail = s.sl_avail + shoelace_arrive.arr_quant,
       sl_color = s.sl_color,
       sl_len = s.sl_len,
       sl_unit = s.sl_unit
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data,
       shoelace old, shoelace new,
       shoelace_data s, unit u
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name;
```

Finally, the rule `log_shoelace` gets applied, producing the extra query tree:

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_ok shoelace_ok,
       shoelace_ok old, shoelace_ok new,
       shoelace shoelace, shoelace old,
       shoelace new, shoelace_data shoelace_data,
       shoelace old, shoelace new,
       shoelace_data s, unit u,
       shoelace_data old, shoelace_data new
       shoelace_log shoelace_log
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND (s.sl_avail + shoelace_arrive.arr_quant) <> s.sl_avail;
```

After that the rule system runs out of rules and returns the generated query trees.

So we end up with two final query trees that are equivalent to the SQL statements:

```
INSERT INTO shoelace_log
SELECT s.sl_name,
       s.sl_avail + shoelace_arrive.arr_quant,
       current_user,
       current_timestamp
  FROM shoelace_arrive shoelace_arrive, shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.arr_name
   AND shoelace_data.sl_name = s.sl_name
   AND s.sl_avail + shoelace_arrive.arr_quant <> s.sl_avail;
```

```
UPDATE shoelace_data
   SET sl_avail = shoelace_data.sl_avail + shoelace_arrive.arr_quant
  FROM shoelace_arrive shoelace_arrive,
       shoelace_data shoelace_data,
       shoelace_data s
 WHERE s.sl_name = shoelace_arrive.sl_name
   AND shoelace_data.sl_name = s.sl_name;
```

The result is that data coming from one relation inserted into another, changed into updates on a third, changed into updating a fourth plus logging that final update in a fifth gets reduced into two queries.

There is a little detail that's a bit ugly. Looking at the two queries, it turns out that the `shoelace_data` relation appears twice in the range table where it could definitely be reduced to one. The planner does not handle it and so the execution plan for the rule systems output of the `INSERT` will be

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on shoelace_arrive
  ->  Seq Scan on shoelace_data
```

while omitting the extra range table entry would result in a

```
Merge Join
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on s
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on shoelace_arrive
```

which produces exactly the same entries in the log table. Thus, the rule system caused one extra scan on the table `shoelace_data` that is absolutely not necessary. And the same redundant scan is done once more in the `UPDATE`. But it was a really hard job to make that all possible at all.

Now we make a final demonstration of the PostgreSQL rule system and its power. Say you add some shoelaces with extraordinary colors to your database:

```
INSERT INTO shoelace VALUES ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
INSERT INTO shoelace VALUES ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

We would like to make a view to check which `shoelace` entries do not fit any shoe in color. The view for this is:

```
CREATE VIEW shoelace_mismatch AS
    SELECT * FROM shoelace WHERE NOT EXISTS
        (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

Its output is:

```
SELECT * FROM shoelace_mismatch;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl9     |        0 | pink     |     35 | inch    |      88.9
 sl10    |     1000 | magenta  |     40 | inch    |     101.6
```

Now we want to set it up so that mismatching shoelaces that are not in stock are deleted from the database. To make it a little harder for PostgreSQL, we don't delete it directly. Instead we create one more view:

```
CREATE VIEW shoelace_can_delete AS
    SELECT * FROM shoelace_mismatch WHERE sl_avail = 0;
```

and do it this way:

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_can_delete
              WHERE sl_name = shoelace.sl_name);
```

Voilà:

```
SELECT * FROM shoelace;

 sl_name | sl_avail | sl_color | sl_len | sl_unit | sl_len_cm
---------+----------+----------+--------+---------+-----------
 sl1     |        5 | black    |     80 | cm      |         80
 sl2     |        6 | black    |    100 | cm      |        100
 sl7     |        6 | brown    |     60 | cm      |         60
 sl4     |        8 | black    |     40 | inch    |      101.6
 sl3     |       10 | black    |     35 | inch    |       88.9
 sl8     |       21 | brown    |     40 | inch    |      101.6
 sl10    |     1000 | magenta  |     40 | inch    |      101.6
 sl5     |        4 | brown    |      1 | m       |        100
 sl6     |       20 | brown    |    0.9 | m       |         90
(9 rows)
```

A `DELETE` on a view, with a subquery qualification that in total uses 4 nesting/joined views, where one of them itself has a subquery qualification containing a view and where calculated view columns are used, gets rewritten into one single query tree that deletes the requested data from a real table.

There are probably only a few situations out in the real world where such a construct is necessary. But it makes you feel comfortable that it works.

38.5. Rules and Privileges

Due to rewriting of queries by the PostgreSQL rule system, other tables/views than those used in the original query get accessed. When update rules are used, this can include write access to tables.

Rewrite rules don't have a separate owner. The owner of a relation (table or view) is automatically the owner of the rewrite rules that are defined for it. The PostgreSQL rule system changes the behavior of the default access control system. Relations that are used due to rules get checked against the privileges of the rule owner, not the user invoking the rule. This means that a user only needs the required privileges for the tables/views that he names explicitly in his queries.

For example: A user has a list of phone numbers where some of them are private, the others are of interest for the secretary of the office. He can construct the following:

```
CREATE TABLE phone_data (person text, phone text, private boolean);
CREATE VIEW phone_number AS
    SELECT person, CASE WHEN NOT private THEN phone END AS phone
    FROM phone_data;
GRANT SELECT ON phone_number TO secretary;
```

Nobody except him (and the database superusers) can access the `phone_data` table. But because of the `GRANT`, the secretary can run a `SELECT` on the `phone_number` view. The rule system will rewrite the `SELECT` from `phone_number` into a `SELECT` from `phone_data`. Since the user is the owner of `phone_number` and therefore the owner of the rule, the read access to `phone_data` is now checked against his privileges and the query is permitted. The check for accessing `phone_number` is also performed, but this is done against the invoking user, so nobody but the user and the secretary can use it.

The privileges are checked rule by rule. So the secretary is for now the only one who can see the public phone numbers. But the secretary can setup another view and grant access to that to the public. Then, anyone can see the `phone_number` data through the secretary's view. What the secretary cannot do is to create a view that directly accesses `phone_data`. (Actually he can, but it will not work since every access will be denied during the permission checks.) And as soon as the user will notice, that the secretary opened his `phone_number` view, he can revoke his access. Immediately, any access to the secretary's view would fail.

One might think that this rule-by-rule checking is a security hole, but in fact it isn't. But if it did not work this way, the secretary could set up a table with the same columns as `phone_number` and copy the data to there once per day. Then it's his own data and he can grant access to everyone he wants. A `GRANT` command means, "I trust you". If someone you trust does the thing above, it's time to think it over and then use `REVOKE`.

Note that while views can be used to hide the contents of certain columns using the technique shown above, they cannot be used to reliably conceal the data in unseen rows unless the `security_barrier` flag has been set. For example, the following view is insecure:

```
CREATE VIEW phone_number AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

This view might seem secure, since the rule system will rewrite any `SELECT` from `phone_number` into a `SELECT` from `phone_data` and add the qualification that only entries where `phone` does not begin with 412 are wanted. But if the user can create his or her own functions, it is not difficult to convince the planner to execute the user-defined function prior to the `NOT LIKE` expression. For example:

```
CREATE FUNCTION tricky(text, text) RETURNS bool AS $$
BEGIN
    RAISE NOTICE '% => %', $1, $2;
    RETURN true;
END;
```

```
$$ LANGUAGE plpgsql COST 0.00000000000000000001;

SELECT * FROM phone_number WHERE tricky(person, phone);
```

Every person and phone number in the phone_data table will be printed as a NOTICE, because the planner will choose to execute the inexpensive tricky function before the more expensive NOT LIKE. Even if the user is prevented from defining new functions, built-in functions can be used in similar attacks. (For example, most casting functions include their input values in the error messages they produce.)

Similar considerations apply to update rules. In the examples of the previous section, the owner of the tables in the example database could grant the privileges SELECT, INSERT, UPDATE, and DELETE on the shoelace view to someone else, but only SELECT on shoelace_log. The rule action to write log entries will still be executed successfully, and that other user could see the log entries. But he cannot create fake entries, nor could he manipulate or remove existing ones. In this case, there is no possibility of subverting the rules by convincing the planner to alter the order of operations, because the only rule which references shoelace_log is an unqualified INSERT. This might not be true in more complex scenarios.

When it is necessary for a view to provide row level security, the security_barrier attribute should be applied to the view. This prevents maliciously-chosen functions and operators from being passed values from rows until after the view has done its work. For example, if the view shown above had been created like this, it would be secure:

```
CREATE VIEW phone_number WITH (security_barrier) AS
    SELECT person, phone FROM phone_data WHERE phone NOT LIKE '412%';
```

Views created with the security_barrier may perform far worse than views created without this option. In general, there is no way to avoid this: the fastest possible plan must be rejected if it may compromise security. For this reason, this option is not enabled by default.

The query planner has more flexibility when dealing with functions that have no side effects. Such functions are referred to as LEAKPROOF, and include many simple, commonly used operators, such as many equality operators. The query planner can safely allow such functions to be evaluated at any point in the query execution process, since invoking them on rows invisible to the user will not leak any information about the unseen rows. Further, functions which do not take arguments or which are not passed any arguments from the security barrier view do not have to be marked as LEAKPROOF to be pushed down, as they never receive data from the view. In contrast, a function that might throw an error depending on the values received as arguments (such as one that throws an error in the event of overflow or division by zero) is not leak-proof, and could provide significant information about the unseen rows if applied before the security view's row filters.

It is important to understand that even a view created with the security_barrier option is intended to be secure only in the limited sense that the contents of the invisible tuples will not be passed to possibly-insecure functions. The user may well have other means of making inferences about the unseen data; for example, they can see the query plan using EXPLAIN, or measure the run time of queries against the view. A malicious attacker might be able to infer something about the amount of unseen data, or even gain some information about the data distribution or most common values (since these things may affect the run time of the plan; or even, since they are also reflected in the optimizer statistics, the choice of plan). If these types of "covert channel" attacks are of concern, it is probably unwise to grant any access to the data at all.

38.6. Rules and Command Status

The PostgreSQL server returns a command status string, such as `INSERT 149592 1`, for each command it receives. This is simple enough when there are no rules involved, but what happens when the query is rewritten by rules?

Rules affect the command status as follows:

- If there is no unconditional `INSTEAD` rule for the query, then the originally given query will be executed, and its command status will be returned as usual. (But note that if there were any conditional `INSTEAD` rules, the negation of their qualifications will have been added to the original query. This might reduce the number of rows it processes, and if so the reported status will be affected.)
- If there is any unconditional `INSTEAD` rule for the query, then the original query will not be executed at all. In this case, the server will return the command status for the last query that was inserted by an `INSTEAD` rule (conditional or unconditional) and is of the same command type (`INSERT`, `UPDATE`, or `DELETE`) as the original query. If no query meeting those requirements is added by any rule, then the returned command status shows the original query type and zeroes for the row-count and OID fields.

The programmer can ensure that any desired `INSTEAD` rule is the one that sets the command status in the second case, by giving it the alphabetically last rule name among the active rules, so that it gets applied last.

38.7. Rules Versus Triggers

Many things that can be done using triggers can also be implemented using the PostgreSQL rule system. One of the things that cannot be implemented by rules are some kinds of constraints, especially foreign keys. It is possible to place a qualified rule that rewrites a command to `NOTHING` if the value of a column does not appear in another table. But then the data is silently thrown away and that's not a good idea. If checks for valid values are required, and in the case of an invalid value an error message should be generated, it must be done by a trigger.

In this chapter, we focused on using rules to update views. All of the update rule examples in this chapter can also be implemented using `INSTEAD OF` triggers on the views. Writing such triggers is often easier than writing rules, particularly if complex logic is required to perform the update.

For the things that can be implemented by both, which is best depends on the usage of the database. A trigger is fired once for each affected row. A rule modifies the query or generates an additional query. So if many rows are affected in one statement, a rule issuing one extra command is likely to be faster than a trigger that is called for every single row and must re-determine what to do many times. However, the trigger approach is conceptually far simpler than the rule approach, and is easier for novices to get right.

Here we show an example of how the choice of rules versus triggers plays out in one situation. There are two tables:

```
CREATE TABLE computer (
    hostname        text,    -- indexed
    manufacturer    text     -- indexed
);
```

```
CREATE TABLE software (
    software        text,    -- indexed
    hostname        text     -- indexed
);
```

Both tables have many thousands of rows and the indexes on `hostname` are unique. The rule or trigger should implement a constraint that deletes rows from `software` that reference a deleted computer. The trigger would use this command:

```
DELETE FROM software WHERE hostname = $1;
```

Since the trigger is called for each individual row deleted from `computer`, it can prepare and save the plan for this command and pass the `hostname` value in the parameter. The rule would be written as:

```
CREATE RULE computer_del AS ON DELETE TO computer
    DO DELETE FROM software WHERE hostname = OLD.hostname;
```

Now we look at different types of deletes. In the case of a:

```
DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

the table `computer` is scanned by index (fast), and the command issued by the trigger would also use an index scan (also fast). The extra command from the rule would be:

```
DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
                       AND software.hostname = computer.hostname;
```

Since there are appropriate indexes setup, the planner will create a plan of

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

So there would be not that much difference in speed between the trigger and the rule implementation.

With the next delete we want to get rid of all the 2000 computers where the `hostname` starts with `old`. There are two possible commands to do that. One is:

```
DELETE FROM computer WHERE hostname >= 'old'
                       AND hostname <  'ole'
```

The command added by the rule will be:

```
DELETE FROM software WHERE computer.hostname >= 'old' AND computer.hostname < 'ole'
                       AND software.hostname = computer.hostname;
```

with the plan

```
Hash Join
  ->  Seq Scan on software
  ->  Hash
    ->  Index Scan using comp_hostidx on computer
```

The other possible command is:

```
DELETE FROM computer WHERE hostname ~ '^old';
```

which results in the following executing plan for the command added by the rule:

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

This shows, that the planner does not realize that the qualification for `hostname` in `computer` could also be used for an index scan on `software` when there are multiple qualification expressions combined with `AND`, which is what it does in the regular-expression version of the command. The trigger will get invoked once for each of the 2000 old computers that have to be deleted, and that will result in one index scan over `computer` and 2000 index scans over `software`. The rule implementation will do it with two commands that use indexes. And it depends on the overall size of the table `software` whether the rule will still be faster in the sequential scan situation. 2000 command executions from the trigger over the SPI manager take some time, even if all the index blocks will soon be in the cache.

The last command we look at is:

```
DELETE FROM computer WHERE manufacturer = 'bim';
```

Again this could result in many rows to be deleted from `computer`. So the trigger will again run many commands through the executor. The command generated by the rule will be:

```
DELETE FROM software WHERE computer.manufacturer = 'bim'
                       AND software.hostname = computer.hostname;
```

The plan for that command will again be the nested loop over two index scans, only using a different index on `computer`:

```
Nestloop
  ->  Index Scan using comp_manufidx on computer
  ->  Index Scan using soft_hostidx on software
```

In any of these cases, the extra commands from the rule system will be more or less independent from the number of affected rows in a command.

The summary is, rules will only be significantly slower than triggers if their actions result in large and badly qualified joins, a situation where the planner fails.

Chapter 39. Procedural Languages

PostgreSQL allows user-defined functions to be written in other languages besides SQL and C. These other languages are generically called *procedural languages* (PLs). For a function written in a procedural language, the database server has no built-in knowledge about how to interpret the function's source text. Instead, the task is passed to a special handler that knows the details of the language. The handler could either do all the work of parsing, syntax analysis, execution, etc. itself, or it could serve as "glue" between PostgreSQL and an existing implementation of a programming language. The handler itself is a C language function compiled into a shared object and loaded on demand, just like any other C function.

There are currently four procedural languages available in the standard PostgreSQL distribution: PL/pgSQL (Chapter 40), PL/Tcl (Chapter 41), PL/Perl (Chapter 42), and PL/Python (Chapter 43). There are additional procedural languages available that are not included in the core distribution. Appendix H has information about finding them. In addition other languages can be defined by users; the basics of developing a new procedural language are covered in Chapter 53.

39.1. Installing Procedural Languages

A procedural language must be "installed" into each database where it is to be used. But procedural languages installed in the database `template1` are automatically available in all subsequently created databases, since their entries in `template1` will be copied by `CREATE DATABASE`. So the database administrator can decide which languages are available in which databases and can make some languages available by default if he chooses.

For the languages supplied with the standard distribution, it is only necessary to execute `CREATE EXTENSION` *`language_name`* to install the language into the current database. Alternatively, the program createlang can be used to do this from the shell command line. For example, to install the language PL/Perl into the database `template1`, use:

```
createlang plperl template1
```

The manual procedure described below is only recommended for installing languages that have not been packaged as extensions.

Manual Procedural Language Installation

A procedural language is installed in a database in five steps, which must be carried out by a database superuser. In most cases the required SQL commands should be packaged as the installation script of an "extension", so that `CREATE EXTENSION` can be used to execute them.

1. The shared object for the language handler must be compiled and installed into an appropriate library directory. This works in the same way as building and installing modules with regular user-defined C functions does; see Section 35.9.6. Often, the language handler will depend on an external library that provides the actual programming language engine; if so, that must be installed as well.

2. The handler must be declared with the command

```
CREATE FUNCTION handler_function_name()
    RETURNS language_handler
    AS 'path-to-shared-object'
```

```
    LANGUAGE C;
```

The special return type of `language_handler` tells the database system that this function does not return one of the defined SQL data types and is not directly usable in SQL statements.

3. Optionally, the language handler can provide an "inline" handler function that executes anonymous code blocks (DO commands) written in this language. If an inline handler function is provided by the language, declare it with a command like

```
CREATE FUNCTION inline_function_name(internal)
    RETURNS void
    AS 'path-to-shared-object'
    LANGUAGE C;
```

4. Optionally, the language handler can provide a "validator" function that checks a function definition for correctness without actually executing it. The validator function is called by `CREATE FUNCTION` if it exists. If a validator function is provided by the language, declare it with a command like

```
CREATE FUNCTION validator_function_name(oid)
    RETURNS void
    AS 'path-to-shared-object'
    LANGUAGE C STRICT;
```

5. Finally, the PL must be declared with the command

```
CREATE [TRUSTED] [PROCEDURAL] LANGUAGE language_name
    HANDLER handler_function_name
    [INLINE inline_function_name]
    [VALIDATOR validator_function_name] ;
```

The optional key word `TRUSTED` specifies that the language does not grant access to data that the user would not otherwise have. Trusted languages are designed for ordinary database users (those without superuser privilege) and allows them to safely create functions and trigger procedures. Since PL functions are executed inside the database server, the `TRUSTED` flag should only be given for languages that do not allow access to database server internals or the file system. The languages PL/pgSQL, PL/Tcl, and PL/Perl are considered trusted; the languages PL/TclU, PL/PerlU, and PL/PythonU are designed to provide unlimited functionality and should *not* be marked trusted.

Example 39-1 shows how the manual installation procedure would work with the language PL/Perl.

Example 39-1. Manual Installation of PL/Perl

The following command tells the database server where to find the shared object for the PL/Perl language's call handler function:

```
CREATE FUNCTION plperl_call_handler() RETURNS language_handler AS
    '$libdir/plperl' LANGUAGE C;
```

PL/Perl has an inline handler function and a validator function, so we declare those too:

```
CREATE FUNCTION plperl_inline_handler(internal) RETURNS void AS
    '$libdir/plperl' LANGUAGE C;

CREATE FUNCTION plperl_validator(oid) RETURNS void AS
    '$libdir/plperl' LANGUAGE C STRICT;
```

The command:

```
CREATE TRUSTED PROCEDURAL LANGUAGE plperl
```

```
    HANDLER plperl_call_handler
    INLINE plperl_inline_handler
    VALIDATOR plperl_validator;
```

then defines that the previously declared functions should be invoked for functions and trigger procedures where the language attribute is `plperl`.

In a default PostgreSQL installation, the handler for the PL/pgSQL language is built and installed into the "library" directory; furthermore, the PL/pgSQL language itself is installed in all databases. If Tcl support is configured in, the handlers for PL/Tcl and PL/TclU are built and installed in the library directory, but the language itself is not installed in any database by default. Likewise, the PL/Perl and PL/PerlU handlers are built and installed if Perl support is configured, and the PL/PythonU handler is installed if Python support is configured, but these languages are not installed by default.

Chapter 40. PL/pgSQL - SQL Procedural Language

40.1. Overview

PL/pgSQL is a loadable procedural language for the PostgreSQL database system. The design goals of PL/pgSQL were to create a loadable procedural language that

- can be used to create functions and trigger procedures,
- adds control structures to the SQL language,
- can perform complex computations,
- inherits all user-defined types, functions, and operators,
- can be defined to be trusted by the server,
- is easy to use.

Functions created with PL/pgSQL can be used anywhere that built-in functions could be used. For example, it is possible to create complex conditional computation functions and later use them to define operators or use them in index expressions.

In PostgreSQL 9.0 and later, PL/pgSQL is installed by default. However it is still a loadable module, so especially security-conscious administrators could choose to remove it.

40.1.1. Advantages of Using PL/pgSQL

SQL is the language PostgreSQL and most other relational databases use as query language. It's portable and easy to learn. But every SQL statement must be executed individually by the database server.

That means that your client application must send each query to the database server, wait for it to be processed, receive and process the results, do some computation, then send further queries to the server. All this incurs interprocess communication and will also incur network overhead if your client is on a different machine than the database server.

With PL/pgSQL you can group a block of computation and a series of queries *inside* the database server, thus having the power of a procedural language and the ease of use of SQL, but with considerable savings of client/server communication overhead.

- Extra round trips between client and server are eliminated
- Intermediate results that the client does not need do not have to be marshaled or transferred between server and client
- Multiple rounds of query parsing can be avoided

This can result in a considerable performance increase as compared to an application that does not use stored functions.

Also, with PL/pgSQL you can use all the data types, operators and functions of SQL.

40.1.2. Supported Argument and Result Data Types

Functions written in PL/pgSQL can accept as arguments any scalar or array data type supported by the server, and they can return a result of any of these types. They can also accept or return any composite type (row type) specified by name. It is also possible to declare a PL/pgSQL function as returning `record`, which means that the result is a row type whose columns are determined by specification in the calling query, as discussed in Section 7.2.1.4.

PL/pgSQL functions can be declared to accept a variable number of arguments by using the `VARIADIC` marker. This works exactly the same way as for SQL functions, as discussed in Section 35.4.5.

PL/pgSQL functions can also be declared to accept and return the polymorphic types `anyelement`, `anyarray`, `anynonarray`, `anyenum`, and `anyrange`. The actual data types handled by a polymorphic function can vary from call to call, as discussed in Section 35.2.5. An example is shown in Section 40.3.1.

PL/pgSQL functions can also be declared to return a "set" (or table) of any data type that can be returned as a single instance. Such a function generates its output by executing `RETURN NEXT` for each desired element of the result set, or by using `RETURN QUERY` to output the result of evaluating a query.

Finally, a PL/pgSQL function can be declared to return `void` if it has no useful return value.

PL/pgSQL functions can also be declared with output parameters in place of an explicit specification of the return type. This does not add any fundamental capability to the language, but it is often convenient, especially for returning multiple values. The `RETURNS TABLE` notation can also be used in place of `RETURNS SETOF`.

Specific examples appear in Section 40.3.1 and Section 40.6.1.

40.2. Structure of PL/pgSQL

Functions written in PL/pgSQL are defined to the server by executing CREATE FUNCTION commands. Such a command would normally look like, say,

```
CREATE FUNCTION somefunc(integer, text) RETURNS integer
AS 'function body text'
LANGUAGE plpgsql;
```

The function body is simply a string literal so far as `CREATE FUNCTION` is concerned. It is often helpful to use dollar quoting (see Section 4.1.2.4) to write the function body, rather than the normal single quote syntax. Without dollar quoting, any single quotes or backslashes in the function body must be escaped by doubling them. Almost all the examples in this chapter use dollar-quoted literals for their function bodies.

PL/pgSQL is a block-structured language. The complete text of a function body must be a *block*. A block is defined as:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

Each declaration and each statement within a block is terminated by a semicolon. A block that appears within another block must have a semicolon after `END`, as shown above; however the final `END` that concludes a function body does not require a semicolon.

> **Tip:** A common mistake is to write a semicolon immediately after `BEGIN`. This is incorrect and will result in a syntax error.

A *label* is only needed if you want to identify the block for use in an `EXIT` statement, or to qualify the names of the variables declared in the block. If a label is given after `END`, it must match the label at the block's beginning.

All key words are case-insensitive. Identifiers are implicitly converted to lower case unless double-quoted, just as they are in ordinary SQL commands.

Comments work the same way in PL/pgSQL code as in ordinary SQL. A double dash (`--`) starts a comment that extends to the end of the line. A `/*` starts a block comment that extends to the matching occurrence of `*/`. Block comments nest.

Any statement in the statement section of a block can be a *subblock*. Subblocks can be used for logical grouping or to localize variables to a small group of statements. Variables declared in a subblock mask any similarly-named variables of outer blocks for the duration of the subblock; but you can access the outer variables anyway if you qualify their names with their block's label. For example:

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 30
    quantity := 50;
    --
    -- Create a subblock
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  -- Prints 50
    END;

    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 50

    RETURN quantity;
```

```
END;
$$ LANGUAGE plpgsql;
```

> **Note:** There is actually a hidden "outer block" surrounding the body of any PL/pgSQL function. This block provides the declarations of the function's parameters (if any), as well as some special variables such as FOUND (see Section 40.5.5). The outer block is labeled with the function's name, meaning that parameters and special variables can be qualified with the function's name.

It is important not to confuse the use of BEGIN/END for grouping statements in PL/pgSQL with the similarly-named SQL commands for transaction control. PL/pgSQL's BEGIN/END are only for grouping; they do not start or end a transaction. Functions and trigger procedures are always executed within a transaction established by an outer query — they cannot start or commit that transaction, since there would be no context for them to execute in. However, a block containing an EXCEPTION clause effectively forms a subtransaction that can be rolled back without affecting the outer transaction. For more about that see Section 40.6.6.

40.3. Declarations

All variables used in a block must be declared in the declarations section of the block. (The only exceptions are that the loop variable of a FOR loop iterating over a range of integer values is automatically declared as an integer variable, and likewise the loop variable of a FOR loop iterating over a cursor's result is automatically declared as a record variable.)

PL/pgSQL variables can have any SQL data type, such as integer, varchar, and char.

Here are some examples of variable declarations:

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

The general syntax of a variable declaration is:

```
name [ CONSTANT ] type [ COLLATE collation_name ] [ NOT NULL ] [ { DEFAULT | := | = } expressi
```

The DEFAULT clause, if given, specifies the initial value assigned to the variable when the block is entered. If the DEFAULT clause is not given then the variable is initialized to the SQL null value. The CONSTANT option prevents the variable from being assigned to after initialization, so that its value will remain constant for the duration of the block. The COLLATE option specifies a collation to use for the variable (see Section 40.3.6). If NOT NULL is specified, an assignment of a null value results in a run-time error. All variables declared as NOT NULL must have a nonnull default value specified. Equal (=) can be used instead of PL/SQL-compliant :=.

A variable's default value is evaluated and assigned to the variable each time the block is entered (not just once per function call). So, for example, assigning `now()` to a variable of type `timestamp` causes the variable to have the time of the current function call, not the time when the function was precompiled.

Examples:

```
quantity integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

40.3.1. Declaring Function Parameters

Parameters passed to functions are named with the identifiers `$1`, `$2`, etc. Optionally, aliases can be declared for *`$n`* parameter names for increased readability. Either the alias or the numeric identifier can then be used to refer to the parameter value.

There are two ways to create an alias. The preferred way is to give a name to the parameter in the `CREATE FUNCTION` command, for example:

```
CREATE FUNCTION sales_tax(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

The other way is to explicitly declare an alias, using the declaration syntax

```
name ALIAS FOR $n;
```

The same example in this style looks like:

```
CREATE FUNCTION sales_tax(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

> **Note:** These two examples are not perfectly equivalent. In the first case, `subtotal` could be referenced as `sales_tax.subtotal`, but in the second case it could not. (Had we attached a label to the inner block, `subtotal` could be qualified with that label, instead.)

Some more examples:

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
```

```
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- some computations using v_string and index here
END;
$$ LANGUAGE plpgsql;


CREATE FUNCTION concat_selected_fields(in_t sometablename) RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;
```

When a PL/pgSQL function is declared with output parameters, the output parameters are given `$n` names and optional aliases in just the same way as the normal input parameters. An output parameter is effectively a variable that starts out NULL; it should be assigned to during the execution of the function. The final value of the parameter is what is returned. For instance, the sales-tax example could also be done this way:

```
CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Notice that we omitted `RETURNS real` — we could have included it, but it would be redundant.

Output parameters are most useful when returning multiple values. A trivial example is:

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$
BEGIN
    sum := x + y;
    prod := x * y;
END;
$$ LANGUAGE plpgsql;
```

As discussed in Section 35.4.4, this effectively creates an anonymous record type for the function's results. If a `RETURNS` clause is given, it must say `RETURNS record`.

Another way to declare a PL/pgSQL function is with `RETURNS TABLE`, for example:

```
CREATE FUNCTION extended_sales(p_itemno int)
RETURNS TABLE(quantity int, total numeric) AS $$
BEGIN
    RETURN QUERY SELECT s.quantity, s.quantity * s.price FROM sales AS s
                 WHERE s.itemno = p_itemno;
END;
$$ LANGUAGE plpgsql;
```

This is exactly equivalent to declaring one or more `OUT` parameters and specifying `RETURNS SETOF` *`sometype`*.

When the return type of a PL/pgSQL function is declared as a polymorphic type (anyelement, anyarray, anynonarray, anyenum, or anyrange), a special parameter $0 is created. Its data type is the actual return type of the function, as deduced from the actual input types (see Section 35.2.5). This allows the function to access its actual return type as shown in Section 40.3.3. $0 is initialized to null and can be modified by the function, so it can be used to hold the return value if desired, though that is not required. $0 can also be given an alias. For example, this function works on any data type that has a + operator:

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement)
RETURNS anyelement AS $$
DECLARE
    result ALIAS FOR $0;
BEGIN
    result := v1 + v2 + v3;
    RETURN result;
END;
$$ LANGUAGE plpgsql;
```

The same effect can be had by declaring one or more output parameters as polymorphic types. In this case the special $0 parameter is not used; the output parameters themselves serve the same purpose. For example:

```
CREATE FUNCTION add_three_values(v1 anyelement, v2 anyelement, v3 anyelement,
                                 OUT sum anyelement)
AS $$
BEGIN
    sum := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

40.3.2. ALIAS

```
newname ALIAS FOR oldname;
```

The ALIAS syntax is more general than is suggested in the previous section: you can declare an alias for any variable, not just function parameters. The main practical use for this is to assign a different name for variables with predetermined names, such as NEW or OLD within a trigger procedure.

Examples:

```
DECLARE
  prior ALIAS FOR old;
  updated ALIAS FOR new;
```

Since ALIAS creates two different ways to name the same object, unrestricted use can be confusing. It's best to use it only for the purpose of overriding predetermined names.

40.3.3. Copying Types

variable%TYPE

%TYPE provides the data type of a variable or table column. You can use this to declare variables that will hold database values. For example, let's say you have a column named user_id in your users table. To declare a variable with the same data type as users.user_id you write:

```
user_id users.user_id%TYPE;
```

By using %TYPE you don't need to know the data type of the structure you are referencing, and most importantly, if the data type of the referenced item changes in the future (for instance: you change the type of user_id from integer to real), you might not need to change your function definition.

%TYPE is particularly valuable in polymorphic functions, since the data types needed for internal variables can change from one call to the next. Appropriate variables can be created by applying %TYPE to the function's arguments or result placeholders.

40.3.4. Row Types

name table_name%ROWTYPE;
name composite_type_name;

A variable of a composite type is called a *row* variable (or *row-type* variable). Such a variable can hold a whole row of a SELECT or FOR query result, so long as that query's column set matches the declared type of the variable. The individual fields of the row value are accessed using the usual dot notation, for example rowvar.field.

A row variable can be declared to have the same type as the rows of an existing table or view, by using the *table_name*%ROWTYPE notation; or it can be declared by giving a composite type's name. (Since every table has an associated composite type of the same name, it actually does not matter in PostgreSQL whether you write %ROWTYPE or not. But the form with %ROWTYPE is more portable.)

Parameters to a function can be composite types (complete table rows). In that case, the corresponding identifier $*n* will be a row variable, and fields can be selected from it, for example $1.user_id.

Only the user-defined columns of a table row are accessible in a row-type variable, not the OID or other system columns (because the row could be from a view). The fields of the row type inherit the table's field size or precision for data types such as char(*n*).

Here is an example of using composite types. table1 and table2 are existing tables having at least the mentioned fields:

```
CREATE FUNCTION merge_fields(t_row table1) RETURNS text AS $$
DECLARE
    t2_row table2%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM table2 WHERE ... ;
    RETURN t_row.f1 || t2_row.f3 || t_row.f5 || t2_row.f7;
END;
$$ LANGUAGE plpgsql;
```

```
SELECT merge_fields(t.*) FROM table1 t WHERE ... ;
```

40.3.5. Record Types

```
name RECORD;
```

Record variables are similar to row-type variables, but they have no predefined structure. They take on the actual row structure of the row they are assigned during a `SELECT` or `FOR` command. The substructure of a record variable can change each time it is assigned to. A consequence of this is that until a record variable is first assigned to, it has no substructure, and any attempt to access a field in it will draw a run-time error.

Note that `RECORD` is not a true data type, only a placeholder. One should also realize that when a PL/pgSQL function is declared to return type `record`, this is not quite the same concept as a record variable, even though such a function might use a record variable to hold its result. In both cases the actual row structure is unknown when the function is written, but for a function returning `record` the actual structure is determined when the calling query is parsed, whereas a record variable can change its row structure on-the-fly.

40.3.6. Collation of PL/pgSQL Variables

When a PL/pgSQL function has one or more parameters of collatable data types, a collation is identified for each function call depending on the collations assigned to the actual arguments, as described in Section 22.2. If a collation is successfully identified (i.e., there are no conflicts of implicit collations among the arguments) then all the collatable parameters are treated as having that collation implicitly. This will affect the behavior of collation-sensitive operations within the function. For example, consider

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b;
END;
$$ LANGUAGE plpgsql;

SELECT less_than(text_field_1, text_field_2) FROM table1;
SELECT less_than(text_field_1, text_field_2 COLLATE "C") FROM table1;
```

The first use of `less_than` will use the common collation of `text_field_1` and `text_field_2` for the comparison, while the second use will use `C` collation.

Furthermore, the identified collation is also assumed as the collation of any local variables that are of collatable types. Thus this function would not work any differently if it were written as

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$
DECLARE
    local_a text := a;
    local_b text := b;
BEGIN
    RETURN local_a < local_b;
```

```
END;
$$ LANGUAGE plpgsql;
```

If there are no parameters of collatable data types, or no common collation can be identified for them, then parameters and local variables use the default collation of their data type (which is usually the database's default collation, but could be different for variables of domain types).

A local variable of a collatable data type can have a different collation associated with it by including the `COLLATE` option in its declaration, for example

```
DECLARE
    local_a text COLLATE "en_US";
```

This option overrides the collation that would otherwise be given to the variable according to the rules above.

Also, of course explicit `COLLATE` clauses can be written inside a function if it is desired to force a particular collation to be used in a particular operation. For example,

```
CREATE FUNCTION less_than_c(a text, b text) RETURNS boolean AS $$
BEGIN
    RETURN a < b COLLATE "C";
END;
$$ LANGUAGE plpgsql;
```

This overrides the collations associated with the table columns, parameters, or local variables used in the expression, just as would happen in a plain SQL command.

40.4. Expressions

All expressions used in PL/pgSQL statements are processed using the server's main SQL executor. For example, when you write a PL/pgSQL statement like

```
IF expression THEN ...
```

PL/pgSQL will evaluate the expression by feeding a query like

```
SELECT expression
```

to the main SQL engine. While forming the `SELECT` command, any occurrences of PL/pgSQL variable names are replaced by parameters, as discussed in detail in Section 40.10.1. This allows the query plan for the `SELECT` to be prepared just once and then reused for subsequent evaluations with different values of the variables. Thus, what really happens on first use of an expression is essentially a `PREPARE` command. For example, if we have declared two integer variables `x` and `y`, and we write

```
IF x < y THEN ...
```

what happens behind the scenes is equivalent to

```
PREPARE statement_name(integer, integer) AS SELECT $1 < $2;
```

and then this prepared statement is EXECUTEd for each execution of the IF statement, with the current values of the PL/pgSQL variables supplied as parameter values. Normally these details are not important to a PL/pgSQL user, but they are useful to know when trying to diagnose a problem. More information appears in Section 40.10.2.

40.5. Basic Statements

In this section and the following ones, we describe all the statement types that are explicitly understood by PL/pgSQL. Anything not recognized as one of these statement types is presumed to be an SQL command and is sent to the main database engine to execute, as described in Section 40.5.2 and Section 40.5.3.

40.5.1. Assignment

An assignment of a value to a PL/pgSQL variable is written as:

```
variable { := | = } expression;
```

As explained previously, the expression in such a statement is evaluated by means of an SQL SELECT command sent to the main database engine. The expression must yield a single value (possibly a row value, if the variable is a row or record variable). The target variable can be a simple variable (optionally qualified with a block name), a field of a row or record variable, or an element of an array that is a simple variable or field. Equal (=) can be used instead of PL/SQL-compliant :=.

If the expression's result data type doesn't match the variable's data type, the value will be coerced as though by an assignment cast (see Section 10.4). If no assignment cast is known for the pair of data types involved, the PL/pgSQL interpreter will attempt to convert the result value textually, that is by applying the result type's output function followed by the variable type's input function. Note that this could result in run-time errors generated by the input function, if the string form of the result value is not acceptable to the input function.

Examples:

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

40.5.2. Executing a Command With No Result

For any SQL command that does not return rows, for example INSERT without a RETURNING clause, you can execute the command within a PL/pgSQL function just by writing the command.

Any PL/pgSQL variable name appearing in the command text is treated as a parameter, and then the current value of the variable is provided as the parameter value at run time. This is exactly like the processing described earlier for expressions; for details see Section 40.10.1.

When executing a SQL command in this way, PL/pgSQL may cache and re-use the execution plan for the command, as discussed in Section 40.10.2.

Sometimes it is useful to evaluate an expression or SELECT query but discard the result, for example when calling a function that has side-effects but no useful result value. To do this in PL/pgSQL, use the PERFORM statement:

```
PERFORM query;
```

This executes *query* and discards the result. Write the *query* the same way you would write an SQL SELECT command, but replace the initial keyword SELECT with PERFORM. For WITH queries, use PERFORM and then place the query in parentheses. (In this case, the query can only return one row.) PL/pgSQL variables will be substituted into the query just as for commands that return no result, and the plan is cached in the same way. Also, the special variable FOUND is set to true if the query produced at least one row, or false if it produced no rows (see Section 40.5.5).

> **Note:** One might expect that writing SELECT directly would accomplish this result, but at present the only accepted way to do it is PERFORM. A SQL command that can return rows, such as SELECT, will be rejected as an error unless it has an INTO clause as discussed in the next section.

An example:

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

40.5.3. Executing a Query with a Single-row Result

The result of a SQL command yielding a single row (possibly of multiple columns) can be assigned to a record variable, row-type variable, or list of scalar variables. This is done by writing the base SQL command and adding an INTO clause. For example,

```
SELECT select_expressions INTO [STRICT] target FROM ...;
INSERT ... RETURNING expressions INTO [STRICT] target;
UPDATE ... RETURNING expressions INTO [STRICT] target;
DELETE ... RETURNING expressions INTO [STRICT] target;
```

where *target* can be a record variable, a row variable, or a comma-separated list of simple variables and record/row fields. PL/pgSQL variables will be substituted into the rest of the query, and the plan is cached, just as described above for commands that do not return rows. This works for SELECT, INSERT/UPDATE/DELETE with RETURNING, and utility commands that return row-set results (such as EXPLAIN). Except for the INTO clause, the SQL command is the same as it would be written outside PL/pgSQL.

> **Tip:** Note that this interpretation of SELECT with INTO is quite different from PostgreSQL's regular SELECT INTO command, wherein the INTO target is a newly created table. If you want to create a table from a SELECT result inside a PL/pgSQL function, use the syntax CREATE TABLE ... AS SELECT.

If a row or a variable list is used as target, the query's result columns must exactly match the structure of the target as to number and data types, or else a run-time error occurs. When a record variable is the target, it automatically configures itself to the row type of the query result columns.

The INTO clause can appear almost anywhere in the SQL command. Customarily it is written either just before or just after the list of *select_expressions* in a SELECT command, or at the end of the command for other command types. It is recommended that you follow this convention in case the PL/pgSQL parser becomes stricter in future versions.

If STRICT is not specified in the INTO clause, then *target* will be set to the first row returned by the query, or to nulls if the query returned no rows. (Note that "the first row" is not well-defined unless you've used ORDER BY.) Any result rows after the first row are discarded. You can check the special FOUND variable (see Section 40.5.5) to determine whether a row was returned:

```
SELECT * INTO myrec FROM emp WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION 'employee % not found', myname;
END IF;
```

If the STRICT option is specified, the query must return exactly one row or a run-time error will be reported, either NO_DATA_FOUND (no rows) or TOO_MANY_ROWS (more than one row). You can use an exception block if you wish to catch the error, for example:

```
BEGIN
    SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RAISE EXCEPTION 'employee % not found', myname;
        WHEN TOO_MANY_ROWS THEN
            RAISE EXCEPTION 'employee % not unique', myname;
END;
```

Successful execution of a command with STRICT always sets FOUND to true.

For INSERT/UPDATE/DELETE with RETURNING, PL/pgSQL reports an error for more than one returned row, even when STRICT is not specified. This is because there is no option such as ORDER BY with which to determine which affected row should be returned.

If print_strict_params is enabled for the function, then when an error is thrown because the requirements of STRICT are not met, the DETAIL part of the error message will include information about the parameters passed to the query. You can change the print_strict_params setting for all functions by setting plpgsql.print_strict_params, though only subsequent function compilations will be affected. You can also enable it on a per-function basis by using a compiler option, for example:

```
CREATE FUNCTION get_userid(username text) RETURNS int
AS $$
#print_strict_params on
DECLARE
userid int;
BEGIN
    SELECT users.userid INTO STRICT userid
        FROM users WHERE users.username = get_userid.username;
    RETURN userid;
END;
```

```
$$ LANGUAGE plpgsql;
```

On failure, this function might produce an error message such as

```
ERROR:  query returned no rows
DETAIL:  parameters: $1 = 'nosuchuser'
CONTEXT:  PL/pgSQL function get_userid(text) line 6 at SQL statement
```

> **Note:** The `STRICT` option matches the behavior of Oracle PL/SQL's `SELECT INTO` and related statements.

To handle cases where you need to process multiple result rows from a SQL query, see Section 40.6.4.

40.5.4. Executing Dynamic Commands

Oftentimes you will want to generate dynamic commands inside your PL/pgSQL functions, that is, commands that will involve different tables or different data types each time they are executed. PL/pgSQL's normal attempts to cache plans for commands (as discussed in Section 40.10.2) will not work in such scenarios. To handle this sort of problem, the `EXECUTE` statement is provided:

```
EXECUTE command-string [ INTO [STRICT] target ] [ USING expression [, ... ] ];
```

where *command-string* is an expression yielding a string (of type `text`) containing the command to be executed. The optional *target* is a record variable, a row variable, or a comma-separated list of simple variables and record/row fields, into which the results of the command will be stored. The optional `USING` expressions supply values to be inserted into the command.

No substitution of PL/pgSQL variables is done on the computed command string. Any required variable values must be inserted in the command string as it is constructed; or you can use parameters as described below.

Also, there is no plan caching for commands executed via `EXECUTE`. Instead, the command is always planned each time the statement is run. Thus the command string can be dynamically created within the function to perform actions on different tables and columns.

The `INTO` clause specifies where the results of a SQL command returning rows should be assigned. If a row or variable list is provided, it must exactly match the structure of the query's results (when a record variable is used, it will configure itself to match the result structure automatically). If multiple rows are returned, only the first will be assigned to the `INTO` variable. If no rows are returned, NULL is assigned to the `INTO` variable(s). If no `INTO` clause is specified, the query results are discarded.

If the `STRICT` option is given, an error is reported unless the query produces exactly one row.

The command string can use parameter values, which are referenced in the command as `$1`, `$2`, etc. These symbols refer to values supplied in the `USING` clause. This method is often preferable to inserting data values into the command string as text: it avoids run-time overhead of converting the values to text and back, and it is much less prone to SQL-injection attacks since there is no need for quoting or escaping. An example is:

```
EXECUTE 'SELECT count(*) FROM mytable WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

Note that parameter symbols can only be used for data values — if you want to use dynamically determined table or column names, you must insert them into the command string textually. For example, if the preceding query needed to be done against a dynamically selected table, you could do this:

```
EXECUTE 'SELECT count(*) FROM '
    || quote_ident(tabname)
    || ' WHERE inserted_by = $1 AND inserted <= $2'
   INTO c
   USING checked_user, checked_date;
```

A cleaner approach is to use `format()`'s `%I` specification for table or column names (strings separated by a newline are concatenated):

```
EXECUTE format('SELECT count(*) FROM %I '
   'WHERE inserted_by = $1 AND inserted <= $2', tabname)
   INTO c
   USING checked_user, checked_date;
```

Another restriction on parameter symbols is that they only work in `SELECT`, `INSERT`, `UPDATE`, and `DELETE` commands. In other statement types (generically called utility statements), you must insert values textually even if they are just data values.

An `EXECUTE` with a simple constant command string and some `USING` parameters, as in the first example above, is functionally equivalent to just writing the command directly in PL/pgSQL and allowing replacement of PL/pgSQL variables to happen automatically. The important difference is that `EXECUTE` will re-plan the command on each execution, generating a plan that is specific to the current parameter values; whereas PL/pgSQL may otherwise create a generic plan and cache it for re-use. In situations where the best plan depends strongly on the parameter values, it can be helpful to use `EXECUTE` to positively ensure that a generic plan is not selected.

`SELECT INTO` is not currently supported within `EXECUTE`; instead, execute a plain `SELECT` command and specify `INTO` as part of the `EXECUTE` itself.

> **Note:** The PL/pgSQL `EXECUTE` statement is not related to the EXECUTE SQL statement supported by the PostgreSQL server. The server's `EXECUTE` statement cannot be used directly within PL/pgSQL functions (and is not needed).

Example 40-1. Quoting Values In Dynamic Queries

When working with dynamic commands you will often have to handle escaping of single quotes. The recommended method for quoting fixed text in your function body is dollar quoting. (If you have legacy code that does not use dollar quoting, please refer to the overview in Section 40.11.1, which can save you some effort when translating said code to a more reasonable scheme.)

Dynamic values require careful handling since they might contain quote characters. An example using `format()` (this assumes that you are dollar quoting the function body so quote marks need not be doubled):

```
EXECUTE format('UPDATE tbl SET %I = $1 '
   'WHERE key = $2', colname) USING newvalue, keyvalue;
```

It is also possible to call the quoting functions directly:

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_literal(newvalue)
        || ' WHERE key = '
        || quote_literal(keyvalue);
```

This example demonstrates the use of the `quote_ident` and `quote_literal` functions (see Section 9.4). For safety, expressions containing column or table identifiers should be passed through `quote_ident` before insertion in a dynamic query. Expressions containing values that should be literal strings in the constructed command should be passed through `quote_literal`. These functions take the appropriate steps to return the input text enclosed in double or single quotes respectively, with any embedded special characters properly escaped.

Because `quote_literal` is labeled `STRICT`, it will always return null when called with a null argument. In the above example, if `newvalue` or `keyvalue` were null, the entire dynamic query string would become null, leading to an error from `EXECUTE`. You can avoid this problem by using the `quote_nullable` function, which works the same as `quote_literal` except that when called with a null argument it returns the string `NULL`. For example,

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = '
        || quote_nullable(newvalue)
        || ' WHERE key = '
        || quote_nullable(keyvalue);
```

If you are dealing with values that might be null, you should usually use `quote_nullable` in place of `quote_literal`.

As always, care must be taken to ensure that null values in a query do not deliver unintended results. For example the `WHERE` clause

```
'WHERE key = ' || quote_nullable(keyvalue)
```

will never succeed if `keyvalue` is null, because the result of using the equality operator `=` with a null operand is always null. If you wish null to work like an ordinary key value, you would need to rewrite the above as

```
'WHERE key IS NOT DISTINCT FROM ' || quote_nullable(keyvalue)
```

(At present, `IS NOT DISTINCT FROM` is handled much less efficiently than `=`, so don't do this unless you must. See Section 9.2 for more information on nulls and `IS DISTINCT`.)

Note that dollar quoting is only useful for quoting fixed text. It would be a very bad idea to try to write this example as:

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(colname)
        || ' = $$'
        || newvalue
        || '$$ WHERE key = '
```

```
        || quote_literal(keyvalue);
```

because it would break if the contents of `newvalue` happened to contain `$$`. The same objection would apply to any other dollar-quoting delimiter you might pick. So, to safely quote text that is not known in advance, you *must* use `quote_literal`, `quote_nullable`, or `quote_ident`, as appropriate.

Dynamic SQL statements can also be safely constructed using the `format` function (see Section 9.4). For example:

```
EXECUTE format('UPDATE tbl SET %I = %L '
   'WHERE key = %L', colname, newvalue, keyvalue);
```

`%I` is equivalent to `quote_ident`, and `%L` is equivalent to `quote_nullable`. The `format` function can be used in conjunction with the `USING` clause:

```
EXECUTE format('UPDATE tbl SET %I = $1 WHERE key = $2', colname)
   USING newvalue, keyvalue;
```

This form is better because the variables are handled in their native data type format, rather than unconditionally converting them to text and quoting them via `%L`. It is also more efficient.

A much larger example of a dynamic command and `EXECUTE` can be seen in Example 40-9, which builds and executes a `CREATE FUNCTION` command to define a new function.

40.5.5. Obtaining the Result Status

There are several ways to determine the effect of a command. The first method is to use the `GET DIAGNOSTICS` command, which has the form:

```
GET [ CURRENT ] DIAGNOSTICS variable { = | := } item [ , ... ];
```

This command allows retrieval of system status indicators. `CURRENT` is a noise word (but see also `GET STACKED DIAGNOSTICS` in Section 40.6.6.1). Each *`item`* is a key word identifying a status value to be assigned to the specified *`variable`* (which should be of the right data type to receive it). The currently available status items are shown in Table 40-1. Colon-equal (`:=`) can be used instead of the SQL-standard `=` token. An example:

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

Table 40-1. Available Diagnostics Items

Name	Type	Description
`ROW_COUNT`	`integer`	the number of rows processed by the most recent SQL command
`RESULT_OID`	`oid`	the OID of the last row inserted by the most recent SQL command (only useful after an `INSERT` command into a table having OIDs)

Name	Type	Description
PG_CONTEXT	text	line(s) of text describing the current call stack (see Section 40.6.7)

The second method to determine the effects of a command is to check the special variable named FOUND, which is of type boolean. FOUND starts out false within each PL/pgSQL function call. It is set by each of the following types of statements:

- A SELECT INTO statement sets FOUND true if a row is assigned, false if no row is returned.
- A PERFORM statement sets FOUND true if it produces (and discards) one or more rows, false if no row is produced.
- UPDATE, INSERT, and DELETE statements set FOUND true if at least one row is affected, false if no row is affected.
- A FETCH statement sets FOUND true if it returns a row, false if no row is returned.
- A MOVE statement sets FOUND true if it successfully repositions the cursor, false otherwise.
- A FOR or FOREACH statement sets FOUND true if it iterates one or more times, else false. FOUND is set this way when the loop exits; inside the execution of the loop, FOUND is not modified by the loop statement, although it might be changed by the execution of other statements within the loop body.
- RETURN QUERY and RETURN QUERY EXECUTE statements set FOUND true if the query returns at least one row, false if no row is returned.

Other PL/pgSQL statements do not change the state of FOUND. Note in particular that EXECUTE changes the output of GET DIAGNOSTICS, but does not change FOUND.

FOUND is a local variable within each PL/pgSQL function; any changes to it affect only the current function.

40.5.6. Doing Nothing At All

Sometimes a placeholder statement that does nothing is useful. For example, it can indicate that one arm of an if/then/else chain is deliberately empty. For this purpose, use the NULL statement:

```
NULL;
```

For example, the following two fragments of code are equivalent:

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        NULL;  -- ignore the error
END;

BEGIN
```

```
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN  -- ignore the error
END;
```

Which is preferable is a matter of taste.

> **Note:** In Oracle's PL/SQL, empty statement lists are not allowed, and so `NULL` statements are *required* for situations such as this. PL/pgSQL allows you to just write nothing, instead.

40.6. Control Structures

Control structures are probably the most useful (and important) part of PL/pgSQL. With PL/pgSQL's control structures, you can manipulate PostgreSQL data in a very flexible and powerful way.

40.6.1. Returning From a Function

There are two commands available that allow you to return data from a function: `RETURN` and `RETURN NEXT`.

40.6.1.1. `RETURN`

```
RETURN expression;
```

`RETURN` with an expression terminates the function and returns the value of *`expression`* to the caller. This form is used for PL/pgSQL functions that do not return a set.

In a function that returns a scalar type, the expression's result will automatically be cast into the function's return type as described for assignments. But to return a composite (row) value, you must write an expression delivering exactly the requested column set. This may require use of explicit casting.

If you declared the function with output parameters, write just `RETURN` with no expression. The current values of the output parameter variables will be returned.

If you declared the function to return `void`, a `RETURN` statement can be used to exit the function early; but do not write an expression following `RETURN`.

The return value of a function cannot be left undefined. If control reaches the end of the top-level block of the function without hitting a `RETURN` statement, a run-time error will occur. This restriction does not apply to functions with output parameters and functions returning `void`, however. In those cases a `RETURN` statement is automatically executed if the top-level block finishes.

Some examples:

```
-- functions returning a scalar type
RETURN 1 + 2;
RETURN scalar_var;
```

```
-- functions returning a composite type
RETURN composite_type_var;
RETURN (1, 2, 'three'::text);  -- must cast columns to correct types
```

40.6.1.2. RETURN NEXT and RETURN QUERY

```
RETURN NEXT expression;
RETURN QUERY query;
RETURN QUERY EXECUTE command-string [ USING expression [, ... ] ];
```

When a PL/pgSQL function is declared to return SETOF *sometype*, the procedure to follow is slightly different. In that case, the individual items to return are specified by a sequence of RETURN NEXT or RETURN QUERY commands, and then a final RETURN command with no argument is used to indicate that the function has finished executing. RETURN NEXT can be used with both scalar and composite data types; with a composite result type, an entire "table" of results will be returned. RETURN QUERY appends the results of executing a query to the function's result set. RETURN NEXT and RETURN QUERY can be freely intermixed in a single set-returning function, in which case their results will be concatenated.

RETURN NEXT and RETURN QUERY do not actually return from the function — they simply append zero or more rows to the function's result set. Execution then continues with the next statement in the PL/pgSQL function. As successive RETURN NEXT or RETURN QUERY commands are executed, the result set is built up. A final RETURN, which should have no argument, causes control to exit the function (or you can just let control reach the end of the function).

RETURN QUERY has a variant RETURN QUERY EXECUTE, which specifies the query to be executed dynamically. Parameter expressions can be inserted into the computed query string via USING, in just the same way as in the EXECUTE command.

If you declared the function with output parameters, write just RETURN NEXT with no expression. On each execution, the current values of the output parameter variable(s) will be saved for eventual return as a row of the result. Note that you must declare the function as returning SETOF record when there are multiple output parameters, or SETOF *sometype* when there is just one output parameter of type *sometype*, in order to create a set-returning function with output parameters.

Here is an example of a function using RETURN NEXT:

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS
$BODY$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN
        SELECT * FROM foo WHERE fooid > 0
    LOOP
        -- can do some processing here
```

```
        RETURN NEXT r; -- return current row of SELECT
    END LOOP;
    RETURN;
END;
$BODY$
LANGUAGE plpgsql;

SELECT * FROM get_all_foo();
```

Here is an example of a function using `RETURN QUERY`:

```
CREATE FUNCTION get_available_flightid(date) RETURNS SETOF integer AS
$BODY$
BEGIN
    RETURN QUERY SELECT flightid
                   FROM flight
                  WHERE flightdate >= $1
                    AND flightdate < ($1 + 1);

    -- Since execution is not finished, we can check whether rows were returned
    -- and raise exception if not.
    IF NOT FOUND THEN
        RAISE EXCEPTION 'No flight at %.', $1;
    END IF;

    RETURN;
 END;
$BODY$
LANGUAGE plpgsql;

-- Returns available flights or raises exception if there are no
-- available flights.
SELECT * FROM get_available_flightid(CURRENT_DATE);
```

Note: The current implementation of `RETURN NEXT` and `RETURN QUERY` stores the entire result set before returning from the function, as discussed above. That means that if a PL/pgSQL function produces a very large result set, performance might be poor: data will be written to disk to avoid memory exhaustion, but the function itself will not return until the entire result set has been generated. A future version of PL/pgSQL might allow users to define set-returning functions that do not have this limitation. Currently, the point at which data begins being written to disk is controlled by the work_mem configuration variable. Administrators who have sufficient memory to store larger result sets in memory should consider increasing this parameter.

40.6.2. Conditionals

IF and CASE statements let you execute alternative commands based on certain conditions. PL/pgSQL has three forms of IF:

- IF ... THEN ... END IF
- IF ... THEN ... ELSE ... END IF
- IF ... THEN ... ELSIF ... THEN ... ELSE ... END IF

and two forms of CASE:

- CASE ... WHEN ... THEN ... ELSE ... END CASE
- CASE WHEN ... THEN ... ELSE ... END CASE

40.6.2.1. IF-THEN

```
IF boolean-expression THEN
    statements
END IF;
```

IF-THEN statements are the simplest form of IF. The statements between THEN and END IF will be executed if the condition is true. Otherwise, they are skipped.

Example:

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

40.6.2.2. IF-THEN-ELSE

```
IF boolean-expression THEN
    statements
ELSE
    statements
END IF;
```

IF-THEN-ELSE statements add to IF-THEN by letting you specify an alternative set of statements that should be executed if the condition is not true. (Note this includes the case where the condition evaluates to NULL.)

Examples:

```
IF parentid IS NULL OR parentid = ''
THEN
    RETURN fullname;
```

```
ELSE
    RETURN hp_true_filename(parentid) || '/' || fullname;
END IF;

IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

40.6.2.3. IF-THEN-ELSIF

```
IF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
[ ELSIF boolean-expression THEN
    statements
    ...]]
[ ELSE
    statements ]
END IF;
```

Sometimes there are more than just two alternatives. IF-THEN-ELSIF provides a convenient method of checking several alternatives in turn. The IF conditions are tested successively until the first one that is true is found. Then the associated statement(s) are executed, after which control passes to the next statement after END IF. (Any subsequent IF conditions are *not* tested.) If none of the IF conditions is true, then the ELSE block (if any) is executed.

Here is an example:

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- hmm, the only other possibility is that number is null
    result := 'NULL';
END IF;
```

The key word ELSIF can also be spelled ELSEIF.

An alternative way of accomplishing the same task is to nest IF-THEN-ELSE statements, as in the following example:

```
IF demo_row.sex = 'm' THEN
```

```
    pretty_sex := 'man';
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
END IF;
```

However, this method requires writing a matching `END IF` for each `IF`, so it is much more cumbersome than using `ELSIF` when there are many alternatives.

40.6.2.4. Simple CASE

```
CASE search-expression
    WHEN expression [, expression [ ... ]] THEN
      statements
  [ WHEN expression [, expression [ ... ]] THEN
      statements
    ... ]
  [ ELSE
      statements ]
END CASE;
```

The simple form of `CASE` provides conditional execution based on equality of operands. The *search-expression* is evaluated (once) and successively compared to each *expression* in the `WHEN` clauses. If a match is found, then the corresponding *statements* are executed, and then control passes to the next statement after `END CASE`. (Subsequent `WHEN` expressions are not evaluated.) If no match is found, the `ELSE` *statements* are executed; but if `ELSE` is not present, then a `CASE_NOT_FOUND` exception is raised.

Here is a simple example:

```
CASE x
    WHEN 1, 2 THEN
        msg := 'one or two';
    ELSE
        msg := 'other value than one or two';
END CASE;
```

40.6.2.5. Searched CASE

```
CASE
    WHEN boolean-expression THEN
      statements
  [ WHEN boolean-expression THEN
      statements
    ... ]
  [ ELSE
```

```
      statements ]
END CASE;
```

The searched form of CASE provides conditional execution based on truth of Boolean expressions. Each WHEN clause's *boolean-expression* is evaluated in turn, until one is found that yields true. Then the corresponding *statements* are executed, and then control passes to the next statement after END CASE. (Subsequent WHEN expressions are not evaluated.) If no true result is found, the ELSE *statements* are executed; but if ELSE is not present, then a CASE_NOT_FOUND exception is raised.

Here is an example:

```
CASE
    WHEN x BETWEEN 0 AND 10 THEN
        msg := 'value is between zero and ten';
    WHEN x BETWEEN 11 AND 20 THEN
        msg := 'value is between eleven and twenty';
END CASE;
```

This form of CASE is entirely equivalent to IF-THEN-ELSIF, except for the rule that reaching an omitted ELSE clause results in an error rather than doing nothing.

40.6.3. Simple Loops

With the LOOP, EXIT, CONTINUE, WHILE, FOR, and FOREACH statements, you can arrange for your PL/pgSQL function to repeat a series of commands.

40.6.3.1. LOOP

```
[ <<label>> ]
LOOP
    statements
END LOOP [ label ];
```

LOOP defines an unconditional loop that is repeated indefinitely until terminated by an EXIT or RETURN statement. The optional *label* can be used by EXIT and CONTINUE statements within nested loops to specify which loop those statements refer to.

40.6.3.2. EXIT

```
EXIT [ label ] [ WHEN boolean-expression ];
```

If no *label* is given, the innermost loop is terminated and the statement following END LOOP is executed next. If *label* is given, it must be the label of the current or some outer level of nested loop or block. Then the named loop or block is terminated and control continues with the statement after the loop's/block's corresponding END.

If WHEN is specified, the loop exit occurs only if *boolean-expression* is true. Otherwise, control passes to the statement after EXIT.

EXIT can be used with all types of loops; it is not limited to use with unconditional loops.

When used with a BEGIN block, EXIT passes control to the next statement after the end of the block. Note that a label must be used for this purpose; an unlabeled EXIT is never considered to match a BEGIN block. (This is a change from pre-8.4 releases of PostgreSQL, which would allow an unlabeled EXIT to match a BEGIN block.)

Examples:

```
LOOP
    -- some computations
    IF count > 0 THEN
        EXIT;  -- exit loop
    END IF;
END LOOP;

LOOP
    -- some computations
    EXIT WHEN count > 0;  -- same result as previous example
END LOOP;

<<ablock>>
BEGIN
    -- some computations
    IF stocks > 100000 THEN
        EXIT ablock;  -- causes exit from the BEGIN block
    END IF;
    -- computations here will be skipped when stocks > 100000
END;
```

40.6.3.3. CONTINUE

```
CONTINUE [ label ] [ WHEN boolean-expression ];
```

If no *label* is given, the next iteration of the innermost loop is begun. That is, all statements remaining in the loop body are skipped, and control returns to the loop control expression (if any) to determine whether another loop iteration is needed. If *label* is present, it specifies the label of the loop whose execution will be continued.

If WHEN is specified, the next iteration of the loop is begun only if *boolean-expression* is true. Otherwise, control passes to the statement after CONTINUE.

CONTINUE can be used with all types of loops; it is not limited to use with unconditional loops.

Examples:

```
LOOP
    -- some computations
    EXIT WHEN count > 100;
```

```
    CONTINUE WHEN count < 50;
    -- some computations for count IN [50 .. 100]
END LOOP;
```

40.6.3.4. WHILE

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

The WHILE statement repeats a sequence of statements so long as the *boolean-expression* evaluates to true. The expression is checked just before each entry to the loop body.

For example:

```
WHILE amount_owed > 0 AND gift_certificate_balance > 0 LOOP
    -- some computations here
END LOOP;

WHILE NOT done LOOP
    -- some computations here
END LOOP;
```

40.6.3.5. FOR (Integer Variant)

```
[ <<label>> ]
FOR name IN [ REVERSE ] expression .. expression [ BY expression ] LOOP
    statements
END LOOP [ label ];
```

This form of FOR creates a loop that iterates over a range of integer values. The variable *name* is automatically defined as type integer and exists only inside the loop (any existing definition of the variable name is ignored within the loop). The two expressions giving the lower and upper bound of the range are evaluated once when entering the loop. If the BY clause isn't specified the iteration step is 1, otherwise it's the value specified in the BY clause, which again is evaluated once on loop entry. If REVERSE is specified then the step value is subtracted, rather than added, after each iteration.

Some examples of integer FOR loops:

```
FOR i IN 1..10 LOOP
    -- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- i will take on the values 10,9,8,7,6,5,4,3,2,1 within the loop
END LOOP;
```

```
FOR i IN REVERSE 10..1 BY 2 LOOP
    -- i will take on the values 10,8,6,4,2 within the loop
END LOOP;
```

If the lower bound is greater than the upper bound (or less than, in the `REVERSE` case), the loop body is not executed at all. No error is raised.

If a *`label`* is attached to the `FOR` loop then the integer loop variable can be referenced with a qualified name, using that *`label`*.

40.6.4. Looping Through Query Results

Using a different type of `FOR` loop, you can iterate through the results of a query and manipulate that data accordingly. The syntax is:

```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

The *`target`* is a record variable, row variable, or comma-separated list of scalar variables. The *`target`* is successively assigned each row resulting from the *`query`* and the loop body is executed for each row. Here is an example:

```
CREATE FUNCTION refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    RAISE NOTICE 'Refreshing all materialized views...';

    FOR mviews IN
       SELECT n.nspname AS mv_schema,
              c.relname AS mv_name,
              pg_catalog.pg_get_userbyid(c.relowner) AS owner
         FROM pg_catalog.pg_class c
    LEFT JOIN pg_catalog.pg_namespace n ON (n.oid = c.relnamespace)
        WHERE c.relkind = 'm'
     ORDER BY 1
    LOOP

        -- Now "mviews" has one record with information about the materialized view

        RAISE NOTICE 'Refreshing materialized view %.% (owner: %)...',
                     quote_ident(mviews.mv_schema),
                     quote_ident(mviews.mv_name),
                     quote_ident(mviews.owner);
        EXECUTE format('REFRESH MATERIALIZED VIEW %I.%I', mviews.mv_schema, mviews.mv_name)
    END LOOP;
```

```
    RAISE NOTICE 'Done refreshing materialized views.';
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

If the loop is terminated by an EXIT statement, the last assigned row value is still accessible after the loop.

The *query* used in this type of FOR statement can be any SQL command that returns rows to the caller: SELECT is the most common case, but you can also use INSERT, UPDATE, or DELETE with a RETURNING clause. Some utility commands such as EXPLAIN will work too.

PL/pgSQL variables are substituted into the query text, and the query plan is cached for possible re-use, as discussed in detail in Section 40.10.1 and Section 40.10.2.

The FOR-IN-EXECUTE statement is another way to iterate over rows:

```
[ <<label>> ]
FOR target IN EXECUTE text_expression [ USING expression [, ... ] ] LOOP
    statements
END LOOP [ label ];
```

This is like the previous form, except that the source query is specified as a string expression, which is evaluated and replanned on each entry to the FOR loop. This allows the programmer to choose the speed of a preplanned query or the flexibility of a dynamic query, just as with a plain EXECUTE statement. As with EXECUTE, parameter values can be inserted into the dynamic command via USING.

Another way to specify the query whose results should be iterated through is to declare it as a cursor. This is described in Section 40.7.4.

40.6.5. Looping Through Arrays

The FOREACH loop is much like a FOR loop, but instead of iterating through the rows returned by a SQL query, it iterates through the elements of an array value. (In general, FOREACH is meant for looping through components of a composite-valued expression; variants for looping through composites besides arrays may be added in future.) The FOREACH statement to loop over an array is:

```
[ <<label>> ]
FOREACH target [ SLICE number ] IN ARRAY expression LOOP
    statements
END LOOP [ label ];
```

Without SLICE, or if SLICE 0 is specified, the loop iterates through individual elements of the array produced by evaluating the *expression*. The *target* variable is assigned each element value in sequence, and the loop body is executed for each element. Here is an example of looping through the elements of an integer array:

```
CREATE FUNCTION sum(int[]) RETURNS int8 AS $$
DECLARE
  s int8 := 0;
  x int;
```

```
BEGIN
  FOREACH x IN ARRAY $1
  LOOP
    s := s + x;
  END LOOP;
  RETURN s;
END;
$$ LANGUAGE plpgsql;
```

The elements are visited in storage order, regardless of the number of array dimensions. Although the *target* is usually just a single variable, it can be a list of variables when looping through an array of composite values (records). In that case, for each array element, the variables are assigned from successive columns of the composite value.

With a positive SLICE value, FOREACH iterates through slices of the array rather than single elements. The SLICE value must be an integer constant not larger than the number of dimensions of the array. The *target* variable must be an array, and it receives successive slices of the array value, where each slice is of the number of dimensions specified by SLICE. Here is an example of iterating through one-dimensional slices:

```
CREATE FUNCTION scan_rows(int[]) RETURNS void AS $$
DECLARE
  x int[];
BEGIN
  FOREACH x SLICE 1 IN ARRAY $1
  LOOP
    RAISE NOTICE 'row = %', x;
  END LOOP;
END;
$$ LANGUAGE plpgsql;

SELECT scan_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);

NOTICE:  row = {1,2,3}
NOTICE:  row = {4,5,6}
NOTICE:  row = {7,8,9}
NOTICE:  row = {10,11,12}
```

40.6.6. Trapping Errors

By default, any error occurring in a PL/pgSQL function aborts execution of the function and the surrounding transaction. You can trap errors and recover from them by using a BEGIN block with an EXCEPTION clause. The syntax is an extension of the normal syntax for a BEGIN block:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
```

```
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    [ WHEN condition [ OR condition ... ] THEN
          handler_statements
      ... ]
END;
```

If no error occurs, this form of block simply executes all the *statements*, and then control passes to the next statement after END. But if an error occurs within the *statements*, further processing of the *statements* is abandoned, and control passes to the EXCEPTION list. The list is searched for the first *condition* matching the error that occurred. If a match is found, the corresponding *handler_statements* are executed, and then control passes to the next statement after END. If no match is found, the error propagates out as though the EXCEPTION clause were not there at all: the error can be caught by an enclosing block with EXCEPTION, or if there is none it aborts processing of the function.

The *condition* names can be any of those shown in Appendix A. A category name matches any error within its category. The special condition name OTHERS matches every error type except QUERY_CANCELED and ASSERT_FAILURE. (It is possible, but often unwise, to trap those two error types by name.) Condition names are not case-sensitive. Also, an error condition can be specified by SQLSTATE code; for example these are equivalent:

```
WHEN division_by_zero THEN ...
WHEN SQLSTATE '22012' THEN ...
```

If a new error occurs within the selected *handler_statements*, it cannot be caught by this EXCEPTION clause, but is propagated out. A surrounding EXCEPTION clause could catch it.

When an error is caught by an EXCEPTION clause, the local variables of the PL/pgSQL function remain as they were when the error occurred, but all changes to persistent database state within the block are rolled back. As an example, consider this fragment:

```
INSERT INTO mytab(firstname, lastname) VALUES('Tom', 'Jones');
BEGIN
    UPDATE mytab SET firstname = 'Joe' WHERE lastname = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'caught division_by_zero';
        RETURN x;
END;
```

When control reaches the assignment to y, it will fail with a division_by_zero error. This will be caught by the EXCEPTION clause. The value returned in the RETURN statement will be the incremented value of x, but the effects of the UPDATE command will have been rolled back. The INSERT command preceding the block is not rolled back, however, so the end result is that the database contains Tom Jones not Joe Jones.

Tip: A block containing an EXCEPTION clause is significantly more expensive to enter and exit than a block without one. Therefore, don't use EXCEPTION without need.

Example 40-2. Exceptions with UPDATE/INSERT

This example uses exception handling to perform either UPDATE or INSERT, as appropriate. It is recommended that applications use INSERT with ON CONFLICT DO UPDATE rather than actually using this pattern. This example serves primarily to illustrate use of PL/pgSQL control flow structures:

```
CREATE TABLE db (a INT PRIMARY KEY, b TEXT);

CREATE FUNCTION merge_db(key INT, data TEXT) RETURNS VOID AS
$$
BEGIN
    LOOP
        -- first try to update the key
        UPDATE db SET b = data WHERE a = key;
        IF found THEN
            RETURN;
        END IF;
        -- not there, so try to insert the key
        -- if someone else inserts the same key concurrently,
        -- we could get a unique-key failure
        BEGIN
            INSERT INTO db(a,b) VALUES (key, data);
            RETURN;
        EXCEPTION WHEN unique_violation THEN
            -- Do nothing, and loop to try the UPDATE again.
        END;
    END LOOP;
END;
$$
LANGUAGE plpgsql;

SELECT merge_db(1, 'david');
SELECT merge_db(1, 'dennis');
```

This coding assumes the unique_violation error is caused by the INSERT, and not by, say, an INSERT in a trigger function on the table. It might also misbehave if there is more than one unique index on the table, since it will retry the operation regardless of which index caused the error. More safety could be had by using the features discussed next to check that the trapped error was the one expected.

40.6.6.1. Obtaining Information About an Error

Exception handlers frequently need to identify the specific error that occurred. There are two ways to get information about the current exception in PL/pgSQL: special variables and the GET STACKED DIAGNOSTICS command.

Within an exception handler, the special variable SQLSTATE contains the error code that corresponds to the exception that was raised (refer to Table A-1 for a list of possible error codes). The special variable SQLERRM contains the error message associated with the exception. These variables are undefined outside exception handlers.

Within an exception handler, one may also retrieve information about the current exception by using the GET STACKED DIAGNOSTICS command, which has the form:

```
GET STACKED DIAGNOSTICS variable { = | := } item [ , ... ];
```

Each *item* is a key word identifying a status value to be assigned to the specified *variable* (which should be of the right data type to receive it). The currently available status items are shown in Table 40-2.

Table 40-2. Error Diagnostics Items

Name	Type	Description
RETURNED_SQLSTATE	text	the SQLSTATE error code of the exception
COLUMN_NAME	text	the name of the column related to exception
CONSTRAINT_NAME	text	the name of the constraint related to exception
PG_DATATYPE_NAME	text	the name of the data type related to exception
MESSAGE_TEXT	text	the text of the exception's primary message
TABLE_NAME	text	the name of the table related to exception
SCHEMA_NAME	text	the name of the schema related to exception
PG_EXCEPTION_DETAIL	text	the text of the exception's detail message, if any
PG_EXCEPTION_HINT	text	the text of the exception's hint message, if any
PG_EXCEPTION_CONTEXT	text	line(s) of text describing the call stack at the time of the exception (see Section 40.6.7)

If the exception did not set a value for an item, an empty string will be returned.

Here is an example:

```
DECLARE
  text_var1 text;
  text_var2 text;
  text_var3 text;
BEGIN
  -- some processing which might cause an exception
  ...
```

```
EXCEPTION WHEN OTHERS THEN
  GET STACKED DIAGNOSTICS text_var1 = MESSAGE_TEXT,
                          text_var2 = PG_EXCEPTION_DETAIL,
                          text_var3 = PG_EXCEPTION_HINT;
END;
```

40.6.7. Obtaining Execution Location Information

The `GET DIAGNOSTICS` command, previously described in Section 40.5.5, retrieves information about current execution state (whereas the `GET STACKED DIAGNOSTICS` command discussed above reports information about the execution state as of a previous error). Its `PG_CONTEXT` status item is useful for identifying the current execution location. `PG_CONTEXT` returns a text string with line(s) of text describing the call stack. The first line refers to the current function and currently executing `GET DIAGNOSTICS` command. The second and any subsequent lines refer to calling functions further up the call stack. For example:

```
CREATE OR REPLACE FUNCTION outer_func() RETURNS integer AS $$
BEGIN
  RETURN inner_func();
END;
$$ LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION inner_func() RETURNS integer AS $$
DECLARE
  stack text;
BEGIN
  GET DIAGNOSTICS stack = PG_CONTEXT;
  RAISE NOTICE E'--- Call Stack ---\n%', stack;
  RETURN 1;
END;
$$ LANGUAGE plpgsql;

SELECT outer_func();

NOTICE:  --- Call Stack ---
PL/pgSQL function inner_func() line 5 at GET DIAGNOSTICS
PL/pgSQL function outer_func() line 3 at RETURN
CONTEXT:  PL/pgSQL function outer_func() line 3 at RETURN
 outer_func
 ------------
           1
(1 row)
```

`GET STACKED DIAGNOSTICS ... PG_EXCEPTION_CONTEXT` returns the same sort of stack trace, but describing the location at which an error was detected, rather than the current location.

40.7. Cursors

Rather than executing a whole query at once, it is possible to set up a *cursor* that encapsulates the query, and then read the query result a few rows at a time. One reason for doing this is to avoid memory overrun when the result contains a large number of rows. (However, PL/pgSQL users do not normally need to worry about that, since FOR loops automatically use a cursor internally to avoid memory problems.) A more interesting usage is to return a reference to a cursor that a function has created, allowing the caller to read the rows. This provides an efficient way to return large row sets from functions.

40.7.1. Declaring Cursor Variables

All access to cursors in PL/pgSQL goes through cursor variables, which are always of the special data type refcursor. One way to create a cursor variable is just to declare it as a variable of type refcursor. Another way is to use the cursor declaration syntax, which in general is:

```
name [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] FOR query;
```

(FOR can be replaced by IS for Oracle compatibility.) If SCROLL is specified, the cursor will be capable of scrolling backward; if NO SCROLL is specified, backward fetches will be rejected; if neither specification appears, it is query-dependent whether backward fetches will be allowed. *arguments*, if specified, is a comma-separated list of pairs *name datatype* that define names to be replaced by parameter values in the given query. The actual values to substitute for these names will be specified later, when the cursor is opened.

Some examples:

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (key integer) FOR SELECT * FROM tenk1 WHERE unique1 = key;
```

All three of these variables have the data type refcursor, but the first can be used with any query, while the second has a fully specified query already *bound* to it, and the last has a parameterized query bound to it. (key will be replaced by an integer parameter value when the cursor is opened.) The variable curs1 is said to be *unbound* since it is not bound to any particular query.

40.7.2. Opening Cursors

Before a cursor can be used to retrieve rows, it must be *opened*. (This is the equivalent action to the SQL command DECLARE CURSOR.) PL/pgSQL has three forms of the OPEN statement, two of which use unbound cursor variables while the third uses a bound cursor variable.

> **Note:** Bound cursor variables can also be used without explicitly opening the cursor, via the FOR statement described in Section 40.7.4.

40.7.2.1. OPEN FOR *query*

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR query;
```

The cursor variable is opened and given the specified query to execute. The cursor cannot be open already, and it must have been declared as an unbound cursor variable (that is, as a simple `refcursor` variable). The query must be a `SELECT`, or something else that returns rows (such as `EXPLAIN`). The query is treated in the same way as other SQL commands in PL/pgSQL: PL/pgSQL variable names are substituted, and the query plan is cached for possible reuse. When a PL/pgSQL variable is substituted into the cursor query, the value that is substituted is the one it has at the time of the `OPEN`; subsequent changes to the variable will not affect the cursor's behavior. The `SCROLL` and `NO SCROLL` options have the same meanings as for a bound cursor.

An example:

```
OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

40.7.2.2. OPEN FOR EXECUTE

```
OPEN unbound_cursorvar [ [ NO ] SCROLL ] FOR EXECUTE query_string
                                        [ USING expression [, ... ] ];
```

The cursor variable is opened and given the specified query to execute. The cursor cannot be open already, and it must have been declared as an unbound cursor variable (that is, as a simple `refcursor` variable). The query is specified as a string expression, in the same way as in the `EXECUTE` command. As usual, this gives flexibility so the query plan can vary from one run to the next (see Section 40.10.2), and it also means that variable substitution is not done on the command string. As with `EXECUTE`, parameter values can be inserted into the dynamic command via `format()` and `USING`. The `SCROLL` and `NO SCROLL` options have the same meanings as for a bound cursor.

An example:

```
OPEN curs1 FOR EXECUTE format('SELECT * FROM %I WHERE col1 = $1',tabname) USING keyvalue;
```

In this example, the table name is inserted into the query via `format()`. The comparison value for `col1` is inserted via a `USING` parameter, so it needs no quoting.

40.7.2.3. Opening a Bound Cursor

```
OPEN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ];
```

This form of `OPEN` is used to open a cursor variable whose query was bound to it when it was declared. The cursor cannot be open already. A list of actual argument value expressions must appear if and only if the cursor was declared to take arguments. These values will be substituted in the query.

The query plan for a bound cursor is always considered cacheable; there is no equivalent of `EXECUTE` in this case. Notice that `SCROLL` and `NO SCROLL` cannot be specified in `OPEN`, as the cursor's scrolling behavior was already determined.

Argument values can be passed using either *positional* or *named* notation. In positional notation, all arguments are specified in order. In named notation, each argument's name is specified using := to separate it from the argument expression. Similar to calling functions, described in Section 4.3, it is also allowed to mix positional and named notation.

Examples (these use the cursor declaration examples above):

```
OPEN curs2;
OPEN curs3(42);
OPEN curs3(key := 42);
```

Because variable substitution is done on a bound cursor's query, there are really two ways to pass values into the cursor: either with an explicit argument to OPEN, or implicitly by referencing a PL/pgSQL variable in the query. However, only variables declared before the bound cursor was declared will be substituted into it. In either case the value to be passed is determined at the time of the OPEN. For example, another way to get the same effect as the curs3 example above is

```
DECLARE
    key integer;
    curs4 CURSOR FOR SELECT * FROM tenk1 WHERE unique1 = key;
BEGIN
    key := 42;
    OPEN curs4;
```

40.7.3. Using Cursors

Once a cursor has been opened, it can be manipulated with the statements described here.

These manipulations need not occur in the same function that opened the cursor to begin with. You can return a refcursor value out of a function and let the caller operate on the cursor. (Internally, a refcursor value is simply the string name of a so-called portal containing the active query for the cursor. This name can be passed around, assigned to other refcursor variables, and so on, without disturbing the portal.)

All portals are implicitly closed at transaction end. Therefore a refcursor value is usable to reference an open cursor only until the end of the transaction.

40.7.3.1. FETCH

```
FETCH [ direction { FROM | IN } ] cursor INTO target;
```

FETCH retrieves the next row from the cursor into a target, which might be a row variable, a record variable, or a comma-separated list of simple variables, just like SELECT INTO. If there is no next row, the target is set to NULL(s). As with SELECT INTO, the special variable FOUND can be checked to see whether a row was obtained or not.

The *direction* clause can be any of the variants allowed in the SQL FETCH command except the ones that can fetch more than one row; namely, it can be NEXT, PRIOR, FIRST, LAST, ABSOLUTE *count*,

RELATIVE *count*, FORWARD, or BACKWARD. Omitting *direction* is the same as specifying NEXT. In the forms using a *count*, the *count* can be any integer-valued expression (unlike the SQL FETCH command, which only allows an integer constant). *direction* values that require moving backward are likely to fail unless the cursor was declared or opened with the SCROLL option.

cursor must be the name of a refcursor variable that references an open cursor portal.

Examples:

```
FETCH curs1 INTO rowvar;
FETCH curs2 INTO foo, bar, baz;
FETCH LAST FROM curs3 INTO x, y;
FETCH RELATIVE -2 FROM curs4 INTO x;
```

40.7.3.2. MOVE

```
MOVE [ direction { FROM | IN } ] cursor;
```

MOVE repositions a cursor without retrieving any data. MOVE works exactly like the FETCH command, except it only repositions the cursor and does not return the row moved to. As with SELECT INTO, the special variable FOUND can be checked to see whether there was a next row to move to.

Examples:

```
MOVE curs1;
MOVE LAST FROM curs3;
MOVE RELATIVE -2 FROM curs4;
MOVE FORWARD 2 FROM curs4;
```

40.7.3.3. UPDATE/DELETE WHERE CURRENT OF

```
UPDATE table SET ... WHERE CURRENT OF cursor;
DELETE FROM table WHERE CURRENT OF cursor;
```

When a cursor is positioned on a table row, that row can be updated or deleted using the cursor to identify the row. There are restrictions on what the cursor's query can be (in particular, no grouping) and it's best to use FOR UPDATE in the cursor. For more information see the DECLARE reference page.

An example:

```
UPDATE foo SET dataval = myval WHERE CURRENT OF curs1;
```

40.7.3.4. CLOSE

```
CLOSE cursor;
```

CLOSE closes the portal underlying an open cursor. This can be used to release resources earlier than end of transaction, or to free up the cursor variable to be opened again.

An example:

```
CLOSE curs1;
```

40.7.3.5. Returning Cursors

PL/pgSQL functions can return cursors to the caller. This is useful to return multiple rows or columns, especially with very large result sets. To do this, the function opens the cursor and returns the cursor name to the caller (or simply opens the cursor using a portal name specified by or otherwise known to the caller). The caller can then fetch rows from the cursor. The cursor can be closed by the caller, or it will be closed automatically when the transaction closes.

The portal name used for a cursor can be specified by the programmer or automatically generated. To specify a portal name, simply assign a string to the refcursor variable before opening it. The string value of the refcursor variable will be used by OPEN as the name of the underlying portal. However, if the refcursor variable is null, OPEN automatically generates a name that does not conflict with any existing portal, and assigns it to the refcursor variable.

> **Note:** A bound cursor variable is initialized to the string value representing its name, so that the portal name is the same as the cursor variable name, unless the programmer overrides it by assignment before opening the cursor. But an unbound cursor variable defaults to the null value initially, so it will receive an automatically-generated unique name, unless overridden.

The following example shows one way a cursor name can be supplied by the caller:

```
CREATE TABLE test (col text);
INSERT INTO test VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM test;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');
FETCH ALL IN funccursor;
COMMIT;
```

The following example uses automatic cursor name generation:

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM test;
    RETURN ref;
END;
' LANGUAGE plpgsql;

-- need to be in a transaction to use cursors.
BEGIN;
SELECT reffunc2();

      reffunc2
--------------------
 <unnamed cursor 1>
(1 row)

FETCH ALL IN "<unnamed cursor 1>";
COMMIT;
```

The following example shows one way to return multiple cursors from a single function:

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM table_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM table_2;
    RETURN NEXT $2;
END;
$$ LANGUAGE plpgsql;

-- need to be in a transaction to use cursors.
BEGIN;

SELECT * FROM myfunc('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

40.7.4. Looping Through a Cursor's Result

There is a variant of the FOR statement that allows iterating through the rows returned by a cursor. The syntax is:

```
[ <<label>> ]
FOR recordvar IN bound_cursorvar [ ( [ argument_name := ] argument_value [, ...] ) ] LOOP
    statements
END LOOP [ label ];
```

The cursor variable must have been bound to some query when it was declared, and it *cannot* be open already. The FOR statement automatically opens the cursor, and it closes the cursor again when the loop exits. A list of actual argument value expressions must appear if and only if the cursor was declared to take arguments. These values will be substituted in the query, in just the same way as during an OPEN (see Section 40.7.2.3).

The variable *recordvar* is automatically defined as type record and exists only inside the loop (any existing definition of the variable name is ignored within the loop). Each row returned by the cursor is successively assigned to this record variable and the loop body is executed.

40.8. Errors and Messages

40.8.1. Reporting Errors and Messages

Use the RAISE statement to report messages and raise errors.

```
RAISE [ level ] 'format' [, expression [, ... ]] [ USING option = expression [, ... ] ];
RAISE [ level ] condition_name [ USING option = expression [, ... ] ];
RAISE [ level ] SQLSTATE 'sqlstate' [ USING option = expression [, ... ] ];
RAISE [ level ] USING option = expression [, ... ];
RAISE ;
```

The *level* option specifies the error severity. Allowed levels are DEBUG, LOG, INFO, NOTICE, WARNING, and EXCEPTION, with EXCEPTION being the default. EXCEPTION raises an error (which normally aborts the current transaction); the other levels only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the log_min_messages and client_min_messages configuration variables. See Chapter 18 for more information.

After *level* if any, you can write a *format* (which must be a simple string literal, not an expression). The format string specifies the error message text to be reported. The format string can be followed by optional argument expressions to be inserted into the message. Inside the format string, % is replaced by the string representation of the next optional argument's value. Write %% to emit a literal %. The number of arguments must match the number of % placeholders in the format string, or an error is raised during the compilation of the function.

In this example, the value of v_job_id will replace the % in the string:

```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

You can attach additional information to the error report by writing `USING` followed by *`option`* = *`expression`* items. Each *`expression`* can be any string-valued expression. The allowed *`option`* key words are:

`MESSAGE`

Sets the error message text. This option can't be used in the form of `RAISE` that includes a format string before `USING`.

`DETAIL`

Supplies an error detail message.

`HINT`

Supplies a hint message.

`ERRCODE`

Specifies the error code (SQLSTATE) to report, either by condition name, as shown in Appendix A, or directly as a five-character SQLSTATE code.

`COLUMN`
`CONSTRAINT`
`DATATYPE`
`TABLE`
`SCHEMA`

Supplies the name of a related object.

This example will abort the transaction with the given error message and hint:

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id
      USING HINT = 'Please check your user ID';
```

These two examples show equivalent ways of setting the SQLSTATE:

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

There is a second `RAISE` syntax in which the main argument is the condition name or SQLSTATE to be reported, for example:

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

In this syntax, `USING` can be used to supply a custom error message, detail, or hint. Another way to do the earlier example is

```
RAISE unique_violation USING MESSAGE = 'Duplicate user ID: ' || user_id;
```

Still another variant is to write RAISE USING or RAISE *level* USING and put everything else into the USING list.

The last variant of RAISE has no parameters at all. This form can only be used inside a BEGIN block's EXCEPTION clause; it causes the error currently being handled to be re-thrown.

> **Note:** Before PostgreSQL 9.1, RAISE without parameters was interpreted as re-throwing the error from the block containing the active exception handler. Thus an EXCEPTION clause nested within that handler could not catch it, even if the RAISE was within the nested EXCEPTION clause's block. This was deemed surprising as well as being incompatible with Oracle's PL/SQL.

If no condition name nor SQLSTATE is specified in a RAISE EXCEPTION command, the default is to use ERRCODE_RAISE_EXCEPTION (P0001). If no message text is specified, the default is to use the condition name or SQLSTATE as message text.

> **Note:** When specifying an error code by SQLSTATE code, you are not limited to the predefined error codes, but can select any error code consisting of five digits and/or upper-case ASCII letters, other than 00000. It is recommended that you avoid throwing error codes that end in three zeroes, because these are category codes and can only be trapped by trapping the whole category.

40.8.2. Checking Assertions

The ASSERT statement is a convenient shorthand for inserting debugging checks into PL/pgSQL functions.

```
ASSERT condition [ , message ];
```

The *condition* is a Boolean expression that is expected to always evaluate to true; if it does, the ASSERT statement does nothing further. If the result is false or null, then an ASSERT_FAILURE exception is raised. (If an error occurs while evaluating the *condition*, it is reported as a normal error.)

If the optional *message* is provided, it is an expression whose result (if not null) replaces the default error message text "assertion failed", should the *condition* fail. The *message* expression is not evaluated in the normal case where the assertion succeeds.

Testing of assertions can be enabled or disabled via the configuration parameter plpgsql.check_asserts, which takes a Boolean value; the default is on. If this parameter is off then ASSERT statements do nothing.

Note that ASSERT is meant for detecting program bugs, not for reporting ordinary error conditions. Use the RAISE statement, described above, for that.

40.9. Trigger Procedures

40.9.1. Triggers on Data Changes

PL/pgSQL can be used to define trigger procedures. A trigger procedure is created with the `CREATE FUNCTION` command, declaring it as a function with no arguments and a return type of `trigger`. Note that the function must be declared with no arguments even if it expects to receive arguments specified in `CREATE TRIGGER` — trigger arguments are passed via `TG_ARGV`, as described below.

When a PL/pgSQL function is called as a trigger, several special variables are created automatically in the top-level block. They are:

`NEW`

Data type `RECORD`; variable holding the new database row for `INSERT`/`UPDATE` operations in row-level triggers. This variable is unassigned in statement-level triggers and for `DELETE` operations.

`OLD`

Data type `RECORD`; variable holding the old database row for `UPDATE`/`DELETE` operations in row-level triggers. This variable is unassigned in statement-level triggers and for `INSERT` operations.

`TG_NAME`

Data type `name`; variable that contains the name of the trigger actually fired.

`TG_WHEN`

Data type `text`; a string of `BEFORE`, `AFTER`, or `INSTEAD OF`, depending on the trigger's definition.

`TG_LEVEL`

Data type `text`; a string of either `ROW` or `STATEMENT` depending on the trigger's definition.

`TG_OP`

Data type `text`; a string of `INSERT`, `UPDATE`, `DELETE`, or `TRUNCATE` telling for which operation the trigger was fired.

`TG_RELID`

Data type `oid`; the object ID of the table that caused the trigger invocation.

`TG_RELNAME`

Data type `name`; the name of the table that caused the trigger invocation. This is now deprecated, and could disappear in a future release. Use `TG_TABLE_NAME` instead.

`TG_TABLE_NAME`

Data type `name`; the name of the table that caused the trigger invocation.

`TG_TABLE_SCHEMA`

Data type `name`; the name of the schema of the table that caused the trigger invocation.

`TG_NARGS`

Data type `integer`; the number of arguments given to the trigger procedure in the `CREATE TRIGGER` statement.

`TG_ARGV[]`

Data type array of `text`; the arguments from the `CREATE TRIGGER` statement. The index counts from 0. Invalid indexes (less than 0 or greater than or equal to `tg_nargs`) result in a null value.

A trigger function must return either `NULL` or a record/row value having exactly the structure of the table the trigger was fired for.

Row-level triggers fired `BEFORE` can return null to signal the trigger manager to skip the rest of the operation for this row (i.e., subsequent triggers are not fired, and the `INSERT`/`UPDATE`/`DELETE` does not occur for this row). If a nonnull value is returned then the operation proceeds with that row value. Returning a row value different from the original value of `NEW` alters the row that will be inserted or updated. Thus, if the trigger function wants the triggering action to succeed normally without altering the row value, `NEW` (or a value equal thereto) has to be returned. To alter the row to be stored, it is possible to replace single values directly in `NEW` and return the modified `NEW`, or to build a complete new record/row to return. In the case of a before-trigger on `DELETE`, the returned value has no direct effect, but it has to be nonnull to allow the trigger action to proceed. Note that `NEW` is null in `DELETE` triggers, so returning that is usually not sensible. The usual idiom in `DELETE` triggers is to return `OLD`.

`INSTEAD OF` triggers (which are always row-level triggers, and may only be used on views) can return null to signal that they did not perform any updates, and that the rest of the operation for this row should be skipped (i.e., subsequent triggers are not fired, and the row is not counted in the rows-affected status for the surrounding `INSERT`/`UPDATE`/`DELETE`). Otherwise a nonnull value should be returned, to signal that the trigger performed the requested operation. For `INSERT` and `UPDATE` operations, the return value should be `NEW`, which the trigger function may modify to support `INSERT RETURNING` and `UPDATE RETURNING` (this will also affect the row value passed to any subsequent triggers, or passed to a special `EXCLUDED` alias reference within an `INSERT` statement with an `ON CONFLICT DO UPDATE` clause). For `DELETE` operations, the return value should be `OLD`.

The return value of a row-level trigger fired `AFTER` or a statement-level trigger fired `BEFORE` or `AFTER` is always ignored; it might as well be null. However, any of these types of triggers might still abort the entire operation by raising an error.

Example 40-3 shows an example of a trigger procedure in PL/pgSQL.

Example 40-3. A PL/pgSQL Trigger Procedure

This example trigger ensures that any time a row is inserted or updated in the table, the current user name and time are stamped into the row. And it checks that an employee's name is given and that the salary is a positive value.

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname IS NULL THEN
```

```
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

        -- Who works for us when she must pay for it?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
$emp_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Another way to log changes to a table involves creating a new table that holds a row for each insert, update, or delete that occurs. This approach can be thought of as auditing changes to a table. Example 40-4 shows an example of an audit trigger procedure in PL/pgSQL.

Example 40-4. A PL/pgSQL Trigger Procedure For Auditing

This example trigger ensures that any insert, update or delete of a row in the `emp` table is recorded (i.e., audited) in the `emp_audit` table. The current time and user name are stamped into the row, together with the type of operation performed on it.

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Create a row in emp_audit to reflect the operation performed on emp,
        -- make use of the special variable TG_OP to work out the operation.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
            RETURN OLD;
```

```
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
            RETURN NEW;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER trigger
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```

A variation of the previous example uses a view joining the main table to the audit table, to show when each entry was last modified. This approach still records the full audit trail of changes to the table, but also presents a simplified view of the audit trail, showing just the last modified timestamp derived from the audit trail for each entry. Example 40-5 shows an example of an audit trigger on a view in PL/pgSQL.

Example 40-5. A PL/pgSQL View Trigger Procedure For Auditing

This example uses a trigger on the view to make it updatable, and ensure that any insert, update or delete of a row in the view is recorded (i.e., audited) in the `emp_audit` table. The current time and user name are recorded, together with the type of operation performed, and the view displays the last modified time of each row.

```
CREATE TABLE emp (
    empname           text PRIMARY KEY,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary            integer,
    stamp             timestamp NOT NULL
);

CREATE VIEW emp_view AS
    SELECT e.empname,
           e.salary,
           max(ea.stamp) AS last_updated
      FROM emp e
      LEFT JOIN emp_audit ea ON ea.empname = e.empname
     GROUP BY 1, 2;

CREATE OR REPLACE FUNCTION update_emp_view() RETURNS TRIGGER AS $$
    BEGIN
        --
        -- Perform the required operation on emp, and create a row in emp_audit
        -- to reflect the change made to emp.
```

```
        --
        IF (TG_OP = 'DELETE') THEN
            DELETE FROM emp WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            OLD.last_updated = now();
            INSERT INTO emp_audit VALUES('D', user, OLD.*);
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            UPDATE emp SET salary = NEW.salary WHERE empname = OLD.empname;
            IF NOT FOUND THEN RETURN NULL; END IF;

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('U', user, NEW.*);
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp VALUES(NEW.empname, NEW.salary);

            NEW.last_updated = now();
            INSERT INTO emp_audit VALUES('I', user, NEW.*);
            RETURN NEW;
        END IF;
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_view
    FOR EACH ROW EXECUTE PROCEDURE update_emp_view();
```

One use of triggers is to maintain a summary table of another table. The resulting summary can be used in place of the original table for certain queries — often with vastly reduced run times. This technique is commonly used in Data Warehousing, where the tables of measured or observed data (called fact tables) might be extremely large. Example 40-6 shows an example of a trigger procedure in PL/pgSQL that maintains a summary table for a fact table in a data warehouse.

Example 40-6. A PL/pgSQL Trigger Procedure For Maintaining A Summary Table

The schema detailed here is partly based on the *Grocery Store* example from *The Data Warehouse Toolkit* by Ralph Kimball.

```
--
-- Main tables - time dimension and sales fact.
--
CREATE TABLE time_dimension (
    time_key                    integer NOT NULL,
    day_of_week                 integer NOT NULL,
    day_of_month                integer NOT NULL,
    month                       integer NOT NULL,
    quarter                     integer NOT NULL,
    year                        integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);
```

```
CREATE TABLE sales_fact (
    time_key                    integer NOT NULL,
    product_key                 integer NOT NULL,
    store_key                   integer NOT NULL,
    amount_sold                 numeric(12,2) NOT NULL,
    units_sold                  integer NOT NULL,
    amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- Summary table - sales by time.
--
CREATE TABLE sales_summary_bytime (
    time_key                    integer NOT NULL,
    amount_sold                 numeric(15,2) NOT NULL,
    units_sold                  numeric(12) NOT NULL,
    amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- Function and trigger to amend summarized column(s) on UPDATE, INSERT, DELETE.
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER
AS $maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
    BEGIN

        -- Work out the increment/decrement amount(s).
        IF (TG_OP = 'DELETE') THEN

            delta_time_key = OLD.time_key;
            delta_amount_sold = -1 * OLD.amount_sold;
            delta_units_sold = -1 * OLD.units_sold;
            delta_amount_cost = -1 * OLD.amount_cost;

        ELSIF (TG_OP = 'UPDATE') THEN

            -- forbid updates that change the time_key -
            -- (probably not too onerous, as DELETE + INSERT is how most
            -- changes will be made).
            IF ( OLD.time_key != NEW.time_key) THEN
                RAISE EXCEPTION 'Update of time_key : % -> % not allowed',
                                                    OLD.time_key, NEW.time_key;
            END IF;

            delta_time_key = OLD.time_key;
            delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
```

```
            delta_units_sold = NEW.units_sold - OLD.units_sold;
            delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

        ELSIF (TG_OP = 'INSERT') THEN

            delta_time_key = NEW.time_key;
            delta_amount_sold = NEW.amount_sold;
            delta_units_sold = NEW.units_sold;
            delta_amount_cost = NEW.amount_cost;

        END IF;


        -- Insert or update the summary row with the new values.
        <<insert_update>>
        LOOP
            UPDATE sales_summary_bytime
                SET amount_sold = amount_sold + delta_amount_sold,
                    units_sold = units_sold + delta_units_sold,
                    amount_cost = amount_cost + delta_amount_cost
                WHERE time_key = delta_time_key;

            EXIT insert_update WHEN found;

            BEGIN
                INSERT INTO sales_summary_bytime (
                            time_key,
                            amount_sold,
                            units_sold,
                            amount_cost)
                    VALUES (
                            delta_time_key,
                            delta_amount_sold,
                            delta_units_sold,
                            delta_amount_cost
                           );

                EXIT insert_update;

            EXCEPTION
                WHEN UNIQUE_VIOLATION THEN
                    -- do nothing
            END;
        END LOOP insert_update;

        RETURN NULL;

    END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
    FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();
```

```
INSERT INTO sales_fact VALUES(1,1,1,10,3,15);
INSERT INTO sales_fact VALUES(1,2,1,20,5,35);
INSERT INTO sales_fact VALUES(2,2,1,40,15,135);
INSERT INTO sales_fact VALUES(2,3,1,10,1,13);
SELECT * FROM sales_summary_bytime;
DELETE FROM sales_fact WHERE product_key = 1;
SELECT * FROM sales_summary_bytime;
UPDATE sales_fact SET units_sold = units_sold * 2;
SELECT * FROM sales_summary_bytime;
```

40.9.2. Triggers on Events

PL/pgSQL can be used to define event triggers. PostgreSQL requires that a procedure that is to be called as an event trigger must be declared as a function with no arguments and a return type of `event_trigger`.

When a PL/pgSQL function is called as an event trigger, several special variables are created automatically in the top-level block. They are:

`TG_EVENT`

Data type `text`; a string representing the event the trigger is fired for.

`TG_TAG`

Data type `text`; variable that contains the command tag for which the trigger is fired.

Example 40-7 shows an example of an event trigger procedure in PL/pgSQL.

Example 40-7. A PL/pgSQL Event Trigger Procedure

This example trigger simply raises a `NOTICE` message each time a supported command is executed.

```
CREATE OR REPLACE FUNCTION snitch() RETURNS event_trigger AS $$
BEGIN
    RAISE NOTICE 'snitch: % %', tg_event, tg_tag;
END;
$$ LANGUAGE plpgsql;

CREATE EVENT TRIGGER snitch ON ddl_command_start EXECUTE PROCEDURE snitch();
```

40.10. PL/pgSQL Under the Hood

This section discusses some implementation details that are frequently important for PL/pgSQL users to know.

40.10.1. Variable Substitution

SQL statements and expressions within a PL/pgSQL function can refer to variables and parameters of the function. Behind the scenes, PL/pgSQL substitutes query parameters for such references. Parameters will only be substituted in places where a parameter or column reference is syntactically allowed. As an extreme case, consider this example of poor programming style:

```
INSERT INTO foo (foo) VALUES (foo);
```

The first occurrence of `foo` must syntactically be a table name, so it will not be substituted, even if the function has a variable named `foo`. The second occurrence must be the name of a column of the table, so it will not be substituted either. Only the third occurrence is a candidate to be a reference to the function's variable.

> **Note:** PostgreSQL versions before 9.0 would try to substitute the variable in all three cases, leading to syntax errors.

Since the names of variables are syntactically no different from the names of table columns, there can be ambiguity in statements that also refer to tables: is a given name meant to refer to a table column, or a variable? Let's change the previous example to

```
INSERT INTO dest (col) SELECT foo + bar FROM src;
```

Here, `dest` and `src` must be table names, and `col` must be a column of `dest`, but `foo` and `bar` might reasonably be either variables of the function or columns of `src`.

By default, PL/pgSQL will report an error if a name in a SQL statement could refer to either a variable or a table column. You can fix such a problem by renaming the variable or column, or by qualifying the ambiguous reference, or by telling PL/pgSQL which interpretation to prefer.

The simplest solution is to rename the variable or column. A common coding rule is to use a different naming convention for PL/pgSQL variables than you use for column names. For example, if you consistently name function variables `v_`*`something`* while none of your column names start with `v_`, no conflicts will occur.

Alternatively you can qualify ambiguous references to make them clear. In the above example, `src.foo` would be an unambiguous reference to the table column. To create an unambiguous reference to a variable, declare it in a labeled block and use the block's label (see Section 40.2). For example,

```
<<block>>
DECLARE
    foo int;
BEGIN
    foo := ...;
    INSERT INTO dest (col) SELECT block.foo + bar FROM src;
```

Here `block.foo` means the variable even if there is a column `foo` in `src`. Function parameters, as well as special variables such as `FOUND`, can be qualified by the function's name, because they are implicitly declared in an outer block labeled with the function's name.

Sometimes it is impractical to fix all the ambiguous references in a large body of PL/pgSQL code. In such cases you can specify that PL/pgSQL should resolve ambiguous references as the variable (which is com-

patible with PL/pgSQL's behavior before PostgreSQL 9.0), or as the table column (which is compatible with some other systems such as Oracle).

To change this behavior on a system-wide basis, set the configuration parameter `plpgsql.variable_conflict` to one of `error`, `use_variable`, or `use_column` (where `error` is the factory default). This parameter affects subsequent compilations of statements in PL/pgSQL functions, but not statements already compiled in the current session. Because changing this setting can cause unexpected changes in the behavior of PL/pgSQL functions, it can only be changed by a superuser.

You can also set the behavior on a function-by-function basis, by inserting one of these special commands at the start of the function text:

```
#variable_conflict error
#variable_conflict use_variable
#variable_conflict use_column
```

These commands affect only the function they are written in, and override the setting of `plpgsql.variable_conflict`. An example is

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    #variable_conflict use_variable
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = curtime, comment = comment
          WHERE users.id = id;
    END;
$$ LANGUAGE plpgsql;
```

In the `UPDATE` command, `curtime`, `comment`, and `id` will refer to the function's variable and parameters whether or not `users` has columns of those names. Notice that we had to qualify the reference to `users.id` in the `WHERE` clause to make it refer to the table column. But we did not have to qualify the reference to `comment` as a target in the `UPDATE` list, because syntactically that must be a column of `users`. We could write the same function without depending on the `variable_conflict` setting in this way:

```
CREATE FUNCTION stamp_user(id int, comment text) RETURNS void AS $$
    <<fn>>
    DECLARE
        curtime timestamp := now();
    BEGIN
        UPDATE users SET last_modified = fn.curtime, comment = stamp_user.comment
          WHERE users.id = stamp_user.id;
    END;
$$ LANGUAGE plpgsql;
```

Variable substitution does not happen in the command string given to `EXECUTE` or one of its variants. If you need to insert a varying value into such a command, do so as part of constructing the string value, or use `USING`, as illustrated in Section 40.5.4.

Variable substitution currently works only in `SELECT`, `INSERT`, `UPDATE`, and `DELETE` commands, because the main SQL engine allows query parameters only in these commands. To use a non-constant name

or value in other statement types (generically called utility statements), you must construct the utility statement as a string and `EXECUTE` it.

40.10.2. Plan Caching

The PL/pgSQL interpreter parses the function's source text and produces an internal binary instruction tree the first time the function is called (within each session). The instruction tree fully translates the PL/pgSQL statement structure, but individual SQL expressions and SQL commands used in the function are not translated immediately.

As each expression and SQL command is first executed in the function, the PL/pgSQL interpreter parses and analyzes the command to create a prepared statement, using the SPI manager's `SPI_prepare` function. Subsequent visits to that expression or command reuse the prepared statement. Thus, a function with conditional code paths that are seldom visited will never incur the overhead of analyzing those commands that are never executed within the current session. A disadvantage is that errors in a specific expression or command cannot be detected until that part of the function is reached in execution. (Trivial syntax errors will be detected during the initial parsing pass, but anything deeper will not be detected until execution.)

PL/pgSQL (or more precisely, the SPI manager) can furthermore attempt to cache the execution plan associated with any particular prepared statement. If a cached plan is not used, then a fresh execution plan is generated on each visit to the statement, and the current parameter values (that is, PL/pgSQL variable values) can be used to optimize the selected plan. If the statement has no parameters, or is executed many times, the SPI manager will consider creating a *generic* plan that is not dependent on specific parameter values, and caching that for re-use. Typically this will happen only if the execution plan is not very sensitive to the values of the PL/pgSQL variables referenced in it. If it is, generating a plan each time is a net win. See PREPARE for more information about the behavior of prepared statements.

Because PL/pgSQL saves prepared statements and sometimes execution plans in this way, SQL commands that appear directly in a PL/pgSQL function must refer to the same tables and columns on every execution; that is, you cannot use a parameter as the name of a table or column in an SQL command. To get around this restriction, you can construct dynamic commands using the PL/pgSQL `EXECUTE` statement — at the price of performing new parse analysis and constructing a new execution plan on every execution.

The mutable nature of record variables presents another problem in this connection. When fields of a record variable are used in expressions or statements, the data types of the fields must not change from one call of the function to the next, since each expression will be analyzed using the data type that is present when the expression is first reached. `EXECUTE` can be used to get around this problem when necessary.

If the same function is used as a trigger for more than one table, PL/pgSQL prepares and caches statements independently for each such table — that is, there is a cache for each trigger function and table combination, not just for each function. This alleviates some of the problems with varying data types; for instance, a trigger function will be able to work successfully with a column named `key` even if it happens to have different types in different tables.

Likewise, functions having polymorphic argument types have a separate statement cache for each combination of actual argument types they have been invoked for, so that data type differences do not cause unexpected failures.

Statement caching can sometimes have surprising effects on the interpretation of time-sensitive values. For example there is a difference between what these two functions do:

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS void AS $$
    BEGIN
        INSERT INTO logtable VALUES (logtxt, 'now');
    END;
$$ LANGUAGE plpgsql;
```

and:

```
CREATE FUNCTION logfunc2(logtxt text) RETURNS void AS $$
    DECLARE
        curtime timestamp;
    BEGIN
        curtime := 'now';
        INSERT INTO logtable VALUES (logtxt, curtime);
    END;
$$ LANGUAGE plpgsql;
```

In the case of `logfunc1`, the PostgreSQL main parser knows when analyzing the `INSERT` that the string `'now'` should be interpreted as `timestamp`, because the target column of `logtable` is of that type. Thus, `'now'` will be converted to a `timestamp` constant when the `INSERT` is analyzed, and then used in all invocations of `logfunc1` during the lifetime of the session. Needless to say, this isn't what the programmer wanted. A better idea is to use the `now()` or `current_timestamp` function.

In the case of `logfunc2`, the PostgreSQL main parser does not know what type `'now'` should become and therefore it returns a data value of type `text` containing the string `now`. During the ensuing assignment to the local variable `curtime`, the PL/pgSQL interpreter casts this string to the `timestamp` type by calling the `textout` and `timestamp_in` functions for the conversion. So, the computed time stamp is updated on each execution as the programmer expects. Even though this happens to work as expected, it's not terribly efficient, so use of the `now()` function would still be a better idea.

40.11. Tips for Developing in PL/pgSQL

One good way to develop in PL/pgSQL is to use the text editor of your choice to create your functions, and in another window, use psql to load and test those functions. If you are doing it this way, it is a good idea to write the function using `CREATE OR REPLACE FUNCTION`. That way you can just reload the file to update the function definition. For example:

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $$
          ....
$$ LANGUAGE plpgsql;
```

While running psql, you can load or reload such a function definition file with:

```
\i filename.sql
```

and then immediately issue SQL commands to test the function.

Another good way to develop in PL/pgSQL is with a GUI database access tool that facilitates development in a procedural language. One example of such a tool is pgAdmin, although others exist. These tools often provide convenient features such as escaping single quotes and making it easier to recreate and debug functions.

40.11.1. Handling of Quotation Marks

The code of a PL/pgSQL function is specified in `CREATE FUNCTION` as a string literal. If you write the string literal in the ordinary way with surrounding single quotes, then any single quotes inside the function body must be doubled; likewise any backslashes must be doubled (assuming escape string syntax is used). Doubling quotes is at best tedious, and in more complicated cases the code can become downright incomprehensible, because you can easily find yourself needing half a dozen or more adjacent quote marks. It's recommended that you instead write the function body as a "dollar-quoted" string literal (see Section 4.1.2.4). In the dollar-quoting approach, you never double any quote marks, but instead take care to choose a different dollar-quoting delimiter for each level of nesting you need. For example, you might write the `CREATE FUNCTION` command as:

```
CREATE OR REPLACE FUNCTION testfunc(integer) RETURNS integer AS $PROC$
          ....
$PROC$ LANGUAGE plpgsql;
```

Within this, you might use quote marks for simple literal strings in SQL commands and `$$` to delimit fragments of SQL commands that you are assembling as strings. If you need to quote text that includes `$$`, you could use `$Q$`, and so on.

The following chart shows what you have to do when writing quote marks without dollar quoting. It might be useful when translating pre-dollar quoting code into something more comprehensible.

1 quotation mark

To begin and end the function body, for example:

```
CREATE FUNCTION foo() RETURNS integer AS '
           ....
' LANGUAGE plpgsql;
```

Anywhere within a single-quoted function body, quote marks *must* appear in pairs.

2 quotation marks

For string literals inside the function body, for example:

```
a_output := ”Blah”;
SELECT * FROM users WHERE f_name=”foobar”;
```

In the dollar-quoting approach, you'd just write:

```
a_output := 'Blah';
SELECT * FROM users WHERE f_name='foobar';
```

which is exactly what the PL/pgSQL parser would see in either case.

4 quotation marks

When you need a single quotation mark in a string constant inside the function body, for example:

```
a_output := a_output || ” AND name LIKE ””foobar”” AND xyz”
```

The value actually appended to `a_output` would be: `AND name LIKE 'foobar' AND xyz`.

In the dollar-quoting approach, you'd write:

```
a_output := a_output || $$ AND name LIKE 'foobar' AND xyz$$
```

being careful that any dollar-quote delimiters around this are not just `$$`.

6 quotation marks

When a single quotation mark in a string inside the function body is adjacent to the end of that string constant, for example:

```
a_output := a_output || '' AND name LIKE ''foobar''''
```

The value appended to `a_output` would then be: `AND name LIKE 'foobar'`.

In the dollar-quoting approach, this becomes:

```
a_output := a_output || $$ AND name LIKE 'foobar'$$
```

10 quotation marks

When you want two single quotation marks in a string constant (which accounts for 8 quotation marks) and this is adjacent to the end of that string constant (2 more). You will probably only need that if you are writing a function that generates other functions, as in Example 40-9. For example:

```
a_output := a_output || '' if v_'' ||
    referrer_keys.kind || '' like ''''''
    || referrer_keys.key_string || ''''''
    then return ''''  || referrer_keys.referrer_type
    || ''''; end if;'';
```

The value of `a_output` would then be:

```
if v_... like ''...'' then return ''...''; end if;
```

In the dollar-quoting approach, this becomes:

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
    || referrer_keys.key_string || $$'
    then return '$$  || referrer_keys.referrer_type
    || $$'; end if;$$;
```

where we assume we only need to put single quote marks into `a_output`, because it will be re-quoted before use.

40.11.2. Additional Compile-time Checks

To aid the user in finding instances of simple but common problems before they cause harm, PL/PgSQL provides additional *checks*. When enabled, depending on the configuration, they can be used to emit either a `WARNING` or an `ERROR` during the compilation of a function. A function which has received a `WARNING` can be executed without producing further messages, so you are advised to test in a separate development environment.

These additional checks are enabled through the configuration variables `plpgsql.extra_warnings` for warnings and `plpgsql.extra_errors` for errors. Both can be set either to a comma-separated list of checks, `"none"` or `"all"`. The default is `"none"`. Currently the list of available checks includes only one:

`shadowed_variables`

Checks if a declaration shadows a previously defined variable.

The following example shows the effect of `plpgsql.extra_warnings` set to `shadowed_variables`:

```
SET plpgsql.extra_warnings TO 'shadowed_variables';

CREATE FUNCTION foo(f1 int) RETURNS int AS $$
DECLARE
f1 int;
BEGIN
RETURN f1;
END;
$$ LANGUAGE plpgsql;
WARNING:  variable "f1" shadows a previously defined variable
LINE 3: f1 int;
        ^
CREATE FUNCTION
```

40.12. Porting from Oracle PL/SQL

This section explains differences between PostgreSQL's PL/pgSQL language and Oracle's PL/SQL language, to help developers who port applications from Oracle® to PostgreSQL.

PL/pgSQL is similar to PL/SQL in many aspects. It is a block-structured, imperative language, and all variables have to be declared. Assignments, loops, and conditionals are similar. The main differences you should keep in mind when porting from PL/SQL to PL/pgSQL are:

- If a name used in a SQL command could be either a column name of a table or a reference to a variable of the function, PL/SQL treats it as a column name. This corresponds to PL/pgSQL's `plpgsql.variable_conflict = use_column` behavior, which is not the default, as explained in Section 40.10.1. It's often best to avoid such ambiguities in the first place, but if you have to port a large amount of code that depends on this behavior, setting `variable_conflict` may be the best solution.
- In PostgreSQL the function body must be written as a string literal. Therefore you need to use dollar quoting or escape single quotes in the function body. (See Section 40.11.1.)
- Instead of packages, use schemas to organize your functions into groups.
- Since there are no packages, there are no package-level variables either. This is somewhat annoying. You can keep per-session state in temporary tables instead.
- Integer `FOR` loops with `REVERSE` work differently: PL/SQL counts down from the second number to the first, while PL/pgSQL counts down from the first number to the second, requiring the loop bounds to be swapped when porting. This incompatibility is unfortunate but is unlikely to be changed. (See Section 40.6.3.5.)

- `FOR` loops over queries (other than cursors) also work differently: the target variable(s) must have been declared, whereas PL/SQL always declares them implicitly. An advantage of this is that the variable values are still accessible after the loop exits.
- There are various notational differences for the use of cursor variables.

40.12.1. Porting Examples

Example 40-8 shows how to port a simple function from PL/SQL to PL/pgSQL.

Example 40-8. Porting a Simple Function from PL/SQL to PL/pgSQL

Here is an Oracle PL/SQL function:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                  v_version varchar)
RETURN varchar IS
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
/
show errors;
```

Let's go through this function and see the differences compared to PL/pgSQL:

- The `RETURN` key word in the function prototype (not the function body) becomes `RETURNS` in PostgreSQL. Also, `IS` becomes `AS`, and you need to add a `LANGUAGE` clause because PL/pgSQL is not the only possible function language.
- In PostgreSQL, the function body is considered to be a string literal, so you need to use quote marks or dollar quotes around it. This substitutes for the terminating `/` in the Oracle approach.
- The `show errors` command does not exist in PostgreSQL, and is not needed since errors are reported automatically.

This is how this function would look when ported to PostgreSQL:

```
CREATE OR REPLACE FUNCTION cs_fmt_browser_version(v_name varchar,
                                                  v_version varchar)
RETURNS varchar AS $$
BEGIN
    IF v_version IS NULL THEN
        RETURN v_name;
    END IF;
    RETURN v_name || '/' || v_version;
END;
$$ LANGUAGE plpgsql;
```

Example 40-9 shows how to port a function that creates another function and how to handle the ensuing quoting problems.

Example 40-9. Porting a Function that Creates Another Function from PL/SQL to PL/pgSQL

The following procedure grabs rows from a `SELECT` statement and builds a large function with the results in `IF` statements, for the sake of efficiency.

This is the Oracle version:

```
CREATE OR REPLACE PROCEDURE cs_update_referrer_type_proc IS
    CURSOR referrer_keys IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_cmd VARCHAR(4000);
BEGIN
    func_cmd := 'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host IN VARCHAR,
                 v_domain IN VARCHAR, v_url IN VARCHAR) RETURN VARCHAR IS BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_cmd := func_cmd ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ''' || referrer_key.key_string
          || ''' THEN RETURN ''' || referrer_key.referrer_type
          || '''; END IF;';
    END LOOP;

    func_cmd := func_cmd || ' RETURN NULL; END;';

    EXECUTE IMMEDIATE func_cmd;
END;
/
show errors;
```

Here is how this function would end up in PostgreSQL:

```
CREATE OR REPLACE FUNCTION cs_update_referrer_type_proc() RETURNS void AS $func$
DECLARE
    referrer_keys CURSOR IS
        SELECT * FROM cs_referrer_keys
        ORDER BY try_order;
    func_body text;
    func_cmd text;
BEGIN
    func_body := 'BEGIN';

    FOR referrer_key IN referrer_keys LOOP
        func_body := func_body ||
          ' IF v_' || referrer_key.kind
          || ' LIKE ' || quote_literal(referrer_key.key_string)
          || ' THEN RETURN ' || quote_literal(referrer_key.referrer_type)
          || '; END IF;' ;
    END LOOP;
```

```
    func_body := func_body || ' RETURN NULL; END;';

    func_cmd :=
      'CREATE OR REPLACE FUNCTION cs_find_referrer_type(v_host varchar,
                                                        v_domain varchar,
                                                        v_url varchar)
        RETURNS varchar AS '
      || quote_literal(func_body)
      || ' LANGUAGE plpgsql;' ;

    EXECUTE func_cmd;
END;
$func$ LANGUAGE plpgsql;
```

Notice how the body of the function is built separately and passed through `quote_literal` to double any quote marks in it. This technique is needed because we cannot safely use dollar quoting for defining the new function: we do not know for sure what strings will be interpolated from the `referrer_key.key_string` field. (We are assuming here that `referrer_key.kind` can be trusted to always be `host`, `domain`, or `url`, but `referrer_key.key_string` might be anything, in particular it might contain dollar signs.) This function is actually an improvement on the Oracle original, because it will not generate broken code when `referrer_key.key_string` or `referrer_key.referrer_type` contain quote marks.

Example 40-10 shows how to port a function with `OUT` parameters and string manipulation. PostgreSQL does not have a built-in `instr` function, but you can create one using a combination of other functions. In Section 40.12.3 there is a PL/pgSQL implementation of `instr` that you can use to make your porting easier.

Example 40-10. Porting a Procedure With String Manipulation and `OUT` Parameters from PL/SQL to PL/pgSQL

The following Oracle PL/SQL procedure is used to parse a URL and return several elements (host, path, and query).

This is the Oracle version:

```
CREATE OR REPLACE PROCEDURE cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- This will be passed back
    v_path OUT VARCHAR,  -- This one too
    v_query OUT VARCHAR) -- And this one
IS
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
```

```
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
/
show errors;
```

Here is a possible translation into PL/pgSQL:

```
CREATE OR REPLACE FUNCTION cs_parse_url(
    v_url IN VARCHAR,
    v_host OUT VARCHAR,  -- This will be passed back
    v_path OUT VARCHAR,  -- This one too
    v_query OUT VARCHAR) -- And this one
AS $$
DECLARE
    a_pos1 INTEGER;
    a_pos2 INTEGER;
BEGIN
    v_host := NULL;
    v_path := NULL;
    v_query := NULL;
    a_pos1 := instr(v_url, '//');

    IF a_pos1 = 0 THEN
        RETURN;
    END IF;
    a_pos2 := instr(v_url, '/', a_pos1 + 2);
    IF a_pos2 = 0 THEN
        v_host := substr(v_url, a_pos1 + 2);
        v_path := '/';
        RETURN;
    END IF;

    v_host := substr(v_url, a_pos1 + 2, a_pos2 - a_pos1 - 2);
    a_pos1 := instr(v_url, '?', a_pos2 + 1);

    IF a_pos1 = 0 THEN
```

```
        v_path := substr(v_url, a_pos2);
        RETURN;
    END IF;

    v_path := substr(v_url, a_pos2, a_pos1 - a_pos2);
    v_query := substr(v_url, a_pos1 + 1);
END;
$$ LANGUAGE plpgsql;
```

This function could be used like this:

```
SELECT * FROM cs_parse_url('http://foobar.com/query.cgi?baz');
```

Example 40-11 shows how to port a procedure that uses numerous features that are specific to Oracle.

Example 40-11. Porting a Procedure from PL/SQL to PL/pgSQL

The Oracle version:

```
CREATE OR REPLACE PROCEDURE cs_create_job(v_job_id IN INTEGER) IS
    a_running_job_count INTEGER;
    PRAGMA AUTONOMOUS_TRANSACTION;❶
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;❷

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        COMMIT; -- free lock❸
        raise_application_error(-20000,
                'Unable to create a new job: a job is currently running.');
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN dup_val_on_index THEN NULL; -- don't worry if it already exists
    END;
    COMMIT;
END;
/
show errors
```

Procedures like this can easily be converted into PostgreSQL functions returning `void`. This procedure in particular is interesting because it can teach us some things:

❶ There is no `PRAGMA` statement in PostgreSQL.

❷ If you do a `LOCK TABLE` in PL/pgSQL, the lock will not be released until the calling transaction is finished.

❸ You cannot issue `COMMIT` in a PL/pgSQL function. The function is running within some outer transaction and so `COMMIT` would imply terminating the function's execution. However, in this particular case it is not necessary anyway, because the lock obtained by the `LOCK TABLE` will be released when we raise an error.

This is how we could port this procedure to PL/pgSQL:

```
CREATE OR REPLACE FUNCTION cs_create_job(v_job_id integer) RETURNS void AS $$
DECLARE
    a_running_job_count integer;
BEGIN
    LOCK TABLE cs_jobs IN EXCLUSIVE MODE;

    SELECT count(*) INTO a_running_job_count FROM cs_jobs WHERE end_stamp IS NULL;

    IF a_running_job_count > 0 THEN
        RAISE EXCEPTION 'Unable to create a new job: a job is currently running';❶
    END IF;

    DELETE FROM cs_active_job;
    INSERT INTO cs_active_job(job_id) VALUES (v_job_id);

    BEGIN
        INSERT INTO cs_jobs (job_id, start_stamp) VALUES (v_job_id, now());
    EXCEPTION
        WHEN unique_violation THEN ❷
            -- don't worry if it already exists
    END;
END;
$$ LANGUAGE plpgsql;
```

❶ The syntax of `RAISE` is considerably different from Oracle's statement, although the basic case `RAISE` *`exception_name`* works similarly.

❷ The exception names supported by PL/pgSQL are different from Oracle's. The set of built-in exception names is much larger (see Appendix A). There is not currently a way to declare user-defined exception names, although you can throw user-chosen SQLSTATE values instead.

The main functional difference between this procedure and the Oracle equivalent is that the exclusive lock on the `cs_jobs` table will be held until the calling transaction completes. Also, if the caller later aborts (for example due to an error), the effects of this procedure will be rolled back.

40.12.2. Other Things to Watch For

This section explains a few other things to watch for when porting Oracle PL/SQL functions to PostgreSQL.

40.12.2.1. Implicit Rollback after Exceptions

In PL/pgSQL, when an exception is caught by an `EXCEPTION` clause, all database changes since the block's `BEGIN` are automatically rolled back. That is, the behavior is equivalent to what you'd get in Oracle with:

```
BEGIN
    SAVEPOINT s1;
    ... code here ...
EXCEPTION
    WHEN ... THEN
        ROLLBACK TO s1;
        ... code here ...
    WHEN ... THEN
        ROLLBACK TO s1;
        ... code here ...
END;
```

If you are translating an Oracle procedure that uses `SAVEPOINT` and `ROLLBACK TO` in this style, your task is easy: just omit the `SAVEPOINT` and `ROLLBACK TO`. If you have a procedure that uses `SAVEPOINT` and `ROLLBACK TO` in a different way then some actual thought will be required.

40.12.2.2. `EXECUTE`

The PL/pgSQL version of `EXECUTE` works similarly to the PL/SQL version, but you have to remember to use `quote_literal` and `quote_ident` as described in Section 40.5.4. Constructs of the type `EXECUTE 'SELECT * FROM $1';` will not work reliably unless you use these functions.

40.12.2.3. Optimizing PL/pgSQL Functions

PostgreSQL gives you two function creation modifiers to optimize execution: "volatility" (whether the function always returns the same result when given the same arguments) and "strictness" (whether the function returns null if any argument is null). Consult the CREATE FUNCTION reference page for details.

When making use of these optimization attributes, your `CREATE FUNCTION` statement might look something like this:

```
CREATE FUNCTION foo(...) RETURNS integer AS $$
...
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

40.12.3. Appendix

This section contains the code for a set of Oracle-compatible `instr` functions that you can use to simplify your porting efforts.

```
--
```

```
-- instr functions that mimic Oracle's counterpart
-- Syntax: instr(string1, string2 [, n [, m]])
-- where [] denotes optional parameters.
--
-- Search string1, beginning at the nth character, for the mth occurrence
-- of string2.  If n is negative, search backwards, starting at the abs(n)'th
-- character from the end of string1.
-- If n is not passed, assume 1 (search starts at first character).
-- If m is not passed, assume 1 (find first occurrence).
-- Returns starting index of string2 in string1, or 0 if string2 is not found.
--

CREATE FUNCTION instr(varchar, varchar) RETURNS integer AS $$
BEGIN
    RETURN instr($1, $2, 1);
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                      beg_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    length integer;
    ss_length integer;
BEGIN
    IF beg_index > 0 THEN
        temp_str := substring(string FROM beg_index);
        pos := position(string_to_search_for IN temp_str);

        IF pos = 0 THEN
            RETURN 0;
        ELSE
            RETURN pos + beg_index - 1;
        END IF;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search_for);
        length := char_length(string);
        beg := length + 1 + beg_index;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            IF string_to_search_for = temp_str THEN
                RETURN beg;
            END IF;

            beg := beg - 1;
        END LOOP;

        RETURN 0;
```

```
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;


CREATE FUNCTION instr(string varchar, string_to_search_for varchar,
                      beg_index integer, occur_index integer)
RETURNS integer AS $$
DECLARE
    pos integer NOT NULL DEFAULT 0;
    occur_number integer NOT NULL DEFAULT 0;
    temp_str varchar;
    beg integer;
    i integer;
    length integer;
    ss_length integer;
BEGIN
    IF occur_index <= 0 THEN
        RAISE 'argument "%" is out of range', occur_index
          USING ERRCODE = '22003';
    END IF;

    IF beg_index > 0 THEN
        beg := beg_index - 1;
        FOR i IN 1..occur_index LOOP
            temp_str := substring(string FROM beg + 1);
            pos := position(string_to_search_for IN temp_str);
            IF pos = 0 THEN
                RETURN 0;
            END IF;
            beg := beg + pos;
        END LOOP;

        RETURN beg;
    ELSIF beg_index < 0 THEN
        ss_length := char_length(string_to_search_for);
        length := char_length(string);
        beg := length + 1 + beg_index;

        WHILE beg > 0 LOOP
            temp_str := substring(string FROM beg FOR ss_length);
            IF string_to_search_for = temp_str THEN
                occur_number := occur_number + 1;
                IF occur_number = occur_index THEN
                    RETURN beg;
                END IF;
            END IF;

            beg := beg - 1;
        END LOOP;
```

```
        RETURN 0;
    ELSE
        RETURN 0;
    END IF;
END;
$$ LANGUAGE plpgsql STRICT IMMUTABLE;
```

Chapter 41. PL/Tcl - Tcl Procedural Language

PL/Tcl is a loadable procedural language for the PostgreSQL database system that enables the Tcl language[1] to be used to write functions and trigger procedures.

41.1. Overview

PL/Tcl offers most of the capabilities a function writer has in the C language, with a few restrictions, and with the addition of the powerful string processing libraries that are available for Tcl.

One compelling *good* restriction is that everything is executed from within the safety of the context of a Tcl interpreter. In addition to the limited command set of safe Tcl, only a few commands are available to access the database via SPI and to raise messages via `elog()`. PL/Tcl provides no way to access internals of the database server or to gain OS-level access under the permissions of the PostgreSQL server process, as a C function can do. Thus, unprivileged database users can be trusted to use this language; it does not give them unlimited authority.

The other notable implementation restriction is that Tcl functions cannot be used to create input/output functions for new data types.

Sometimes it is desirable to write Tcl functions that are not restricted to safe Tcl. For example, one might want a Tcl function that sends email. To handle these cases, there is a variant of PL/Tcl called `PL/TclU` (for untrusted Tcl). This is exactly the same language except that a full Tcl interpreter is used. *If PL/TclU is used, it must be installed as an untrusted procedural language* so that only database superusers can create functions in it. The writer of a PL/TclU function must take care that the function cannot be used to do anything unwanted, since it will be able to do anything that could be done by a user logged in as the database administrator.

The shared object code for the PL/Tcl and PL/TclU call handlers is automatically built and installed in the PostgreSQL library directory if Tcl support is specified in the configuration step of the installation procedure. To install PL/Tcl and/or PL/TclU in a particular database, use the `CREATE EXTENSION` command or the `createlang` program, for example `createlang pltcl` *`dbname`* or `createlang pltclu` *`dbname`*.

41.2. PL/Tcl Functions and Arguments

To create a function in the PL/Tcl language, use the standard CREATE FUNCTION syntax:

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS $$
    # PL/Tcl function body
$$ LANGUAGE pltcl;
```

PL/TclU is the same, except that the language has to be specified as `pltclu`.

The body of the function is simply a piece of Tcl script. When the function is called, the argument values are passed as variables `$1` ... `$n` to the Tcl script. The result is returned from the Tcl code in the usual way, with a `return` statement.

1. https://www.tcl.tk/

For example, a function returning the greater of two integer values could be defined as:

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl STRICT;
```

Note the clause `STRICT`, which saves us from having to think about null input values: if a null value is passed, the function will not be called at all, but will just return a null result automatically.

In a nonstrict function, if the actual value of an argument is null, the corresponding `$n` variable will be set to an empty string. To detect whether a particular argument is null, use the function `argisnull`. For example, suppose that we wanted `tcl_max` with one null and one nonnull argument to return the nonnull argument, rather than null:

```
CREATE FUNCTION tcl_max(integer, integer) RETURNS integer AS $$
    if {[argisnull 1]} {
        if {[argisnull 2]} { return_null }
        return $2
    }
    if {[argisnull 2]} { return $1 }
    if {$1 > $2} {return $1}
    return $2
$$ LANGUAGE pltcl;
```

As shown above, to return a null value from a PL/Tcl function, execute `return_null`. This can be done whether the function is strict or not.

Composite-type arguments are passed to the function as Tcl arrays. The element names of the array are the attribute names of the composite type. If an attribute in the passed row has the null value, it will not appear in the array. Here is an example:

```
CREATE TABLE employee (
    name text,
    salary integer,
    age integer
);

CREATE FUNCTION overpaid(employee) RETURNS boolean AS $$
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
$$ LANGUAGE pltcl;
```

There is currently no support for returning a composite-type result value, nor for returning sets.

PL/Tcl does not currently have full support for domain types: it treats a domain the same as the underlying scalar type. This means that constraints associated with the domain will not be enforced. This is not an issue for function arguments, but it is a hazard if you declare a PL/Tcl function as returning a domain type.

41.3. Data Values in PL/Tcl

The argument values supplied to a PL/Tcl function's code are simply the input arguments converted to text form (just as if they had been displayed by a `SELECT` statement). Conversely, the `return` command will accept any string that is acceptable input format for the function's declared return type. So, within the PL/Tcl function, all values are just text strings.

41.4. Global Data in PL/Tcl

Sometimes it is useful to have some global data that is held between two calls to a function or is shared between different functions. This is easily done in PL/Tcl, but there are some restrictions that must be understood.

For security reasons, PL/Tcl executes functions called by any one SQL role in a separate Tcl interpreter for that role. This prevents accidental or malicious interference by one user with the behavior of another user's PL/Tcl functions. Each such interpreter will have its own values for any "global" Tcl variables. Thus, two PL/Tcl functions will share the same global variables if and only if they are executed by the same SQL role. In an application wherein a single session executes code under multiple SQL roles (via `SECURITY DEFINER` functions, use of `SET ROLE`, etc) you may need to take explicit steps to ensure that PL/Tcl functions can share data. To do that, make sure that functions that should communicate are owned by the same user, and mark them `SECURITY DEFINER`. You must of course take care that such functions can't be used to do anything unintended.

All PL/TclU functions used in a session execute in the same Tcl interpreter, which of course is distinct from the interpreter(s) used for PL/Tcl functions. So global data is automatically shared between PL/TclU functions. This is not considered a security risk because all PL/TclU functions execute at the same trust level, namely that of a database superuser.

To help protect PL/Tcl functions from unintentionally interfering with each other, a global array is made available to each function via the `upvar` command. The global name of this variable is the function's internal name, and the local name is `GD`. It is recommended that `GD` be used for persistent private data of a function. Use regular Tcl global variables only for values that you specifically intend to be shared among multiple functions. (Note that the `GD` arrays are only global within a particular interpreter, so they do not bypass the security restrictions mentioned above.)

An example of using `GD` appears in the `spi_execp` example below.

41.5. Database Access from PL/Tcl

The following commands are available to access the database from the body of a PL/Tcl function:

`spi_exec ?-count` *n*`? ?-array` *name*`?` *command* `?`*loop-body*`?`

Executes an SQL command given as a string. An error in the command causes an error to be raised. Otherwise, the return value of `spi_exec` is the number of rows processed (selected, inserted, updated, or deleted) by the command, or zero if the command is a utility statement. In addition, if the command is a `SELECT` statement, the values of the selected columns are placed in Tcl variables as described below.

The optional `-count` value tells `spi_exec` the maximum number of rows to process in the command. The effect of this is comparable to setting up a query as a cursor and then saying `FETCH` *n*.

If the command is a `SELECT` statement, the values of the result columns are placed into Tcl variables named after the columns. If the `-array` option is given, the column values are instead stored into elements of the named associative array, with the column names used as array indexes. In addition, the current row number within the result (counting from zero) is stored into the array element named "`.tupno`", unless that name is in use as a column name in the result.

If the command is a `SELECT` statement and no *loop-body* script is given, then only the first row of results are stored into Tcl variables or array elements; remaining rows, if any, are ignored. No storing occurs if the query returns no rows. (This case can be detected by checking the result of `spi_exec`.) For example:

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

will set the Tcl variable `$cnt` to the number of rows in the `pg_proc` system catalog.

If the optional *loop-body* argument is given, it is a piece of Tcl script that is executed once for each row in the query result. (*loop-body* is ignored if the given command is not a `SELECT`.) The values of the current row's columns are stored into Tcl variables or array elements before each iteration. For example:

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

will print a log message for every row of `pg_class`. This feature works similarly to other Tcl looping constructs; in particular `continue` and `break` work in the usual way inside the loop body.

If a column of a query result is null, the target variable for it is "unset" rather than being set.

`spi_prepare` *query typelist*

Prepares and saves a query plan for later execution. The saved plan will be retained for the life of the current session.

The query can use parameters, that is, placeholders for values to be supplied whenever the plan is actually executed. In the query string, refer to parameters by the symbols `$1` ... `$`*n*. If the query uses parameters, the names of the parameter types must be given as a Tcl list. (Write an empty list for *typelist* if no parameters are used.)

The return value from `spi_prepare` is a query ID to be used in subsequent calls to `spi_execp`. See `spi_execp` for an example.

`spi_execp ?-count n? ?-array name? ?-nulls string? queryid ?value-list? ?loop-body?`

Executes a query previously prepared with `spi_prepare`. *queryid* is the ID returned by `spi_prepare`. If the query references parameters, a *value-list* must be supplied. This is a Tcl list of actual values for the parameters. The list must be the same length as the parameter type list previously given to `spi_prepare`. Omit *value-list* if the query has no parameters.

The optional value for `-nulls` is a string of spaces and `'n'` characters telling `spi_execp` which of the parameters are null values. If given, it must have exactly the same length as the *value-list*. If it is not given, all the parameter values are nonnull.

Except for the way in which the query and its parameters are specified, `spi_execp` works just like `spi_exec`. The `-count`, `-array`, and *loop-body* options are the same, and so is the result value.

Here's an example of a PL/Tcl function using a prepared plan:

```
CREATE FUNCTION t1_count(integer, integer) RETURNS integer AS $$
    if {![ info exists GD(plan) ]} {
        # prepare the saved plan on the first call
        set GD(plan) [ spi_prepare \
                "SELECT count(*) AS cnt FROM t1 WHERE num >= \$1 AND num <= \$2" \
                [ list int4 int4 ] ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
$$ LANGUAGE pltcl;
```

We need backslashes inside the query string given to `spi_prepare` to ensure that the `$n` markers will be passed through to `spi_prepare` as-is, and not replaced by Tcl variable substitution.

`spi_lastoid`

Returns the OID of the row inserted by the last `spi_exec` or `spi_execp`, if the command was a single-row `INSERT` and the modified table contained OIDs. (If not, you get zero.)

`quote string`

Doubles all occurrences of single quote and backslash characters in the given string. This can be used to safely quote strings that are to be inserted into SQL commands given to `spi_exec` or `spi_prepare`. For example, think about an SQL command string like:

```
"SELECT '$val' AS ret"
```

where the Tcl variable `val` actually contains `doesn't`. This would result in the final command string:

```
SELECT 'doesn't' AS ret
```

which would cause a parse error during `spi_exec` or `spi_prepare`. To work properly, the submitted command should contain:

```
SELECT 'doesn''t' AS ret
```

which can be formed in PL/Tcl using:

```
"SELECT '[ quote $val ]' AS ret"
```

One advantage of `spi_execp` is that you don't have to quote parameter values like this, since the parameters are never parsed as part of an SQL command string.

`elog` *level msg*

Emits a log or error message. Possible levels are `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, `ERROR`, and `FATAL`. `ERROR` raises an error condition; if this is not trapped by the surrounding Tcl code, the error propagates out to the calling query, causing the current transaction or subtransaction to be aborted. This is effectively the same as the Tcl `error` command. `FATAL` aborts the transaction and causes the current session to shut down. (There is probably no good reason to use this error level in PL/Tcl functions, but it's provided for completeness.) The other levels only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the log_min_messages and client_min_messages configuration variables. See Chapter 18 for more information.

41.6. Trigger Procedures in PL/Tcl

Trigger procedures can be written in PL/Tcl. PostgreSQL requires that a procedure that is to be called as a trigger must be declared as a function with no arguments and a return type of `trigger`.

The information from the trigger manager is passed to the procedure body in the following variables:

`$TG_name`

The name of the trigger from the `CREATE TRIGGER` statement.

`$TG_relid`

The object ID of the table that caused the trigger procedure to be invoked.

`$TG_table_name`

The name of the table that caused the trigger procedure to be invoked.

`$TG_table_schema`

The schema of the table that caused the trigger procedure to be invoked.

`$TG_relatts`

A Tcl list of the table column names, prefixed with an empty list element. So looking up a column name in the list with Tcl's `lsearch` command returns the element's number starting with 1 for the first column, the same way the columns are customarily numbered in PostgreSQL. (Empty list elements also appear in the positions of columns that have been dropped, so that the attribute numbering is correct for columns to their right.)

`$TG_when`

The string `BEFORE`, `AFTER`, or `INSTEAD OF`, depending on the type of trigger event.

`$TG_level`

The string `ROW` or `STATEMENT` depending on the type of trigger event.

`$TG_op`

The string `INSERT`, `UPDATE`, `DELETE`, or `TRUNCATE` depending on the type of trigger event.

`$NEW`

An associative array containing the values of the new table row for `INSERT` or `UPDATE` actions, or empty for `DELETE`. The array is indexed by column name. Columns that are null will not appear in the array. This is not set for statement-level triggers.

`$OLD`

An associative array containing the values of the old table row for `UPDATE` or `DELETE` actions, or empty for `INSERT`. The array is indexed by column name. Columns that are null will not appear in the array. This is not set for statement-level triggers.

`$args`

A Tcl list of the arguments to the procedure as given in the `CREATE TRIGGER` statement. These arguments are also accessible as `$1` ... `$n` in the procedure body.

The return value from a trigger procedure can be one of the strings `OK` or `SKIP`, or a list of column name/value pairs. If the return value is `OK`, the operation (`INSERT`/`UPDATE`/`DELETE`) that fired the trigger will proceed normally. `SKIP` tells the trigger manager to silently suppress the operation for this row. If a list is returned, it tells PL/Tcl to return a modified row to the trigger manager; the contents of the modified row are specified by the column names and values in the list. Any columns not mentioned in the list are set to null. Returning a modified row is only meaningful for row-level `BEFORE INSERT` or `UPDATE` triggers, for which the modified row will be inserted instead of the one given in `$NEW`; or for row-level `INSTEAD OF INSERT` or `UPDATE` triggers where the returned row is used as the source data for `INSERT RETURNING` or `UPDATE RETURNING` clauses. In row-level `BEFORE DELETE` or `INSTEAD OF DELETE` triggers, returning a modified row has the same effect as returning `OK`, that is the operation proceeds. The trigger return value is ignored for all other types of triggers.

Tip: The result list can be made from an array representation of the modified tuple with the `array get` Tcl command.

Here's a little example trigger procedure that forces an integer value in a table to keep track of the number of updates that are performed on the row. For new rows inserted, the value is initialized to 0 and then incremented on every update operation.

```
CREATE FUNCTION trigfunc_modcount() RETURNS trigger AS $$
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
$$ LANGUAGE pltcl;
```

```
CREATE TABLE mytab (num integer, description text, modcnt integer);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
    FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

Notice that the trigger procedure itself does not know the column name; that's supplied from the trigger arguments. This lets the trigger procedure be reused with different tables.

41.7. Event Trigger Procedures in PL/Tcl

Event trigger procedures can be written in PL/Tcl. PostgreSQL requires that a procedure that is to be called as an event trigger must be declared as a function with no arguments and a return type of `event_trigger`.

The information from the trigger manager is passed to the procedure body in the following variables:

`$TG_event`

The name of the event the trigger is fired for.

`$TG_tag`

The command tag for which the trigger is fired.

The return value of the trigger procedure is ignored.

Here's a little example event trigger procedure that simply raises a `NOTICE` message each time a supported command is executed:

```
CREATE OR REPLACE FUNCTION tclsnitch() RETURNS event_trigger AS $$
  elog NOTICE "tclsnitch: $TG_event $TG_tag"
$$ LANGUAGE pltcl;

CREATE EVENT TRIGGER tcl_a_snitch ON ddl_command_start EXECUTE PROCEDURE tclsnitch();
```

41.8. Modules and the `unknown` Command

PL/Tcl has support for autoloading Tcl code when used. It recognizes a special table, `pltcl_modules`, which is presumed to contain modules of Tcl code. If this table exists, the module `unknown` is fetched from the table and loaded into the Tcl interpreter immediately before the first execution of a PL/Tcl function in a database session. (This happens separately for each Tcl interpreter, if more than one is used in a session; see Section 41.4.)

While the `unknown` module could actually contain any initialization script you need, it normally defines a Tcl `unknown` procedure that is invoked whenever Tcl does not recognize an invoked procedure name. PL/Tcl's standard version of this procedure tries to find a module in `pltcl_modules` that will define the required procedure. If one is found, it is loaded into the interpreter, and then execution is allowed to

proceed with the originally attempted procedure call. A secondary table `pltcl_modfuncs` provides an index of which functions are defined by which modules, so that the lookup is reasonably quick.

The PostgreSQL distribution includes support scripts to maintain these tables: `pltcl_loadmod`, `pltcl_listmod`, `pltcl_delmod`, as well as source for the standard `unknown` module in `share/unknown.pltcl`. This module must be loaded into each database initially to support the autoloading mechanism.

The tables `pltcl_modules` and `pltcl_modfuncs` must be readable by all, but it is wise to make them owned and writable only by the database administrator. As a security precaution, PL/Tcl will ignore `pltcl_modules` (and thus, not attempt to load the `unknown` module) unless it is owned by a superuser. But update privileges on this table can be granted to other users, if you trust them sufficiently.

41.9. Tcl Procedure Names

In PostgreSQL, the same function name can be used for different function definitions as long as the number of arguments or their types differ. Tcl, however, requires all procedure names to be distinct. PL/Tcl deals with this by making the internal Tcl procedure names contain the object ID of the function from the system table `pg_proc` as part of their name. Thus, PostgreSQL functions with the same name and different argument types will be different Tcl procedures, too. This is not normally a concern for a PL/Tcl programmer, but it might be visible when debugging.

Chapter 42. PL/Perl - Perl Procedural Language

PL/Perl is a loadable procedural language that enables you to write PostgreSQL functions in the Perl programming language[1].

The main advantage to using PL/Perl is that this allows use, within stored functions, of the manyfold "string munging" operators and functions available for Perl. Parsing complex strings might be easier using Perl than it is with the string functions and control structures provided in PL/pgSQL.

To install PL/Perl in a particular database, use `CREATE EXTENSION plperl`, or from the shell command line use `createlang plperl` *`dbname`*.

> **Tip:** If a language is installed into `template1`, all subsequently created databases will have the language installed automatically.

> **Note:** Users of source packages must specially enable the build of PL/Perl during the installation process. (Refer to Chapter 15 for more information.) Users of binary packages might find PL/Perl in a separate subpackage.

42.1. PL/Perl Functions and Arguments

To create a function in the PL/Perl language, use the standard CREATE FUNCTION syntax:

```
CREATE FUNCTION funcname (argument-types) RETURNS return-type AS $$
    # PL/Perl function body
$$ LANGUAGE plperl;
```

The body of the function is ordinary Perl code. In fact, the PL/Perl glue code wraps it inside a Perl subroutine. A PL/Perl function is called in a scalar context, so it can't return a list. You can return non-scalar values (arrays, records, and sets) by returning a reference, as discussed below.

PL/Perl also supports anonymous code blocks called with the DO statement:

```
DO $$
    # PL/Perl code
$$ LANGUAGE plperl;
```

An anonymous code block receives no arguments, and whatever value it might return is discarded. Otherwise it behaves just like a function.

> **Note:** The use of named nested subroutines is dangerous in Perl, especially if they refer to lexical variables in the enclosing scope. Because a PL/Perl function is wrapped in a subroutine, any named subroutine you place inside one will be nested. In general, it is far safer to create anonymous subroutines which you call via a coderef. For more information, see the entries for `Variable "%s" will`

1. https://www.perl.org

not stay shared and Variable "%s" is not available in the perldiag man page, or search the Internet for "perl nested named subroutine".

The syntax of the CREATE FUNCTION command requires the function body to be written as a string constant. It is usually most convenient to use dollar quoting (see Section 4.1.2.4) for the string constant. If you choose to use escape string syntax E'', you must double any single quote marks (') and backslashes (\) used in the body of the function (see Section 4.1.2.1).

Arguments and results are handled as in any other Perl subroutine: arguments are passed in @_, and a result value is returned with return or as the last expression evaluated in the function.

For example, a function returning the greater of two integer values could be defined as:

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    if ($_[0] > $_[1]) { return $_[0]; }
    return $_[1];
$$ LANGUAGE plperl;
```

Note: Arguments will be converted from the database's encoding to UTF-8 for use inside PL/Perl, and then converted from UTF-8 back to the database encoding upon return.

If an SQL null value is passed to a function, the argument value will appear as "undefined" in Perl. The above function definition will not behave very nicely with null inputs (in fact, it will act as though they are zeroes). We could add STRICT to the function definition to make PostgreSQL do something more reasonable: if a null value is passed, the function will not be called at all, but will just return a null result automatically. Alternatively, we could check for undefined inputs in the function body. For example, suppose that we wanted perl_max with one null and one nonnull argument to return the nonnull argument, rather than a null value:

```
CREATE FUNCTION perl_max (integer, integer) RETURNS integer AS $$
    my ($x, $y) = @_;
    if (not defined $x) {
        return undef if not defined $y;
        return $y;
    }
    return $x if not defined $y;
    return $x if $x > $y;
    return $y;
$$ LANGUAGE plperl;
```

As shown above, to return an SQL null value from a PL/Perl function, return an undefined value. This can be done whether the function is strict or not.

Anything in a function argument that is not a reference is a string, which is in the standard PostgreSQL external text representation for the relevant data type. In the case of ordinary numeric or text types, Perl will just do the right thing and the programmer will normally not have to worry about it. However, in other cases the argument will need to be converted into a form that is more usable in Perl. For example, the decode_bytea function can be used to convert an argument of type bytea into unescaped binary.

Similarly, values passed back to PostgreSQL must be in the external text representation format. For example, the `encode_bytea` function can be used to escape binary data for a return value of type `bytea`.

Perl can return PostgreSQL arrays as references to Perl arrays. Here is an example:

```
CREATE OR REPLACE function returns_array()
RETURNS text[][] AS $$
    return [['a"b','c,d'],['e\\f','g']];
$$ LANGUAGE plperl;

select returns_array();
```

Perl passes PostgreSQL arrays as a blessed `PostgreSQL::InServer::ARRAY` object. This object may be treated as an array reference or a string, allowing for backward compatibility with Perl code written for PostgreSQL versions below 9.1 to run. For example:

```
CREATE OR REPLACE FUNCTION concat_array_elements(text[]) RETURNS TEXT AS $$
    my $arg = shift;
    my $result = "";
    return undef if (!defined $arg);

    # as an array reference
    for (@$arg) {
        $result .= $_;
    }

    # also works as a string
    $result .= $arg;

    return $result;
$$ LANGUAGE plperl;

SELECT concat_array_elements(ARRAY['PL','/','Perl']);
```

> **Note:** Multidimensional arrays are represented as references to lower-dimensional arrays of references in a way common to every Perl programmer.

Composite-type arguments are passed to the function as references to hashes. The keys of the hash are the attribute names of the composite type. Here is an example:

```
CREATE TABLE employee (
    name text,
    basesalary integer,
    bonus integer
);

CREATE FUNCTION empcomp(employee) RETURNS integer AS $$
    my ($emp) = @_;
```

```
    return $emp->{basesalary} + $emp->{bonus};
$$ LANGUAGE plperl;

SELECT name, empcomp(employee.*) FROM employee;
```

A PL/Perl function can return a composite-type result using the same approach: return a reference to a hash that has the required attributes. For example:

```
CREATE TYPE testrowperl AS (f1 integer, f2 text, f3 text);

CREATE OR REPLACE FUNCTION perl_row() RETURNS testrowperl AS $$
    return {f2 => 'hello', f1 => 1, f3 => 'world'};
$$ LANGUAGE plperl;

SELECT * FROM perl_row();
```

Any columns in the declared result data type that are not present in the hash will be returned as null values.

PL/Perl functions can also return sets of either scalar or composite types. Usually you'll want to return rows one at a time, both to speed up startup time and to keep from queuing up the entire result set in memory. You can do this with `return_next` as illustrated below. Note that after the last `return_next`, you must put either `return` or (better) `return undef`.

```
CREATE OR REPLACE FUNCTION perl_set_int(int)
RETURNS SETOF INTEGER AS $$
    foreach (0..$_[0]) {
        return_next($_);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set()
RETURNS SETOF testrowperl AS $$
    return_next({ f1 => 1, f2 => 'Hello', f3 => 'World' });
    return_next({ f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' });
    return_next({ f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' });
    return undef;
$$ LANGUAGE plperl;
```

For small result sets, you can return a reference to an array that contains either scalars, references to arrays, or references to hashes for simple types, array types, and composite types, respectively. Here are some simple examples of returning the entire result set as an array reference:

```
CREATE OR REPLACE FUNCTION perl_set_int(int) RETURNS SETOF INTEGER AS $$
    return [0..$_[0]];
$$ LANGUAGE plperl;

SELECT * FROM perl_set_int(5);

CREATE OR REPLACE FUNCTION perl_set() RETURNS SETOF testrowperl AS $$
```

```
    return [
        { f1 => 1, f2 => 'Hello', f3 => 'World' },
        { f1 => 2, f2 => 'Hello', f3 => 'PostgreSQL' },
        { f1 => 3, f2 => 'Hello', f3 => 'PL/Perl' }
    ];
$$ LANGUAGE plperl;

SELECT * FROM perl_set();
```

If you wish to use the `strict` pragma with your code you have a few options. For temporary global use you can `SET plperl.use_strict` to true. This will affect subsequent compilations of PL/Perl functions, but not functions already compiled in the current session. For permanent global use you can set `plperl.use_strict` to true in the `postgresql.conf` file.

For permanent use in specific functions you can simply put:

```
use strict;
```

at the top of the function body.

The `feature` pragma is also available to `use` if your Perl is version 5.10.0 or higher.

42.2. Data Values in PL/Perl

The argument values supplied to a PL/Perl function's code are simply the input arguments converted to text form (just as if they had been displayed by a `SELECT` statement). Conversely, the `return` and `return_next` commands will accept any string that is acceptable input format for the function's declared return type.

42.3. Built-in Functions

42.3.1. Database Access from PL/Perl

Access to the database itself from your Perl function can be done via the following functions:

`spi_exec_query(`*`query`* `[,` *`max-rows`*`])`

`spi_exec_query` executes an SQL command and returns the entire row set as a reference to an array of hash references. *You should only use this command when you know that the result set will be relatively small.* Here is an example of a query (`SELECT` command) with the optional maximum number of rows:

```
$rv = spi_exec_query('SELECT * FROM my_table', 5);
```

This returns up to 5 rows from the table `my_table`. If `my_table` has a column `my_column`, you can get that value from row `$i` of the result like this:

```
$foo = $rv->{rows}[$i]->{my_column};
```

The total number of rows returned from a SELECT query can be accessed like this:

```
$nrows = $rv->{processed}
```

Here is an example using a different command type:

```
$query = "INSERT INTO my_table VALUES (1, 'test')";
$rv = spi_exec_query($query);
```

You can then access the command status (e.g., SPI_OK_INSERT) like this:

```
$res = $rv->{status};
```

To get the number of rows affected, do:

```
$nrows = $rv->{processed};
```

Here is a complete example:

```
CREATE TABLE test (
    i int,
    v varchar
);

INSERT INTO test (i, v) VALUES (1, 'first line');
INSERT INTO test (i, v) VALUES (2, 'second line');
INSERT INTO test (i, v) VALUES (3, 'third line');
INSERT INTO test (i, v) VALUES (4, 'immortal');

CREATE OR REPLACE FUNCTION test_munge() RETURNS SETOF test AS $$
    my $rv = spi_exec_query('select i, v from test;');
    my $status = $rv->{status};
    my $nrows = $rv->{processed};
    foreach my $rn (0 .. $nrows - 1) {
        my $row = $rv->{rows}[$rn];
        $row->{i} += 200 if defined($row->{i});
        $row->{v} =~ tr/A-Za-z/a-zA-Z/ if (defined($row->{v}));
        return_next($row);
    }
    return undef;
$$ LANGUAGE plperl;

SELECT * FROM test_munge();
```

spi_query(*command*)

spi_fetchrow(*cursor*)

spi_cursor_close(*cursor*)

spi_query and spi_fetchrow work together as a pair for row sets which might be large, or for cases where you wish to return rows as they arrive. spi_fetchrow works *only* with spi_query. The following example illustrates how you use them together:

```
CREATE TYPE foo_type AS (the_num INTEGER, the_text TEXT);

CREATE OR REPLACE FUNCTION lotsa_md5 (INTEGER) RETURNS SETOF foo_type AS $$
    use Digest::MD5 qw(md5_hex);
    my $file = '/usr/share/dict/words';
    my $t = localtime;
    elog(NOTICE, "opening file $file at $t" );
    open my $fh, '<', $file # ooh, it's a file access!
```

```
        or elog(ERROR, "cannot open $file for reading: $!");
    my @words = <$fh>;
    close $fh;
    $t = localtime;
    elog(NOTICE, "closed file $file at $t");
    chomp(@words);
    my $row;
    my $sth = spi_query("SELECT * FROM generate_series(1,$_[0]) AS b(a)");
    while (defined ($row = spi_fetchrow($sth))) {
        return_next({
            the_num => $row->{a},
            the_text => md5_hex($words[rand @words])
        });
    }
    return;
$$ LANGUAGE plperlu;

SELECT * from lotsa_md5(500);
```

Normally, `spi_fetchrow` should be repeated until it returns `undef`, indicating that there are no more rows to read. The cursor returned by `spi_query` is automatically freed when `spi_fetchrow` returns `undef`. If you do not wish to read all the rows, instead call `spi_cursor_close` to free the cursor. Failure to do so will result in memory leaks.

`spi_prepare(`*`command`*`, `*`argument types`*`)`
`spi_query_prepared(`*`plan`*`, `*`arguments`*`)`
`spi_exec_prepared(`*`plan`*` [, `*`attributes`*`], `*`arguments`*`)`
`spi_freeplan(`*`plan`*`)`

`spi_prepare`, `spi_query_prepared`, `spi_exec_prepared`, and `spi_freeplan` implement the same functionality but for prepared queries. `spi_prepare` accepts a query string with numbered argument placeholders ($1, $2, etc) and a string list of argument types:

```
$plan = spi_prepare('SELECT * FROM test WHERE id > $1 AND name = $2',
                    'INTEGER', 'TEXT');
```

Once a query plan is prepared by a call to `spi_prepare`, the plan can be used instead of the string query, either in `spi_exec_prepared`, where the result is the same as returned by `spi_exec_query`, or in `spi_query_prepared` which returns a cursor exactly as `spi_query` does, which can be later passed to `spi_fetchrow`. The optional second parameter to `spi_exec_prepared` is a hash reference of attributes; the only attribute currently supported is `limit`, which sets the maximum number of rows returned by a query.

The advantage of prepared queries is that is it possible to use one prepared plan for more than one query execution. After the plan is not needed anymore, it can be freed with `spi_freeplan`:

```
CREATE OR REPLACE FUNCTION init() RETURNS VOID AS $$
        $_SHARED{my_plan} = spi_prepare('SELECT (now() + $1)::date AS now',
                                        'INTERVAL');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION add_time( INTERVAL ) RETURNS TEXT AS $$
        return spi_exec_prepared(
                $_SHARED{my_plan},
                $_[0]
```

```
        )->{rows}->[0]->{now};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION done() RETURNS VOID AS $$
        spi_freeplan( $_SHARED{my_plan});
        undef $_SHARED{my_plan};
$$ LANGUAGE plperl;

SELECT init();
SELECT add_time('1 day'), add_time('2 days'), add_time('3 days');
SELECT done();

  add_time  |  add_time  |  add_time
------------+------------+------------
 2005-12-10 | 2005-12-11 | 2005-12-12
```

Note that the parameter subscript in `spi_prepare` is defined via $1, $2, $3, etc, so avoid declaring query strings in double quotes that might easily lead to hard-to-catch bugs.

Another example illustrates usage of an optional parameter in `spi_exec_prepared`:

```
CREATE TABLE hosts AS SELECT id, ('192.168.1.'||id)::inet AS address
                     FROM generate_series(1,3) AS id;

CREATE OR REPLACE FUNCTION init_hosts_query() RETURNS VOID AS $$
        $_SHARED{plan} = spi_prepare('SELECT * FROM hosts
                                      WHERE address << $1', 'inet');
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION query_hosts(inet) RETURNS SETOF hosts AS $$
        return spi_exec_prepared(
                $_SHARED{plan},
                {limit => 2},
                $_[0]
        )->{rows};
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION release_hosts_query() RETURNS VOID AS $$
        spi_freeplan($_SHARED{plan});
        undef $_SHARED{plan};
$$ LANGUAGE plperl;

SELECT init_hosts_query();
SELECT query_hosts('192.168.1.0/30');
SELECT release_hosts_query();

    query_hosts
-----------------
 (1,192.168.1.1)
 (2,192.168.1.2)
(2 rows)
```

42.3.2. Utility Functions in PL/Perl

`elog(level, msg)`

Emit a log or error message. Possible levels are `DEBUG`, `LOG`, `INFO`, `NOTICE`, `WARNING`, and `ERROR`. `ERROR` raises an error condition; if this is not trapped by the surrounding Perl code, the error propagates out to the calling query, causing the current transaction or subtransaction to be aborted. This is effectively the same as the Perl `die` command. The other levels only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the log_min_messages and client_min_messages configuration variables. See Chapter 18 for more information.

`quote_literal(string)`

Return the given string suitably quoted to be used as a string literal in an SQL statement string. Embedded single-quotes and backslashes are properly doubled. Note that `quote_literal` returns undef on undef input; if the argument might be undef, `quote_nullable` is often more suitable.

`quote_nullable(string)`

Return the given string suitably quoted to be used as a string literal in an SQL statement string; or, if the argument is undef, return the unquoted string "NULL". Embedded single-quotes and backslashes are properly doubled.

`quote_ident(string)`

Return the given string suitably quoted to be used as an identifier in an SQL statement string. Quotes are added only if necessary (i.e., if the string contains non-identifier characters or would be case-folded). Embedded quotes are properly doubled.

`decode_bytea(string)`

Return the unescaped binary data represented by the contents of the given string, which should be `bytea` encoded.

`encode_bytea(string)`

Return the `bytea` encoded form of the binary data contents of the given string.

`encode_array_literal(array)`
`encode_array_literal(array, delimiter)`

Returns the contents of the referenced array as a string in array literal format (see Section 8.15.2). Returns the argument value unaltered if it's not a reference to an array. The delimiter used between elements of the array literal defaults to `", "` if a delimiter is not specified or is undef.

`encode_typed_literal(value, typename)`

Converts a Perl variable to the value of the data type passed as a second argument and returns a string representation of this value. Correctly handles nested arrays and values of composite types.

`encode_array_constructor(array)`

Returns the contents of the referenced array as a string in array constructor format (see Section 4.2.12). Individual values are quoted using `quote_nullable`. Returns the argument value, quoted using `quote_nullable`, if it's not a reference to an array.

`looks_like_number(`*`string`*`)`

Returns a true value if the content of the given string looks like a number, according to Perl, returns false otherwise. Returns undef if the argument is undef. Leading and trailing space is ignored. `Inf` and `Infinity` are regarded as numbers.

`is_array_ref(`*`argument`*`)`

Returns a true value if the given argument may be treated as an array reference, that is, if ref of the argument is `ARRAY` or `PostgreSQL::InServer::ARRAY`. Returns false otherwise.

42.4. Global Values in PL/Perl

You can use the global hash `%_SHARED` to store data, including code references, between function calls for the lifetime of the current session.

Here is a simple example for shared data:

```
CREATE OR REPLACE FUNCTION set_var(name text, val text) RETURNS text AS $$
    if ($_SHARED{$_[0]} = $_[1]) {
        return 'ok';
    } else {
        return "cannot set shared variable $_[0] to $_[1]";
    }
$$ LANGUAGE plperl;

CREATE OR REPLACE FUNCTION get_var(name text) RETURNS text AS $$
    return $_SHARED{$_[0]};
$$ LANGUAGE plperl;

SELECT set_var('sample', 'Hello, PL/Perl!  How"s tricks?');
SELECT get_var('sample');
```

Here is a slightly more complicated example using a code reference:

```
CREATE OR REPLACE FUNCTION myfuncs() RETURNS void AS $$
    $_SHARED{myquote} = sub {
        my $arg = shift;
        $arg =~ s/(['\\])/\\$1/g;
        return "'$arg'";
    };
$$ LANGUAGE plperl;

SELECT myfuncs(); /* initializes the function */

/* Set up a function that uses the quote function */

CREATE OR REPLACE FUNCTION use_quote(TEXT) RETURNS text AS $$
    my $text_to_quote = shift;
    my $qfunc = $_SHARED{myquote};
```

```
    return &$qfunc($text_to_quote);
$$ LANGUAGE plperl;
```

(You could have replaced the above with the one-liner `return $_SHARED{myquote}->($_[0]);` at the expense of readability.)

For security reasons, PL/Perl executes functions called by any one SQL role in a separate Perl interpreter for that role. This prevents accidental or malicious interference by one user with the behavior of another user's PL/Perl functions. Each such interpreter has its own value of the `%_SHARED` variable and other global state. Thus, two PL/Perl functions will share the same value of `%_SHARED` if and only if they are executed by the same SQL role. In an application wherein a single session executes code under multiple SQL roles (via `SECURITY DEFINER` functions, use of `SET ROLE`, etc) you may need to take explicit steps to ensure that PL/Perl functions can share data via `%_SHARED`. To do that, make sure that functions that should communicate are owned by the same user, and mark them `SECURITY DEFINER`. You must of course take care that such functions can't be used to do anything unintended.

42.5. Trusted and Untrusted PL/Perl

Normally, PL/Perl is installed as a "trusted" programming language named `plperl`. In this setup, certain Perl operations are disabled to preserve security. In general, the operations that are restricted are those that interact with the environment. This includes file handle operations, `require`, and `use` (for external modules). There is no way to access internals of the database server process or to gain OS-level access with the permissions of the server process, as a C function can do. Thus, any unprivileged database user can be permitted to use this language.

Here is an example of a function that will not work because file system operations are not allowed for security reasons:

```
CREATE FUNCTION badfunc() RETURNS integer AS $$
    my $tmpfile = "/tmp/badfile";
    open my $fh, '>', $tmpfile
        or elog(ERROR, qq{could not open the file "$tmpfile": $!});
    print $fh "Testing writing to a file\n";
    close $fh or elog(ERROR, qq{could not close the file "$tmpfile": $!});
    return 1;
$$ LANGUAGE plperl;
```

The creation of this function will fail as its use of a forbidden operation will be caught by the validator.

Sometimes it is desirable to write Perl functions that are not restricted. For example, one might want a Perl function that sends mail. To handle these cases, PL/Perl can also be installed as an "untrusted" language (usually called PL/PerlU). In this case the full Perl language is available. When installing the language, the language name `plperlu` will select the untrusted PL/Perl variant.

The writer of a PL/PerlU function must take care that the function cannot be used to do anything unwanted, since it will be able to do anything that could be done by a user logged in as the database administrator. Note that the database system allows only database superusers to create functions in untrusted languages.

If the above function was created by a superuser using the language `plperlu`, execution would succeed.

In the same way, anonymous code blocks written in Perl can use restricted operations if the language is specified as `plperlu` rather than `plperl`, but the caller must be a superuser.

> **Note:** While PL/Perl functions run in a separate Perl interpreter for each SQL role, all PL/PerlU functions executed in a given session run in a single Perl interpreter (which is not any of the ones used for PL/Perl functions). This allows PL/PerlU functions to share data freely, but no communication can occur between PL/Perl and PL/PerlU functions.

> **Note:** Perl cannot support multiple interpreters within one process unless it was built with the appropriate flags, namely either `usemultiplicity` or `useithreads`. (`usemultiplicity` is preferred unless you actually need to use threads. For more details, see the perlembed man page.) If PL/Perl is used with a copy of Perl that was not built this way, then it is only possible to have one Perl interpreter per session, and so any one session can only execute either PL/PerlU functions, or PL/Perl functions that are all called by the same SQL role.

42.6. PL/Perl Triggers

PL/Perl can be used to write trigger functions. In a trigger function, the hash reference `$_TD` contains information about the current trigger event. `$_TD` is a global variable, which gets a separate local value for each invocation of the trigger. The fields of the `$_TD` hash reference are:

`$_TD->{new}{foo}`

`NEW` value of column `foo`

`$_TD->{old}{foo}`

`OLD` value of column `foo`

`$_TD->{name}`

Name of the trigger being called

`$_TD->{event}`

Trigger event: `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, or `UNKNOWN`

`$_TD->{when}`

When the trigger was called: `BEFORE`, `AFTER`, `INSTEAD OF`, or `UNKNOWN`

`$_TD->{level}`

The trigger level: `ROW`, `STATEMENT`, or `UNKNOWN`

`$_TD->{relid}`

OID of the table on which the trigger fired

`$_TD->{table_name}`

Name of the table on which the trigger fired

`$_TD->{relname}`

Name of the table on which the trigger fired. This has been deprecated, and could be removed in a future release. Please use $_TD->{table_name} instead.

`$_TD->{table_schema}`

Name of the schema in which the table on which the trigger fired, is

`$_TD->{argc}`

Number of arguments of the trigger function

`@{$_TD->{args}}`

Arguments of the trigger function. Does not exist if `$_TD->{argc}` is 0.

Row-level triggers can return one of the following:

`return;`

Execute the operation

`"SKIP"`

Don't execute the operation

`"MODIFY"`

Indicates that the `NEW` row was modified by the trigger function

Here is an example of a trigger function, illustrating some of the above:

```
CREATE TABLE test (
    i int,
    v varchar
);

CREATE OR REPLACE FUNCTION valid_id() RETURNS trigger AS $$
    if (($_TD->{new}{i} >= 100) || ($_TD->{new}{i} <= 0)) {
        return "SKIP";    # skip INSERT/UPDATE command
    } elsif ($_TD->{new}{v} ne "immortal") {
        $_TD->{new}{v} .= "(modified by trigger)";
        return "MODIFY";  # modify row and execute INSERT/UPDATE command
    } else {
        return;           # execute INSERT/UPDATE command
    }
$$ LANGUAGE plperl;

CREATE TRIGGER test_valid_id_trig
    BEFORE INSERT OR UPDATE ON test
    FOR EACH ROW EXECUTE PROCEDURE valid_id();
```

42.7. PL/Perl Event Triggers

PL/Perl can be used to write event trigger functions. In an event trigger function, the hash reference `$_TD` contains information about the current trigger event. `$_TD` is a global variable, which gets a separate local value for each invocation of the trigger. The fields of the `$_TD` hash reference are:

`$_TD->{event}`

The name of the event the trigger is fired for.

`$_TD->{tag}`

The command tag for which the trigger is fired.

The return value of the trigger procedure is ignored.

Here is an example of an event trigger function, illustrating some of the above:

```
CREATE OR REPLACE FUNCTION perlsnitch() RETURNS event_trigger AS $$
  elog(NOTICE, "perlsnitch: " . $_TD->{event} . " " . $_TD->{tag} . " ");
$$ LANGUAGE plperl;

CREATE EVENT TRIGGER perl_a_snitch
    ON ddl_command_start
    EXECUTE PROCEDURE perlsnitch();
```

42.8. PL/Perl Under the Hood

42.8.1. Configuration

This section lists configuration parameters that affect PL/Perl.

`plperl.on_init` (`string`)

Specifies Perl code to be executed when a Perl interpreter is first initialized, before it is specialized for use by `plperl` or `plperlu`. The SPI functions are not available when this code is executed. If the code fails with an error it will abort the initialization of the interpreter and propagate out to the calling query, causing the current transaction or subtransaction to be aborted.

The Perl code is limited to a single string. Longer code can be placed into a module and loaded by the `on_init` string. Examples:

```
plperl.on_init = 'require "plperlinit.pl"'
plperl.on_init = 'use lib "/my/app"; use MyApp::PgInit;'
```

Any modules loaded by `plperl.on_init`, either directly or indirectly, will be available for use by `plperl`. This may create a security risk. To see what modules have been loaded you can use:

```
DO 'elog(WARNING, join ", ", sort keys %INC)' LANGUAGE plperl;
```

Initialization will happen in the postmaster if the plperl library is included in shared_preload_libraries, in which case extra consideration should be given to the risk of destabilizing the postmaster. The principal reason for making use of this feature is that Perl modules loaded by `plperl.on_init` need be loaded only at postmaster start, and will be instantly available without loading overhead in individual database sessions. However, keep in mind that the overhead is avoided only for the first Perl interpreter used by a database session — either PL/PerlU, or PL/Perl for the first SQL role that calls a PL/Perl function. Any additional Perl interpreters created in a database session will have to execute `plperl.on_init` afresh. Also, on Windows there will be no savings whatsoever from preloading, since the Perl interpreter created in the postmaster process does not propagate to child processes.

This parameter can only be set in the `postgresql.conf` file or on the server command line.

`plperl.on_plperl_init` (`string`)
`plperl.on_plperlu_init` (`string`)

These parameters specify Perl code to be executed when a Perl interpreter is specialized for `plperl` or `plperlu` respectively. This will happen when a PL/Perl or PL/PerlU function is first executed in a database session, or when an additional interpreter has to be created because the other language is called or a PL/Perl function is called by a new SQL role. This follows any initialization done by `plperl.on_init`. The SPI functions are not available when this code is executed. The Perl code in `plperl.on_plperl_init` is executed after "locking down" the interpreter, and thus it can only perform trusted operations.

If the code fails with an error it will abort the initialization and propagate out to the calling query, causing the current transaction or subtransaction to be aborted. Any actions already done within Perl won't be undone; however, that interpreter won't be used again. If the language is used again the initialization will be attempted again within a fresh Perl interpreter.

Only superusers can change these settings. Although these settings can be changed within a session, such changes will not affect Perl interpreters that have already been used to execute functions.

`plperl.use_strict` (`boolean`)

When set true subsequent compilations of PL/Perl functions will have the `strict` pragma enabled. This parameter does not affect functions already compiled in the current session.

42.8.2. Limitations and Missing Features

The following features are currently missing from PL/Perl, but they would make welcome contributions.

- PL/Perl functions cannot call each other directly.
- SPI is not yet fully implemented.
- If you are fetching very large data sets using `spi_exec_query`, you should be aware that these will all go into memory. You can avoid this by using `spi_query`/`spi_fetchrow` as illustrated earlier.

  A similar problem occurs if a set-returning function passes a large set of rows back to PostgreSQL via `return`. You can avoid this problem too by instead using `return_next` for each row returned, as shown previously.

- When a session ends normally, not due to a fatal error, any `END` blocks that have been defined are executed. Currently no other actions are performed. Specifically, file handles are not automatically flushed and objects are not automatically destroyed.

Chapter 43. PL/Python - Python Procedural Language

The PL/Python procedural language allows PostgreSQL functions to be written in the Python language[1].

To install PL/Python in a particular database, use `CREATE EXTENSION plpythonu`, or from the shell command line use `createlang plpythonu` *dbname* (but see also Section 43.1).

> **Tip:** If a language is installed into `template1`, all subsequently created databases will have the language installed automatically.

PL/Python is only available as an "untrusted" language, meaning it does not offer any way of restricting what users can do in it and is therefore named `plpythonu`. A trusted variant `plpython` might become available in the future if a secure execution mechanism is developed in Python. The writer of a function in untrusted PL/Python must take care that the function cannot be used to do anything unwanted, since it will be able to do anything that could be done by a user logged in as the database administrator. Only superusers can create functions in untrusted languages such as `plpythonu`.

> **Note:** Users of source packages must specially enable the build of PL/Python during the installation process. (Refer to the installation instructions for more information.) Users of binary packages might find PL/Python in a separate subpackage.

43.1. Python 2 vs. Python 3

PL/Python supports both the Python 2 and Python 3 language variants. (The PostgreSQL installation instructions might contain more precise information about the exact supported minor versions of Python.) Because the Python 2 and Python 3 language variants are incompatible in some important aspects, the following naming and transitioning scheme is used by PL/Python to avoid mixing them:

- The PostgreSQL language named `plpython2u` implements PL/Python based on the Python 2 language variant.
- The PostgreSQL language named `plpython3u` implements PL/Python based on the Python 3 language variant.
- The language named `plpythonu` implements PL/Python based on the default Python language variant, which is currently Python 2. (This default is independent of what any local Python installations might consider to be their "default", for example, what `/usr/bin/python` might be.) The default will probably be changed to Python 3 in a distant future release of PostgreSQL, depending on the progress of the migration to Python 3 in the Python community.

1. http://www.python.org

This scheme is analogous to the recommendations in PEP 394[2] regarding the naming and transitioning of the `python` command.

It depends on the build configuration or the installed packages whether PL/Python for Python 2 or Python 3 or both are available.

> **Tip:** The built variant depends on which Python version was found during the installation or which version was explicitly set using the `PYTHON` environment variable; see Section 15.4. To make both variants of PL/Python available in one installation, the source tree has to be configured and built twice.

This results in the following usage and migration strategy:

- Existing users and users who are currently not interested in Python 3 use the language name `plpythonu` and don't have to change anything for the foreseeable future. It is recommended to gradually "future-proof" the code via migration to Python 2.6/2.7 to simplify the eventual migration to Python 3.

  In practice, many PL/Python functions will migrate to Python 3 with few or no changes.
- Users who know that they have heavily Python 2 dependent code and don't plan to ever change it can make use of the `plpython2u` language name. This will continue to work into the very distant future, until Python 2 support might be completely dropped by PostgreSQL.
- Users who want to dive into Python 3 can use the `plpython3u` language name, which will keep working forever by today's standards. In the distant future, when Python 3 might become the default, they might like to remove the "3" for aesthetic reasons.
- Daredevils, who want to build a Python-3-only operating system environment, can change the contents of `pg_pltemplate` to make `plpythonu` be equivalent to `plpython3u`, keeping in mind that this would make their installation incompatible with most of the rest of the world.

See also the document What's New In Python 3.0[3] for more information about porting to Python 3.

It is not allowed to use PL/Python based on Python 2 and PL/Python based on Python 3 in the same session, because the symbols in the dynamic modules would clash, which could result in crashes of the PostgreSQL server process. There is a check that prevents mixing Python major versions in a session, which will abort the session if a mismatch is detected. It is possible, however, to use both PL/Python variants in the same database, from separate sessions.

43.2. PL/Python Functions

Functions in PL/Python are declared via the standard CREATE FUNCTION syntax:

```
CREATE FUNCTION funcname (argument-list)
  RETURNS return-type
```

2. http://www.python.org/dev/peps/pep-0394/
3. http://docs.python.org/py3k/whatsnew/3.0.html

```
AS $$
  # PL/Python function body
$$ LANGUAGE plpythonu;
```

The body of a function is simply a Python script. When the function is called, its arguments are passed as elements of the list `args`; named arguments are also passed as ordinary variables to the Python script. Use of named arguments is usually more readable. The result is returned from the Python code in the usual way, with `return` or `yield` (in case of a result-set statement). If you do not provide a return value, Python returns the default `None`. PL/Python translates Python's `None` into the SQL null value.

For example, a function to return the greater of two integers can be defined as:

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

The Python code that is given as the body of the function definition is transformed into a Python function. For example, the above results in:

```
def __plpython_procedure_pymax_23456():
  if a > b:
    return a
  return b
```

assuming that 23456 is the OID assigned to the function by PostgreSQL.

The arguments are set as global variables. Because of the scoping rules of Python, this has the subtle consequence that an argument variable cannot be reassigned inside the function to the value of an expression that involves the variable name itself, unless the variable is redeclared as global in the block. For example, the following won't work:

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  x = x.strip()  # error
  return x
$$ LANGUAGE plpythonu;
```

because assigning to `x` makes `x` a local variable for the entire block, and so the `x` on the right-hand side of the assignment refers to a not-yet-assigned local variable `x`, not the PL/Python function parameter. Using the `global` statement, this can be made to work:

```
CREATE FUNCTION pystrip(x text)
  RETURNS text
AS $$
  global x
  x = x.strip()  # ok now
  return x
```

```
$$ LANGUAGE plpythonu;
```

But it is advisable not to rely on this implementation detail of PL/Python. It is better to treat the function parameters as read-only.

43.3. Data Values

Generally speaking, the aim of PL/Python is to provide a "natural" mapping between the PostgreSQL and the Python worlds. This informs the data mapping rules described below.

43.3.1. Data Type Mapping

When a PL/Python function is called, its arguments are converted from their PostgreSQL data type to a corresponding Python type:

- PostgreSQL `boolean` is converted to Python `bool`.
- PostgreSQL `smallint` and `int` are converted to Python `int`. PostgreSQL `bigint` and `oid` are converted to `long` in Python 2 and to `int` in Python 3.
- PostgreSQL `real` and `double` are converted to Python `float`.
- PostgreSQL `numeric` is converted to Python `Decimal`. This type is imported from the `cdecimal` package if that is available. Otherwise, `decimal.Decimal` from the standard library will be used. `cdecimal` is significantly faster than `decimal`. In Python 3.3 and up, however, `cdecimal` has been integrated into the standard library under the name `decimal`, so there is no longer any difference.
- PostgreSQL `bytea` is converted to Python `str` in Python 2 and to `bytes` in Python 3. In Python 2, the string should be treated as a byte sequence without any character encoding.
- All other data types, including the PostgreSQL character string types, are converted to a Python `str`. In Python 2, this string will be in the PostgreSQL server encoding; in Python 3, it will be a Unicode string like all strings.
- For nonscalar data types, see below.

When a PL/Python function returns, its return value is converted to the function's declared PostgreSQL return data type as follows:

- When the PostgreSQL return type is `boolean`, the return value will be evaluated for truth according to the *Python* rules. That is, 0 and empty string are false, but notably `'f'` is true.
- When the PostgreSQL return type is `bytea`, the return value will be converted to a string (Python 2) or bytes (Python 3) using the respective Python built-ins, with the result being converted to `bytea`.
- For all other PostgreSQL return types, the return value is converted to a string using the Python built-in `str`, and the result is passed to the input function of the PostgreSQL data type. (If the Python value is a `float`, it is converted using the `repr` built-in instead of `str`, to avoid loss of precision.)

Strings in Python 2 are required to be in the PostgreSQL server encoding when they are passed to PostgreSQL. Strings that are not valid in the current server encoding will raise an error, but not all encoding mismatches can be detected, so garbage data can still result when this is not done correctly. Unicode strings are converted to the correct encoding automatically, so it can be safer and more convenient to use those. In Python 3, all strings are Unicode strings.

- For nonscalar data types, see below.

Note that logical mismatches between the declared PostgreSQL return type and the Python data type of the actual return object are not flagged; the value will be converted in any case.

43.3.2. Null, None

If an SQL null value is passed to a function, the argument value will appear as `None` in Python. For example, the function definition of `pymax` shown in Section 43.2 will return the wrong answer for null inputs. We could add `STRICT` to the function definition to make PostgreSQL do something more reasonable: if a null value is passed, the function will not be called at all, but will just return a null result automatically. Alternatively, we could check for null inputs in the function body:

```
CREATE FUNCTION pymax (a integer, b integer)
  RETURNS integer
AS $$
  if (a is None) or (b is None):
    return None
  if a > b:
    return a
  return b
$$ LANGUAGE plpythonu;
```

As shown above, to return an SQL null value from a PL/Python function, return the value `None`. This can be done whether the function is strict or not.

43.3.3. Arrays, Lists

SQL array values are passed into PL/Python as a Python list. To return an SQL array value out of a PL/Python function, return a Python sequence, for example a list or tuple:

```
CREATE FUNCTION return_arr()
  RETURNS int[]
AS $$
return (1, 2, 3, 4, 5)
$$ LANGUAGE plpythonu;

SELECT return_arr();
 return_arr
-------------
 {1,2,3,4,5}
(1 row)
```

Note that in Python, strings are sequences, which can have undesirable effects that might be familiar to Python programmers:

```
CREATE FUNCTION return_str_arr()
  RETURNS varchar[]
AS $$
return "hello"
$$ LANGUAGE plpythonu;

SELECT return_str_arr();
 return_str_arr
----------------
 {h,e,l,l,o}
(1 row)
```

43.3.4. Composite Types

Composite-type arguments are passed to the function as Python mappings. The element names of the mapping are the attribute names of the composite type. If an attribute in the passed row has the null value, it has the value `None` in the mapping. Here is an example:

```
CREATE TABLE employee (
  name text,
  salary integer,
  age integer
);

CREATE FUNCTION overpaid (e employee)
  RETURNS boolean
AS $$
  if e["salary"] > 200000:
    return True
  if (e["age"] < 30) and (e["salary"] > 100000):
    return True
  return False
$$ LANGUAGE plpythonu;
```

There are multiple ways to return row or composite types from a Python function. The following examples assume we have:

```
CREATE TYPE named_value AS (
  name   text,
  value  integer
);
```

A composite result can be returned as a:

Sequence type (a tuple or list, but not a set because it is not indexable)

Returned sequence objects must have the same number of items as the composite result type has fields. The item with index 0 is assigned to the first field of the composite type, 1 to the second and so on. For example:

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return [ name, value ]
  # or alternatively, as tuple: return ( name, value )
$$ LANGUAGE plpythonu;
```

To return a SQL null for any column, insert `None` at the corresponding position.

Mapping (dictionary)

The value for each result type column is retrieved from the mapping with the column name as key. Example:

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  return { "name": name, "value": value }
$$ LANGUAGE plpythonu;
```

Any extra dictionary key/value pairs are ignored. Missing keys are treated as errors. To return a SQL null value for any column, insert `None` with the corresponding column name as the key.

Object (any object providing method `__getattr__`)

This works the same as a mapping. Example:

```
CREATE FUNCTION make_pair (name text, value integer)
  RETURNS named_value
AS $$
  class named_value:
    def __init__ (self, n, v):
      self.name = n
      self.value = v
  return named_value(name, value)

  # or simply
  class nv: pass
  nv.name = name
  nv.value = value
  return nv
$$ LANGUAGE plpythonu;
```

Functions with `OUT` parameters are also supported. For example:

```
CREATE FUNCTION multiout_simple(OUT i integer, OUT j integer) AS $$
return (1, 2)
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple();
```

43.3.5. Set-returning Functions

A PL/Python function can also return sets of scalar or composite types. There are several ways to achieve this because the returned object is internally turned into an iterator. The following examples assume we have composite type:

```
CREATE TYPE greeting AS (
  how text,
  who text
);
```

A set result can be returned from a:

Sequence type (tuple, list, set)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  # return tuple containing lists as composite types
  # all other combinations work also
  return ( [ how, "World" ], [ how, "PostgreSQL" ], [ how, "PL/Python" ] )
$$ LANGUAGE plpythonu;
```

Iterator (any object providing `__iter__` and `next` methods)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  class producer:
    def __init__ (self, how, who):
      self.how = how
      self.who = who
      self.ndx = -1

    def __iter__ (self):
      return self

    def next (self):
      self.ndx += 1
      if self.ndx == len(self.who):
        raise StopIteration
      return ( self.how, self.who[self.ndx] )

  return producer(how, [ "World", "PostgreSQL", "PL/Python" ])
$$ LANGUAGE plpythonu;
```

Generator (`yield`)

```
CREATE FUNCTION greet (how text)
  RETURNS SETOF greeting
AS $$
  for who in [ "World", "PostgreSQL", "PL/Python" ]:
    yield ( how, who )
$$ LANGUAGE plpythonu;
```

Warning

Due to Python bug #1483133[4], some debug versions of Python 2.4 (configured and compiled with option `--with-pydebug`) are known to crash the PostgreSQL server when using an iterator to return a set result. Unpatched versions of Fedora 4 contain this bug. It does not happen in production versions of Python or on patched versions of Fedora 4.

Set-returning functions with `OUT` parameters (using `RETURNS SETOF record`) are also supported. For example:

```
CREATE FUNCTION multiout_simple_setof(n integer, OUT integer, OUT integer) RETURNS SETOF re
return [(1, 2)] * n
$$ LANGUAGE plpythonu;

SELECT * FROM multiout_simple_setof(3);
```

43.4. Sharing Data

The global dictionary `SD` is available to store private data between repeated calls to the same function. The global dictionary `GD` is public data, that is available to all Python functions within a session; use with care.

Each function gets its own execution environment in the Python interpreter, so that global data and function arguments from `myfunc` are not available to `myfunc2`. The exception is the data in the `GD` dictionary, as mentioned above.

43.5. Anonymous Code Blocks

PL/Python also supports anonymous code blocks called with the DO statement:

```
DO $$
    # PL/Python code
$$ LANGUAGE plpythonu;
```

An anonymous code block receives no arguments, and whatever value it might return is discarded. Otherwise it behaves just like a function.

43.6. Trigger Functions

When a function is used as a trigger, the dictionary `TD` contains trigger-related values:

`TD["event"]`

contains the event as a string: `INSERT`, `UPDATE`, `DELETE`, or `TRUNCATE`.

`TD["when"]`

contains one of `BEFORE`, `AFTER`, or `INSTEAD OF`.

`TD["level"]`

contains `ROW` or `STATEMENT`.

`TD["new"]`
`TD["old"]`

For a row-level trigger, one or both of these fields contain the respective trigger rows, depending on the trigger event.

`TD["name"]`

contains the trigger name.

`TD["table_name"]`

contains the name of the table on which the trigger occurred.

`TD["table_schema"]`

contains the schema of the table on which the trigger occurred.

`TD["relid"]`

contains the OID of the table on which the trigger occurred.

`TD["args"]`

If the `CREATE TRIGGER` command included arguments, they are available in `TD["args"][0]` to `TD["args"][`*n*`-1]`.

If `TD["when"]` is `BEFORE` or `INSTEAD OF` and `TD["level"]` is `ROW`, you can return `None` or `"OK"` from the Python function to indicate the row is unmodified, `"SKIP"` to abort the event, or if `TD["event"]` is `INSERT` or `UPDATE` you can return `"MODIFY"` to indicate you've modified the new row. Otherwise the return value is ignored.

43.7. Database Access

The PL/Python language module automatically imports a Python module called `plpy`. The functions and constants in this module are available to you in the Python code as `plpy.foo`.

43.7.1. Database Access Functions

The `plpy` module provides several functions to execute database commands:

`plpy.execute(query [, max-rows])`

Calling `plpy.execute` with a query string and an optional row limit argument causes that query to be run and the result to be returned in a result object.

The result object emulates a list or dictionary object. The result object can be accessed by row number and column name. For example:

```
rv = plpy.execute("SELECT * FROM my_table", 5)
```

returns up to 5 rows from `my_table`. If `my_table` has a column `my_column`, it would be accessed as:

```
foo = rv[i]["my_column"]
```

The number of rows returned can be obtained using the built-in `len` function.

The result object has these additional methods:

`nrows()`

Returns the number of rows processed by the command. Note that this is not necessarily the same as the number of rows returned. For example, an `UPDATE` command will set this value but won't return any rows (unless `RETURNING` is used).

`status()`

The `SPI_execute()` return value.

`colnames()`
`coltypes()`
`coltypmods()`

Return a list of column names, list of column type OIDs, and list of type-specific type modifiers for the columns, respectively.

These methods raise an exception when called on a result object from a command that did not produce a result set, e.g., `UPDATE` without `RETURNING`, or `DROP TABLE`. But it is OK to use these methods on a result set containing zero rows.

`__str__()`

The standard `__str__` method is defined so that it is possible for example to debug query execution results using `plpy.debug(rv)`.

The result object can be modified.

Note that calling `plpy.execute` will cause the entire result set to be read into memory. Only use that function when you are sure that the result set will be relatively small. If you don't want to risk excessive memory usage when fetching large results, use `plpy.cursor` rather than `plpy.execute`.

`plpy.prepare(`*query* `[, `*argtypes*`])`
`plpy.execute(`*plan* `[, `*arguments* `[, `*max-rows*`]])`

`plpy.prepare` prepares the execution plan for a query. It is called with a query string and a list of parameter types, if you have parameter references in the query. For example:

```
plan = plpy.prepare("SELECT last_name FROM my_users WHERE first_name = $1", ["text"])
```

`text` is the type of the variable you will be passing for `$1`. The second argument is optional if you don't want to pass any parameters to the query.

After preparing a statement, you use a variant of the function `plpy.execute` to run it:

```
rv = plpy.execute(plan, ["name"], 5)
```

Pass the plan as the first argument (instead of the query string), and a list of values to substitute into the query as the second argument. The second argument is optional if the query does not expect any parameters. The third argument is the optional row limit as before.

Query parameters and result row fields are converted between PostgreSQL and Python data types as described in Section 43.3.

When you prepare a plan using the PL/Python module it is automatically saved. Read the SPI documentation (Chapter 44) for a description of what this means. In order to make effective use of this across function calls one needs to use one of the persistent storage dictionaries `SD` or `GD` (see Section 43.4). For example:

```
CREATE FUNCTION usesavedplan() RETURNS trigger AS $$
    if "plan" in SD:
        plan = SD["plan"]
    else:
        plan = plpy.prepare("SELECT 1")
        SD["plan"] = plan
    # rest of function
$$ LANGUAGE plpythonu;
```

`plpy.cursor(`*query*`)`
`plpy.cursor(`*plan* `[, `*arguments*`])`

The `plpy.cursor` function accepts the same arguments as `plpy.execute` (except for the row limit) and returns a cursor object, which allows you to process large result sets in smaller chunks. As with `plpy.execute`, either a query string or a plan object along with a list of arguments can be used.

The cursor object provides a `fetch` method that accepts an integer parameter and returns a result object. Each time you call `fetch`, the returned object will contain the next batch of rows, never larger than the parameter value. Once all rows are exhausted, `fetch` starts returning an empty result object. Cursor objects also provide an iterator interface[5], yielding one row at a time until all rows are exhausted. Data fetched that way is not returned as result objects, but rather as dictionaries, each dictionary corresponding to a single result row.

An example of two ways of processing data from a large table is:

5. http://docs.python.org/library/stdtypes.html#iterator-types

```
CREATE FUNCTION count_odd_iterator() RETURNS integer AS $$
odd = 0
for row in plpy.cursor("select num from largetable"):
    if row['num'] % 2:
         odd += 1
return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_fetch(batch_size integer) RETURNS integer AS $$
odd = 0
cursor = plpy.cursor("select num from largetable")
while True:
    rows = cursor.fetch(batch_size)
    if not rows:
        break
    for row in rows:
        if row['num'] % 2:
            odd += 1
return odd
$$ LANGUAGE plpythonu;

CREATE FUNCTION count_odd_prepared() RETURNS integer AS $$
odd = 0
plan = plpy.prepare("select num from largetable where num % $1 <> 0", ["integer"])
rows = list(plpy.cursor(plan, [2]))

return len(rows)
$$ LANGUAGE plpythonu;
```

Cursors are automatically disposed of. But if you want to explicitly release all resources held by a cursor, use the `close` method. Once closed, a cursor cannot be fetched from anymore.

> **Tip:** Do not confuse objects created by `plpy.cursor` with DB-API cursors as defined by the Python Database API specification[6]. They don't have anything in common except for the name.

43.7.2. Trapping Errors

Functions accessing the database might encounter errors, which will cause them to abort and raise an exception. Both `plpy.execute` and `plpy.prepare` can raise an instance of a subclass of `plpy.SPIError`, which by default will terminate the function. This error can be handled just like any other Python exception, by using the `try/except` construct. For example:

```
CREATE FUNCTION try_adding_joe() RETURNS text AS $$
    try:
        plpy.execute("INSERT INTO users(username) VALUES ('joe')")
    except plpy.SPIError:
```

6. http://www.python.org/dev/peps/pep-0249/

```
        return "something went wrong"
    else:
        return "Joe added"
$$ LANGUAGE plpythonu;
```

The actual class of the exception being raised corresponds to the specific condition that caused the error. Refer to Table A-1 for a list of possible conditions. The module `plpy.spiexceptions` defines an exception class for each PostgreSQL condition, deriving their names from the condition name. For instance, `division_by_zero` becomes `DivisionByZero`, `unique_violation` becomes `UniqueViolation`, `fdw_error` becomes `FdwError`, and so on. Each of these exception classes inherits from `SPIError`. This separation makes it easier to handle specific errors, for instance:

```
CREATE FUNCTION insert_fraction(numerator int, denominator int) RETURNS text AS $$
from plpy import spiexceptions
try:
    plan = plpy.prepare("INSERT INTO fractions (frac) VALUES ($1 / $2)", ["int", "int"])
    plpy.execute(plan, [numerator, denominator])
except spiexceptions.DivisionByZero:
    return "denominator cannot equal zero"
except spiexceptions.UniqueViolation:
    return "already have that fraction"
except plpy.SPIError, e:
    return "other error, SQLSTATE %s" % e.sqlstate
else:
    return "fraction inserted"
$$ LANGUAGE plpythonu;
```

Note that because all exceptions from the `plpy.spiexceptions` module inherit from `SPIError`, an `except` clause handling it will catch any database access error.

As an alternative way of handling different error conditions, you can catch the `SPIError` exception and determine the specific error condition inside the `except` block by looking at the `sqlstate` attribute of the exception object. This attribute is a string value containing the "SQLSTATE" error code. This approach provides approximately the same functionality

43.8. Explicit Subtransactions

Recovering from errors caused by database access as described in Section 43.7.2 can lead to an undesirable situation where some operations succeed before one of them fails, and after recovering from that error the data is left in an inconsistent state. PL/Python offers a solution to this problem in the form of explicit subtransactions.

43.8.1. Subtransaction Context Managers

Consider a function that implements a transfer between two accounts:

```
CREATE FUNCTION transfer_funds() RETURNS void AS $$
```

```
try:
    plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe'")
    plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mary'")
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

If the second `UPDATE` statement results in an exception being raised, this function will report the error, but the result of the first `UPDATE` will nevertheless be committed. In other words, the funds will be withdrawn from Joe's account, but will not be transferred to Mary's account.

To avoid such issues, you can wrap your `plpy.execute` calls in an explicit subtransaction. The `plpy` module provides a helper object to manage explicit subtransactions that gets created with the `plpy.subtransaction()` function. Objects created by this function implement the context manager interface[7]. Using explicit subtransactions we can rewrite our function as:

```
CREATE FUNCTION transfer_funds2() RETURNS void AS $$
try:
    with plpy.subtransaction():
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mar
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"
plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

Note that the use of `try/catch` is still required. Otherwise the exception would propagate to the top of the Python stack and would cause the whole function to abort with a PostgreSQL error, so that the `operations` table would not have any row inserted into it. The subtransaction context manager does not trap errors, it only assures that all database operations executed inside its scope will be atomically committed or rolled back. A rollback of the subtransaction block occurs on any kind of exception exit, not only ones caused by errors originating from database access. A regular Python exception raised inside an explicit subtransaction block would also cause the subtransaction to be rolled back.

43.8.2. Older Python Versions

Context managers syntax using the `with` keyword is available by default in Python 2.6. If using PL/Python with an older Python version, it is still possible to use explicit subtransactions, although not as transparently. You can call the subtransaction manager's `__enter__` and `__exit__` functions using the `enter` and `exit` convenience aliases. The example function that transfers funds could be written as:

```
CREATE FUNCTION transfer_funds_old() RETURNS void AS $$
```

7. http://docs.python.org/library/stdtypes.html#context-manager-types

```
try:
    subxact = plpy.subtransaction()
    subxact.enter()
    try:
        plpy.execute("UPDATE accounts SET balance = balance - 100 WHERE account_name = 'joe
        plpy.execute("UPDATE accounts SET balance = balance + 100 WHERE account_name = 'mar
    except:
        import sys
        subxact.exit(*sys.exc_info())
        raise
    else:
        subxact.exit(None, None, None)
except plpy.SPIError, e:
    result = "error transferring funds: %s" % e.args
else:
    result = "funds transferred correctly"

plan = plpy.prepare("INSERT INTO operations (result) VALUES ($1)", ["text"])
plpy.execute(plan, [result])
$$ LANGUAGE plpythonu;
```

Note: Although context managers were implemented in Python 2.5, to use the `with` syntax in that version you need to use a future statement[8]. Because of implementation details, however, you cannot use future statements in PL/Python functions.

43.9. Utility Functions

The `plpy` module also provides the functions `plpy.debug(msg)`, `plpy.log(msg)`, `plpy.info(msg)`, `plpy.notice(msg)`, `plpy.warning(msg)`, `plpy.error(msg)`, and `plpy.fatal(msg)`. `plpy.error` and `plpy.fatal` actually raise a Python exception which, if uncaught, propagates out to the calling query, causing the current transaction or subtransaction to be aborted. `raise plpy.Error(msg)` and `raise plpy.Fatal(msg)` are equivalent to calling `plpy.error` and `plpy.fatal`, respectively. The other functions only generate messages of different priority levels. Whether messages of a particular priority are reported to the client, written to the server log, or both is controlled by the log_min_messages and client_min_messages configuration variables. See Chapter 18 for more information.

Another set of utility functions are `plpy.quote_literal(string)`, `plpy.quote_nullable(string)`, and `plpy.quote_ident(string)`. They are equivalent to the built-in quoting functions described in Section 9.4. They are useful when constructing ad-hoc queries. A PL/Python equivalent of dynamic SQL from Example 40-1 would be:

```
plpy.execute("UPDATE tbl SET %s = %s WHERE key = %s" % (
    plpy.quote_ident(colname),
```

8. http://docs.python.org/release/2.5/ref/future.html

```
    plpy.quote_nullable(newvalue),
    plpy.quote_literal(keyvalue)))
```

43.10. Environment Variables

Some of the environment variables that are accepted by the Python interpreter can also be used to affect PL/Python behavior. They would need to be set in the environment of the main PostgreSQL server process, for example in a start script. The available environment variables depend on the version of Python; see the Python documentation for details. At the time of this writing, the following environment variables have an affect on PL/Python, assuming an adequate Python version:

- `PYTHONHOME`
- `PYTHONPATH`
- `PYTHONY2K`
- `PYTHONOPTIMIZE`
- `PYTHONDEBUG`
- `PYTHONVERBOSE`
- `PYTHONCASEOK`
- `PYTHONDONTWRITEBYTECODE`
- `PYTHONIOENCODING`
- `PYTHONUSERBASE`
- `PYTHONHASHSEED`

(It appears to be a Python implementation detail beyond the control of PL/Python that some of the environment variables listed on the `python` man page are only effective in a command-line interpreter and not an embedded Python interpreter.)

Chapter 44. Server Programming Interface

The *Server Programming Interface* (SPI) gives writers of user-defined C functions the ability to run SQL commands inside their functions. SPI is a set of interface functions to simplify access to the parser, planner, and executor. SPI also does some memory management.

> **Note:** The available procedural languages provide various means to execute SQL commands from procedures. Most of these facilities are based on SPI, so this documentation might be of use for users of those languages as well.

To avoid misunderstanding we'll use the term "function" when we speak of SPI interface functions and "procedure" for a user-defined C-function that is using SPI.

Note that if a command invoked via SPI fails, then control will not be returned to your procedure. Rather, the transaction or subtransaction in which your procedure executes will be rolled back. (This might seem surprising given that the SPI functions mostly have documented error-return conventions. Those conventions only apply for errors detected within the SPI functions themselves, however.) It is possible to recover control after an error by establishing your own subtransaction surrounding SPI calls that might fail.

SPI functions return a nonnegative result on success (either via a returned integer value or in the global variable `SPI_result`, as described below). On error, a negative result or `NULL` will be returned.

Source code files that use SPI must include the header file `executor/spi.h`.

44.1. Interface Functions

SPI_connect

Name

`SPI_connect` — connect a procedure to the SPI manager

Synopsis

```
int SPI_connect(void)
```

Description

`SPI_connect` opens a connection from a procedure invocation to the SPI manager. You must call this function if you want to execute commands through SPI. Some utility SPI functions can be called from unconnected procedures.

If your procedure is already connected, `SPI_connect` will return the error code SPI_ERROR_CONNECT. This could happen if a procedure that has called `SPI_connect` directly calls another procedure that calls `SPI_connect`. While recursive calls to the SPI manager are permitted when an SQL command called through SPI invokes another function that uses SPI, directly nested calls to `SPI_connect` and `SPI_finish` are forbidden. (But see `SPI_push` and `SPI_pop`.)

Return Value

`SPI_OK_CONNECT`

on success

`SPI_ERROR_CONNECT`

on error

SPI_finish

Name

SPI_finish — disconnect a procedure from the SPI manager

Synopsis

```
int SPI_finish(void)
```

Description

SPI_finish closes an existing connection to the SPI manager. You must call this function after completing the SPI operations needed during your procedure's current invocation. You do not need to worry about making this happen, however, if you abort the transaction via elog(ERROR). In that case SPI will clean itself up automatically.

If SPI_finish is called without having a valid connection, it will return SPI_ERROR_UNCONNECTED. There is no fundamental problem with this; it means that the SPI manager has nothing to do.

Return Value

SPI_OK_FINISH

: if properly disconnected

SPI_ERROR_UNCONNECTED

: if called from an unconnected procedure

SPI_push

Name

SPI_push — push SPI stack to allow recursive SPI usage

Synopsis

```
void SPI_push(void)
```

Description

SPI_push should be called before executing another procedure that might itself wish to use SPI. After SPI_push, SPI is no longer in a "connected" state, and SPI function calls will be rejected unless a fresh SPI_connect is done. This ensures a clean separation between your procedure's SPI state and that of another procedure you call. After the other procedure returns, call SPI_pop to restore access to your own SPI state.

Note that SPI_execute and related functions automatically do the equivalent of SPI_push before passing control back to the SQL execution engine, so it is not necessary for you to worry about this when using those functions. Only when you are directly calling arbitrary code that might contain SPI_connect calls do you need to issue SPI_push and SPI_pop.

SPI_pop

Name

`SPI_pop` — pop SPI stack to return from recursive SPI usage

Synopsis

```
void SPI_pop(void)
```

Description

`SPI_pop` pops the previous environment from the SPI call stack. See `SPI_push`.

SPI_execute

Name

SPI_execute — execute a command

Synopsis

```
int SPI_execute(const char * command, bool read_only, long count)
```

Description

SPI_execute executes the specified SQL command for count rows. If read_only is true, the command must be read-only, and execution overhead is somewhat reduced.

This function can only be called from a connected procedure.

If count is zero then the command is executed for all rows that it applies to. If count is greater than zero, then no more than count rows will be retrieved; execution stops when the count is reached, much like adding a LIMIT clause to the query. For example,

```
SPI_execute("SELECT * FROM foo", true, 5);
```

will retrieve at most 5 rows from the table. Note that such a limit is only effective when the command actually returns rows. For example,

```
SPI_execute("INSERT INTO foo SELECT * FROM bar", false, 5);
```

inserts all rows from bar, ignoring the count parameter. However, with

```
SPI_execute("INSERT INTO foo SELECT * FROM bar RETURNING *", false, 5);
```

at most 5 rows would be inserted, since execution would stop after the fifth RETURNING result row is retrieved.

You can pass multiple commands in one string; SPI_execute returns the result for the command executed last. The count limit applies to each command separately (even though only the last result will actually be returned). The limit is not applied to any hidden commands generated by rules.

When read_only is false, SPI_execute increments the command counter and computes a new *snapshot* before executing each command in the string. The snapshot does not actually change if the current transaction isolation level is SERIALIZABLE or REPEATABLE READ, but in READ COMMITTED mode the snapshot update allows each command to see the results of newly committed transactions from other sessions. This is essential for consistent behavior when the commands are modifying the database.

When read_only is true, SPI_execute does not update either the snapshot or the command counter, and it allows only plain SELECT commands to appear in the command string. The commands are executed using the snapshot previously established for the surrounding query. This execution mode is somewhat faster than the read/write mode due to eliminating per-command overhead. It also allows genuinely *stable*

functions to be built: since successive executions will all use the same snapshot, there will be no change in the results.

It is generally unwise to mix read-only and read-write commands within a single function using SPI; that could result in very confusing behavior, since the read-only queries would not see the results of any database updates done by the read-write queries.

The actual number of rows for which the (last) command was executed is returned in the global variable `SPI_processed`. If the return value of the function is `SPI_OK_SELECT`, `SPI_OK_INSERT_RETURNING`, `SPI_OK_DELETE_RETURNING`, or `SPI_OK_UPDATE_RETURNING`, then you can use the global pointer `SPITupleTable *SPI_tuptable` to access the result rows. Some utility commands (such as `EXPLAIN`) also return row sets, and `SPI_tuptable` will contain the result in these cases too. Some utility commands (`COPY`, `CREATE TABLE AS`) don't return a row set, so `SPI_tuptable` is NULL, but they still return the number of rows processed in `SPI_processed`.

The structure `SPITupleTable` is defined thus:

```
typedef struct
{
    MemoryContext tuptabcxt;    /* memory context of result table */
    uint32      alloced;        /* number of alloced vals */
    uint32      free;           /* number of free vals */
    TupleDesc   tupdesc;        /* row descriptor */
    HeapTuple  *vals;           /* rows */
} SPITupleTable;
```

`vals` is an array of pointers to rows. (The number of valid entries is given by `SPI_processed`.) `tupdesc` is a row descriptor which you can pass to SPI functions dealing with rows. `tuptabcxt`, `alloced`, and `free` are internal fields not intended for use by SPI callers.

`SPI_finish` frees all `SPITupleTable`s allocated during the current procedure. You can free a particular result table earlier, if you are done with it, by calling `SPI_freetuptable`.

Arguments

`const char * command`

string containing command to execute

`bool read_only`

`true` for read-only execution

`long count`

maximum number of rows to return, or `0` for no limit

Return Value

If the execution of the command was successful then one of the following (nonnegative) values will be returned:

`SPI_OK_SELECT`

if a `SELECT` (but not `SELECT INTO`) was executed

`SPI_OK_SELINTO`

if a `SELECT INTO` was executed

`SPI_OK_INSERT`

if an `INSERT` was executed

`SPI_OK_DELETE`

if a `DELETE` was executed

`SPI_OK_UPDATE`

if an `UPDATE` was executed

`SPI_OK_INSERT_RETURNING`

if an `INSERT RETURNING` was executed

`SPI_OK_DELETE_RETURNING`

if a `DELETE RETURNING` was executed

`SPI_OK_UPDATE_RETURNING`

if an `UPDATE RETURNING` was executed

`SPI_OK_UTILITY`

if a utility command (e.g., `CREATE TABLE`) was executed

`SPI_OK_REWRITTEN`

if the command was rewritten into another kind of command (e.g., `UPDATE` became an `INSERT`) by a rule.

On error, one of the following negative values is returned:

`SPI_ERROR_ARGUMENT`

if `command` is `NULL` or `count` is less than 0

`SPI_ERROR_COPY`

if `COPY TO stdout` or `COPY FROM stdin` was attempted

`SPI_ERROR_TRANSACTION`

if a transaction manipulation command was attempted (`BEGIN`, `COMMIT`, `ROLLBACK`, `SAVEPOINT`, `PREPARE TRANSACTION`, `COMMIT PREPARED`, `ROLLBACK PREPARED`, or any variant thereof)

`SPI_ERROR_OPUNKNOWN`

if the command type is unknown (shouldn't happen)

`SPI_ERROR_UNCONNECTED`

if called from an unconnected procedure

Notes

All SPI query-execution functions set both `SPI_processed` and `SPI_tuptable` (just the pointer, not the contents of the structure). Save these two global variables into local procedure variables if you need to access the result table of `SPI_execute` or another query-execution function across later calls.

SPI_exec

Name

SPI_exec — execute a read/write command

Synopsis

```
int SPI_exec(const char * command, long count)
```

Description

SPI_exec is the same as SPI_execute, with the latter's read_only parameter always taken as false.

Arguments

const char * command

string containing command to execute

long count

maximum number of rows to return, or 0 for no limit

Return Value

See SPI_execute.

SPI_execute_with_args

Name

SPI_execute_with_args — execute a command with out-of-line parameters

Synopsis

```
int SPI_execute_with_args(const char *command,
                          int nargs, Oid *argtypes,
                          Datum *values, const char *nulls,
                          bool read_only, long count)
```

Description

SPI_execute_with_args executes a command that might include references to externally supplied parameters. The command text refers to a parameter as $*n*, and the call specifies data types and values for each such symbol. read_only and count have the same interpretation as in SPI_execute.

The main advantage of this routine compared to SPI_execute is that data values can be inserted into the command without tedious quoting/escaping, and thus with much less risk of SQL-injection attacks.

Similar results can be achieved with SPI_prepare followed by SPI_execute_plan; however, when using this function the query plan is always customized to the specific parameter values provided. For one-time query execution, this function should be preferred. If the same command is to be executed with many different parameters, either method might be faster, depending on the cost of re-planning versus the benefit of custom plans.

Arguments

const char * command

command string

int nargs

number of input parameters ($1, $2, etc.)

Oid * argtypes

an array of length nargs, containing the OIDs of the data types of the parameters

Datum * values

an array of length nargs, containing the actual parameter values

const char * nulls

an array of length nargs, describing which parameters are null

If `nulls` is `NULL` then `SPI_execute_with_args` assumes that no parameters are null. Otherwise, each entry of the `nulls` array should be `' '` if the corresponding parameter value is non-null, or `'n'` if the corresponding parameter value is null. (In the latter case, the actual value in the corresponding `values` entry doesn't matter.) Note that `nulls` is not a text string, just an array: it does not need a `'\0'` terminator.

`bool read_only`

`true` for read-only execution

`long count`

maximum number of rows to return, or `0` for no limit

Return Value

The return value is the same as for `SPI_execute`.

`SPI_processed` and `SPI_tuptable` are set as in `SPI_execute` if successful.

SPI_prepare

Name

SPI_prepare — prepare a statement, without executing it yet

Synopsis

```
SPIPlanPtr SPI_prepare(const char * command, int nargs, Oid * argtypes)
```

Description

SPI_prepare creates and returns a prepared statement for the specified command, but doesn't execute the command. The prepared statement can later be executed repeatedly using SPI_execute_plan.

When the same or a similar command is to be executed repeatedly, it is generally advantageous to perform parse analysis only once, and might furthermore be advantageous to re-use an execution plan for the command. SPI_prepare converts a command string into a prepared statement that encapsulates the results of parse analysis. The prepared statement also provides a place for caching an execution plan if it is found that generating a custom plan for each execution is not helpful.

A prepared command can be generalized by writing parameters ($1, $2, etc.) in place of what would be constants in a normal command. The actual values of the parameters are then specified when SPI_execute_plan is called. This allows the prepared command to be used over a wider range of situations than would be possible without parameters.

The statement returned by SPI_prepare can be used only in the current invocation of the procedure, since SPI_finish frees memory allocated for such a statement. But the statement can be saved for longer using the functions SPI_keepplan or SPI_saveplan.

Arguments

`const char * command`

command string

`int nargs`

number of input parameters ($1, $2, etc.)

`Oid * argtypes`

pointer to an array containing the OIDs of the data types of the parameters

Return Value

`SPI_prepare` returns a non-null pointer to an `SPIPlan`, which is an opaque struct representing a prepared statement. On error, `NULL` will be returned, and `SPI_result` will be set to one of the same error codes used by `SPI_execute`, except that it is set to `SPI_ERROR_ARGUMENT` if `command` is `NULL`, or if `nargs` is less than 0, or if `nargs` is greater than 0 and `argtypes` is `NULL`.

Notes

If no parameters are defined, a generic plan will be created at the first use of `SPI_execute_plan`, and used for all subsequent executions as well. If there are parameters, the first few uses of `SPI_execute_plan` will generate custom plans that are specific to the supplied parameter values. After enough uses of the same prepared statement, `SPI_execute_plan` will build a generic plan, and if that is not too much more expensive than the custom plans, it will start using the generic plan instead of re-planning each time. If this default behavior is unsuitable, you can alter it by passing the `CURSOR_OPT_GENERIC_PLAN` or `CURSOR_OPT_CUSTOM_PLAN` flag to `SPI_prepare_cursor`, to force use of generic or custom plans respectively.

Although the main point of a prepared statement is to avoid repeated parse analysis and planning of the statement, PostgreSQL will force re-analysis and re-planning of the statement before using it whenever database objects used in the statement have undergone definitional (DDL) changes since the previous use of the prepared statement. Also, if the value of search_path changes from one use to the next, the statement will be re-parsed using the new `search_path`. (This latter behavior is new as of PostgreSQL 9.3.) See PREPARE for more information about the behavior of prepared statements.

This function should only be called from a connected procedure.

`SPIPlanPtr` is declared as a pointer to an opaque struct type in `spi.h`. It is unwise to try to access its contents directly, as that makes your code much more likely to break in future revisions of PostgreSQL.

The name `SPIPlanPtr` is somewhat historical, since the data structure no longer necessarily contains an execution plan.

SPI_prepare_cursor

Name

SPI_prepare_cursor — prepare a statement, without executing it yet

Synopsis

```
SPIPlanPtr SPI_prepare_cursor(const char * command, int nargs,
                              Oid * argtypes, int cursorOptions)
```

Description

SPI_prepare_cursor is identical to SPI_prepare, except that it also allows specification of the planner's "cursor options" parameter. This is a bit mask having the values shown in nodes/parsenodes.h for the options field of DeclareCursorStmt. SPI_prepare always takes the cursor options as zero.

Arguments

const char * command

command string

int nargs

number of input parameters ($1, $2, etc.)

Oid * argtypes

pointer to an array containing the OIDs of the data types of the parameters

int cursorOptions

integer bit mask of cursor options; zero produces default behavior

Return Value

SPI_prepare_cursor has the same return conventions as SPI_prepare.

Notes

Useful bits to set in cursorOptions include CURSOR_OPT_SCROLL, CURSOR_OPT_NO_SCROLL, CURSOR_OPT_FAST_PLAN, CURSOR_OPT_GENERIC_PLAN, and CURSOR_OPT_CUSTOM_PLAN. Note in particular that CURSOR_OPT_HOLD is ignored.

SPI_prepare_params

Name

SPI_prepare_params — prepare a statement, without executing it yet

Synopsis

```
SPIPlanPtr SPI_prepare_params(const char * command,
                              ParserSetupHook parserSetup,
                              void * parserSetupArg,
                              int cursorOptions)
```

Description

SPI_prepare_params creates and returns a prepared statement for the specified command, but doesn't execute the command. This function is equivalent to SPI_prepare_cursor, with the addition that the caller can specify parser hook functions to control the parsing of external parameter references.

Arguments

const char * command

command string

ParserSetupHook parserSetup

Parser hook setup function

void * parserSetupArg

pass-through argument for parserSetup

int cursorOptions

integer bit mask of cursor options; zero produces default behavior

Return Value

SPI_prepare_params has the same return conventions as SPI_prepare.

SPI_getargcount

Name

SPI_getargcount — return the number of arguments needed by a statement prepared by SPI_prepare

Synopsis

```
int SPI_getargcount(SPIPlanPtr plan)
```

Description

SPI_getargcount returns the number of arguments needed to execute a statement prepared by SPI_prepare.

Arguments

SPIPlanPtr plan

prepared statement (returned by SPI_prepare)

Return Value

The count of expected arguments for the plan. If the plan is NULL or invalid, SPI_result is set to SPI_ERROR_ARGUMENT and -1 is returned.

SPI_getargtypeid

Name

SPI_getargtypeid — return the data type OID for an argument of a statement prepared by SPI_prepare

Synopsis

```
Oid SPI_getargtypeid(SPIPlanPtr plan, int argIndex)
```

Description

SPI_getargtypeid returns the OID representing the type for the argIndex'th argument of a statement prepared by SPI_prepare. First argument is at index zero.

Arguments

SPIPlanPtr plan

prepared statement (returned by SPI_prepare)

int argIndex

zero based index of the argument

Return Value

The type OID of the argument at the given index. If the plan is NULL or invalid, or argIndex is less than 0 or not less than the number of arguments declared for the plan, SPI_result is set to SPI_ERROR_ARGUMENT and InvalidOid is returned.

SPI_is_cursor_plan

Name

SPI_is_cursor_plan — return true if a statement prepared by SPI_prepare can be used with SPI_cursor_open

Synopsis

```
bool SPI_is_cursor_plan(SPIPlanPtr plan)
```

Description

SPI_is_cursor_plan returns true if a statement prepared by SPI_prepare can be passed as an argument to SPI_cursor_open, or false if that is not the case. The criteria are that the plan represents one single command and that this command returns tuples to the caller; for example, SELECT is allowed unless it contains an INTO clause, and UPDATE is allowed only if it contains a RETURNING clause.

Arguments

SPIPlanPtr plan

prepared statement (returned by SPI_prepare)

Return Value

true or false to indicate if the plan can produce a cursor or not, with SPI_result set to zero. If it is not possible to determine the answer (for example, if the plan is NULL or invalid, or if called when not connected to SPI), then SPI_result is set to a suitable error code and false is returned.

SPI_execute_plan

Name

SPI_execute_plan — execute a statement prepared by SPI_prepare

Synopsis

```
int SPI_execute_plan(SPIPlanPtr plan, Datum * values, const char * nulls,
                     bool read_only, long count)
```

Description

SPI_execute_plan executes a statement prepared by SPI_prepare or one of its siblings. read_only and count have the same interpretation as in SPI_execute.

Arguments

SPIPlanPtr plan

prepared statement (returned by SPI_prepare)

Datum * values

An array of actual parameter values. Must have same length as the statement's number of arguments.

const char * nulls

An array describing which parameters are null. Must have same length as the statement's number of arguments.

If nulls is NULL then SPI_execute_plan assumes that no parameters are null. Otherwise, each entry of the nulls array should be ' ' if the corresponding parameter value is non-null, or 'n' if the corresponding parameter value is null. (In the latter case, the actual value in the corresponding values entry doesn't matter.) Note that nulls is not a text string, just an array: it does not need a '\0' terminator.

bool read_only

true for read-only execution

long count

maximum number of rows to return, or 0 for no limit

Return Value

The return value is the same as for `SPI_execute`, with the following additional possible error (negative) results:

`SPI_ERROR_ARGUMENT`

if `plan` is `NULL` or invalid, or `count` is less than 0

`SPI_ERROR_PARAM`

if `values` is `NULL` and `plan` was prepared with some parameters

`SPI_processed` and `SPI_tuptable` are set as in `SPI_execute` if successful.

SPI_execute_plan_with_paramlist

Name

SPI_execute_plan_with_paramlist — execute a statement prepared by SPI_prepare

Synopsis

```
int SPI_execute_plan_with_paramlist(SPIPlanPtr plan,
                                    ParamListInfo params,
                                    bool read_only,
                                    long count)
```

Description

SPI_execute_plan_with_paramlist executes a statement prepared by SPI_prepare. This function is equivalent to SPI_execute_plan except that information about the parameter values to be passed to the query is presented differently. The ParamListInfo representation can be convenient for passing down values that are already available in that format. It also supports use of dynamic parameter sets via hook functions specified in ParamListInfo.

Arguments

SPIPlanPtr plan

prepared statement (returned by SPI_prepare)

ParamListInfo params

data structure containing parameter types and values; NULL if none

bool read_only

true for read-only execution

long count

maximum number of rows to return, or 0 for no limit

Return Value

The return value is the same as for SPI_execute_plan.

SPI_processed and SPI_tuptable are set as in SPI_execute_plan if successful.

SPI_execp

Name

SPI_execp — execute a statement in read/write mode

Synopsis

```
int SPI_execp(SPIPlanPtr plan, Datum * values, const char * nulls, long count)
```

Description

SPI_execp is the same as SPI_execute_plan, with the latter's read_only parameter always taken as false.

Arguments

SPIPlanPtr plan

prepared statement (returned by SPI_prepare)

Datum * values

An array of actual parameter values. Must have same length as the statement's number of arguments.

const char * nulls

An array describing which parameters are null. Must have same length as the statement's number of arguments.

If nulls is NULL then SPI_execp assumes that no parameters are null. Otherwise, each entry of the nulls array should be ' ' if the corresponding parameter value is non-null, or 'n' if the corresponding parameter value is null. (In the latter case, the actual value in the corresponding values entry doesn't matter.) Note that nulls is not a text string, just an array: it does not need a '\0' terminator.

long count

maximum number of rows to return, or 0 for no limit

Return Value

See SPI_execute_plan.

SPI_processed and SPI_tuptable are set as in SPI_execute if successful.

SPI_cursor_open

Name

SPI_cursor_open — set up a cursor using a statement created with SPI_prepare

Synopsis

```
Portal SPI_cursor_open(const char * name, SPIPlanPtr plan,
                       Datum * values, const char * nulls,
                       bool read_only)
```

Description

SPI_cursor_open sets up a cursor (internally, a portal) that will execute a statement prepared by SPI_prepare. The parameters have the same meanings as the corresponding parameters to SPI_execute_plan.

Using a cursor instead of executing the statement directly has two benefits. First, the result rows can be retrieved a few at a time, avoiding memory overrun for queries that return many rows. Second, a portal can outlive the current procedure (it can, in fact, live to the end of the current transaction). Returning the portal name to the procedure's caller provides a way of returning a row set as result.

The passed-in parameter data will be copied into the cursor's portal, so it can be freed while the cursor still exists.

Arguments

`const char * name`

name for portal, or NULL to let the system select a name

`SPIPlanPtr plan`

prepared statement (returned by SPI_prepare)

`Datum * values`

An array of actual parameter values. Must have same length as the statement's number of arguments.

`const char * nulls`

An array describing which parameters are null. Must have same length as the statement's number of arguments.

If nulls is NULL then SPI_cursor_open assumes that no parameters are null. Otherwise, each entry of the nulls array should be `' '` if the corresponding parameter value is non-null, or `'n'` if the corresponding parameter value is null. (In the latter case, the actual value in the corresponding values entry doesn't matter.) Note that nulls is not a text string, just an array: it does not need a `'\0'` terminator.

`bool read_only`

`true` for read-only execution

Return Value

Pointer to portal containing the cursor. Note there is no error return convention; any error will be reported via `elog`.

SPI_cursor_open_with_args

Name

SPI_cursor_open_with_args — set up a cursor using a query and parameters

Synopsis

```
Portal SPI_cursor_open_with_args(const char *name,
                                 const char *command,
                                 int nargs, Oid *argtypes,
                                 Datum *values, const char *nulls,
                                 bool read_only, int cursorOptions)
```

Description

SPI_cursor_open_with_args sets up a cursor (internally, a portal) that will execute the specified query. Most of the parameters have the same meanings as the corresponding parameters to SPI_prepare_cursor and SPI_cursor_open.

For one-time query execution, this function should be preferred over SPI_prepare_cursor followed by SPI_cursor_open. If the same command is to be executed with many different parameters, either method might be faster, depending on the cost of re-planning versus the benefit of custom plans.

The passed-in parameter data will be copied into the cursor's portal, so it can be freed while the cursor still exists.

Arguments

`const char * name`

name for portal, or NULL to let the system select a name

`const char * command`

command string

`int nargs`

number of input parameters ($1, $2, etc.)

`Oid * argtypes`

an array of length nargs, containing the OIDs of the data types of the parameters

`Datum * values`

an array of length nargs, containing the actual parameter values

`const char * nulls`

an array of length `nargs`, describing which parameters are null

If `nulls` is `NULL` then `SPI_cursor_open_with_args` assumes that no parameters are null. Otherwise, each entry of the `nulls` array should be `' '` if the corresponding parameter value is non-null, or `'n'` if the corresponding parameter value is null. (In the latter case, the actual value in the corresponding `values` entry doesn't matter.) Note that `nulls` is not a text string, just an array: it does not need a `'\0'` terminator.

`bool read_only`

`true` for read-only execution

`int cursorOptions`

integer bit mask of cursor options; zero produces default behavior

Return Value

Pointer to portal containing the cursor. Note there is no error return convention; any error will be reported via `elog`.

SPI_cursor_open_with_paramlist

Name

SPI_cursor_open_with_paramlist — set up a cursor using parameters

Synopsis

```
Portal SPI_cursor_open_with_paramlist(const char *name,
                                      SPIPlanPtr plan,
                                      ParamListInfo params,
                                      bool read_only)
```

Description

SPI_cursor_open_with_paramlist sets up a cursor (internally, a portal) that will execute a statement prepared by SPI_prepare. This function is equivalent to SPI_cursor_open except that information about the parameter values to be passed to the query is presented differently. The ParamListInfo representation can be convenient for passing down values that are already available in that format. It also supports use of dynamic parameter sets via hook functions specified in ParamListInfo.

The passed-in parameter data will be copied into the cursor's portal, so it can be freed while the cursor still exists.

Arguments

const char * name

name for portal, or NULL to let the system select a name

SPIPlanPtr plan

prepared statement (returned by SPI_prepare)

ParamListInfo params

data structure containing parameter types and values; NULL if none

bool read_only

true for read-only execution

Return Value

Pointer to portal containing the cursor. Note there is no error return convention; any error will be reported via elog.

SPI_cursor_find

Name

`SPI_cursor_find` — find an existing cursor by name

Synopsis

```
Portal SPI_cursor_find(const char * name)
```

Description

`SPI_cursor_find` finds an existing portal by name. This is primarily useful to resolve a cursor name returned as text by some other function.

Arguments

`const char * name`

name of the portal

Return Value

pointer to the portal with the specified name, or `NULL` if none was found

SPI_cursor_fetch

Name

SPI_cursor_fetch — fetch some rows from a cursor

Synopsis

```
void SPI_cursor_fetch(Portal portal, bool forward, long count)
```

Description

SPI_cursor_fetch fetches some rows from a cursor. This is equivalent to a subset of the SQL command FETCH (see SPI_scroll_cursor_fetch for more functionality).

Arguments

Portal portal

portal containing the cursor

bool forward

true for fetch forward, false for fetch backward

long count

maximum number of rows to fetch

Return Value

SPI_processed and SPI_tuptable are set as in SPI_execute if successful.

Notes

Fetching backward may fail if the cursor's plan was not created with the CURSOR_OPT_SCROLL option.

SPI_cursor_move

Name

`SPI_cursor_move` — move a cursor

Synopsis

```
void SPI_cursor_move(Portal portal, bool forward, long count)
```

Description

`SPI_cursor_move` skips over some number of rows in a cursor. This is equivalent to a subset of the SQL command `MOVE` (see `SPI_scroll_cursor_move` for more functionality).

Arguments

`Portal portal`

portal containing the cursor

`bool forward`

true for move forward, false for move backward

`long count`

maximum number of rows to move

Notes

Moving backward may fail if the cursor's plan was not created with the `CURSOR_OPT_SCROLL` option.

SPI_scroll_cursor_fetch

Name

SPI_scroll_cursor_fetch — fetch some rows from a cursor

Synopsis

```
void SPI_scroll_cursor_fetch(Portal portal, FetchDirection direction,
                             long count)
```

Description

SPI_scroll_cursor_fetch fetches some rows from a cursor. This is equivalent to the SQL command FETCH.

Arguments

Portal portal

portal containing the cursor

FetchDirection direction

one of FETCH_FORWARD, FETCH_BACKWARD, FETCH_ABSOLUTE or FETCH_RELATIVE

long count

number of rows to fetch for FETCH_FORWARD or FETCH_BACKWARD; absolute row number to fetch for FETCH_ABSOLUTE; or relative row number to fetch for FETCH_RELATIVE

Return Value

SPI_processed and SPI_tuptable are set as in SPI_execute if successful.

Notes

See the SQL FETCH command for details of the interpretation of the direction and count parameters.

Direction values other than FETCH_FORWARD may fail if the cursor's plan was not created with the CURSOR_OPT_SCROLL option.

SPI_scroll_cursor_move

Name

SPI_scroll_cursor_move — move a cursor

Synopsis

```
void SPI_scroll_cursor_move(Portal portal, FetchDirection direction,
                            long count)
```

Description

SPI_scroll_cursor_move skips over some number of rows in a cursor. This is equivalent to the SQL command MOVE.

Arguments

Portal portal

portal containing the cursor

FetchDirection direction

one of FETCH_FORWARD, FETCH_BACKWARD, FETCH_ABSOLUTE or FETCH_RELATIVE

long count

number of rows to move for FETCH_FORWARD or FETCH_BACKWARD; absolute row number to move to for FETCH_ABSOLUTE; or relative row number to move to for FETCH_RELATIVE

Return Value

SPI_processed is set as in SPI_execute if successful. SPI_tuptable is set to NULL, since no rows are returned by this function.

Notes

See the SQL FETCH command for details of the interpretation of the direction and count parameters.

Direction values other than FETCH_FORWARD may fail if the cursor's plan was not created with the CURSOR_OPT_SCROLL option.

SPI_cursor_close

Name

`SPI_cursor_close` — close a cursor

Synopsis

```
void SPI_cursor_close(Portal portal)
```

Description

`SPI_cursor_close` closes a previously created cursor and releases its portal storage.

All open cursors are closed automatically at the end of a transaction. `SPI_cursor_close` need only be invoked if it is desirable to release resources sooner.

Arguments

`Portal portal`

portal containing the cursor

SPI_keepplan

Name

SPI_keepplan — save a prepared statement

Synopsis

```
int SPI_keepplan(SPIPlanPtr plan)
```

Description

SPI_keepplan saves a passed statement (prepared by SPI_prepare) so that it will not be freed by SPI_finish nor by the transaction manager. This gives you the ability to reuse prepared statements in the subsequent invocations of your procedure in the current session.

Arguments

SPIPlanPtr plan

the prepared statement to be saved

Return Value

0 on success; SPI_ERROR_ARGUMENT if plan is NULL or invalid

Notes

The passed-in statement is relocated to permanent storage by means of pointer adjustment (no data copying is required). If you later wish to delete it, use SPI_freeplan on it.

SPI_saveplan

Name

SPI_saveplan — save a prepared statement

Synopsis

```
SPIPlanPtr SPI_saveplan(SPIPlanPtr plan)
```

Description

SPI_saveplan copies a passed statement (prepared by SPI_prepare) into memory that will not be freed by SPI_finish nor by the transaction manager, and returns a pointer to the copied statement. This gives you the ability to reuse prepared statements in the subsequent invocations of your procedure in the current session.

Arguments

SPIPlanPtr plan

the prepared statement to be saved

Return Value

Pointer to the copied statement; or NULL if unsuccessful. On error, SPI_result is set thus:

SPI_ERROR_ARGUMENT

if plan is NULL or invalid

SPI_ERROR_UNCONNECTED

if called from an unconnected procedure

Notes

The originally passed-in statement is not freed, so you might wish to do SPI_freeplan on it to avoid leaking memory until SPI_finish.

In most cases, SPI_keepplan is preferred to this function, since it accomplishes largely the same result without needing to physically copy the prepared statement's data structures.

44.2. Interface Support Functions

The functions described here provide an interface for extracting information from result sets returned by `SPI_execute` and other SPI functions.

All functions described in this section can be used by both connected and unconnected procedures.

SPI_fname

Name

`SPI_fname` — determine the column name for the specified column number

Synopsis

```
char * SPI_fname(TupleDesc rowdesc, int colnumber)
```

Description

`SPI_fname` returns a copy of the column name of the specified column. (You can use `pfree` to release the copy of the name when you don't need it anymore.)

Arguments

`TupleDesc rowdesc`

input row description

`int colnumber`

column number (count starts at 1)

Return Value

The column name; `NULL` if `colnumber` is out of range. `SPI_result` set to `SPI_ERROR_NOATTRIBUTE` on error.

SPI_fnumber

Name

`SPI_fnumber` — determine the column number for the specified column name

Synopsis

```
int SPI_fnumber(TupleDesc rowdesc, const char * colname)
```

Description

`SPI_fnumber` returns the column number for the column with the specified name.

If `colname` refers to a system column (e.g., `oid`) then the appropriate negative column number will be returned. The caller should be careful to test the return value for exact equality to `SPI_ERROR_NOATTRIBUTE` to detect an error; testing the result for less than or equal to 0 is not correct unless system columns should be rejected.

Arguments

`TupleDesc rowdesc`

input row description

`const char * colname`

column name

Return Value

Column number (count starts at 1), or `SPI_ERROR_NOATTRIBUTE` if the named column was not found.

SPI_getvalue

Name

`SPI_getvalue` — return the string value of the specified column

Synopsis

```
char * SPI_getvalue(HeapTuple row, TupleDesc rowdesc, int colnumber)
```

Description

`SPI_getvalue` returns the string representation of the value of the specified column.

The result is returned in memory allocated using `palloc`. (You can use `pfree` to release the memory when you don't need it anymore.)

Arguments

`HeapTuple row`

input row to be examined

`TupleDesc rowdesc`

input row description

`int colnumber`

column number (count starts at 1)

Return Value

Column value, or `NULL` if the column is null, `colnumber` is out of range (`SPI_result` is set to `SPI_ERROR_NOATTRIBUTE`), or no output function is available (`SPI_result` is set to `SPI_ERROR_NOOUTFUNC`).

SPI_getbinval

Name

SPI_getbinval — return the binary value of the specified column

Synopsis

```
Datum SPI_getbinval(HeapTuple row, TupleDesc rowdesc, int colnumber,
                    bool * isnull)
```

Description

SPI_getbinval returns the value of the specified column in the internal form (as type Datum).

This function does not allocate new space for the datum. In the case of a pass-by-reference data type, the return value will be a pointer into the passed row.

Arguments

HeapTuple row

input row to be examined

TupleDesc rowdesc

input row description

int colnumber

column number (count starts at 1)

bool * isnull

flag for a null value in the column

Return Value

The binary value of the column is returned. The variable pointed to by isnull is set to true if the column is null, else to false.

SPI_result is set to SPI_ERROR_NOATTRIBUTE on error.

SPI_gettype

Name

SPI_gettype — return the data type name of the specified column

Synopsis

```
char * SPI_gettype(TupleDesc rowdesc, int colnumber)
```

Description

SPI_gettype returns a copy of the data type name of the specified column. (You can use pfree to release the copy of the name when you don't need it anymore.)

Arguments

TupleDesc rowdesc

input row description

int colnumber

column number (count starts at 1)

Return Value

The data type name of the specified column, or NULL on error. SPI_result is set to SPI_ERROR_NOATTRIBUTE on error.

SPI_gettypeid

Name

SPI_gettypeid — return the data type OID of the specified column

Synopsis

```
Oid SPI_gettypeid(TupleDesc rowdesc, int colnumber)
```

Description

SPI_gettypeid returns the OID of the data type of the specified column.

Arguments

TupleDesc rowdesc

input row description

int colnumber

column number (count starts at 1)

Return Value

The OID of the data type of the specified column or InvalidOid on error. On error, SPI_result is set to SPI_ERROR_NOATTRIBUTE.

SPI_getrelname

Name

`SPI_getrelname` — return the name of the specified relation

Synopsis

```
char * SPI_getrelname(Relation rel)
```

Description

`SPI_getrelname` returns a copy of the name of the specified relation. (You can use `pfree` to release the copy of the name when you don't need it anymore.)

Arguments

`Relation rel`

input relation

Return Value

The name of the specified relation.

SPI_getnspname

Name

`SPI_getnspname` — return the namespace of the specified relation

Synopsis

```
char * SPI_getnspname(Relation rel)
```

Description

`SPI_getnspname` returns a copy of the name of the namespace that the specified `Relation` belongs to. This is equivalent to the relation's schema. You should `pfree` the return value of this function when you are finished with it.

Arguments

`Relation rel`

input relation

Return Value

The name of the specified relation's namespace.

44.3. Memory Management

PostgreSQL allocates memory within *memory contexts*, which provide a convenient method of managing allocations made in many different places that need to live for differing amounts of time. Destroying a context releases all the memory that was allocated in it. Thus, it is not necessary to keep track of individual objects to avoid memory leaks; instead only a relatively small number of contexts have to be managed. `palloc` and related functions allocate memory from the "current" context.

`SPI_connect` creates a new memory context and makes it current. `SPI_finish` restores the previous current memory context and destroys the context created by `SPI_connect`. These actions ensure that transient memory allocations made inside your procedure are reclaimed at procedure exit, avoiding memory leakage.

However, if your procedure needs to return an object in allocated memory (such as a value of a pass-by-reference data type), you cannot allocate that memory using `palloc`, at least not while you are connected to SPI. If you try, the object will be deallocated by `SPI_finish`, and your procedure will not work reliably. To solve this problem, use `SPI_palloc` to allocate memory for your return object. `SPI_palloc` allocates memory in the "upper executor context", that is, the memory context that was current when `SPI_connect` was called, which is precisely the right context for a value returned from your procedure.

If `SPI_palloc` is called while the procedure is not connected to SPI, then it acts the same as a normal `palloc`. Before a procedure connects to the SPI manager, the current memory context is the upper executor context, so all allocations made by the procedure via `palloc` or by SPI utility functions are made in this context.

When `SPI_connect` is called, the private context of the procedure, which is created by `SPI_connect`, is made the current context. All allocations made by `palloc`, `repalloc`, or SPI utility functions (except for `SPI_copytuple`, `SPI_returntuple`, `SPI_modifytuple`, and `SPI_palloc`) are made in this context. When a procedure disconnects from the SPI manager (via `SPI_finish`) the current context is restored to the upper executor context, and all allocations made in the procedure memory context are freed and cannot be used any more.

All functions described in this section can be used by both connected and unconnected procedures. In an unconnected procedure, they act the same as the underlying ordinary server functions (`palloc`, etc.).

SPI_palloc

Name

`SPI_palloc` — allocate memory in the upper executor context

Synopsis

```
void * SPI_palloc(Size size)
```

Description

`SPI_palloc` allocates memory in the upper executor context.

Arguments

`Size size`

size in bytes of storage to allocate

Return Value

pointer to new storage space of the specified size

SPI_repalloc

Name

`SPI_repalloc` — reallocate memory in the upper executor context

Synopsis

```
void * SPI_repalloc(void * pointer, Size size)
```

Description

`SPI_repalloc` changes the size of a memory segment previously allocated using `SPI_palloc`.

This function is no longer different from plain `repalloc`. It's kept just for backward compatibility of existing code.

Arguments

`void * pointer`

pointer to existing storage to change

`Size size`

size in bytes of storage to allocate

Return Value

pointer to new storage space of specified size with the contents copied from the existing area

SPI_pfree

Name

`SPI_pfree` — free memory in the upper executor context

Synopsis

```
void SPI_pfree(void * pointer)
```

Description

`SPI_pfree` frees memory previously allocated using `SPI_palloc` or `SPI_repalloc`.

This function is no longer different from plain `pfree`. It's kept just for backward compatibility of existing code.

Arguments

`void * pointer`

pointer to existing storage to free

SPI_copytuple

Name

SPI_copytuple — make a copy of a row in the upper executor context

Synopsis

```
HeapTuple SPI_copytuple(HeapTuple row)
```

Description

SPI_copytuple makes a copy of a row in the upper executor context. This is normally used to return a modified row from a trigger. In a function declared to return a composite type, use SPI_returntuple instead.

Arguments

HeapTuple row

row to be copied

Return Value

the copied row; NULL only if tuple is NULL

SPI_returntuple

Name

`SPI_returntuple` — prepare to return a tuple as a Datum

Synopsis

```
HeapTupleHeader SPI_returntuple(HeapTuple row, TupleDesc rowdesc)
```

Description

`SPI_returntuple` makes a copy of a row in the upper executor context, returning it in the form of a row type `Datum`. The returned pointer need only be converted to `Datum` via `PointerGetDatum` before returning.

Note that this should be used for functions that are declared to return composite types. It is not used for triggers; use `SPI_copytuple` for returning a modified row in a trigger.

Arguments

`HeapTuple row`

row to be copied

`TupleDesc rowdesc`

descriptor for row (pass the same descriptor each time for most effective caching)

Return Value

`HeapTupleHeader` pointing to copied row; `NULL` only if `row` or `rowdesc` is `NULL`

SPI_modifytuple

Name

SPI_modifytuple — create a row by replacing selected fields of a given row

Synopsis

```
HeapTuple SPI_modifytuple(Relation rel, HeapTuple row, int ncols,
                          int * colnum, Datum * values, const char * nulls)
```

Description

SPI_modifytuple creates a new row by substituting new values for selected columns, copying the original row's columns at other positions. The input row is not modified.

Arguments

Relation rel

Used only as the source of the row descriptor for the row. (Passing a relation rather than a row descriptor is a misfeature.)

HeapTuple row

row to be modified

int ncols

number of columns to be changed

int * colnum

an array of length ncols, containing the numbers of the columns that are to be changed (column numbers start at 1)

Datum * values

an array of length ncols, containing the new values for the specified columns

const char * nulls

an array of length ncols, describing which new values are null

If nulls is NULL then SPI_modifytuple assumes that no new values are null. Otherwise, each entry of the nulls array should be ' ' if the corresponding new value is non-null, or 'n' if the corresponding new value is null. (In the latter case, the actual value in the corresponding values entry doesn't matter.) Note that nulls is not a text string, just an array: it does not need a '\0' terminator.

Return Value

new row with modifications, allocated in the upper executor context; `NULL` only if `row` is `NULL`

On error, `SPI_result` is set as follows:

`SPI_ERROR_ARGUMENT`

if `rel` is `NULL`, or if `row` is `NULL`, or if `ncols` is less than or equal to 0, or if `colnum` is `NULL`, or if `values` is `NULL`.

`SPI_ERROR_NOATTRIBUTE`

if `colnum` contains an invalid column number (less than or equal to 0 or greater than the number of column in `row`)

SPI_freetuple

Name

`SPI_freetuple` — free a row allocated in the upper executor context

Synopsis

```
void SPI_freetuple(HeapTuple row)
```

Description

`SPI_freetuple` frees a row previously allocated in the upper executor context.

This function is no longer different from plain `heap_freetuple`. It's kept just for backward compatibility of existing code.

Arguments

`HeapTuple row`

row to free

SPI_freetuptable

Name

`SPI_freetuptable` — free a row set created by `SPI_execute` or a similar function

Synopsis

```
void SPI_freetuptable(SPITupleTable * tuptable)
```

Description

`SPI_freetuptable` frees a row set created by a prior SPI command execution function, such as `SPI_execute`. Therefore, this function is often called with the global variable `SPI_tuptable` as argument.

This function is useful if a SPI procedure needs to execute multiple commands and does not want to keep the results of earlier commands around until it ends. Note that any unfreed row sets will be freed anyway at `SPI_finish`. Also, if a subtransaction is started and then aborted within execution of a SPI procedure, SPI automatically frees any row sets created while the subtransaction was running.

Beginning in PostgreSQL 9.3, `SPI_freetuptable` contains guard logic to protect against duplicate deletion requests for the same row set. In previous releases, duplicate deletions would lead to crashes.

Arguments

`SPITupleTable * tuptable`

pointer to row set to free, or NULL to do nothing

SPI_freeplan

Name

SPI_freeplan — free a previously saved prepared statement

Synopsis

```
int SPI_freeplan(SPIPlanPtr plan)
```

Description

SPI_freeplan releases a prepared statement previously returned by SPI_prepare or saved by SPI_keepplan or SPI_saveplan.

Arguments

SPIPlanPtr plan

pointer to statement to free

Return Value

0 on success; SPI_ERROR_ARGUMENT if plan is NULL or invalid

44.4. Visibility of Data Changes

The following rules govern the visibility of data changes in functions that use SPI (or any other C function):

- During the execution of an SQL command, any data changes made by the command are invisible to the command itself. For example, in:

  ```
  INSERT INTO a SELECT * FROM a;
  ```
  the inserted rows are invisible to the `SELECT` part.
- Changes made by a command C are visible to all commands that are started after C, no matter whether they are started inside C (during the execution of C) or after C is done.
- Commands executed via SPI inside a function called by an SQL command (either an ordinary function or a trigger) follow one or the other of the above rules depending on the read/write flag passed to SPI. Commands executed in read-only mode follow the first rule: they cannot see changes of the calling command. Commands executed in read-write mode follow the second rule: they can see all changes made so far.
- All standard procedural languages set the SPI read-write mode depending on the volatility attribute of the function. Commands of `STABLE` and `IMMUTABLE` functions are done in read-only mode, while commands of `VOLATILE` functions are done in read-write mode. While authors of C functions are able to violate this convention, it's unlikely to be a good idea to do so.

The next section contains an example that illustrates the application of these rules.

44.5. Examples

This section contains a very simple example of SPI usage. The procedure `execq` takes an SQL command as its first argument and a row count as its second, executes the command using `SPI_exec` and returns the number of rows that were processed by the command. You can find more complex examples for SPI in the source tree in `src/test/regress/regress.c` and in the spi module.

```
#include "postgres.h"

#include "executor/spi.h"
#include "utils/builtins.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

int execq(text *sql, int cnt);

int
execq(text *sql, int cnt)
{
    char *command;
```

```
    int ret;
    int proc;

    /* Convert given text object to a C string */
    command = text_to_cstring(sql);

    SPI_connect();

    ret = SPI_exec(command, cnt);

    proc = SPI_processed;
    /*
     * If some rows were fetched, print them via elog(INFO).
     */
    if (ret > 0 && SPI_tuptable != NULL)
    {
        TupleDesc tupdesc = SPI_tuptable->tupdesc;
        SPITupleTable *tuptable = SPI_tuptable;
        char buf[8192];
        int i, j;

        for (j = 0; j < proc; j++)
        {
            HeapTuple tuple = tuptable->vals[j];

            for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
                snprintf(buf + strlen (buf), sizeof(buf) - strlen(buf), " %s%s",
                        SPI_getvalue(tuple, tupdesc, i),
                        (i == tupdesc->natts) ? " " : " |");
            elog(INFO, "EXECQ: %s", buf);
        }
    }

    SPI_finish();
    pfree(command);

    return (proc);
}
```

(This function uses call convention version 0, to make the example easier to understand. In real applications you should use the new version 1 interface.)

This is how you declare the function after having compiled it into a shared library (details are in Section 35.9.6.):

```
CREATE FUNCTION execq(text, integer) RETURNS integer
    AS 'filename'
    LANGUAGE C;
```

Here is a sample session:

```
=> SELECT execq('CREATE TABLE a (x integer)', 0);
```

```
 execq
-------
     0
(1 row)

=> INSERT INTO a VALUES (execq('INSERT INTO a VALUES (0)', 0));
INSERT 0 1
=> SELECT execq('SELECT * FROM a', 0);
INFO:  EXECQ:  0    -- inserted by execq
INFO:  EXECQ:  1    -- returned by execq and inserted by upper INSERT

 execq
-------
     2
(1 row)

=> SELECT execq('INSERT INTO a SELECT x + 2 FROM a', 1);
 execq
-------
     1
(1 row)

=> SELECT execq('SELECT * FROM a', 10);
INFO:  EXECQ:  0
INFO:  EXECQ:  1
INFO:  EXECQ:  2    -- 0 + 2, only one row inserted - as specified

 execq
-------
     3              -- 10 is the max value only, 3 is the real number of rows
(1 row)

=> DELETE FROM a;
DELETE 3
=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1                  -- no rows in a (0) + 1
(1 row)

=> INSERT INTO a VALUES (execq('SELECT * FROM a', 0) + 1);
INFO:  EXECQ:  1
INSERT 0 1
=> SELECT * FROM a;
 x
---
 1
 2                  -- there was one row in a + 1
(2 rows)

-- This demonstrates the data changes visibility rule:
```

```
=> INSERT INTO a SELECT execq('SELECT * FROM a', 0) * x FROM a;
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  1
INFO:  EXECQ:  2
INFO:  EXECQ:  2
INSERT 0 2
=> SELECT * FROM a;
 x
---
 1
 2
 2                  -- 2 rows * 1 (x in first row)
 6                  -- 3 rows (2 + 1 just inserted) * 2 (x in second row)
(4 rows)               ^^^^^^
                       rows visible to execq() in different invocations
```

Chapter 45. Background Worker Processes

PostgreSQL can be extended to run user-supplied code in separate processes. Such processes are started, stopped and monitored by `postgres`, which permits them to have a lifetime closely linked to the server's status. These processes have the option to attach to PostgreSQL's shared memory area and to connect to databases internally; they can also run multiple transactions serially, just like a regular client-connected server process. Also, by linking to libpq they can connect to the server and behave like a regular client application.

> **Warning**
>
> There are considerable robustness and security risks in using background worker processes because, being written in the `C` language, they have unrestricted access to data. Administrators wishing to enable modules that include background worker processes should exercise extreme caution. Only carefully audited modules should be permitted to run background worker processes.

Background workers can be initialized at the time that PostgreSQL is started by including the module name in `shared_preload_libraries`. A module wishing to run a background worker can register it by calling `RegisterBackgroundWorker(BackgroundWorker *worker)` from its `_PG_init()`. Background workers can also be started after the system is up and running by calling the function `RegisterDynamicBackgroundWorker(BackgroundWorker *worker, BackgroundWorkerHandle **handle)`. Unlike `RegisterBackgroundWorker`, which can only be called from within the postmaster, `RegisterDynamicBackgroundWorker` must be called from a regular backend.

The structure `BackgroundWorker` is defined thus:

```
typedef void (*bgworker_main_type)(Datum main_arg);
typedef struct BackgroundWorker
{
    char        bgw_name[BGW_MAXLEN];
    int         bgw_flags;
    BgWorkerStartTime bgw_start_time;
    int         bgw_restart_time;       /* in seconds, or BGW_NEVER_RESTART */
    bgworker_main_type bgw_main;
    char        bgw_library_name[BGW_MAXLEN];   /* only if bgw_main is NULL */
    char        bgw_function_name[BGW_MAXLEN];  /* only if bgw_main is NULL */
    Datum       bgw_main_arg;
    char        bgw_extra[BGW_EXTRALEN];
    int         bgw_notify_pid;
} BackgroundWorker;
```

`bgw_name` is a string to be used in log messages, process listings and similar contexts.

`bgw_flags` is a bitwise-or'd bit mask indicating the capabilities that the module wants. Possible values are `BGWORKER_SHMEM_ACCESS` (requesting shared memory access) and `BGWORKER_BACKEND_DATABASE_CONNECTION` (requesting the ability to establish a database connection, through which it can later run transactions and queries). A background worker using

`BGWORKER_BACKEND_DATABASE_CONNECTION` to connect to a database must also attach shared memory using `BGWORKER_SHMEM_ACCESS`, or worker start-up will fail.

`bgw_start_time` is the server state during which `postgres` should start the process; it can be one of `BgWorkerStart_PostmasterStart` (start as soon as `postgres` itself has finished its own initialization; processes requesting this are not eligible for database connections), `BgWorkerStart_ConsistentState` (start as soon as a consistent state has been reached in a hot standby, allowing processes to connect to databases and run read-only queries), and `BgWorkerStart_RecoveryFinished` (start as soon as the system has entered normal read-write state). Note the last two values are equivalent in a server that's not a hot standby. Note that this setting only indicates when the processes are to be started; they do not stop when a different state is reached.

`bgw_restart_time` is the interval, in seconds, that `postgres` should wait before restarting the process, in case it crashes. It can be any positive value, or `BGW_NEVER_RESTART`, indicating not to restart the process in case of a crash.

`bgw_main` is a pointer to the function to run when the process is started. This function must take a single argument of type `Datum` and return `void`. `bgw_main_arg` will be passed to it as its only argument. Note that the global variable `MyBgworkerEntry` points to a copy of the `BackgroundWorker` structure passed at registration time. `bgw_main` may be NULL; in that case, `bgw_library_name` and `bgw_function_name` will be used to determine the entry point. This is useful for background workers launched after postmaster startup, where the postmaster does not have the requisite library loaded.

`bgw_library_name` is the name of a library in which the initial entry point for the background worker should be sought. It is ignored unless `bgw_main` is NULL. But if `bgw_main` is NULL, then the named library will be dynamically loaded by the worker process and `bgw_function_name` will be used to identify the function to be called.

`bgw_function_name` is the name of a function in a dynamically loaded library which should be used as the initial entry point for a new background worker. It is ignored unless `bgw_main` is NULL.

`bgw_extra` can contain extra data to be passed to the background worker. Unlike `bgw_main_arg`, this data is not passed as an argument to the worker's main function, but it can be accessed via `MyBgworkerEntry`, as discussed above.

`bgw_notify_pid` is the PID of a PostgreSQL backend process to which the postmaster should send `SIGUSR1` when the process is started or exits. It should be 0 for workers registered at postmaster startup time, or when the backend registering the worker does not wish to wait for the worker to start up. Otherwise, it should be initialized to `MyProcPid`.

Once running, the process can connect to a database by calling `BackgroundWorkerInitializeConnection(char *dbname, char *username)` or `BackgroundWorkerInitializeConnectionByOid(Oid dboid, Oid useroid)`. This allows the process to run transactions and queries using the `SPI` interface. If `dbname` is NULL or `dboid` is `InvalidOid`, the session is not connected to any particular database, but shared catalogs can be accessed. If `username` is NULL or `useroid` is `InvalidOid`, the process will run as the superuser created during `initdb`. A background worker can only call one of these two functions, and only once. It is not possible to switch databases.

Signals are initially blocked when control reaches the `bgw_main` function, and must be unblocked by it; this is to allow the process to customize its signal handlers, if necessary. Signals can be unblocked in the new process by calling `BackgroundWorkerUnblockSignals` and blocked by calling `BackgroundWorkerBlockSignals`.

If `bgw_restart_time` for a background worker is configured as `BGW_NEVER_RESTART`, or if it exits with an exit code of 0 or is terminated by `TerminateBackgroundWorker`, it will be automatically unregistered by the postmaster on exit. Otherwise, it will be restarted after the time period configured via `bgw_restart_time`, or immediately if the postmaster reinitializes the cluster due to a backend failure. Backends which need to suspend execution only temporarily should use an interruptible sleep rather than exiting; this can be achieved by calling `WaitLatch()`. Make sure the `WL_POSTMASTER_DEATH` flag is set when calling that function, and verify the return code for a prompt exit in the emergency case that `postgres` itself has terminated.

When a background worker is registered using the `RegisterDynamicBackgroundWorker` function, it is possible for the backend performing the registration to obtain information regarding the status of the worker. Backends wishing to do this should pass the address of a `BackgroundWorkerHandle *` as the second argument to `RegisterDynamicBackgroundWorker`. If the worker is successfully registered, this pointer will be initialized with an opaque handle that can subsequently be passed to `GetBackgroundWorkerPid(BackgroundWorkerHandle *, pid_t *)` or `TerminateBackgroundWorker(BackgroundWorkerHandle *)`. `GetBackgroundWorkerPid` can be used to poll the status of the worker: a return value of `BGWH_NOT_YET_STARTED` indicates that the worker has not yet been started by the postmaster; `BGWH_STOPPED` indicates that it has been started but is no longer running; and `BGWH_STARTED` indicates that it is currently running. In this last case, the PID will also be returned via the second argument. `TerminateBackgroundWorker` causes the postmaster to send `SIGTERM` to the worker if it is running, and to unregister it as soon as it is not.

In some cases, a process which registers a background worker may wish to wait for the worker to start up. This can be accomplished by initializing `bgw_notify_pid` to `MyProcPid` and then passing the `BackgroundWorkerHandle *` obtained at registration time to `WaitForBackgroundWorkerStartup(BackgroundWorkerHandle *handle, pid_t *)` function. This function will block until the postmaster has attempted to start the background worker, or until the postmaster dies. If the background runner is running, the return value will `BGWH_STARTED`, and the PID will be written to the provided address. Otherwise, the return value will be `BGWH_STOPPED` or `BGWH_POSTMASTER_DIED`.

The `src/test/modules/worker_spi` module contains a working example, which demonstrates some useful techniques.

The maximum number of registered background workers is limited by max_worker_processes.

Chapter 46. Logical Decoding

PostgreSQL provides infrastructure to stream the modifications performed via SQL to external consumers. This functionality can be used for a variety of purposes, including replication solutions and auditing.

Changes are sent out in streams identified by logical replication slots. Each stream outputs each change exactly once.

The format in which those changes are streamed is determined by the output plugin used. An example plugin is provided in the PostgreSQL distribution. Additional plugins can be written to extend the choice of available formats without modifying any core code. Every output plugin has access to each individual new row produced by `INSERT` and the new row version created by `UPDATE`. Availability of old row versions for `UPDATE` and `DELETE` depends on the configured replica identity (see *`REPLICA IDENTITY`*).

Changes can be consumed either using the streaming replication protocol (see Section 50.3 and Section 46.3), or by calling functions via SQL (see Section 46.4). It is also possible to write additional methods of consuming the output of a replication slot without modifying core code (see Section 46.7).

46.1. Logical Decoding Examples

The following example demonstrates controlling logical decoding using the SQL interface.

Before you can use logical decoding, you must set wal_level to `logical` and max_replication_slots to at least 1. Then, you should connect to the target database (in the example below, `postgres`) as a superuser.

```
postgres=# -- Create a slot named 'regression_slot' using the output plugin 'test_decoding'
postgres=# SELECT * FROM pg_create_logical_replication_slot('regression_slot', 'test_decodi
    slot_name    | xlog_position
-----------------+---------------
 regression_slot | 0/16B1970
(1 row)

postgres=# SELECT slot_name, plugin, slot_type, database, active, restart_lsn FROM pg_repli
    slot_name    |    plugin     | slot_type | database | active | restart_lsn
-----------------+---------------+-----------+----------+--------+-------------
 regression_slot | test_decoding | logical   | postgres | f      | 0/16A4408
(1 row)

postgres=# -- There are no changes to see yet
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
 location | xid | data
----------+-----+------
(0 rows)

postgres=# CREATE TABLE data(id serial primary key, data text);
CREATE TABLE

postgres=# -- DDL isn't replicated, so all you'll see is the transaction
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
 location  | xid |    data
-----------+-----+------------
```

```
 0/16D5D48 | 688 | BEGIN 688
 0/16E0380 | 688 | COMMIT 688
(2 rows)

postgres=# -- Once changes are read, they're consumed and not emitted
postgres=# -- in a subsequent call:
postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
 location | xid | data
----------+-----+------
(0 rows)

postgres=# BEGIN;
postgres=# INSERT INTO data(data) VALUES('1');
postgres=# INSERT INTO data(data) VALUES('2');
postgres=# COMMIT;

postgres=# SELECT * FROM pg_logical_slot_get_changes('regression_slot', NULL, NULL);
 location  | xid |                      data
-----------+-----+-------------------------------------------------
 0/16E0478 | 689 | BEGIN 689
 0/16E0478 | 689 | table public.data: INSERT: id[integer]:1 data[text]:'1'
 0/16E0580 | 689 | table public.data: INSERT: id[integer]:2 data[text]:'2'
 0/16E0650 | 689 | COMMIT 689
(4 rows)

postgres=# INSERT INTO data(data) VALUES('3');

postgres=# -- You can also peek ahead in the change stream without consuming changes
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL);
 location  | xid |                      data
-----------+-----+-------------------------------------------------
 0/16E09C0 | 690 | BEGIN 690
 0/16E09C0 | 690 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/16E0B90 | 690 | COMMIT 690
(3 rows)

postgres=# -- The next call to pg_logical_slot_peek_changes() returns the same changes agai
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL);
 location  | xid |                      data
-----------+-----+-------------------------------------------------
 0/16E09C0 | 690 | BEGIN 690
 0/16E09C0 | 690 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/16E0B90 | 690 | COMMIT 690
(3 rows)

postgres=# -- options can be passed to output plugin, to influence the formatting
postgres=# SELECT * FROM pg_logical_slot_peek_changes('regression_slot', NULL, NULL, 'inclu
 location  | xid |                      data
-----------+-----+-------------------------------------------------
 0/16E09C0 | 690 | BEGIN 690
 0/16E09C0 | 690 | table public.data: INSERT: id[integer]:3 data[text]:'3'
 0/16E0B90 | 690 | COMMIT 690 (at 2014-02-27 16:41:51.863092+01)
(3 rows)
```

```
postgres=# -- Remember to destroy a slot you no longer need to stop it consuming
postgres=# -- server resources:
postgres=# SELECT pg_drop_replication_slot('regression_slot');
 pg_drop_replication_slot
--------------------------

(1 row)
```

The following example shows how logical decoding is controlled over the streaming replication protocol, using the program pg_recvlogical included in the PostgreSQL distribution. This requires that client authentication is set up to allow replication connections (see Section 25.2.5.1) and that `max_wal_senders` is set sufficiently high to allow an additional connection.

```
$ pg_recvlogical -d postgres --slot test --create-slot
$ pg_recvlogical -d postgres --slot test --start -f -
Control+Z
$ psql -d postgres -c "INSERT INTO data(data) VALUES('4');"
$ fg
BEGIN 693
table public.data: INSERT: id[integer]:4 data[text]:'4'
COMMIT 693
Control+C
$ pg_recvlogical -d postgres --slot test --drop-slot
```

46.2. Logical Decoding Concepts

46.2.1. Logical Decoding

Logical decoding is the process of extracting all persistent changes to a database's tables into a coherent, easy to understand format which can be interpreted without detailed knowledge of the database's internal state.

In PostgreSQL, logical decoding is implemented by decoding the contents of the write-ahead log, which describe changes on a storage level, into an application-specific form such as a stream of tuples or SQL statements.

46.2.2. Replication Slots

In the context of logical replication, a slot represents a stream of changes that can be replayed to a client in the order they were made on the origin server. Each slot streams a sequence of changes from a single database, sending each change exactly once (except when peeking forward in the stream).

> **Note:** PostgreSQL also has streaming replication slots (see Section 25.2.5), but they are used somewhat differently there.

A replication slot has an identifier that is unique across all databases in a PostgreSQL cluster. Slots persist independently of the connection using them and are crash-safe.

Multiple independent slots may exist for a single database. Each slot has its own state, allowing different consumers to receive changes from different points in the database change stream. For most applications, a separate slot will be required for each consumer.

A logical replication slot knows nothing about the state of the receiver(s). It's even possible to have multiple different receivers using the same slot at different times; they'll just get the changes following on from when the last receiver stopped consuming them. Only one receiver may consume changes from a slot at any given time.

> **Note:** Replication slots persist across crashes and know nothing about the state of their consumer(s). They will prevent removal of required resources even when there is no connection using them. This consumes storage because neither required WAL nor required rows from the system catalogs can be removed by `VACUUM` as long as they are required by a replication slot. So if a slot is no longer required it should be dropped.

46.2.3. Output Plugins

Output plugins transform the data from the write-ahead log's internal representation into the format the consumer of a replication slot desires.

46.2.4. Exported Snapshots

When a new replication slot is created using the streaming replication interface, a snapshot is exported (see Section 9.26.5), which will show exactly the state of the database after which all changes will be included in the change stream. This can be used to create a new replica by using `SET TRANSACTION SNAPSHOT` to read the state of the database at the moment the slot was created. This transaction can then be used to dump the database's state at that point in time, which afterwards can be updated using the slot's contents without losing any changes.

46.3. Streaming Replication Protocol Interface

The commands

- `CREATE_REPLICATION_SLOT` *`slot_name`* `LOGICAL` *`output_plugin`*
- `DROP_REPLICATION_SLOT` *`slot_name`*
- `START_REPLICATION SLOT` *`slot_name`* `LOGICAL ...`

are used to create, drop, and stream changes from a replication slot, respectively. These commands are only available over a replication connection; they cannot be used via SQL. See Section 50.3 for details on these commands.

The command pg_recvlogical can be used to control logical decoding over a streaming replication connection. (It uses these commands internally.)

46.4. Logical Decoding SQL Interface

See Section 9.26.6 for detailed documentation on the SQL-level API for interacting with logical decoding.

Synchronous replication (see Section 25.2.8) is only supported on replication slots used over the streaming replication interface. The function interface and additional, non-core interfaces do not support synchronous replication.

46.5. System Catalogs Related to Logical Decoding

The `pg_replication_slots` view and the `pg_stat_replication` view provide information about the current state of replication slots and streaming replication connections respectively. These views apply to both physical and logical replication.

46.6. Logical Decoding Output Plugins

An example output plugin can be found in the `contrib/test_decoding` subdirectory of the PostgreSQL source tree.

46.6.1. Initialization Function

An output plugin is loaded by dynamically loading a shared library with the output plugin's name as the library base name. The normal library search path is used to locate the library. To provide the required output plugin callbacks and to indicate that the library is actually an output plugin it needs to provide a function named `_PG_output_plugin_init`. This function is passed a struct that needs to be filled with the callback function pointers for individual actions.

```
typedef struct OutputPluginCallbacks
{
    LogicalDecodeStartupCB startup_cb;
    LogicalDecodeBeginCB begin_cb;
    LogicalDecodeChangeCB change_cb;
    LogicalDecodeCommitCB commit_cb;
    LogicalDecodeFilterByOriginCB filter_by_origin_cb;
    LogicalDecodeShutdownCB shutdown_cb;
} OutputPluginCallbacks;

typedef void (*LogicalOutputPluginInit) (struct OutputPluginCallbacks *cb);
```

The begin_cb, change_cb and commit_cb callbacks are required, while startup_cb, filter_by_origin_cb and shutdown_cb are optional.

46.6.2. Capabilities

To decode, format and output changes, output plugins can use most of the backend's normal infrastructure, including calling output functions. Read only access to relations is permitted as long as only relations are accessed that either have been created by initdb in the pg_catalog schema, or have been marked as user provided catalog tables using

```
ALTER TABLE user_catalog_table SET (user_catalog_table = true);
CREATE TABLE another_catalog_table(data text) WITH (user_catalog_table = true);
```

Any actions leading to transaction ID assignment are prohibited. That, among others, includes writing to tables, performing DDL changes, and calling txid_current().

46.6.3. Output Modes

Output plugin callbacks can pass data to the consumer in nearly arbitrary formats. For some use cases, like viewing the changes via SQL, returning data in a data type that can contain arbitrary data (e.g., bytea) is cumbersome. If the output plugin only outputs textual data in the server's encoding, it can declare that by setting OutputPluginOptions.output_type to OUTPUT_PLUGIN_TEXTUAL_OUTPUT instead of OUTPUT_PLUGIN_BINARY_OUTPUT in the startup callback. In that case, all the data has to be in the server's encoding so that a text datum can contain it. This is checked in assertion-enabled builds.

46.6.4. Output Plugin Callbacks

An output plugin gets notified about changes that are happening via various callbacks it needs to provide.

Concurrent transactions are decoded in commit order, and only changes belonging to a specific transaction are decoded between the begin and commit callbacks. Transactions that were rolled back explicitly or implicitly never get decoded. Successful savepoints are folded into the transaction containing them in the order they were executed within that transaction.

> **Note:** Only transactions that have already safely been flushed to disk will be decoded. That can lead to a COMMIT not immediately being decoded in a directly following pg_logical_slot_get_changes() when synchronous_commit is set to off.

46.6.4.1. Startup Callback

The optional startup_cb callback is called whenever a replication slot is created or asked to stream changes, independent of the number of changes that are ready to be put out.

```
typedef void (*LogicalDecodeStartupCB) (struct LogicalDecodingContext *ctx,
                                        OutputPluginOptions *options,
```

```
                                        bool is_init);
```

The `is_init` parameter will be true when the replication slot is being created and false otherwise. `options` points to a struct of options that output plugins can set:

```
typedef struct OutputPluginOptions
{
    OutputPluginOutputType output_type;
} OutputPluginOptions;
```

`output_type` has to either be set to `OUTPUT_PLUGIN_TEXTUAL_OUTPUT` or `OUTPUT_PLUGIN_BINARY_OUTPUT`. See also Section 46.6.3.

The startup callback should validate the options present in `ctx->output_plugin_options`. If the output plugin needs to have a state, it can use `ctx->output_plugin_private` to store it.

46.6.4.2. Shutdown Callback

The optional `shutdown_cb` callback is called whenever a formerly active replication slot is not used anymore and can be used to deallocate resources private to the output plugin. The slot isn't necessarily being dropped, streaming is just being stopped.

```
typedef void (*LogicalDecodeShutdownCB) (struct LogicalDecodingContext *ctx);
```

46.6.4.3. Transaction Begin Callback

The required `begin_cb` callback is called whenever a start of a committed transaction has been decoded. Aborted transactions and their contents never get decoded.

```
typedef void (*LogicalDecodeBeginCB) (struct LogicalDecodingContext *ctx,
                                      ReorderBufferTXN *txn);
```

The `txn` parameter contains meta information about the transaction, like the time stamp at which it has been committed and its XID.

46.6.4.4. Transaction End Callback

The required `commit_cb` callback is called whenever a transaction commit has been decoded. The `change_cb` callbacks for all modified rows will have been called before this, if there have been any modified rows.

```
typedef void (*LogicalDecodeCommitCB) (struct LogicalDecodingContext *ctx,
                                       ReorderBufferTXN *txn,
                                       XLogRecPtr commit_lsn);
```

46.6.4.5. Change Callback

The required `change_cb` callback is called for every individual row modification inside a transaction, may it be an `INSERT`, `UPDATE`, or `DELETE`. Even if the original command modified several rows at once the callback will be called individually for each row.

```
typedef void (*LogicalDecodeChangeCB) (struct LogicalDecodingContext *ctx,
                                       ReorderBufferTXN *txn,
                                       Relation relation,
                                       ReorderBufferChange *change);
```

The `ctx` and `txn` parameters have the same contents as for the `begin_cb` and `commit_cb` callbacks, but additionally the relation descriptor `relation` points to the relation the row belongs to and a struct `change` describing the row modification are passed in.

> **Note:** Only changes in user defined tables that are not unlogged (see *`UNLOGGED`*) and not temporary (see *`TEMPORARY`* *or* *`TEMP`*) can be extracted using logical decoding.

46.6.4.6. Origin Filter Callback

The optional `filter_by_origin_cb` callback is called to determine whether data that has been replayed from `origin_id` is of interest to the output plugin.

```
typedef bool (*LogicalDecodeFilterByOriginCB) (struct LogicalDecodingContext *ctx,
                                               RepOriginId origin_id);
```

The `ctx` parameter has the same contents as for the other callbacks. No information but the origin is available. To signal that changes originating on the passed in node are irrelevant, return true, causing them to be filtered away; false otherwise. The other callbacks will not be called for transactions and changes that have been filtered away.

This is useful when implementing cascading or multidirectional replication solutions. Filtering by the origin allows to prevent replicating the same changes back and forth in such setups. While transactions and changes also carry information about the origin, filtering via this callback is noticeably more efficient.

46.6.5. Functions for Producing Output

To actually produce output, output plugins can write data to the `StringInfo` output buffer in `ctx->out` when inside the `begin_cb`, `commit_cb`, or `change_cb` callbacks. Before writing to the output buffer, `OutputPluginPrepareWrite(ctx, last_write)` has to be called, and after finishing writing to the buffer, `OutputPluginWrite(ctx, last_write)` has to be called to perform the write. The `last_write` indicates whether a particular write was the callback's last write.

The following example shows how to output data to the consumer of an output plugin:

```
OutputPluginPrepareWrite(ctx, true);
appendStringInfo(ctx->out, "BEGIN %u", txn->xid);
```

```
OutputPluginWrite(ctx, true);
```

46.7. Logical Decoding Output Writers

It is possible to add more output methods for logical decoding. For details, see `src/backend/replication/logical/logicalfuncs.c`. Essentially, three functions need to be provided: one to read WAL, one to prepare writing output, and one to write the output (see Section 46.6.5).

46.8. Synchronous Replication Support for Logical Decoding

Logical decoding can be used to build synchronous replication solutions with the same user interface as synchronous replication for streaming replication. To do this, the streaming replication interface (see Section 46.3) must be used to stream out data. Clients have to send `Standby status update (F)` (see Section 50.3) messages, just like streaming replication clients do.

> **Note:** A synchronous replica receiving changes via logical decoding will work in the scope of a single database. Since, in contrast to that, `synchronous_standby_names` currently is server wide, this means this technique will not work properly if more than one database is actively used.

Chapter 47. Replication Progress Tracking

Replication origins are intended to make it easier to implement logical replication solutions on top of logical decoding. They provide a solution to two common problems:

- How to safely keep track of replication progress
- How to change replication behavior based on the origin of a row; for example, to prevent loops in bi-directional replication setups

Replication origins have just two properties, a name and an OID. The name, which is what should be used to refer to the origin across systems, is free-form `text`. It should be used in a way that makes conflicts between replication origins created by different replication solutions unlikely; e.g., by prefixing the replication solution's name to it. The OID is used only to avoid having to store the long version in situations where space efficiency is important. It should never be shared across systems.

Replication origins can be created using the function `pg_replication_origin_create()`; dropped using `pg_replication_origin_drop()`; and seen in the `pg_replication_origin` system catalog.

One nontrivial part of building a replication solution is to keep track of replay progress in a safe manner. When the applying process, or the whole cluster, dies, it needs to be possible to find out up to where data has successfully been replicated. Naive solutions to this, such as updating a row in a table for every replayed transaction, have problems like run-time overhead and database bloat.

Using the replication origin infrastructure a session can be marked as replaying from a remote node (using the `pg_replication_origin_session_setup()` function). Additionally the LSN and commit time stamp of every source transaction can be configured on a per transaction basis using `pg_replication_origin_xact_setup()`. If that's done replication progress will persist in a crash safe manner. Replay progress for all replication origins can be seen in the `pg_replication_origin_status` view. An individual origin's progress, e.g., when resuming replication, can be acquired using `pg_replication_origin_progress()` for any origin or `pg_replication_origin_session_progress()` for the origin configured in the current session.

In replication topologies more complex than replication from exactly one system to one other system, another problem can be that it is hard to avoid replicating replayed rows again. That can lead both to cycles in the replication and inefficiencies. Replication origins provide an optional mechanism to recognize and prevent that. When configured using the functions referenced in the previous paragraph, every change and transaction passed to output plugin callbacks (see Section 46.6) generated by the session is tagged with the replication origin of the generating session. This allows treating them differently in the output plugin, e.g., ignoring all but locally-originating rows. Additionally the `filter_by_origin_cb` callback can be used to filter the logical decoding change stream based on the source. While less flexible, filtering via that callback is considerably more efficient than doing it in the output plugin.

VI. Reference

The entries in this Reference are meant to provide in reasonable length an authoritative, complete, and formal summary about their respective subjects. More information about the use of PostgreSQL, in narrative, tutorial, or example form, can be found in other parts of this book. See the cross-references listed on each reference page.

The reference entries are also available as traditional “man” pages.

I. SQL Commands

This part contains reference information for the SQL commands supported by PostgreSQL. By “SQL” the language in general is meant; information about the standards conformance and compatibility of each command can be found on the respective reference page.

ABORT

Name

ABORT — abort the current transaction

Synopsis

```
ABORT [ WORK | TRANSACTION ]
```

Description

ABORT rolls back the current transaction and causes all the updates made by the transaction to be discarded. This command is identical in behavior to the standard SQL command ROLLBACK, and is present only for historical reasons.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

Notes

Use COMMIT to successfully terminate a transaction.

Issuing ABORT outside of a transaction block emits a warning and otherwise has no effect.

Examples

To abort all changes:

```
ABORT;
```

Compatibility

This command is a PostgreSQL extension present for historical reasons. `ROLLBACK` is the equivalent standard SQL command.

See Also

BEGIN, COMMIT, ROLLBACK

ALTER AGGREGATE

Name

ALTER AGGREGATE — change the definition of an aggregate function

Synopsis

```
ALTER AGGREGATE name ( aggregate_signature ) RENAME TO new_name
ALTER AGGREGATE name ( aggregate_signature )
                 OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER AGGREGATE name ( aggregate_signature ) SET SCHEMA new_schema

where aggregate_signature is:

* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ] argtype [ , ...
```

Description

ALTER AGGREGATE changes the definition of an aggregate function.

You must own the aggregate function to use ALTER AGGREGATE. To change the schema of an aggregate function, you must also have CREATE privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the aggregate function's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the aggregate function. However, a superuser can alter ownership of any aggregate function anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing aggregate function.

argmode

The mode of an argument: IN or VARIADIC. If omitted, the default is IN.

argname

The name of an argument. Note that ALTER AGGREGATE does not actually pay any attention to argument names, since only the argument data types are needed to determine the aggregate function's identity.

argtype

An input data type on which the aggregate function operates. To reference a zero-argument aggregate function, write * in place of the list of argument specifications. To reference an ordered-set aggregate function, write ORDER BY between the direct and aggregated argument specifications.

new_name

The new name of the aggregate function.

new_owner

The new owner of the aggregate function.

new_schema

The new schema for the aggregate function.

Notes

The recommended syntax for referencing an ordered-set aggregate is to write ORDER BY between the direct and aggregated argument specifications, in the same style as in CREATE AGGREGATE. However, it will also work to omit ORDER BY and just run the direct and aggregated argument specifications into a single list. In this abbreviated form, if VARIADIC "any" was used in both the direct and aggregated argument lists, write VARIADIC "any" only once.

Examples

To rename the aggregate function myavg for type integer to my_average:

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

To change the owner of the aggregate function myavg for type integer to joe:

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

To move the ordered-set aggregate mypercentile with direct argument of type float8 and aggregated argument of type integer into schema myschema:

```
ALTER AGGREGATE mypercentile(float8 ORDER BY integer) SET SCHEMA myschema;
```

This will work too:

```
ALTER AGGREGATE mypercentile(float8, integer) SET SCHEMA myschema;
```

Compatibility

There is no `ALTER AGGREGATE` statement in the SQL standard.

See Also

CREATE AGGREGATE, DROP AGGREGATE

ALTER COLLATION

Name

ALTER COLLATION — change the definition of a collation

Synopsis

```
ALTER COLLATION name RENAME TO new_name
ALTER COLLATION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER COLLATION name SET SCHEMA new_schema
```

Description

ALTER COLLATION changes the definition of a collation.

You must own the collation to use ALTER COLLATION. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the collation's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the collation. However, a superuser can alter ownership of any collation anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing collation.

new_name

The new name of the collation.

new_owner

The new owner of the collation.

new_schema

The new schema for the collation.

Examples

To rename the collation de_DE to german:

```
ALTER COLLATION "de_DE" RENAME TO german;
```

To change the owner of the collation en_US to joe:

```
ALTER COLLATION "en_US" OWNER TO joe;
```

Compatibility

There is no `ALTER COLLATION` statement in the SQL standard.

See Also

CREATE COLLATION, DROP COLLATION

ALTER CONVERSION

Name

ALTER CONVERSION — change the definition of a conversion

Synopsis

```
ALTER CONVERSION name RENAME TO new_name
ALTER CONVERSION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER CONVERSION name SET SCHEMA new_schema
```

Description

ALTER CONVERSION changes the definition of a conversion.

You must own the conversion to use ALTER CONVERSION. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the conversion's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the conversion. However, a superuser can alter ownership of any conversion anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing conversion.

new_name

The new name of the conversion.

new_owner

The new owner of the conversion.

new_schema

The new schema for the conversion.

Examples

To rename the conversion iso_8859_1_to_utf8 to latin1_to_unicode:

```
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO latin1_to_unicode;
```

To change the owner of the conversion iso_8859_1_to_utf8 to joe:

```
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

Compatibility

There is no `ALTER CONVERSION` statement in the SQL standard.

See Also

CREATE CONVERSION, DROP CONVERSION

ALTER DATABASE

Name

ALTER DATABASE — change a database

Synopsis

```
ALTER DATABASE name [ [ WITH ] option [ ... ] ]

where option can be:

    ALLOW_CONNECTIONS allowconn
    CONNECTION LIMIT connlimit
    IS_TEMPLATE istemplate

ALTER DATABASE name RENAME TO new_name

ALTER DATABASE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER DATABASE name SET TABLESPACE new_tablespace

ALTER DATABASE name SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER DATABASE name SET configuration_parameter FROM CURRENT
ALTER DATABASE name RESET configuration_parameter
ALTER DATABASE name RESET ALL
```

Description

ALTER DATABASE changes the attributes of a database.

The first form changes certain per-database settings. (See below for details.) Only the database owner or a superuser can change these settings.

The second form changes the name of the database. Only the database owner or a superuser can rename a database; non-superuser owners must also have the CREATEDB privilege. The current database cannot be renamed. (Connect to a different database if you need to do that.)

The third form changes the owner of the database. To alter the owner, you must own the database and also be a direct or indirect member of the new owning role, and you must have the CREATEDB privilege. (Note that superusers have all these privileges automatically.)

The fourth form changes the default tablespace of the database. Only the database owner or a superuser can do this; you must also have create privilege for the new tablespace. This command physically moves any tables or indexes in the database's old default tablespace to the new tablespace. The new default tablespace must be empty for this database, and no one can be connected to the database. Tables and indexes in non-default tablespaces are unaffected.

The remaining forms change the session default for a run-time configuration variable for a PostgreSQL database. Whenever a new session is subsequently started in that database, the specified value becomes

the session default value. The database-specific default overrides whatever setting is present in `postgresql.conf` or has been received from the `postgres` command line. Only the database owner or a superuser can change the session defaults for a database. Certain variables cannot be set this way, or can only be set by a superuser.

Parameters

name

The name of the database whose attributes are to be altered.

allowconn

If false then no one can connect to this database.

connlimit

How many concurrent connections can be made to this database. -1 means no limit.

istemplate

If true, then this database can be cloned by any user with `CREATEDB` privileges; if false, then only superusers or the owner of the database can clone it.

new_name

The new name of the database.

new_owner

The new owner of the database.

new_tablespace

The new default tablespace of the database.

configuration_parameter
value

Set this database's session default for the specified configuration parameter to the given value. If *value* is `DEFAULT` or, equivalently, `RESET` is used, the database-specific setting is removed, so the system-wide default setting will be inherited in new sessions. Use `RESET ALL` to clear all database-specific settings. `SET FROM CURRENT` saves the session's current value of the parameter as the database-specific value.

See SET and Chapter 18 for more information about allowed parameter names and values.

Notes

It is also possible to tie a session default to a specific role rather than to a database; see ALTER ROLE. Role-specific settings override database-specific ones if there is a conflict.

Examples

To disable index scans by default in the database `test`:

```
ALTER DATABASE test SET enable_indexscan TO off;
```

Compatibility

The `ALTER DATABASE` statement is a PostgreSQL extension.

See Also

CREATE DATABASE, DROP DATABASE, SET, CREATE TABLESPACE

ALTER DEFAULT PRIVILEGES

Name

ALTER DEFAULT PRIVILEGES — define default access privileges

Synopsis

```
ALTER DEFAULT PRIVILEGES
    [ FOR { ROLE | USER } target_role [, ...] ]
    [ IN SCHEMA schema_name [, ...] ]
    abbreviated_grant_or_revoke

where abbreviated_grant_or_revoke is one of:

GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON TABLES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON SEQUENCES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON FUNCTIONS
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
```

```
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

Description

`ALTER DEFAULT PRIVILEGES` allows you to set the privileges that will be applied to objects created in the future. (It does not affect privileges assigned to already-existing objects.) Currently, only the privileges for tables (including views and foreign tables), sequences, functions, and types (including domains) can be altered.

You can change default privileges only for objects that will be created by yourself or by roles that you are a member of. The privileges can be set globally (i.e., for all objects created in the current database), or just for objects created in specified schemas.

As explained under GRANT, the default privileges for any object type normally grant all grantable permissions to the object owner, and may grant some privileges to `PUBLIC` as well. However, this behavior can be changed by altering the global default privileges with `ALTER DEFAULT PRIVILEGES`.

Default privileges that are specified per-schema are added to whatever the global default privileges are for the particular object type. This means you cannot revoke privileges per-schema if they are granted globally (either by default, or according to a previous `ALTER DEFAULT PRIVILEGES` command that did not specify a schema). Per-schema `REVOKE` is only useful to reverse the effects of a previous per-schema `GRANT`.

Parameters

target_role

The name of an existing role of which the current role is a member. If `FOR ROLE` is omitted, the current role is assumed.

schema_name

The name of an existing schema. If specified, the default privileges are altered for objects later created in that schema. If `IN SCHEMA` is omitted, the global default privileges are altered.

role_name

The name of an existing role to grant or revoke privileges for. This parameter, and all the other parameters in *abbreviated_grant_or_revoke*, act as described under GRANT or REVOKE, except that one is setting permissions for a whole class of objects rather than specific named objects.

Notes

Use psql's `\ddp` command to obtain information about existing assignments of default privileges. The meaning of the privilege values is the same as explained for `\dp` under GRANT.

If you wish to drop a role for which the default privileges have been altered, it is necessary to reverse the changes in its default privileges or use `DROP OWNED BY` to get rid of the default privileges entry for the role.

Examples

Grant SELECT privilege to everyone for all tables (and views) you subsequently create in schema `myschema`, and allow role `webuser` to INSERT into them too:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT ON TABLES TO PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT INSERT ON TABLES TO webuser;
```

Undo the above, so that subsequently-created tables won't have any more permissions than normal:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE SELECT ON TABLES FROM PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE INSERT ON TABLES FROM webuser;
```

Remove the public EXECUTE permission that is normally granted on functions, for all functions subsequently created by role `admin`:

```
ALTER DEFAULT PRIVILEGES FOR ROLE admin REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

Note however that you *cannot* accomplish that effect with a command limited to a single schema. This command has no effect, unless it is undoing a matching `GRANT`:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public REVOKE EXECUTE ON FUNCTIONS FROM PUBLIC;
```

That's because per-schema default privileges can only add privileges to the global setting, not remove privileges granted by it.

Compatibility

There is no `ALTER DEFAULT PRIVILEGES` statement in the SQL standard.

See Also

GRANT, REVOKE

ALTER DOMAIN

Name

ALTER DOMAIN — change the definition of a domain

Synopsis

```
ALTER DOMAIN name
    { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN name
    { SET | DROP } NOT NULL
ALTER DOMAIN name
    ADD domain_constraint [ NOT VALID ]
ALTER DOMAIN name
    DROP CONSTRAINT [ IF EXISTS ] constraint_name [ RESTRICT | CASCADE ]
ALTER DOMAIN name
     RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER DOMAIN name
    VALIDATE CONSTRAINT constraint_name
ALTER DOMAIN name
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER DOMAIN name
    RENAME TO new_name
ALTER DOMAIN name
    SET SCHEMA new_schema
```

Description

ALTER DOMAIN changes the definition of an existing domain. There are several sub-forms:

SET/DROP DEFAULT

These forms set or remove the default value for a domain. Note that defaults only apply to subsequent INSERT commands; they do not affect rows already in a table using the domain.

SET/DROP NOT NULL

These forms change whether a domain is marked to allow NULL values or to reject NULL values. You can only SET NOT NULL when the columns using the domain contain no null values.

ADD *domain_constraint* [NOT VALID]

This form adds a new constraint to a domain using the same syntax as CREATE DOMAIN. When a new constraint is added to a domain, all columns using that domain will be checked against the newly added constraint. These checks can be suppressed by adding the new constraint using the NOT VALID option; the constraint can later be made valid using ALTER DOMAIN ... VALIDATE CONSTRAINT. Newly inserted or updated rows are always checked against all constraints, even those marked NOT VALID. NOT VALID is only accepted for CHECK constraints.

DROP CONSTRAINT [IF EXISTS]

This form drops constraints on a domain. If `IF EXISTS` is specified and the constraint does not exist, no error is thrown. In this case a notice is issued instead.

RENAME CONSTRAINT

This form changes the name of a constraint on a domain.

VALIDATE CONSTRAINT

This form validates a constraint previously added as `NOT VALID`, that is, it verifies that all values in table columns of the domain type satisfy the specified constraint.

OWNER

This form changes the owner of the domain to the specified user.

`RENAME`

This form changes the name of the domain.

SET SCHEMA

This form changes the schema of the domain. Any constraints associated with the domain are moved into the new schema as well.

You must own the domain to use `ALTER DOMAIN`. To change the schema of a domain, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the domain's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the domain. However, a superuser can alter ownership of any domain anyway.)

Parameters

`name`

The name (possibly schema-qualified) of an existing domain to alter.

`domain_constraint`

New domain constraint for the domain.

`constraint_name`

Name of an existing constraint to drop or rename.

`NOT VALID`

Do not verify existing stored data for constraint validity.

`CASCADE`

Automatically drop objects that depend on the constraint.

`RESTRICT`

Refuse to drop the constraint if there are any dependent objects. This is the default behavior.

new_name

The new name for the domain.

new_constraint_name

The new name for the constraint.

new_owner

The user name of the new owner of the domain.

new_schema

The new schema for the domain.

Notes

Although `ALTER DOMAIN ADD CONSTRAINT` attempts to verify that existing stored data satisfies the new constraint, this check is not bulletproof, because the command cannot "see" table rows that are newly inserted or updated and not yet committed. If there is a hazard that concurrent operations might insert bad data, the way to proceed is to add the constraint using the `NOT VALID` option, commit that command, wait until all transactions started before that commit have finished, and then issue `ALTER DOMAIN VALIDATE CONSTRAINT` to search for data violating the constraint. This method is reliable because once the constraint is committed, all new transactions are guaranteed to enforce it against new values of the domain type.

Currently, `ALTER DOMAIN ADD CONSTRAINT`, `ALTER DOMAIN VALIDATE CONSTRAINT`, and `ALTER DOMAIN SET NOT NULL` will fail if the validated named domain or any derived domain is used within a composite-type column of any table in the database. They should eventually be improved to be able to verify the new constraint for such nested columns.

Examples

To add a `NOT NULL` constraint to a domain:

```
ALTER DOMAIN zipcode SET NOT NULL;
```

To remove a `NOT NULL` constraint from a domain:

```
ALTER DOMAIN zipcode DROP NOT NULL;
```

To add a check constraint to a domain:

```
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK (char_length(VALUE) = 5);
```

To remove a check constraint from a domain:

```
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;
```

To rename a check constraint on a domain:

```
ALTER DOMAIN zipcode RENAME CONSTRAINT zipchk TO zip_check;
```

To move the domain into a different schema:

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

Compatibility

`ALTER DOMAIN` conforms to the SQL standard, except for the `OWNER`, `RENAME`, `SET SCHEMA`, and `VALIDATE CONSTRAINT` variants, which are PostgreSQL extensions. The `NOT VALID` clause of the `ADD CONSTRAINT` variant is also a PostgreSQL extension.

See Also

CREATE DOMAIN, DROP DOMAIN

ALTER EVENT TRIGGER

Name

ALTER EVENT TRIGGER — change the definition of an event trigger

Synopsis

```
ALTER EVENT TRIGGER name DISABLE
ALTER EVENT TRIGGER name ENABLE [ REPLICA | ALWAYS ]
ALTER EVENT TRIGGER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER EVENT TRIGGER name RENAME TO new_name
```

Description

ALTER EVENT TRIGGER changes properties of an existing event trigger.

You must be superuser to alter an event trigger.

Parameters

name

The name of an existing trigger to alter.

new_owner

The user name of the new owner of the event trigger.

new_name

The new name of the event trigger.

DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

These forms configure the firing of event triggers. A disabled trigger is still known to the system, but is not executed when its triggering event occurs. See also session_replication_role.

Compatibility

There is no ALTER EVENT TRIGGER statement in the SQL standard.

See Also

CREATE EVENT TRIGGER, DROP EVENT TRIGGER

ALTER EXTENSION

Name

ALTER EXTENSION — change the definition of an extension

Synopsis

```
ALTER EXTENSION name UPDATE [ TO new_version ]
ALTER EXTENSION name SET SCHEMA new_schema
ALTER EXTENSION name ADD member_object
ALTER EXTENSION name DROP member_object

where member_object is:

  AGGREGATE aggregate_name ( aggregate_signature ) |
  CAST (source_type AS target_type) |
  COLLATION object_name |
  CONVERSION object_name |
  DOMAIN object_name |
  EVENT TRIGGER object_name |
  FOREIGN DATA WRAPPER object_name |
  FOREIGN TABLE object_name |
  FUNCTION function_name ( [ [ argmode ] [ argname ] argtype [, ...] ] ) |
  MATERIALIZED VIEW object_name |
  OPERATOR operator_name (left_type, right_type) |
  OPERATOR CLASS object_name USING index_method |
  OPERATOR FAMILY object_name USING index_method |
  [ PROCEDURAL ] LANGUAGE object_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  SERVER object_name |
  TABLE object_name |
  TEXT SEARCH CONFIGURATION object_name |
  TEXT SEARCH DICTIONARY object_name |
  TEXT SEARCH PARSER object_name |
  TEXT SEARCH TEMPLATE object_name |
  TRANSFORM FOR type_name LANGUAGE lang_name |
  TYPE object_name |
  VIEW object_name

and aggregate_signature is:

* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ] argtype [ , ...
```

Description

`ALTER EXTENSION` changes the definition of an installed extension. There are several subforms:

`UPDATE`

This form updates the extension to a newer version. The extension must supply a suitable update script (or series of scripts) that can modify the currently-installed version into the requested version.

`SET SCHEMA`

This form moves the extension's objects into another schema. The extension has to be *relocatable* for this command to succeed.

`ADD` *`member_object`*

This form adds an existing object to the extension. This is mainly useful in extension update scripts. The object will subsequently be treated as a member of the extension; notably, it can only be dropped by dropping the extension.

`DROP` *`member_object`*

This form removes a member object from the extension. This is mainly useful in extension update scripts. The object is not dropped, only disassociated from the extension.

See Section 35.15 for more information about these operations.

You must own the extension to use `ALTER EXTENSION`. The `ADD`/`DROP` forms require ownership of the added/dropped object as well.

Parameters

`name`

The name of an installed extension.

`new_version`

The desired new version of the extension. This can be written as either an identifier or a string literal. If not specified, `ALTER EXTENSION UPDATE` attempts to update to whatever is shown as the default version in the extension's control file.

`new_schema`

The new schema for the extension.

`object_name`
`aggregate_name`
`function_name`
`operator_name`

The name of an object to be added to or removed from the extension. Names of tables, aggregates, domains, foreign tables, functions, operators, operator classes, operator families, sequences, text search objects, types, and views can be schema-qualified.

source_type

The name of the source data type of the cast.

target_type

The name of the target data type of the cast.

argmode

The mode of a function or aggregate argument: IN, OUT, INOUT, or VARIADIC. If omitted, the default is IN. Note that ALTER EXTENSION does not actually pay any attention to OUT arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list the IN, INOUT, and VARIADIC arguments.

argname

The name of a function or aggregate argument. Note that ALTER EXTENSION does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity.

argtype

The data type of a function or aggregate argument.

left_type
right_type

The data type(s) of the operator's arguments (optionally schema-qualified). Write NONE for the missing argument of a prefix or postfix operator.

PROCEDURAL

This is a noise word.

type_name

The name of the data type of the transform.

lang_name

The name of the language of the transform.

Examples

To update the hstore extension to version 2.0:

```
ALTER EXTENSION hstore UPDATE TO '2.0';
```

To change the schema of the hstore extension to utils:

```
ALTER EXTENSION hstore SET SCHEMA utils;
```

To add an existing function to the hstore extension:

```
ALTER EXTENSION hstore ADD FUNCTION populate_record(anyelement, hstore);
```

Compatibility

`ALTER EXTENSION` is a PostgreSQL extension.

See Also

CREATE EXTENSION, DROP EXTENSION

ALTER FOREIGN DATA WRAPPER

Name

ALTER FOREIGN DATA WRAPPER — change the definition of a foreign-data wrapper

Synopsis

```
ALTER FOREIGN DATA WRAPPER name
    [ HANDLER handler_function | NO HANDLER ]
    [ VALIDATOR validator_function | NO VALIDATOR ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ]) ]
ALTER FOREIGN DATA WRAPPER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER FOREIGN DATA WRAPPER name RENAME TO new_name
```

Description

ALTER FOREIGN DATA WRAPPER changes the definition of a foreign-data wrapper. The first form of the command changes the support functions or the generic options of the foreign-data wrapper (at least one clause is required). The second form changes the owner of the foreign-data wrapper.

Only superusers can alter foreign-data wrappers. Additionally, only superusers can own foreign-data wrappers.

Parameters

name

The name of an existing foreign-data wrapper.

HANDLER *handler_function*

Specifies a new handler function for the foreign-data wrapper.

NO HANDLER

This is used to specify that the foreign-data wrapper should no longer have a handler function.

Note that foreign tables that use a foreign-data wrapper with no handler cannot be accessed.

VALIDATOR *validator_function*

Specifies a new validator function for the foreign-data wrapper.

Note that it is possible that pre-existing options of the foreign-data wrapper, or of dependent servers, user mappings, or foreign tables, are invalid according to the new validator. PostgreSQL does not check for this. It is up to the user to make sure that these options are correct before using the modified foreign-data wrapper. However, any options specified in this ALTER FOREIGN DATA WRAPPER command will be checked using the new validator.

`NO VALIDATOR`

This is used to specify that the foreign-data wrapper should no longer have a validator function.

`OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )`

Change options for the foreign-data wrapper. `ADD`, `SET`, and `DROP` specify the action to be performed. `ADD` is assumed if no operation is explicitly specified. Option names must be unique; names and values are also validated using the foreign data wrapper's validator function, if any.

new_owner

The user name of the new owner of the foreign-data wrapper.

new_name

The new name for the foreign-data wrapper.

Examples

Change a foreign-data wrapper `dbi`, add option `foo`, drop `bar`:

```
ALTER FOREIGN DATA WRAPPER dbi OPTIONS (ADD foo '1', DROP 'bar');
```

Change the foreign-data wrapper `dbi` validator to `bob.myvalidator`:

```
ALTER FOREIGN DATA WRAPPER dbi VALIDATOR bob.myvalidator;
```

Compatibility

`ALTER FOREIGN DATA WRAPPER` conforms to ISO/IEC 9075-9 (SQL/MED), except that the `HANDLER`, `VALIDATOR`, `OWNER TO`, and `RENAME` clauses are extensions.

See Also

CREATE FOREIGN DATA WRAPPER, DROP FOREIGN DATA WRAPPER

ALTER FOREIGN TABLE

Name

ALTER FOREIGN TABLE — change the definition of a foreign table

Synopsis

```
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    action [, ... ]
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema
```

where *action* is one of:

```
    ADD [ COLUMN ] column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation ]
    ALTER [ COLUMN ] column_name SET DEFAULT expression
    ALTER [ COLUMN ] column_name DROP DEFAULT
    ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
    ALTER [ COLUMN ] column_name OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ])
    ADD table_constraint [ NOT VALID ]
    VALIDATE CONSTRAINT constraint_name
    DROP CONSTRAINT [ IF EXISTS ]  constraint_name [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE REPLICA TRIGGER trigger_name
    ENABLE ALWAYS TRIGGER trigger_name
    SET WITH OIDS
    SET WITHOUT OIDS
    INHERIT parent_table
    NO INHERIT parent_table
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
    OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ])
```

Description

ALTER FOREIGN TABLE changes the definition of an existing foreign table. There are several subforms:

ADD COLUMN

This form adds a new column to the foreign table, using the same syntax as CREATE FOREIGN TABLE. Unlike the case when adding a column to a regular table, nothing happens to the underlying storage: this action simply declares that some new column is now accessible through the foreign table.

DROP COLUMN [IF EXISTS]

This form drops a column from a foreign table. You will need to say CASCADE if anything outside the table depends on the column; for example, views. If IF EXISTS is specified and the column does not exist, no error is thrown. In this case a notice is issued instead.

SET DATA TYPE

This form changes the type of a column of a foreign table. Again, this has no effect on any underlying storage: this action simply changes the type that PostgreSQL believes the column to have.

SET/DROP DEFAULT

These forms set or remove the default value for a column. Default values only apply in subsequent INSERT or UPDATE commands; they do not cause rows already in the table to change.

SET/DROP NOT NULL

Mark a column as allowing, or not allowing, null values.

SET STATISTICS

This form sets the per-column statistics-gathering target for subsequent ANALYZE operations. See the similar form of ALTER TABLE for more details.

SET (*attribute_option* = *value* [, ...])
RESET (*attribute_option* [, ...])

This form sets or resets per-attribute options. See the similar form of ALTER TABLE for more details.

SET STORAGE

This form sets the storage mode for a column. See the similar form of ALTER TABLE for more details. Note that the storage mode has no effect unless the table's foreign-data wrapper chooses to pay attention to it.

ADD *table_constraint* [NOT VALID]

This form adds a new constraint to a foreign table, using the same syntax as CREATE FOREIGN TABLE. Currently only CHECK constraints are supported.

Unlike the case when adding a constraint to a regular table, nothing is done to verify the constraint is correct; rather, this action simply declares that some new condition should be assumed to hold for all rows in the foreign table. (See the discussion in CREATE FOREIGN TABLE.) If the constraint is marked NOT VALID, then it isn't assumed to hold, but is only recorded for possible future use.

`VALIDATE CONSTRAINT`

This form marks as valid a constraint that was previously marked as `NOT VALID`. No action is taken to verify the constraint, but future queries will assume that it holds.

`DROP CONSTRAINT [ IF EXISTS ]`

This form drops the specified constraint on a foreign table. If `IF EXISTS` is specified and the constraint does not exist, no error is thrown. In this case a notice is issued instead.

`DISABLE/ENABLE [ REPLICA | ALWAYS ] TRIGGER`

These forms configure the firing of trigger(s) belonging to the foreign table. See the similar form of ALTER TABLE for more details.

`SET WITH OIDS`

This form adds an `oid` system column to the table (see Section 5.4). It does nothing if the table already has OIDs. Unless the table's foreign-data wrapper supports OIDs, this column will simply read as zeroes.

Note that this is not equivalent to `ADD COLUMN oid oid`; that would add a normal column that happened to be named `oid`, not a system column.

`SET WITHOUT OIDS`

This form removes the `oid` system column from the table. This is exactly equivalent to `DROP COLUMN oid RESTRICT`, except that it will not complain if there is already no `oid` column.

`INHERIT` *`parent_table`*

This form adds the target foreign table as a new child of the specified parent table. See the similar form of ALTER TABLE for more details.

`NO INHERIT` *`parent_table`*

This form removes the target foreign table from the list of children of the specified parent table.

`OWNER`

This form changes the owner of the foreign table to the specified user.

`OPTIONS ( [ ADD | SET | DROP ]` *`option`* `['`*`value`*`'] [, ... ] )`

Change options for the foreign table or one of its columns. `ADD`, `SET`, and `DROP` specify the action to be performed. `ADD` is assumed if no operation is explicitly specified. Duplicate option names are not allowed (although it's OK for a table option and a column option to have the same name). Option names and values are also validated using the foreign data wrapper library.

`RENAME`

The `RENAME` forms change the name of a foreign table or the name of an individual column in a foreign table.

`SET SCHEMA`

This form moves the foreign table into another schema.

All the actions except `RENAME` and `SET SCHEMA` can be combined into a list of multiple alterations to apply in parallel. For example, it is possible to add several columns and/or alter the type of several columns in a single command.

If the command is written as `ALTER FOREIGN TABLE IF EXISTS ...` and the foreign table does not exist, no error is thrown. A notice is issued in this case.

You must own the table to use `ALTER FOREIGN TABLE`. To change the schema of a foreign table, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the table's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the table. However, a superuser can alter ownership of any table anyway.) To add a column or alter a column type, you must also have `USAGE` privilege on the data type.

Parameters

name

The name (possibly schema-qualified) of an existing foreign table to alter. If `ONLY` is specified before the table name, only that table is altered. If `ONLY` is not specified, the table and all its descendant tables (if any) are altered. Optionally, `*` can be specified after the table name to explicitly indicate that descendant tables are included.

column_name

Name of a new or existing column.

new_column_name

New name for an existing column.

new_name

New name for the table.

data_type

Data type of the new column, or new data type for an existing column.

table_constraint

New table constraint for the foreign table.

constraint_name

Name of an existing constraint to drop.

`CASCADE`

Automatically drop objects that depend on the dropped column or constraint (for example, views referencing the column).

`RESTRICT`

Refuse to drop the column or constraint if there are any dependent objects. This is the default behavior.

trigger_name

Name of a single trigger to disable or enable.

ALL

Disable or enable all triggers belonging to the foreign table. (This requires superuser privilege if any of the triggers are internally generated triggers. The core system does not add such triggers to foreign tables, but add-on code could do so.)

USER

Disable or enable all triggers belonging to the foreign table except for internally generated triggers.

parent_table

A parent table to associate or de-associate with this foreign table.

new_owner

The user name of the new owner of the table.

new_schema

The name of the schema to which the table will be moved.

Notes

The key word COLUMN is noise and can be omitted.

Consistency with the foreign server is not checked when a column is added or removed with ADD COLUMN or DROP COLUMN, a NOT NULL or CHECK constraint is added, or a column type is changed with SET DATA TYPE. It is the user's responsibility to ensure that the table definition matches the remote side.

Refer to CREATE FOREIGN TABLE for a further description of valid parameters.

Examples

To mark a column as not-null:

```
ALTER FOREIGN TABLE distributors ALTER COLUMN street SET NOT NULL;
```

To change options of a foreign table:

```
ALTER FOREIGN TABLE myschema.distributors OPTIONS (ADD opt1 'value', SET opt2 'value2', DRO
```

Compatibility

The forms ADD, DROP, and SET DATA TYPE conform with the SQL standard. The other forms are PostgreSQL extensions of the SQL standard. Also, the ability to specify more than one manipulation in a single ALTER FOREIGN TABLE command is an extension.

ALTER FOREIGN TABLE DROP COLUMN can be used to drop the only column of a foreign table, leaving a zero-column table. This is an extension of SQL, which disallows zero-column foreign tables.

See Also

CREATE FOREIGN TABLE, DROP FOREIGN TABLE

ALTER FUNCTION

Name

ALTER FUNCTION — change the definition of a function

Synopsis

```
ALTER FUNCTION name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
    action [ ... ] [ RESTRICT ]
ALTER FUNCTION name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
    RENAME TO new_name
ALTER FUNCTION name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER FUNCTION name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
    SET SCHEMA new_schema
```

where *action* is one of:

```
    CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    COST execution_cost
    ROWS result_rows
    SET configuration_parameter { TO | = } { value | DEFAULT }
    SET configuration_parameter FROM CURRENT
    RESET configuration_parameter
    RESET ALL
```

Description

ALTER FUNCTION changes the definition of a function.

You must own the function to use ALTER FUNCTION. To change a function's schema, you must also have CREATE privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the function's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the function. However, a superuser can alter ownership of any function anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing function.

argmode

The mode of an argument: IN, OUT, INOUT, or VARIADIC. If omitted, the default is IN. Note that ALTER FUNCTION does not actually pay any attention to OUT arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list the IN, INOUT, and VARIADIC arguments.

argname

The name of an argument. Note that ALTER FUNCTION does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity.

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any.

new_name

The new name of the function.

new_owner

The new owner of the function. Note that if the function is marked SECURITY DEFINER, it will subsequently execute as the new owner.

new_schema

The new schema for the function.

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

CALLED ON NULL INPUT changes the function so that it will be invoked when some or all of its arguments are null. RETURNS NULL ON NULL INPUT or STRICT changes the function so that it is not invoked if any of its arguments are null; instead, a null result is assumed automatically. See CREATE FUNCTION for more information.

IMMUTABLE
STABLE
VOLATILE

Change the volatility of the function to the specified setting. See CREATE FUNCTION for details.

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

Change whether the function is a security definer or not. The key word EXTERNAL is ignored for SQL conformance. See CREATE FUNCTION for more information about this capability.

LEAKPROOF

Change whether the function is considered leakproof or not. See CREATE FUNCTION for more information about this capability.

COST *execution_cost*

Change the estimated execution cost of the function. See CREATE FUNCTION for more information.

ROWS *result_rows*

Change the estimated number of rows returned by a set-returning function. See CREATE FUNCTION for more information.

configuration_parameter
value

Add or change the assignment to be made to a configuration parameter when the function is called. If *value* is DEFAULT or, equivalently, RESET is used, the function-local setting is removed, so that the function executes with the value present in its environment. Use RESET ALL to clear all function-local settings. SET FROM CURRENT saves the value of the parameter that is current when ALTER FUNCTION is executed as the value to be applied when the function is entered.

See SET and Chapter 18 for more information about allowed parameter names and values.

RESTRICT

Ignored for conformance with the SQL standard.

Examples

To rename the function sqrt for type integer to square_root:

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

To change the owner of the function sqrt for type integer to joe:

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

To change the schema of the function sqrt for type integer to maths:

```
ALTER FUNCTION sqrt(integer) SET SCHEMA maths;
```

To adjust the search path that is automatically set for a function:

```
ALTER FUNCTION check_password(text) SET search_path = admin, pg_temp;
```

To disable automatic setting of search_path for a function:

```
ALTER FUNCTION check_password(text) RESET search_path;
```

The function will now execute with whatever search path is used by its caller.

Compatibility

This statement is partially compatible with the `ALTER FUNCTION` statement in the SQL standard. The standard allows more properties of a function to be modified, but does not provide the ability to rename a function, make a function a security definer, attach configuration parameter values to a function, or change the owner, schema, or volatility of a function. The standard also requires the `RESTRICT` key word, which is optional in PostgreSQL.

See Also

CREATE FUNCTION, DROP FUNCTION

ALTER GROUP

Name

ALTER GROUP — change role name or membership

Synopsis

```
ALTER GROUP role_specification ADD USER user_name [, ... ]
ALTER GROUP role_specification DROP USER user_name [, ... ]

where role_specification can be:

    role_name
  | CURRENT_USER
  | SESSION_USER

ALTER GROUP group_name RENAME TO new_name
```

Description

ALTER GROUP changes the attributes of a user group. This is an obsolete command, though still accepted for backwards compatibility, because groups (and users too) have been superseded by the more general concept of roles.

The first two variants add users to a group or remove them from a group. (Any role can play the part of either a "user" or a "group" for this purpose.) These variants are effectively equivalent to granting or revoking membership in the role named as the "group"; so the preferred way to do this is to use GRANT or REVOKE.

The third variant changes the name of the group. This is exactly equivalent to renaming the role with ALTER ROLE.

Parameters

group_name

The name of the group (role) to modify.

user_name

Users (roles) that are to be added to or removed from the group. The users must already exist; ALTER GROUP does not create or drop users.

new_name

The new name of the group.

Examples

Add users to a group:

```
ALTER GROUP staff ADD USER karl, john;
```

Remove a user from a group:

```
ALTER GROUP workers DROP USER beth;
```

Compatibility

There is no `ALTER GROUP` statement in the SQL standard.

See Also

GRANT, REVOKE, ALTER ROLE

ALTER INDEX

Name

ALTER INDEX — change the definition of an index

Synopsis

```
ALTER INDEX [ IF EXISTS ] name RENAME TO new_name
ALTER INDEX [ IF EXISTS ] name SET TABLESPACE tablespace_name
ALTER INDEX [ IF EXISTS ] name SET ( storage_parameter [= value] [, ... ] )
ALTER INDEX [ IF EXISTS ] name RESET ( storage_parameter [, ... ] )
ALTER INDEX ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]
```

Description

ALTER INDEX changes the definition of an existing index. There are several subforms:

RENAME

The RENAME form changes the name of the index. There is no effect on the stored data.

SET TABLESPACE

This form changes the index's tablespace to the specified tablespace and moves the data file(s) associated with the index to the new tablespace. To change the tablespace of an index, you must own the index and have CREATE privilege on the new tablespace. All indexes in the current database in a tablespace can be moved by using the ALL IN TABLESPACE form, which will lock all indexes to be moved and then move each one. This form also supports OWNED BY, which will only move indexes owned by the roles specified. If the NOWAIT option is specified then the command will fail if it is unable to acquire all of the locks required immediately. Note that system catalogs will not be moved by this command, use ALTER DATABASE or explicit ALTER INDEX invocations instead if desired. See also CREATE TABLESPACE.

SET (*storage_parameter* [= *value*] [, ...])

This form changes one or more index-method-specific storage parameters for the index. See CREATE INDEX for details on the available parameters. Note that the index contents will not be modified immediately by this command; depending on the parameter you might need to rebuild the index with REINDEX to get the desired effects.

RESET (*storage_parameter* [, ...])

This form resets one or more index-method-specific storage parameters to their defaults. As with SET, a REINDEX might be needed to update the index entirely.

Parameters

`IF EXISTS`

Do not throw an error if the index does not exist. A notice is issued in this case.

`name`

The name (possibly schema-qualified) of an existing index to alter.

`new_name`

The new name for the index.

`tablespace_name`

The tablespace to which the index will be moved.

`storage_parameter`

The name of an index-method-specific storage parameter.

`value`

The new value for an index-method-specific storage parameter. This might be a number or a word depending on the parameter.

Notes

These operations are also possible using ALTER TABLE. `ALTER INDEX` is in fact just an alias for the forms of `ALTER TABLE` that apply to indexes.

There was formerly an `ALTER INDEX OWNER` variant, but this is now ignored (with a warning). An index cannot have an owner different from its table's owner. Changing the table's owner automatically changes the index as well.

Changing any part of a system catalog index is not permitted.

Examples

To rename an existing index:

```
ALTER INDEX distributors RENAME TO suppliers;
```

To move an index to a different tablespace:

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

To change an index's fill factor (assuming that the index method supports it):

```
ALTER INDEX distributors SET (fillfactor = 75);
REINDEX INDEX distributors;
```

Compatibility

`ALTER INDEX` is a PostgreSQL extension.

See Also

CREATE INDEX, REINDEX

ALTER LANGUAGE

Name

ALTER LANGUAGE — change the definition of a procedural language

Synopsis

```
ALTER [ PROCEDURAL ] LANGUAGE name RENAME TO new_name
ALTER [ PROCEDURAL ] LANGUAGE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

Description

ALTER LANGUAGE changes the definition of a procedural language. The only functionality is to rename the language or assign a new owner. You must be superuser or owner of the language to use ALTER LANGUAGE.

Parameters

name

Name of a language

new_name

The new name of the language

new_owner

The new owner of the language

Compatibility

There is no ALTER LANGUAGE statement in the SQL standard.

See Also

CREATE LANGUAGE, DROP LANGUAGE

ALTER LARGE OBJECT

Name

ALTER LARGE OBJECT — change the definition of a large object

Synopsis

```
ALTER LARGE OBJECT large_object_oid OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

Description

ALTER LARGE OBJECT changes the definition of a large object.

You must own the large object to use ALTER LARGE OBJECT. To alter the owner, you must also be a direct or indirect member of the new owning role. (However, a superuser can alter any large object anyway.) Currently, the only functionality is to assign a new owner, so both restrictions always apply.

Parameters

large_object_oid

OID of the large object to be altered

new_owner

The new owner of the large object

Compatibility

There is no ALTER LARGE OBJECT statement in the SQL standard.

See Also

Chapter 32

ALTER MATERIALIZED VIEW

Name

ALTER MATERIALIZED VIEW — change the definition of a materialized view

Synopsis

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    action [, ... ]
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    RENAME TO new_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    SET SCHEMA new_schema
ALTER MATERIALIZED VIEW ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]

where action is one of:

    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
    CLUSTER ON index_name
    SET WITHOUT CLUSTER
    SET ( storage_parameter [= value] [, ... ] )
    RESET ( storage_parameter [, ... ] )
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

Description

ALTER MATERIALIZED VIEW changes various auxiliary properties of an existing materialized view.

You must own the materialized view to use ALTER MATERIALIZED VIEW. To change a materialized view's schema, you must also have CREATE privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the materialized view's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the materialized view. However, a superuser can alter ownership of any view anyway.)

The statement subforms and actions available for ALTER MATERIALIZED VIEW are a subset of those available for ALTER TABLE, and have the same meaning when used for materialized views. See the descriptions for ALTER TABLE for details.

Parameters

name

The name (optionally schema-qualified) of an existing materialized view.

column_name

Name of a new or existing column.

new_column_name

New name for an existing column.

new_owner

The user name of the new owner of the materialized view.

new_name

The new name for the materialized view.

new_schema

The new schema for the materialized view.

Examples

To rename the materialized view `foo` to `bar`:

```
ALTER MATERIALIZED VIEW foo RENAME TO bar;
```

Compatibility

`ALTER MATERIALIZED VIEW` is a PostgreSQL extension.

See Also

CREATE MATERIALIZED VIEW, DROP MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

ALTER OPERATOR

Name

ALTER OPERATOR — change the definition of an operator

Synopsis

```
ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
    SET SCHEMA new_schema
```

Description

ALTER OPERATOR changes the definition of an operator. The only currently available functionality is to change the owner of the operator.

You must own the operator to use ALTER OPERATOR. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the operator's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the operator. However, a superuser can alter ownership of any operator anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing operator.

left_type

The data type of the operator's left operand; write NONE if the operator has no left operand.

right_type

The data type of the operator's right operand; write NONE if the operator has no right operand.

new_owner

The new owner of the operator.

new_schema

The new schema for the operator.

Examples

Change the owner of a custom operator `a @@ b` for type `text`:

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

Compatibility

There is no `ALTER OPERATOR` statement in the SQL standard.

See Also

CREATE OPERATOR, DROP OPERATOR

ALTER OPERATOR CLASS

Name

ALTER OPERATOR CLASS — change the definition of an operator class

Synopsis

```
ALTER OPERATOR CLASS name USING index_method
    RENAME TO new_name

ALTER OPERATOR CLASS name USING index_method
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER OPERATOR CLASS name USING index_method
    SET SCHEMA new_schema
```

Description

ALTER OPERATOR CLASS changes the definition of an operator class.

You must own the operator class to use ALTER OPERATOR CLASS. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the operator class's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the operator class. However, a superuser can alter ownership of any operator class anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing operator class.

index_method

The name of the index method this operator class is for.

new_name

The new name of the operator class.

new_owner

The new owner of the operator class.

new_schema

The new schema for the operator class.

Compatibility

There is no `ALTER OPERATOR CLASS` statement in the SQL standard.

See Also

CREATE OPERATOR CLASS, DROP OPERATOR CLASS, ALTER OPERATOR FAMILY

ALTER OPERATOR FAMILY

Name

ALTER OPERATOR FAMILY — change the definition of an operator family

Synopsis

```
ALTER OPERATOR FAMILY name USING index_method ADD
  {  OPERATOR strategy_number operator_name ( op_type, op_type )
              [ FOR SEARCH | FOR ORDER BY sort_family_name ]
   | FUNCTION support_number [ ( op_type [ , op_type ] ) ]
              function_name ( argument_type [, ...] )
  } [, ... ]

ALTER OPERATOR FAMILY name USING index_method DROP
  {  OPERATOR strategy_number ( op_type [ , op_type ] )
   | FUNCTION support_number ( op_type [ , op_type ] )
  } [, ... ]

ALTER OPERATOR FAMILY name USING index_method
    RENAME TO new_name

ALTER OPERATOR FAMILY name USING index_method
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER OPERATOR FAMILY name USING index_method
    SET SCHEMA new_schema
```

Description

ALTER OPERATOR FAMILY changes the definition of an operator family. You can add operators and support functions to the family, remove them from the family, or change the family's name or owner.

When operators and support functions are added to a family with ALTER OPERATOR FAMILY, they are not part of any specific operator class within the family, but are just "loose" within the family. This indicates that these operators and functions are compatible with the family's semantics, but are not required for correct functioning of any specific index. (Operators and functions that are so required should be declared as part of an operator class, instead; see CREATE OPERATOR CLASS.) PostgreSQL will allow loose members of a family to be dropped from the family at any time, but members of an operator class cannot be dropped without dropping the whole class and any indexes that depend on it. Typically, single-data-type operators and functions are part of operator classes because they are needed to support an index on that specific data type, while cross-data-type operators and functions are made loose members of the family.

You must be a superuser to use ALTER OPERATOR FAMILY. (This restriction is made because an erroneous operator family definition could confuse or even crash the server.)

`ALTER OPERATOR FAMILY` does not presently check whether the operator family definition includes all the operators and functions required by the index method, nor whether the operators and functions form a self-consistent set. It is the user's responsibility to define a valid operator family.

Refer to Section 35.14 for further information.

Parameters

name

The name (optionally schema-qualified) of an existing operator family.

index_method

The name of the index method this operator family is for.

strategy_number

The index method's strategy number for an operator associated with the operator family.

operator_name

The name (optionally schema-qualified) of an operator associated with the operator family.

op_type

In an `OPERATOR` clause, the operand data type(s) of the operator, or `NONE` to signify a left-unary or right-unary operator. Unlike the comparable syntax in `CREATE OPERATOR CLASS`, the operand data types must always be specified.

In an `ADD FUNCTION` clause, the operand data type(s) the function is intended to support, if different from the input data type(s) of the function. For B-tree comparison functions and hash functions it is not necessary to specify *op_type* since the function's input data type(s) are always the correct ones to use. For B-tree sort support functions and all functions in GiST, SP-GiST and GIN operator classes, it is necessary to specify the operand data type(s) the function is to be used with.

In a `DROP FUNCTION` clause, the operand data type(s) the function is intended to support must be specified.

sort_family_name

The name (optionally schema-qualified) of an existing `btree` operator family that describes the sort ordering associated with an ordering operator.

If neither `FOR SEARCH` nor `FOR ORDER BY` is specified, `FOR SEARCH` is the default.

support_number

The index method's support procedure number for a function associated with the operator family.

function_name

The name (optionally schema-qualified) of a function that is an index method support procedure for the operator family.

argument_type

The parameter data type(s) of the function.

new_name

The new name of the operator family.

new_owner

The new owner of the operator family.

new_schema

The new schema for the operator family.

The OPERATOR and FUNCTION clauses can appear in any order.

Notes

Notice that the DROP syntax only specifies the "slot" in the operator family, by strategy or support number and input data type(s). The name of the operator or function occupying the slot is not mentioned. Also, for DROP FUNCTION the type(s) to specify are the input data type(s) the function is intended to support; for GiST, SP-GiST and GIN indexes this might have nothing to do with the actual input argument types of the function.

Because the index machinery does not check access permissions on functions before using them, including a function or operator in an operator family is tantamount to granting public execute permission on it. This is usually not an issue for the sorts of functions that are useful in an operator family.

The operators should not be defined by SQL functions. A SQL function is likely to be inlined into the calling query, which will prevent the optimizer from recognizing that the query matches an index.

Before PostgreSQL 8.4, the OPERATOR clause could include a RECHECK option. This is no longer supported because whether an index operator is "lossy" is now determined on-the-fly at run time. This allows efficient handling of cases where an operator might or might not be lossy.

Examples

The following example command adds cross-data-type operators and support functions to an operator family that already contains B-tree operator classes for data types int4 and int2.

```
ALTER OPERATOR FAMILY integer_ops USING btree ADD

  -- int4 vs int2
  OPERATOR 1 < (int4, int2) ,
  OPERATOR 2 <= (int4, int2) ,
  OPERATOR 3 = (int4, int2) ,
  OPERATOR 4 >= (int4, int2) ,
  OPERATOR 5 > (int4, int2) ,
  FUNCTION 1 btint42cmp(int4, int2) ,

  -- int2 vs int4
  OPERATOR 1 < (int2, int4) ,
  OPERATOR 2 <= (int2, int4) ,
  OPERATOR 3 = (int2, int4) ,
  OPERATOR 4 >= (int2, int4) ,
```

```
  OPERATOR 5 > (int2, int4) ,
  FUNCTION 1 btint24cmp(int2, int4) ;
```

To remove these entries again:

```
ALTER OPERATOR FAMILY integer_ops USING btree DROP

  -- int4 vs int2
  OPERATOR 1 (int4, int2) ,
  OPERATOR 2 (int4, int2) ,
  OPERATOR 3 (int4, int2) ,
  OPERATOR 4 (int4, int2) ,
  OPERATOR 5 (int4, int2) ,
  FUNCTION 1 (int4, int2) ,

  -- int2 vs int4
  OPERATOR 1 (int2, int4) ,
  OPERATOR 2 (int2, int4) ,
  OPERATOR 3 (int2, int4) ,
  OPERATOR 4 (int2, int4) ,
  OPERATOR 5 (int2, int4) ,
  FUNCTION 1 (int2, int4) ;
```

Compatibility

There is no `ALTER OPERATOR FAMILY` statement in the SQL standard.

See Also

CREATE OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

ALTER POLICY

Name

ALTER POLICY — change the definition of a row level security policy

Synopsis

```
ALTER POLICY name ON table_name RENAME TO new_name

ALTER POLICY name ON table_name
    [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( using_expression ) ]
    [ WITH CHECK ( check_expression ) ]
```

Description

ALTER POLICY changes the definition of an existing row-level security policy.

To use ALTER POLICY, you must own the table that the policy applies to.

In the second form of ALTER POLICY, the role list, *using_expression*, and *check_expression* are replaced independently if specified. When one of those clauses is omitted, the corresponding part of the policy is unchanged.

Parameters

name

The name of an existing policy to alter.

table_name

The name (optionally schema-qualified) of the table that the policy is on.

new_name

The new name for the policy.

role_name

The role(s) to which the policy applies. Multiple roles can be specified at one time. To apply the policy to all roles, use PUBLIC.

using_expression

The USING expression for the policy. See CREATE POLICY for details.

check_expression

The WITH CHECK expression for the policy. See CREATE POLICY for details.

Compatibility

`ALTER POLICY` is a PostgreSQL extension.

See Also

CREATE POLICY, DROP POLICY

ALTER ROLE

Name

ALTER ROLE — change a database role

Synopsis

```
ALTER ROLE role_specification [ WITH ] option [ ... ]

where option can be:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'

ALTER ROLE name RENAME TO new_name

ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] SET configuration_parameter
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] SET configuration_parameter
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] RESET configuration_parame
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] RESET ALL

where role_specification can be:

    role_name
  | CURRENT_USER
  | SESSION_USER
```

Description

ALTER ROLE changes the attributes of a PostgreSQL role.

The first variant of this command listed in the synopsis can change many of the role attributes that can be specified in CREATE ROLE. (All the possible attributes are covered, except that there are no options for adding or removing memberships; use GRANT and REVOKE for that.) Attributes not mentioned in the command retain their previous settings. Database superusers can change any of these settings for any role. Roles having CREATEROLE privilege can change any of these settings except SUPERUSER, REPLICATION, and BYPASSRLS; but only for non-superuser and non-replication roles. Ordinary roles can only change their own password.

The second variant changes the name of the role. Database superusers can rename any role. Roles having `CREATEROLE` privilege can rename non-superuser roles. The current session user cannot be renamed. (Connect as a different user if you need to do that.) Because `MD5`-encrypted passwords use the role name as cryptographic salt, renaming a role clears its password if the password is `MD5`-encrypted.

The remaining variants change a role's session default for a configuration variable, either for all databases or, when the `IN DATABASE` clause is specified, only for sessions in the named database. If `ALL` is specified instead of a role name, this changes the setting for all roles. Using `ALL` with `IN DATABASE` is effectively the same as using the command `ALTER DATABASE ... SET ...`.

Whenever the role subsequently starts a new session, the specified value becomes the session default, overriding whatever setting is present in `postgresql.conf` or has been received from the `postgres` command line. This only happens at login time; executing SET ROLE or SET SESSION AUTHORIZATION does not cause new configuration values to be set. Settings set for all databases are overridden by database-specific settings attached to a role. Settings for specific databases or specific roles override settings for all roles.

Superusers can change anyone's session defaults. Roles having `CREATEROLE` privilege can change defaults for non-superuser roles. Ordinary roles can only set defaults for themselves. Certain configuration variables cannot be set this way, or can only be set if a superuser issues the command. Only superusers can change a setting for all roles in all databases.

Parameters

name

The name of the role whose attributes are to be altered.

CURRENT_USER

Alter the current user instead of an explicitly identified role.

SESSION_USER

Alter the current session user instead of an explicitly identified role.

SUPERUSER
NOSUPERUSER
CREATEDB
NOCREATEDB
CREATEROLE
NOCREATEROLE
CREATEUSER
NOCREATEUSER
INHERIT
NOINHERIT
LOGIN
NOLOGIN
REPLICATION
NOREPLICATION
BYPASSRLS
NOBYPASSRLS
CONNECTION LIMIT *connlimit*
PASSWORD *password*
ENCRYPTED
UNENCRYPTED
VALID UNTIL '*timestamp*'

These clauses alter attributes originally set by CREATE ROLE. For more information, see the CREATE ROLE reference page.

new_name

The new name of the role.

database_name

The name of the database the configuration variable should be set in.

configuration_parameter
value

Set this role's session default for the specified configuration parameter to the given value. If *value* is DEFAULT or, equivalently, RESET is used, the role-specific variable setting is removed, so the role will inherit the system-wide default setting in new sessions. Use RESET ALL to clear all role-specific settings. SET FROM CURRENT saves the session's current value of the parameter as the role-specific value. If IN DATABASE is specified, the configuration parameter is set or removed for the given role and database only.

Role-specific variable settings take effect only at login; SET ROLE and SET SESSION AUTHORIZATION do not process role-specific variable settings.

See SET and Chapter 18 for more information about allowed parameter names and values.

Notes

Use CREATE ROLE to add new roles, and DROP ROLE to remove a role.

ALTER ROLE cannot change a role's memberships. Use GRANT and REVOKE to do that.

Caution must be exercised when specifying an unencrypted password with this command. The password will be transmitted to the server in cleartext, and it might also be logged in the client's command history or the server log. psql contains a command `\password` that can be used to change a role's password without exposing the cleartext password.

It is also possible to tie a session default to a specific database rather than to a role; see ALTER DATABASE. If there is a conflict, database-role-specific settings override role-specific ones, which in turn override database-specific ones.

Examples

Change a role's password:

```
ALTER ROLE davide WITH PASSWORD 'hu8jmn3';
```

Remove a role's password:

```
ALTER ROLE davide WITH PASSWORD NULL;
```

Change a password expiration date, specifying that the password should expire at midday on 4th May 2015 using the time zone which is one hour ahead of UTC:

```
ALTER ROLE chris VALID UNTIL 'May 4 12:00:00 2015 +1';
```

Make a password valid forever:

```
ALTER ROLE fred VALID UNTIL 'infinity';
```

Give a role the ability to create other roles and new databases:

```
ALTER ROLE miriam CREATEROLE CREATEDB;
```

Give a role a non-default setting of the maintenance_work_mem parameter:

```
ALTER ROLE worker_bee SET maintenance_work_mem = 100000;
```

Give a role a non-default, database-specific setting of the client_min_messages parameter:

```
ALTER ROLE fred IN DATABASE devel SET client_min_messages = DEBUG;
```

Compatibility

The `ALTER ROLE` statement is a PostgreSQL extension.

See Also

CREATE ROLE, DROP ROLE, ALTER DATABASE, SET

ALTER RULE

Name

ALTER RULE — change the definition of a rule

Synopsis

```
ALTER RULE name ON table_name RENAME TO new_name
```

Description

ALTER RULE changes properties of an existing rule. Currently, the only available action is to change the rule's name.

To use ALTER RULE, you must own the table or view that the rule applies to.

Parameters

name

The name of an existing rule to alter.

table_name

The name (optionally schema-qualified) of the table or view that the rule applies to.

new_name

The new name for the rule.

Examples

To rename an existing rule:

```
ALTER RULE notify_all ON emp RENAME TO notify_me;
```

Compatibility

ALTER RULE is a PostgreSQL language extension, as is the entire query rewrite system.

See Also

CREATE RULE, DROP RULE

ALTER SCHEMA

Name

ALTER SCHEMA — change the definition of a schema

Synopsis

```
ALTER SCHEMA name RENAME TO new_name
ALTER SCHEMA name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

Description

ALTER SCHEMA changes the definition of a schema.

You must own the schema to use ALTER SCHEMA. To rename a schema you must also have the CREATE privilege for the database. To alter the owner, you must also be a direct or indirect member of the new owning role, and you must have the CREATE privilege for the database. (Note that superusers have all these privileges automatically.)

Parameters

name

The name of an existing schema.

new_name

The new name of the schema. The new name cannot begin with pg_, as such names are reserved for system schemas.

new_owner

The new owner of the schema.

Compatibility

There is no ALTER SCHEMA statement in the SQL standard.

See Also

CREATE SCHEMA, DROP SCHEMA

ALTER SEQUENCE

Name

ALTER SEQUENCE — change the definition of a sequence generator

Synopsis

```
ALTER SEQUENCE [ IF EXISTS ] name [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ]
    [ RESTART [ [ WITH ] restart ] ]
    [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
ALTER SEQUENCE [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER SEQUENCE [ IF EXISTS ] name RENAME TO new_name
ALTER SEQUENCE [ IF EXISTS ] name SET SCHEMA new_schema
```

Description

ALTER SEQUENCE changes the parameters of an existing sequence generator. Any parameters not specifically set in the ALTER SEQUENCE command retain their prior settings.

You must own the sequence to use ALTER SEQUENCE. To change a sequence's schema, you must also have CREATE privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the sequence's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the sequence. However, a superuser can alter ownership of any sequence anyway.)

Parameters

name

The name (optionally schema-qualified) of a sequence to be altered.

IF EXISTS

Do not throw an error if the sequence does not exist. A notice is issued in this case.

increment

The clause INCREMENT BY *increment* is optional. A positive value will make an ascending sequence, a negative one a descending sequence. If unspecified, the old increment value will be maintained.

minvalue
NO MINVALUE

The optional clause MINVALUE *minvalue* determines the minimum value a sequence can generate. If NO MINVALUE is specified, the defaults of 1 and -2^{63}-1 for ascending and descending sequences, respectively, will be used. If neither option is specified, the current minimum value will be maintained.

maxvalue
NO MAXVALUE

The optional clause MAXVALUE *maxvalue* determines the maximum value for the sequence. If NO MAXVALUE is specified, the defaults are 2^{63}-1 and -1 for ascending and descending sequences, respectively, will be used. If neither option is specified, the current maximum value will be maintained.

start

The optional clause START WITH *start* changes the recorded start value of the sequence. This has no effect on the *current* sequence value; it simply sets the value that future ALTER SEQUENCE RESTART commands will use.

restart

The optional clause RESTART [WITH *restart*] changes the current value of the sequence. This is equivalent to calling the setval function with is_called = false: the specified value will be returned by the *next* call of nextval. Writing RESTART with no *restart* value is equivalent to supplying the start value that was recorded by CREATE SEQUENCE or last set by ALTER SEQUENCE START WITH.

cache

The clause CACHE *cache* enables sequence numbers to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., no cache). If unspecified, the old cache value will be maintained.

CYCLE

The optional CYCLE key word can be used to enable the sequence to wrap around when the *maxvalue* or *minvalue* has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the *minvalue* or *maxvalue*, respectively.

NO CYCLE

If the optional NO CYCLE key word is specified, any calls to nextval after the sequence has reached its maximum value will return an error. If neither CYCLE or NO CYCLE are specified, the old cycle behavior will be maintained.

OWNED BY *table_name.column_name*
OWNED BY NONE

The OWNED BY option causes the sequence to be associated with a specific table column, such that if that column (or its whole table) is dropped, the sequence will be automatically dropped as well. If specified, this association replaces any previously specified association for the sequence. The specified table must have the same owner and be in the same schema as the sequence. Specifying OWNED BY NONE removes any existing association, making the sequence “free-standing”.

new_owner

The user name of the new owner of the sequence.

new_name

The new name for the sequence.

new_schema

The new schema for the sequence.

Notes

To avoid blocking of concurrent transactions that obtain numbers from the same sequence, ALTER SEQUENCE's effects on the sequence generation parameters are never rolled back; those changes take effect immediately and are not reversible. However, the OWNED BY, OWNER TO, RENAME TO, and SET SCHEMA clauses cause ordinary catalog updates that can be rolled back.

ALTER SEQUENCE will not immediately affect nextval results in backends, other than the current one, that have preallocated (cached) sequence values. They will use up all cached values prior to noticing the changed sequence generation parameters. The current backend will be affected immediately.

ALTER SEQUENCE does not affect the currval status for the sequence. (Before PostgreSQL 8.3, it sometimes did.)

For historical reasons, ALTER TABLE can be used with sequences too; but the only variants of ALTER TABLE that are allowed with sequences are equivalent to the forms shown above.

Examples

Restart a sequence called serial, at 105:

```
ALTER SEQUENCE serial RESTART WITH 105;
```

Compatibility

ALTER SEQUENCE conforms to the SQL standard, except for the START WITH, OWNED BY, OWNER TO, RENAME TO, and SET SCHEMA clauses, which are PostgreSQL extensions.

See Also

CREATE SEQUENCE, DROP SEQUENCE

ALTER SERVER

Name

ALTER SERVER — change the definition of a foreign server

Synopsis

```
ALTER SERVER name [ VERSION 'new_version' ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] ) ]
ALTER SERVER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER SERVER name RENAME TO new_name
```

Description

ALTER SERVER changes the definition of a foreign server. The first form changes the server version string or the generic options of the server (at least one clause is required). The second form changes the owner of the server.

To alter the server you must be the owner of the server. Additionally to alter the owner, you must own the server and also be a direct or indirect member of the new owning role, and you must have USAGE privilege on the server's foreign-data wrapper. (Note that superusers satisfy all these criteria automatically.)

Parameters

name

The name of an existing server.

new_version

New server version.

OPTIONS ([ADD | SET | DROP] *option* ['*value*'] [, ...])

Change options for the server. ADD, SET, and DROP specify the action to be performed. ADD is assumed if no operation is explicitly specified. Option names must be unique; names and values are also validated using the server's foreign-data wrapper library.

new_owner

The user name of the new owner of the foreign server.

new_name

The new name for the foreign server.

Examples

Alter server `foo`, add connection options:

```
ALTER SERVER foo OPTIONS (host 'foo', dbname 'foodb');
```

Alter server `foo`, change version, change `host` option:

```
ALTER SERVER foo VERSION '8.4' OPTIONS (SET host 'baz');
```

Compatibility

`ALTER SERVER` conforms to ISO/IEC 9075-9 (SQL/MED). The `OWNER TO` and `RENAME` forms are PostgreSQL extensions.

See Also

CREATE SERVER, DROP SERVER

ALTER SYSTEM

Name

ALTER SYSTEM — change a server configuration parameter

Synopsis

```
ALTER SYSTEM SET configuration_parameter { TO | = } { value | 'value' | DEFAULT }

ALTER SYSTEM RESET configuration_parameter
ALTER SYSTEM RESET ALL
```

Description

ALTER SYSTEM is used for changing server configuration parameters across the entire database cluster. It can be more convenient than the traditional method of manually editing the postgresql.conf file. ALTER SYSTEM writes the given parameter setting to the postgresql.auto.conf file, which is read in addition to postgresql.conf. Setting a parameter to DEFAULT, or using the RESET variant, removes that configuration entry from the postgresql.auto.conf file. Use RESET ALL to remove all such configuration entries.

Values set with ALTER SYSTEM will be effective after the next server configuration reload (SIGHUP or pg_ctl reload), or after the next server restart in the case of parameters that can only be changed at server start.

Only superusers can use ALTER SYSTEM. Also, since this command acts directly on the file system and cannot be rolled back, it is not allowed inside a transaction block or function.

Parameters

configuration_parameter

Name of a settable configuration parameter. Available parameters are documented in Chapter 18.

value

New value of the parameter. Values can be specified as string constants, identifiers, numbers, or comma-separated lists of these, as appropriate for the particular parameter. DEFAULT can be written to specify removing the parameter and its value from postgresql.auto.conf.

Notes

This command can't be used to set data_directory, nor parameters that are not allowed in postgresql.conf (e.g., preset options).

See Section 18.1 for other ways to set the parameters.

Examples

Set the `wal_level`:

```
ALTER SYSTEM SET wal_level = hot_standby;
```

Undo that, restoring whatever setting was effective in `postgresql.conf`:

```
ALTER SYSTEM RESET wal_level;
```

Compatibility

The `ALTER SYSTEM` statement is a PostgreSQL extension.

See Also

SET, SHOW

ALTER TABLE

Name

ALTER TABLE — change the definition of a table

Synopsis

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    action [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema
ALTER TABLE ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]

where action is one of:

    ADD [ COLUMN ] column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation ] [ USING exp
    ALTER [ COLUMN ] column_name SET DEFAULT expression
    ALTER [ COLUMN ] column_name DROP DEFAULT
    ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED | MAIN }
    ADD table_constraint [ NOT VALID ]
    ADD table_constraint_using_index
    ALTER CONSTRAINT constraint_name [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | I
    VALIDATE CONSTRAINT constraint_name
    DROP CONSTRAINT [ IF EXISTS ]  constraint_name [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE REPLICA TRIGGER trigger_name
    ENABLE ALWAYS TRIGGER trigger_name
    DISABLE RULE rewrite_rule_name
    ENABLE RULE rewrite_rule_name
    ENABLE REPLICA RULE rewrite_rule_name
    ENABLE ALWAYS RULE rewrite_rule_name
    DISABLE ROW LEVEL SECURITY
    ENABLE ROW LEVEL SECURITY
    FORCE ROW LEVEL SECURITY
    NO FORCE ROW LEVEL SECURITY
    CLUSTER ON index_name
```

```
    SET WITHOUT CLUSTER
    SET WITH OIDS
    SET WITHOUT OIDS
    SET TABLESPACE new_tablespace
    SET { LOGGED | UNLOGGED }
    SET ( storage_parameter [= value] [, ... ] )
    RESET ( storage_parameter [, ... ] )
    INHERIT parent_table
    NO INHERIT parent_table
    OF type_name
    NOT OF
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
    REPLICA IDENTITY { DEFAULT | USING INDEX index_name | FULL | NOTHING }

and table_constraint_using_index is:

    [ CONSTRAINT constraint_name ]
    { UNIQUE | PRIMARY KEY } USING INDEX index_name
    [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

Description

`ALTER TABLE` changes the definition of an existing table. There are several subforms described below. Note that the lock level required may differ for each subform. An `ACCESS EXCLUSIVE` lock is acquired unless explicitly noted. When multiple subcommands are given, the lock acquired will be the strictest one required by any subcommand.

`ADD COLUMN`

This form adds a new column to the table, using the same syntax as CREATE TABLE.

`DROP COLUMN [ IF EXISTS ]`

This form drops a column from a table. Indexes and table constraints involving the column will be automatically dropped as well. You will need to say `CASCADE` if anything outside the table depends on the column, for example, foreign key references or views. If `IF EXISTS` is specified and the column does not exist, no error is thrown. In this case a notice is issued instead.

`SET DATA TYPE`

This form changes the type of a column of a table. Indexes and simple table constraints involving the column will be automatically converted to use the new column type by reparsing the originally supplied expression. The optional `COLLATE` clause specifies a collation for the new column; if omitted, the collation is the default for the new column type. The optional `USING` clause specifies how to compute the new column value from the old; if omitted, the default conversion is the same as an assignment cast from old data type to new. A `USING` clause must be provided if there is no implicit or assignment cast from old to new type.

`SET/DROP DEFAULT`

These forms set or remove the default value for a column. Default values only apply in subsequent `INSERT` or `UPDATE` commands; they do not cause rows already in the table to change.

`SET/DROP NOT NULL`

These forms change whether a column is marked to allow null values or to reject null values. You can only use `SET NOT NULL` when the column contains no null values.

`SET STATISTICS`

This form sets the per-column statistics-gathering target for subsequent ANALYZE operations. The target can be set in the range 0 to 10000; alternatively, set it to -1 to revert to using the system default statistics target (default_statistics_target). For more information on the use of statistics by the PostgreSQL query planner, refer to Section 14.2.

`SET STATISTICS` acquires a `SHARE UPDATE EXCLUSIVE` lock.

`SET ( *attribute_option* = *value* [, ... ] )`
`RESET ( *attribute_option* [, ... ] )`

This form sets or resets per-attribute options. Currently, the only defined per-attribute options are `n_distinct` and `n_distinct_inherited`, which override the number-of-distinct-values estimates made by subsequent ANALYZE operations. `n_distinct` affects the statistics for the table itself, while `n_distinct_inherited` affects the statistics gathered for the table plus its inheritance children. When set to a positive value, `ANALYZE` will assume that the column contains exactly the specified number of distinct nonnull values. When set to a negative value, which must be greater than or equal to -1, `ANALYZE` will assume that the number of distinct nonnull values in the column is linear in the size of the table; the exact count is to be computed by multiplying the estimated table size by the absolute value of the given number. For example, a value of -1 implies that all values in the column are distinct, while a value of -0.5 implies that each value appears twice on the average. This can be useful when the size of the table changes over time, since the multiplication by the number of rows in the table is not performed until query planning time. Specify a value of 0 to revert to estimating the number of distinct values normally. For more information on the use of statistics by the PostgreSQL query planner, refer to Section 14.2.

Changing per-attribute options acquires a `SHARE UPDATE EXCLUSIVE` lock.

`SET STORAGE`

This form sets the storage mode for a column. This controls whether this column is held inline or in a secondary TOAST table, and whether the data should be compressed or not. `PLAIN` must be used for fixed-length values such as `integer` and is inline, uncompressed. `MAIN` is for inline, compressible data. `EXTERNAL` is for external, uncompressed data, and `EXTENDED` is for external, compressed data. `EXTENDED` is the default for most data types that support non-`PLAIN` storage. Use of `EXTERNAL` will make substring operations on very large `text` and `bytea` values run faster, at the penalty of increased storage space. Note that `SET STORAGE` doesn't itself change anything in the table, it just sets the strategy to be pursued during future table updates. See Section 63.2 for more information.

`ADD *table_constraint* [ NOT VALID ]`

This form adds a new constraint to a table using the same constraint syntax as CREATE TABLE, plus the option `NOT VALID`, which is currently only allowed for foreign key and CHECK constraints.

Normally, this form will cause a scan of the table to verify that all existing rows in the table satisfy the new constraint. But if the `NOT VALID` option is used, this potentially-lengthy scan is skipped. The constraint will still be enforced against subsequent inserts or updates (that is, they'll fail unless there is a matching row in the referenced table, in the case of foreign keys, or they'll fail unless the new row matches the specified check condition). But the database will not assume that the constraint

holds for all rows in the table, until it is validated by using the `VALIDATE CONSTRAINT` option. See *Notes* below for more information about using the `NOT VALID` option.

Although most forms of `ADD` *`table_constraint`* require an `ACCESS EXCLUSIVE` lock, `ADD FOREIGN KEY` requires only a `SHARE ROW EXCLUSIVE` lock. Note that `ADD FOREIGN KEY` also acquires a `SHARE ROW EXCLUSIVE` lock on the referenced table, in addition to the lock on the table on which the constraint is declared.

`ADD` *`table_constraint_using_index`*

This form adds a new `PRIMARY KEY` or `UNIQUE` constraint to a table based on an existing unique index. All the columns of the index will be included in the constraint.

The index cannot have expression columns nor be a partial index. Also, it must be a b-tree index with default sort ordering. These restrictions ensure that the index is equivalent to one that would be built by a regular `ADD PRIMARY KEY` or `ADD UNIQUE` command.

If `PRIMARY KEY` is specified, and the index's columns are not already marked `NOT NULL`, then this command will attempt to do `ALTER COLUMN SET NOT NULL` against each such column. That requires a full table scan to verify the column(s) contain no nulls. In all other cases, this is a fast operation.

If a constraint name is provided then the index will be renamed to match the constraint name. Otherwise the constraint will be named the same as the index.

After this command is executed, the index is "owned" by the constraint, in the same way as if the index had been built by a regular `ADD PRIMARY KEY` or `ADD UNIQUE` command. In particular, dropping the constraint will make the index disappear too.

> **Note:** Adding a constraint using an existing index can be helpful in situations where a new constraint needs to be added without blocking table updates for a long time. To do that, create the index using `CREATE INDEX CONCURRENTLY`, and then install it as an official constraint using this syntax. See the example below.

`ALTER CONSTRAINT`

This form alters the attributes of a constraint that was previously created. Currently only foreign key constraints may be altered.

`VALIDATE CONSTRAINT`

This form validates a foreign key or check constraint that was previously created as `NOT VALID`, by scanning the table to ensure there are no rows for which the constraint is not satisfied. Nothing happens if the constraint is already marked valid. (See *Notes* below for an explanation of the usefulness of this command.)

`DROP CONSTRAINT [ IF EXISTS ]`

This form drops the specified constraint on a table. If `IF EXISTS` is specified and the constraint does not exist, no error is thrown. In this case a notice is issued instead.

`DISABLE/ENABLE [ REPLICA | ALWAYS ] TRIGGER`

These forms configure the firing of trigger(s) belonging to the table. A disabled trigger is still known to the system, but is not executed when its triggering event occurs. For a deferred trigger, the enable

status is checked when the event occurs, not when the trigger function is actually executed. One can disable or enable a single trigger specified by name, or all triggers on the table, or only user triggers (this option excludes internally generated constraint triggers such as those that are used to implement foreign key constraints or deferrable uniqueness and exclusion constraints). Disabling or enabling internally generated constraint triggers requires superuser privileges; it should be done with caution since of course the integrity of the constraint cannot be guaranteed if the triggers are not executed. The trigger firing mechanism is also affected by the configuration variable session_replication_role. Simply enabled triggers will fire when the replication role is "origin" (the default) or "local". Triggers configured as `ENABLE REPLICA` will only fire if the session is in "replica" mode, and triggers configured as `ENABLE ALWAYS` will fire regardless of the current replication mode.

This command acquires a `SHARE ROW EXCLUSIVE` lock.

`DISABLE/ENABLE [ REPLICA | ALWAYS ] RULE`

These forms configure the firing of rewrite rules belonging to the table. A disabled rule is still known to the system, but is not applied during query rewriting. The semantics are as for disabled/enabled triggers. This configuration is ignored for `ON SELECT` rules, which are always applied in order to keep views working even if the current session is in a non-default replication role.

`DISABLE/ENABLE ROW LEVEL SECURITY`

These forms control the application of row security policies belonging to the table. If enabled and no policies exist for the table, then a default-deny policy is applied. Note that policies can exist for a table even if row level security is disabled - in this case, the policies will NOT be applied and the policies will be ignored. See also CREATE POLICY.

`NO FORCE/FORCE ROW LEVEL SECURITY`

These forms control the application of row security policies belonging to the table when the user is the table owner. If enabled, row level security policies will be applied when the user is the table owner. If disabled (the default) then row level security will not be applied when the user is the table owner. See also CREATE POLICY.

`CLUSTER ON`

This form selects the default index for future CLUSTER operations. It does not actually re-cluster the table.

Changing cluster options acquires a `SHARE UPDATE EXCLUSIVE` lock.

`SET WITHOUT CLUSTER`

This form removes the most recently used CLUSTER index specification from the table. This affects future cluster operations that don't specify an index.

Changing cluster options acquires a `SHARE UPDATE EXCLUSIVE` lock.

`SET WITH OIDS`

This form adds an `oid` system column to the table (see Section 5.4). It does nothing if the table already has OIDs.

Note that this is not equivalent to `ADD COLUMN oid oid`; that would add a normal column that happened to be named `oid`, not a system column.

`SET WITHOUT OIDS`

This form removes the `oid` system column from the table. This is exactly equivalent to `DROP COLUMN oid RESTRICT`, except that it will not complain if there is already no `oid` column.

`SET TABLESPACE`

This form changes the table's tablespace to the specified tablespace and moves the data file(s) associated with the table to the new tablespace. Indexes on the table, if any, are not moved; but they can be moved separately with additional `SET TABLESPACE` commands. All tables in the current database in a tablespace can be moved by using the `ALL IN TABLESPACE` form, which will lock all tables to be moved first and then move each one. This form also supports `OWNED BY`, which will only move tables owned by the roles specified. If the `NOWAIT` option is specified then the command will fail if it is unable to acquire all of the locks required immediately. Note that system catalogs are not moved by this command, use `ALTER DATABASE` or explicit `ALTER TABLE` invocations instead if desired. The `information_schema` relations are not considered part of the system catalogs and will be moved. See also CREATE TABLESPACE.

`SET { LOGGED | UNLOGGED }`

This form changes the table from unlogged to logged or vice-versa (see *`UNLOGGED`*). It cannot be applied to a temporary table.

`SET ( storage_parameter [= value] [, ... ] )`

This form changes one or more storage parameters for the table. See *Storage Parameters* for details on the available parameters. Note that the table contents will not be modified immediately by this command; depending on the parameter you might need to rewrite the table to get the desired effects. That can be done with VACUUM FULL, CLUSTER or one of the forms of `ALTER TABLE` that forces a table rewrite.

> **Note:** While `CREATE TABLE` allows `OIDS` to be specified in the `WITH (storage_parameter)` syntax, `ALTER TABLE` does not treat `OIDS` as a storage parameter. Instead use the `SET WITH OIDS` and `SET WITHOUT OIDS` forms to change OID status.

`RESET ( storage_parameter [, ... ] )`

This form resets one or more storage parameters to their defaults. As with `SET`, a table rewrite might be needed to update the table entirely.

`INHERIT parent_table`

This form adds the target table as a new child of the specified parent table. Subsequently, queries against the parent will include records of the target table. To be added as a child, the target table must already contain all the same columns as the parent (it could have additional columns, too). The columns must have matching data types, and if they have `NOT NULL` constraints in the parent then they must also have `NOT NULL` constraints in the child.

There must also be matching child-table constraints for all `CHECK` constraints of the parent, except those marked non-inheritable (that is, created with `ALTER TABLE ... ADD CONSTRAINT ... NO INHERIT`) in the parent, which are ignored; all child-table constraints matched must not be marked non-inheritable. Currently `UNIQUE`, `PRIMARY KEY`, and `FOREIGN KEY` constraints are not considered, but this might change in the future.

`NO INHERIT` *parent_table*

This form removes the target table from the list of children of the specified parent table. Queries against the parent table will no longer include records drawn from the target table.

`OF` *type_name*

This form links the table to a composite type as though `CREATE TABLE OF` had formed it. The table's list of column names and types must precisely match that of the composite type; the presence of an `oid` system column is permitted to differ. The table must not inherit from any other table. These restrictions ensure that `CREATE TABLE OF` would permit an equivalent table definition.

`NOT OF`

This form dissociates a typed table from its type.

`OWNER`

This form changes the owner of the table, sequence, view, materialized view, or foreign table to the specified user.

`REPLICA IDENTITY`

This form changes the information which is written to the write-ahead log to identify rows which are updated or deleted. This option has no effect except when logical replication is in use. `DEFAULT` (the default for non-system tables) records the old values of the columns of the primary key, if any. `USING INDEX` records the old values of the columns covered by the named index, which must be unique, not partial, not deferrable, and include only columns marked `NOT NULL`. `FULL` records the old values of all columns in the row. `NOTHING` records no information about the old row. (This is the default for system tables.) In all cases, no old values are logged unless at least one of the columns that would be logged differs between the old and new versions of the row.

`RENAME`

The `RENAME` forms change the name of a table (or an index, sequence, view, materialized view, or foreign table), the name of an individual column in a table, or the name of a constraint of the table. There is no effect on the stored data.

`SET SCHEMA`

This form moves the table into another schema. Associated indexes, constraints, and sequences owned by table columns are moved as well.

All the forms of ALTER TABLE that act on a single table, except `RENAME`, and `SET SCHEMA` can be combined into a list of multiple alterations to applied together. For example, it is possible to add several columns and/or alter the type of several columns in a single command. This is particularly useful with large tables, since only one pass over the table need be made.

You must own the table to use `ALTER TABLE`. To change the schema or tablespace of a table, you must also have `CREATE` privilege on the new schema or tablespace. To add the table as a new child of a parent table, you must own the parent table as well. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the table's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the table. However, a superuser can alter ownership of any table anyway.) To add a column or alter a column type or use the `OF` clause, you must also have `USAGE` privilege on the data type.

Parameters

`IF EXISTS`

Do not throw an error if the table does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing table to alter. If `ONLY` is specified before the table name, only that table is altered. If `ONLY` is not specified, the table and all its descendant tables (if any) are altered. Optionally, `*` can be specified after the table name to explicitly indicate that descendant tables are included.

`column_name`

Name of a new or existing column.

`new_column_name`

New name for an existing column.

`new_name`

New name for the table.

`data_type`

Data type of the new column, or new data type for an existing column.

`table_constraint`

New table constraint for the table.

`constraint_name`

Name of a new or existing constraint.

`CASCADE`

Automatically drop objects that depend on the dropped column or constraint (for example, views referencing the column).

`RESTRICT`

Refuse to drop the column or constraint if there are any dependent objects. This is the default behavior.

`trigger_name`

Name of a single trigger to disable or enable.

`ALL`

Disable or enable all triggers belonging to the table. (This requires superuser privilege if any of the triggers are internally generated constraint triggers such as those that are used to implement foreign key constraints or deferrable uniqueness and exclusion constraints.)

`USER`

Disable or enable all triggers belonging to the table except for internally generated constraint triggers such as those that are used to implement foreign key constraints or deferrable uniqueness and exclusion constraints.

index_name

The name of an existing index.

storage_parameter

The name of a table storage parameter.

value

The new value for a table storage parameter. This might be a number or a word depending on the parameter.

parent_table

A parent table to associate or de-associate with this table.

new_owner

The user name of the new owner of the table.

new_tablespace

The name of the tablespace to which the table will be moved.

new_schema

The name of the schema to which the table will be moved.

Notes

The key word `COLUMN` is noise and can be omitted.

When a column is added with `ADD COLUMN`, all existing rows in the table are initialized with the column's default value (NULL if no `DEFAULT` clause is specified). If there is no `DEFAULT` clause, this is merely a metadata change and does not require any immediate update of the table's data; the added NULL values are supplied on readout, instead.

Adding a column with a `DEFAULT` clause or changing the type of an existing column will require the entire table and its indexes to be rewritten. As an exception when changing the type of an existing column, if the `USING` clause does not change the column contents and the old type is either binary coercible to the new type or an unconstrained domain over the new type, a table rewrite is not needed; but any indexes on the affected columns must still be rebuilt. Adding or removing a system `oid` column also requires rewriting the entire table. Table and/or index rebuilds may take a significant amount of time for a large table; and will temporarily require as much as double the disk space.

Adding a `CHECK` or `NOT NULL` constraint requires scanning the table to verify that existing rows meet the constraint, but does not require a table rewrite.

The main reason for providing the option to specify multiple changes in a single `ALTER TABLE` is that multiple table scans or rewrites can thereby be combined into a single pass over the table.

Scanning a large table to verify a new foreign key or check constraint can take a long time, and other updates to the table are locked out until the `ALTER TABLE ADD CONSTRAINT` command is committed. The main purpose of the `NOT VALID` constraint option is to reduce the impact of adding a constraint on concurrent updates. With `NOT VALID`, the `ADD CONSTRAINT` command does not scan the table and can be committed immediately. After that, a `VALIDATE CONSTRAINT` command can be issued to verify that existing rows satisfy the constraint. The validation step does not need to lock out concurrent updates, since

it knows that other transactions will be enforcing the constraint for rows that they insert or update; only pre-existing rows need to be checked. Hence, validation acquires only a `SHARE UPDATE EXCLUSIVE` lock on the table being altered. (If the constraint is a foreign key then a `ROW SHARE` lock is also required on the table referenced by the constraint.) In addition to improving concurrency, it can be useful to use `NOT VALID` and `VALIDATE CONSTRAINT` in cases where the table is known to contain pre-existing violations. Once the constraint is in place, no new violations can be inserted, and the existing problems can be corrected at leisure until `VALIDATE CONSTRAINT` finally succeeds.

The `DROP COLUMN` form does not physically remove the column, but simply makes it invisible to SQL operations. Subsequent insert and update operations in the table will store a null value for the column. Thus, dropping a column is quick but it will not immediately reduce the on-disk size of your table, as the space occupied by the dropped column is not reclaimed. The space will be reclaimed over time as existing rows are updated. (These statements do not apply when dropping the system `oid` column; that is done with an immediate rewrite.)

To force immediate reclamation of space occupied by a dropped column, you can execute one of the forms of `ALTER TABLE` that performs a rewrite of the whole table. This results in reconstructing each row with the dropped column replaced by a null value.

The rewriting forms of `ALTER TABLE` are not MVCC-safe. After a table rewrite, the table will appear empty to concurrent transactions, if they are using a snapshot taken before the rewrite occurred. See Section 13.5 for more details.

The `USING` option of `SET DATA TYPE` can actually specify any expression involving the old values of the row; that is, it can refer to other columns as well as the one being converted. This allows very general conversions to be done with the `SET DATA TYPE` syntax. Because of this flexibility, the `USING` expression is not applied to the column's default value (if any); the result might not be a constant expression as required for a default. This means that when there is no implicit or assignment cast from old to new type, `SET DATA TYPE` might fail to convert the default even though a `USING` clause is supplied. In such cases, drop the default with `DROP DEFAULT`, perform the `ALTER TYPE`, and then use `SET DEFAULT` to add a suitable new default. Similar considerations apply to indexes and constraints involving the column.

If a table has any descendant tables, it is not permitted to add, rename, or change the type of a column, or rename an inherited constraint in the parent table without doing the same to the descendants. That is, `ALTER TABLE ONLY` will be rejected. This ensures that the descendants always have columns matching the parent.

A recursive `DROP COLUMN` operation will remove a descendant table's column only if the descendant does not inherit that column from any other parents and never had an independent definition of the column. A nonrecursive `DROP COLUMN` (i.e., `ALTER TABLE ONLY ... DROP COLUMN`) never removes any descendant columns, but instead marks them as independently defined rather than inherited.

The `TRIGGER`, `CLUSTER`, `OWNER`, and `TABLESPACE` actions never recurse to descendant tables; that is, they always act as though `ONLY` were specified. Adding a constraint recurses only for `CHECK` constraints that are not marked `NO INHERIT`.

Changing any part of a system catalog table is not permitted.

Refer to CREATE TABLE for a further description of valid parameters. Chapter 5 has further information on inheritance.

Examples

To add a column of type `varchar` to a table:

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

To drop a column from a table:

```
ALTER TABLE distributors DROP COLUMN address RESTRICT;
```

To change the types of two existing columns in one operation:

```
ALTER TABLE distributors
    ALTER COLUMN address TYPE varchar(80),
    ALTER COLUMN name TYPE varchar(100);
```

To change an integer column containing Unix timestamps to `timestamp with time zone` via a `USING` clause:

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp SET DATA TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second';
```

The same, when the column has a default expression that won't automatically cast to the new data type:

```
ALTER TABLE foo
    ALTER COLUMN foo_timestamp DROP DEFAULT,
    ALTER COLUMN foo_timestamp TYPE timestamp with time zone
    USING
        timestamp with time zone 'epoch' + foo_timestamp * interval '1 second',
    ALTER COLUMN foo_timestamp SET DEFAULT now();
```

To rename an existing column:

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

To rename an existing table:

```
ALTER TABLE distributors RENAME TO suppliers;
```

To rename an existing constraint:

```
ALTER TABLE distributors RENAME CONSTRAINT zipchk TO zip_check;
```

To add a not-null constraint to a column:

```
ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;
```

To remove a not-null constraint from a column:

```
ALTER TABLE distributors ALTER COLUMN street DROP NOT NULL;
```

To add a check constraint to a table and all its children:

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5);
```

To add a check constraint only to a table and not to its children:

```
ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5) NO INHERIT;
```

(The check constraint will not be inherited by future children, either.)

To remove a check constraint from a table and all its children:

```
ALTER TABLE distributors DROP CONSTRAINT zipchk;
```

To remove a check constraint from one table only:

```
ALTER TABLE ONLY distributors DROP CONSTRAINT zipchk;
```

(The check constraint remains in place for any child tables.)

To add a foreign key constraint to a table:

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES addresses (
```

To add a foreign key constraint to a table with the least impact on other work:

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES addresses (
ALTER TABLE distributors VALIDATE CONSTRAINT distfk;
```

To add a (multicolumn) unique constraint to a table:

```
ALTER TABLE distributors ADD CONSTRAINT dist_id_zipcode_key UNIQUE (dist_id, zipcode);
```

To add an automatically named primary key constraint to a table, noting that a table can only ever have one primary key:

```
ALTER TABLE distributors ADD PRIMARY KEY (dist_id);
```

To move a table to a different tablespace:

```
ALTER TABLE distributors SET TABLESPACE fasttablespace;
```

To move a table to a different schema:

```
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
```

To recreate a primary key constraint, without blocking updates while the index is rebuilt:

```
CREATE UNIQUE INDEX CONCURRENTLY dist_id_temp_idx ON distributors (dist_id);
ALTER TABLE distributors DROP CONSTRAINT distributors_pkey,
    ADD CONSTRAINT distributors_pkey PRIMARY KEY USING INDEX dist_id_temp_idx;
```

Compatibility

The forms `ADD` (without `USING INDEX`), `DROP`, `SET DEFAULT`, and `SET DATA TYPE` (without `USING`) conform with the SQL standard. The other forms are PostgreSQL extensions of the SQL standard. Also, the ability to specify more than one manipulation in a single `ALTER TABLE` command is an extension.

`ALTER TABLE DROP COLUMN` can be used to drop the only column of a table, leaving a zero-column table. This is an extension of SQL, which disallows zero-column tables.

See Also

CREATE TABLE

ALTER TABLESPACE

Name

ALTER TABLESPACE — change the definition of a tablespace

Synopsis

```
ALTER TABLESPACE name RENAME TO new_name
ALTER TABLESPACE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TABLESPACE name SET ( tablespace_option = value [, ... ] )
ALTER TABLESPACE name RESET ( tablespace_option [, ... ] )
```

Description

ALTER TABLESPACE can be used to change the definition of a tablespace.

You must own the tablespace to change the definition of a tablespace. To alter the owner, you must also be a direct or indirect member of the new owning role. (Note that superusers have these privileges automatically.)

Parameters

name

The name of an existing tablespace.

new_name

The new name of the tablespace. The new name cannot begin with pg_, as such names are reserved for system tablespaces.

new_owner

The new owner of the tablespace.

tablespace_option

A tablespace parameter to be set or reset. Currently, the only available parameters are seq_page_cost and random_page_cost. Setting either value for a particular tablespace will override the planner's usual estimate of the cost of reading pages from tables in that tablespace, as established by the configuration parameters of the same name (see seq_page_cost, random_page_cost). This may be useful if one tablespace is located on a disk which is faster or slower than the remainder of the I/O subsystem.

Examples

Rename tablespace `index_space` to `fast_raid`:

```
ALTER TABLESPACE index_space RENAME TO fast_raid;
```

Change the owner of tablespace `index_space`:

```
ALTER TABLESPACE index_space OWNER TO mary;
```

Compatibility

There is no `ALTER TABLESPACE` statement in the SQL standard.

See Also

CREATE TABLESPACE, DROP TABLESPACE

ALTER TEXT SEARCH CONFIGURATION

Name

ALTER TEXT SEARCH CONFIGURATION — change the definition of a text search configuration

Synopsis

```
ALTER TEXT SEARCH CONFIGURATION name
    ADD MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING REPLACE old_dictionary WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    ALTER MAPPING FOR token_type [, ... ] REPLACE old_dictionary WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
    DROP MAPPING [ IF EXISTS ] FOR token_type [, ... ]
ALTER TEXT SEARCH CONFIGURATION name RENAME TO new_name
ALTER TEXT SEARCH CONFIGURATION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TEXT SEARCH CONFIGURATION name SET SCHEMA new_schema
```

Description

ALTER TEXT SEARCH CONFIGURATION changes the definition of a text search configuration. You can modify its mappings from token types to dictionaries, or change the configuration's name or owner.

You must be the owner of the configuration to use ALTER TEXT SEARCH CONFIGURATION.

Parameters

name

The name (optionally schema-qualified) of an existing text search configuration.

token_type

The name of a token type that is emitted by the configuration's parser.

dictionary_name

The name of a text search dictionary to be consulted for the specified token type(s). If multiple dictionaries are listed, they are consulted in the specified order.

old_dictionary

The name of a text search dictionary to be replaced in the mapping.

new_dictionary

The name of a text search dictionary to be substituted for *old_dictionary*.

new_name

The new name of the text search configuration.

new_owner

The new owner of the text search configuration.

new_schema

The new schema for the text search configuration.

The ADD MAPPING FOR form installs a list of dictionaries to be consulted for the specified token type(s); it is an error if there is already a mapping for any of the token types. The ALTER MAPPING FOR form does the same, but first removing any existing mapping for those token types. The ALTER MAPPING REPLACE forms substitute *new_dictionary* for *old_dictionary* anywhere the latter appears. This is done for only the specified token types when FOR appears, or for all mappings of the configuration when it doesn't. The DROP MAPPING form removes all dictionaries for the specified token type(s), causing tokens of those types to be ignored by the text search configuration. It is an error if there is no mapping for the token types, unless IF EXISTS appears.

Examples

The following example replaces the english dictionary with the swedish dictionary anywhere that english is used within my_config.

```
ALTER TEXT SEARCH CONFIGURATION my_config
  ALTER MAPPING REPLACE english WITH swedish;
```

Compatibility

There is no ALTER TEXT SEARCH CONFIGURATION statement in the SQL standard.

See Also

CREATE TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

ALTER TEXT SEARCH DICTIONARY

Name

ALTER TEXT SEARCH DICTIONARY — change the definition of a text search dictionary

Synopsis

```
ALTER TEXT SEARCH DICTIONARY name (
    option [ = value ] [, ... ]
)
ALTER TEXT SEARCH DICTIONARY name RENAME TO new_name
ALTER TEXT SEARCH DICTIONARY name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TEXT SEARCH DICTIONARY name SET SCHEMA new_schema
```

Description

ALTER TEXT SEARCH DICTIONARY changes the definition of a text search dictionary. You can change the dictionary's template-specific options, or change the dictionary's name or owner.

You must be the owner of the dictionary to use ALTER TEXT SEARCH DICTIONARY.

Parameters

name

The name (optionally schema-qualified) of an existing text search dictionary.

option

The name of a template-specific option to be set for this dictionary.

value

The new value to use for a template-specific option. If the equal sign and value are omitted, then any previous setting for the option is removed from the dictionary, allowing the default to be used.

new_name

The new name of the text search dictionary.

new_owner

The new owner of the text search dictionary.

new_schema

The new schema for the text search dictionary.

Template-specific options can appear in any order.

Examples

The following example command changes the stopword list for a Snowball-based dictionary. Other parameters remain unchanged.

```
ALTER TEXT SEARCH DICTIONARY my_dict ( StopWords = newrussian );
```

The following example command changes the language option to `dutch`, and removes the stopword option entirely.

```
ALTER TEXT SEARCH DICTIONARY my_dict ( language = dutch, StopWords );
```

The following example command "updates" the dictionary's definition without actually changing anything.

```
ALTER TEXT SEARCH DICTIONARY my_dict ( dummy );
```

(The reason this works is that the option removal code doesn't complain if there is no such option.) This trick is useful when changing configuration files for the dictionary: the `ALTER` will force existing database sessions to re-read the configuration files, which otherwise they would never do if they had read them earlier.

Compatibility

There is no `ALTER TEXT SEARCH DICTIONARY` statement in the SQL standard.

See Also

CREATE TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY

ALTER TEXT SEARCH PARSER

Name

ALTER TEXT SEARCH PARSER — change the definition of a text search parser

Synopsis

```
ALTER TEXT SEARCH PARSER name RENAME TO new_name
ALTER TEXT SEARCH PARSER name SET SCHEMA new_schema
```

Description

ALTER TEXT SEARCH PARSER changes the definition of a text search parser. Currently, the only supported functionality is to change the parser's name.

You must be a superuser to use ALTER TEXT SEARCH PARSER.

Parameters

name

The name (optionally schema-qualified) of an existing text search parser.

new_name

The new name of the text search parser.

new_schema

The new schema for the text search parser.

Compatibility

There is no ALTER TEXT SEARCH PARSER statement in the SQL standard.

See Also

CREATE TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER

ALTER TEXT SEARCH TEMPLATE

Name

ALTER TEXT SEARCH TEMPLATE — change the definition of a text search template

Synopsis

```
ALTER TEXT SEARCH TEMPLATE name RENAME TO new_name
ALTER TEXT SEARCH TEMPLATE name SET SCHEMA new_schema
```

Description

ALTER TEXT SEARCH TEMPLATE changes the definition of a text search template. Currently, the only supported functionality is to change the template's name.

You must be a superuser to use ALTER TEXT SEARCH TEMPLATE.

Parameters

name

The name (optionally schema-qualified) of an existing text search template.

new_name

The new name of the text search template.

new_schema

The new schema for the text search template.

Compatibility

There is no ALTER TEXT SEARCH TEMPLATE statement in the SQL standard.

See Also

CREATE TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

ALTER TRIGGER

Name

ALTER TRIGGER — change the definition of a trigger

Synopsis

```
ALTER TRIGGER name ON table_name RENAME TO new_name
```

Description

ALTER TRIGGER changes properties of an existing trigger. The RENAME clause changes the name of the given trigger without otherwise changing the trigger definition.

You must own the table on which the trigger acts to be allowed to change its properties.

Parameters

name

The name of an existing trigger to alter.

table_name

The name of the table on which this trigger acts.

new_name

The new name for the trigger.

Notes

The ability to temporarily enable or disable a trigger is provided by ALTER TABLE, not by ALTER TRIGGER, because ALTER TRIGGER has no convenient way to express the option of enabling or disabling all of a table's triggers at once.

Examples

To rename an existing trigger:

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

Compatibility

`ALTER TRIGGER` is a PostgreSQL extension of the SQL standard.

See Also

ALTER TABLE

ALTER TYPE

Name

ALTER TYPE — change the definition of a type

Synopsis

```
ALTER TYPE name action [, ... ]
ALTER TYPE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TYPE name RENAME ATTRIBUTE attribute_name TO new_attribute_name [ CASCADE | RESTRICT ]
ALTER TYPE name RENAME TO new_name
ALTER TYPE name SET SCHEMA new_schema
ALTER TYPE name ADD VALUE [ IF NOT EXISTS ] new_enum_value [ { BEFORE | AFTER } existing_enum_

where action is one of:

    ADD ATTRIBUTE attribute_name data_type [ COLLATE collation ] [ CASCADE | RESTRICT ]
    DROP ATTRIBUTE [ IF EXISTS ] attribute_name [ CASCADE | RESTRICT ]
    ALTER ATTRIBUTE attribute_name [ SET DATA ] TYPE data_type [ COLLATE collation ] [ CASCADE
```

Description

ALTER TYPE changes the definition of an existing type. There are several subforms:

ADD ATTRIBUTE

This form adds a new attribute to a composite type, using the same syntax as CREATE TYPE.

DROP ATTRIBUTE [IF EXISTS]

This form drops an attribute from a composite type. If IF EXISTS is specified and the attribute does not exist, no error is thrown. In this case a notice is issued instead.

SET DATA TYPE

This form changes the type of an attribute of a composite type.

OWNER

This form changes the owner of the type.

RENAME

This form changes the name of the type or the name of an individual attribute of a composite type.

SET SCHEMA

This form moves the type into another schema.

`ADD VALUE [ IF NOT EXISTS ] [ BEFORE | AFTER ]`

This form adds a new value to an enum type. The new value's place in the enum's ordering can be specified as being `BEFORE` or `AFTER` one of the existing values. Otherwise, the new item is added at the end of the list of values.

If `IF NOT EXISTS` is specified, it is not an error if the type already contains the new value: a notice is issued but no other action is taken. Otherwise, an error will occur if the new value is already present.

`CASCADE`

Automatically propagate the operation to typed tables of the type being altered, and their descendants.

`RESTRICT`

Refuse the operation if the type being altered is the type of a typed table. This is the default.

The `ADD ATTRIBUTE`, `DROP ATTRIBUTE`, and `ALTER ATTRIBUTE` actions can be combined into a list of multiple alterations to apply in parallel. For example, it is possible to add several attributes and/or alter the type of several attributes in a single command.

You must own the type to use `ALTER TYPE`. To change the schema of a type, you must also have `CREATE` privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have `CREATE` privilege on the type's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the type. However, a superuser can alter ownership of any type anyway.) To add an attribute or alter an attribute type, you must also have `USAGE` privilege on the data type.

Parameters

`name`

The name (possibly schema-qualified) of an existing type to alter.

`new_name`

The new name for the type.

`new_owner`

The user name of the new owner of the type.

`new_schema`

The new schema for the type.

`attribute_name`

The name of the attribute to add, alter, or drop.

`new_attribute_name`

The new name of the attribute to be renamed.

data_type

The data type of the attribute to add, or the new type of the attribute to alter.

new_enum_value

The new value to be added to an enum type's list of values. Like all enum literals, it needs to be quoted.

existing_enum_value

The existing enum value that the new value should be added immediately before or after in the enum type's sort ordering. Like all enum literals, it needs to be quoted.

Notes

`ALTER TYPE ... ADD VALUE` (the form that adds a new value to an enum type) cannot be executed inside a transaction block.

Comparisons involving an added enum value will sometimes be slower than comparisons involving only original members of the enum type. This will usually only occur if `BEFORE` or `AFTER` is used to set the new value's sort position somewhere other than at the end of the list. However, sometimes it will happen even though the new value is added at the end (this occurs if the OID counter "wrapped around" since the original creation of the enum type). The slowdown is usually insignificant; but if it matters, optimal performance can be regained by dropping and recreating the enum type, or by dumping and reloading the database.

Examples

To rename a data type:

```
ALTER TYPE electronic_mail RENAME TO email;
```

To change the owner of the type `email` to `joe`:

```
ALTER TYPE email OWNER TO joe;
```

To change the schema of the type `email` to `customers`:

```
ALTER TYPE email SET SCHEMA customers;
```

To add a new attribute to a type:

```
ALTER TYPE compfoo ADD ATTRIBUTE f3 int;
```

To add a new value to an enum type in a particular sort position:

```
ALTER TYPE colors ADD VALUE 'orange' AFTER 'red';
```

Compatibility

The variants to add and drop attributes are part of the SQL standard; the other variants are PostgreSQL extensions.

See Also

CREATE TYPE, DROP TYPE

ALTER USER

Name

ALTER USER — change a database role

Synopsis

```
ALTER USER role_specification [ WITH ] option [ ... ]

where option can be:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'

ALTER USER name RENAME TO new_name

ALTER USER { role_specification | ALL } [ IN DATABASE database_name ] SET configuration_parameter
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ] SET configuration_parameter
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ] RESET configuration_parame
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ] RESET ALL

where role_specification can be:

    role_name
  | CURRENT_USER
  | SESSION_USER
```

Description

ALTER USER is now an alias for ALTER ROLE.

Compatibility

The ALTER USER statement is a PostgreSQL extension. The SQL standard leaves the definition of users to the implementation.

See Also

ALTER ROLE

ALTER USER MAPPING

Name

ALTER USER MAPPING — change the definition of a user mapping

Synopsis

```
ALTER USER MAPPING FOR { user_name | USER | CURRENT_USER | SESSION_USER | PUBLIC }
    SERVER server_name
    OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )
```

Description

ALTER USER MAPPING changes the definition of a user mapping.

The owner of a foreign server can alter user mappings for that server for any user. Also, a user can alter a user mapping for his own user name if USAGE privilege on the server has been granted to the user.

Parameters

user_name

User name of the mapping. CURRENT_USER and USER match the name of the current user. PUBLIC is used to match all present and future user names in the system.

server_name

Server name of the user mapping.

OPTIONS ([ADD | SET | DROP] *option* ['*value*'] [, ...])

Change options for the user mapping. The new options override any previously specified options. ADD, SET, and DROP specify the action to be performed. ADD is assumed if no operation is explicitly specified. Option names must be unique; options are also validated by the server's foreign-data wrapper.

Examples

Change the password for user mapping bob, server foo:

```
ALTER USER MAPPING FOR bob SERVER foo OPTIONS (SET password 'public');
```

Compatibility

`ALTER USER MAPPING` conforms to ISO/IEC 9075-9 (SQL/MED). There is a subtle syntax issue: The standard omits the `FOR` key word. Since both `CREATE USER MAPPING` and `DROP USER MAPPING` use `FOR` in analogous positions, and IBM DB2 (being the other major SQL/MED implementation) also requires it for `ALTER USER MAPPING`, PostgreSQL diverges from the standard here in the interest of consistency and interoperability.

See Also

CREATE USER MAPPING, DROP USER MAPPING

ALTER VIEW

Name

ALTER VIEW — change the definition of a view

Synopsis

```
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name SET DEFAULT expression
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name DROP DEFAULT
ALTER VIEW [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER VIEW [ IF EXISTS ] name RENAME TO new_name
ALTER VIEW [ IF EXISTS ] name SET SCHEMA new_schema
ALTER VIEW [ IF EXISTS ] name SET ( view_option_name [= view_option_value] [, ... ] )
ALTER VIEW [ IF EXISTS ] name RESET ( view_option_name [, ... ] )
```

Description

ALTER VIEW changes various auxiliary properties of a view. (If you want to modify the view's defining query, use CREATE OR REPLACE VIEW.)

You must own the view to use ALTER VIEW. To change a view's schema, you must also have CREATE privilege on the new schema. To alter the owner, you must also be a direct or indirect member of the new owning role, and that role must have CREATE privilege on the view's schema. (These restrictions enforce that altering the owner doesn't do anything you couldn't do by dropping and recreating the view. However, a superuser can alter ownership of any view anyway.)

Parameters

name

The name (optionally schema-qualified) of an existing view.

IF EXISTS

Do not throw an error if the view does not exist. A notice is issued in this case.

SET/DROP DEFAULT

These forms set or remove the default value for a column. A view column's default value is substituted into any INSERT or UPDATE command whose target is the view, before applying any rules or triggers for the view. The view's default will therefore take precedence over any default values from underlying relations.

new_owner

The user name of the new owner of the view.

new_name

The new name for the view.

new_schema

The new schema for the view.

```
SET ( view_option_name [= view_option_value] [, ... ] )
RESET ( view_option_name [, ... ] )
```

Sets or resets a view option. Currently supported options are:

`check_option` (`string`)

Changes the check option of the view. The value must be `local` or `cascaded`.

`security_barrier` (`boolean`)

Changes the security-barrier property of the view. The value must be Boolean value, such as `true` or `false`.

Notes

For historical reasons, `ALTER TABLE` can be used with views too; but the only variants of `ALTER TABLE` that are allowed with views are equivalent to the ones shown above.

Examples

To rename the view `foo` to `bar`:

```
ALTER VIEW foo RENAME TO bar;
```

To attach a default column value to an updatable view:

```
CREATE TABLE base_table (id int, ts timestamptz);
CREATE VIEW a_view AS SELECT * FROM base_table;
ALTER VIEW a_view ALTER COLUMN ts SET DEFAULT now();
INSERT INTO base_table(id) VALUES(1);  -- ts will receive a NULL
INSERT INTO a_view(id) VALUES(2);  -- ts will receive the current time
```

Compatibility

`ALTER VIEW` is a PostgreSQL extension of the SQL standard.

See Also

CREATE VIEW, DROP VIEW

ANALYZE

Name

ANALYZE — collect statistics about a database

Synopsis

```
ANALYZE [ VERBOSE ] [ table_name [ ( column_name [, ...] ) ] ]
```

Description

ANALYZE collects statistics about the contents of tables in the database, and stores the results in the pg_statistic system catalog. Subsequently, the query planner uses these statistics to help determine the most efficient execution plans for queries.

With no parameter, ANALYZE examines every table in the current database. With a parameter, ANALYZE examines only that table. It is further possible to give a list of column names, in which case only the statistics for those columns are collected.

Parameters

VERBOSE

Enables display of progress messages.

table_name

The name (possibly schema-qualified) of a specific table to analyze. If omitted, all regular tables (but not foreign tables) in the current database are analyzed.

column_name

The name of a specific column to analyze. Defaults to all columns.

Outputs

When VERBOSE is specified, ANALYZE emits progress messages to indicate which table is currently being processed. Various statistics about the tables are printed as well.

Notes

To analyze a table, one must ordinarily be the table's owner or a superuser. However, database owners are allowed to analyze all tables in their databases, except shared catalogs. (The restriction for shared catalogs

means that a true database-wide ANALYZE can only be performed by a superuser.) ANALYZE will skip over any tables that the calling user does not have permission to analyze.

Foreign tables are analyzed only when explicitly selected. Not all foreign data wrappers support ANALYZE. If the table's wrapper does not support ANALYZE, the command prints a warning and does nothing.

In the default PostgreSQL configuration, the autovacuum daemon (see Section 23.1.6) takes care of automatic analyzing of tables when they are first loaded with data, and as they change throughout regular operation. When autovacuum is disabled, it is a good idea to run ANALYZE periodically, or just after making major changes in the contents of a table. Accurate statistics will help the planner to choose the most appropriate query plan, and thereby improve the speed of query processing. A common strategy for read-mostly databases is to run VACUUM and ANALYZE once a day during a low-usage time of day. (This will not be sufficient if there is heavy update activity.)

ANALYZE requires only a read lock on the target table, so it can run in parallel with other activity on the table.

The statistics collected by ANALYZE usually include a list of some of the most common values in each column and a histogram showing the approximate data distribution in each column. One or both of these can be omitted if ANALYZE deems them uninteresting (for example, in a unique-key column, there are no common values) or if the column data type does not support the appropriate operators. There is more information about the statistics in Chapter 23.

For large tables, ANALYZE takes a random sample of the table contents, rather than examining every row. This allows even very large tables to be analyzed in a small amount of time. Note, however, that the statistics are only approximate, and will change slightly each time ANALYZE is run, even if the actual table contents did not change. This might result in small changes in the planner's estimated costs shown by EXPLAIN. In rare situations, this non-determinism will cause the planner's choices of query plans to change after ANALYZE is run. To avoid this, raise the amount of statistics collected by ANALYZE, as described below.

The extent of analysis can be controlled by adjusting the default_statistics_target configuration variable, or on a column-by-column basis by setting the per-column statistics target with ALTER TABLE ... ALTER COLUMN ... SET STATISTICS (see ALTER TABLE). The target value sets the maximum number of entries in the most-common-value list and the maximum number of bins in the histogram. The default target value is 100, but this can be adjusted up or down to trade off accuracy of planner estimates against the time taken for ANALYZE and the amount of space occupied in pg_statistic. In particular, setting the statistics target to zero disables collection of statistics for that column. It might be useful to do that for columns that are never used as part of the WHERE, GROUP BY, or ORDER BY clauses of queries, since the planner will have no use for statistics on such columns.

The largest statistics target among the columns being analyzed determines the number of table rows sampled to prepare the statistics. Increasing the target causes a proportional increase in the time and space needed to do ANALYZE.

One of the values estimated by ANALYZE is the number of distinct values that appear in each column. Because only a subset of the rows are examined, this estimate can sometimes be quite inaccurate, even with the largest possible statistics target. If this inaccuracy leads to bad query plans, a more accurate value can be determined manually and then installed with ALTER TABLE ... ALTER COLUMN ... SET (n_distinct = ...) (see ALTER TABLE).

If the table being analyzed has one or more children, ANALYZE will gather statistics twice: once on the rows of the parent table only, and a second time on the rows of the parent table with all of its children.

This second set of statistics is needed when planning queries that traverse the entire inheritance tree. The autovacuum daemon, however, will only consider inserts or updates on the parent table itself when deciding whether to trigger an automatic analyze for that table. If that table is rarely inserted into or updated, the inheritance statistics will not be up to date unless you run `ANALYZE` manually.

If any of the child tables are foreign tables whose foreign data wrappers do not support `ANALYZE`, those child tables are ignored while gathering inheritance statistics.

If the table being analyzed is completely empty, `ANALYZE` will not record new statistics for that table. Any existing statistics will be retained.

Compatibility

There is no `ANALYZE` statement in the SQL standard.

See Also

VACUUM, vacuumdb, Section 18.4.4, Section 23.1.6

BEGIN

Name

BEGIN — start a transaction block

Synopsis

```
BEGIN [ WORK | TRANSACTION ] [ transaction_mode [, ...] ]

where transaction_mode is one of:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

Description

BEGIN initiates a transaction block, that is, all statements after a BEGIN command will be executed in a single transaction until an explicit COMMIT or ROLLBACK is given. By default (without BEGIN), PostgreSQL executes transactions in "autocommit" mode, that is, each statement is executed in its own transaction and a commit is implicitly performed at the end of the statement (if execution was successful, otherwise a rollback is done).

Statements are executed more quickly in a transaction block, because transaction start/commit requires significant CPU and disk activity. Execution of multiple statements inside a transaction is also useful to ensure consistency when making several related changes: other sessions will be unable to see the intermediate states wherein not all the related updates have been done.

If the isolation level, read/write mode, or deferrable mode is specified, the new transaction has those characteristics, as if SET TRANSACTION was executed.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

Refer to SET TRANSACTION for information on the meaning of the other parameters to this statement.

Notes

START TRANSACTION has the same functionality as BEGIN.

Use COMMIT or ROLLBACK to terminate a transaction block.

Issuing `BEGIN` when already inside a transaction block will provoke a warning message. The state of the transaction is not affected. To nest transactions within a transaction block, use savepoints (see SAVEPOINT).

For reasons of backwards compatibility, the commas between successive *`transaction_modes`* can be omitted.

Examples

To begin a transaction block:

```
BEGIN;
```

Compatibility

`BEGIN` is a PostgreSQL language extension. It is equivalent to the SQL-standard command START TRANSACTION, whose reference page contains additional compatibility information.

The `DEFERRABLE` *`transaction_mode`* is a PostgreSQL language extension.

Incidentally, the `BEGIN` key word is used for a different purpose in embedded SQL. You are advised to be careful about the transaction semantics when porting database applications.

See Also

COMMIT, ROLLBACK, START TRANSACTION, SAVEPOINT

CHECKPOINT

Name

CHECKPOINT — force a transaction log checkpoint

Synopsis

```
CHECKPOINT
```

Description

A checkpoint is a point in the transaction log sequence at which all data files have been updated to reflect the information in the log. All data files will be flushed to disk. Refer to Section 29.4 for more details about what happens during a checkpoint.

The CHECKPOINT command forces an immediate checkpoint when the command is issued, without waiting for a regular checkpoint scheduled by the system (controlled by the settings in Section 18.5.2). CHECKPOINT is not intended for use during normal operation.

If executed during recovery, the CHECKPOINT command will force a restartpoint (see Section 29.4) rather than writing a new checkpoint.

Only superusers can call CHECKPOINT.

Compatibility

The CHECKPOINT command is a PostgreSQL language extension.

CLOSE

Name

CLOSE — close a cursor

Synopsis

```
CLOSE { name | ALL }
```

Description

CLOSE frees the resources associated with an open cursor. After the cursor is closed, no subsequent operations are allowed on it. A cursor should be closed when it is no longer needed.

Every non-holdable open cursor is implicitly closed when a transaction is terminated by COMMIT or ROLLBACK. A holdable cursor is implicitly closed if the transaction that created it aborts via ROLLBACK. If the creating transaction successfully commits, the holdable cursor remains open until an explicit CLOSE is executed, or the client disconnects.

Parameters

name

The name of an open cursor to close.

ALL

Close all open cursors.

Notes

PostgreSQL does not have an explicit OPEN cursor statement; a cursor is considered open when it is declared. Use the DECLARE statement to declare a cursor.

You can see all available cursors by querying the pg_cursors system view.

If a cursor is closed after a savepoint which is later rolled back, the CLOSE is not rolled back; that is, the cursor remains closed.

Examples

Close the cursor liahona:

```
CLOSE liahona;
```

Compatibility

`CLOSE` is fully conforming with the SQL standard. `CLOSE ALL` is a PostgreSQL extension.

See Also

DECLARE, FETCH, MOVE

CLUSTER

Name

CLUSTER — cluster a table according to an index

Synopsis

```
CLUSTER [VERBOSE] table_name [ USING index_name ]
CLUSTER [VERBOSE]
```

Description

CLUSTER instructs PostgreSQL to cluster the table specified by *table_name* based on the index specified by *index_name*. The index must already have been defined on *table_name*.

When a table is clustered, it is physically reordered based on the index information. Clustering is a one-time operation: when the table is subsequently updated, the changes are not clustered. That is, no attempt is made to store new or updated rows according to their index order. (If one wishes, one can periodically recluster by issuing the command again. Also, setting the table's fillfactor storage parameter to less than 100% can aid in preserving cluster ordering during updates, since updated rows are kept on the same page if enough space is available there.)

When a table is clustered, PostgreSQL remembers which index it was clustered by. The form CLUSTER *table_name* reclusters the table using the same index as before. You can also use the CLUSTER or SET WITHOUT CLUSTER forms of ALTER TABLE to set the index to be used for future cluster operations, or to clear any previous setting.

CLUSTER without any parameter reclusters all the previously-clustered tables in the current database that the calling user owns, or all such tables if called by a superuser. This form of CLUSTER cannot be executed inside a transaction block.

When a table is being clustered, an ACCESS EXCLUSIVE lock is acquired on it. This prevents any other database operations (both reads and writes) from operating on the table until the CLUSTER is finished.

Parameters

table_name

The name (possibly schema-qualified) of a table.

index_name

The name of an index.

VERBOSE

Prints a progress report as each table is clustered.

Notes

In cases where you are accessing single rows randomly within a table, the actual order of the data in the table is unimportant. However, if you tend to access some data more than others, and there is an index that groups them together, you will benefit from using `CLUSTER`. If you are requesting a range of indexed values from a table, or a single indexed value that has multiple rows that match, `CLUSTER` will help because once the index identifies the table page for the first row that matches, all other rows that match are probably already on the same table page, and so you save disk accesses and speed up the query.

`CLUSTER` can re-sort the table using either an index scan on the specified index, or (if the index is a b-tree) a sequential scan followed by sorting. It will attempt to choose the method that will be faster, based on planner cost parameters and available statistical information.

When an index scan is used, a temporary copy of the table is created that contains the table data in the index order. Temporary copies of each index on the table are created as well. Therefore, you need free space on disk at least equal to the sum of the table size and the index sizes.

When a sequential scan and sort is used, a temporary sort file is also created, so that the peak temporary space requirement is as much as double the table size, plus the index sizes. This method is often faster than the index scan method, but if the disk space requirement is intolerable, you can disable this choice by temporarily setting enable_sort to `off`.

It is advisable to set maintenance_work_mem to a reasonably large value (but not more than the amount of RAM you can dedicate to the `CLUSTER` operation) before clustering.

Because the planner records statistics about the ordering of tables, it is advisable to run ANALYZE on the newly clustered table. Otherwise, the planner might make poor choices of query plans.

Because `CLUSTER` remembers which indexes are clustered, one can cluster the tables one wants clustered manually the first time, then set up a periodic maintenance script that executes `CLUSTER` without any parameters, so that the desired tables are periodically reclustered.

Examples

Cluster the table `employees` on the basis of its index `employees_ind`:

```
CLUSTER employees USING employees_ind;
```

Cluster the `employees` table using the same index that was used before:

```
CLUSTER employees;
```

Cluster all tables in the database that have previously been clustered:

```
CLUSTER;
```

Compatibility

There is no `CLUSTER` statement in the SQL standard.

The syntax

```
CLUSTER index_name ON table_name
```

is also supported for compatibility with pre-8.3 PostgreSQL versions.

See Also

clusterdb

COMMENT

Name

COMMENT — define or change the comment of an object

Synopsis

```
COMMENT ON
{
  AGGREGATE aggregate_name ( aggregate_signature ) |
  CAST (source_type AS target_type) |
  COLLATION object_name |
  COLUMN relation_name.column_name |
  CONSTRAINT constraint_name ON table_name |
  CONSTRAINT constraint_name ON DOMAIN domain_name |
  CONVERSION object_name |
  DATABASE object_name |
  DOMAIN object_name |
  EXTENSION object_name |
  EVENT TRIGGER object_name |
  FOREIGN DATA WRAPPER object_name |
  FOREIGN TABLE object_name |
  FUNCTION function_name ( [ [ argmode ] [ argname ] argtype [, ...] ] ) |
  INDEX object_name |
  LARGE OBJECT large_object_oid |
  MATERIALIZED VIEW object_name |
  OPERATOR operator_name (left_type, right_type) |
  OPERATOR CLASS object_name USING index_method |
  OPERATOR FAMILY object_name USING index_method |
  POLICY policy_name ON table_name |
  [ PROCEDURAL ] LANGUAGE object_name |
  ROLE object_name |
  RULE rule_name ON table_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  SERVER object_name |
  TABLE object_name |
  TABLESPACE object_name |
  TEXT SEARCH CONFIGURATION object_name |
  TEXT SEARCH DICTIONARY object_name |
  TEXT SEARCH PARSER object_name |
  TEXT SEARCH TEMPLATE object_name |
  TRANSFORM FOR type_name LANGUAGE lang_name |
  TRIGGER trigger_name ON table_name |
  TYPE object_name |
  VIEW object_name
} IS 'text'
```

where *aggregate_signature* is:

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ] argtype [ , ...
```

Description

COMMENT stores a comment about a database object.

Only one comment string is stored for each object, so to modify a comment, issue a new COMMENT command for the same object. To remove a comment, write NULL in place of the text string. Comments are automatically dropped when their object is dropped.

For most kinds of object, only the object's owner can set the comment. Roles don't have owners, so the rule for COMMENT ON ROLE is that you must be superuser to comment on a superuser role, or have the CREATEROLE privilege to comment on non-superuser roles. Of course, a superuser can comment on anything.

Comments can be viewed using psql's \d family of commands. Other user interfaces to retrieve comments can be built atop the same built-in functions that psql uses, namely obj_description, col_description, and shobj_description (see Table 9-62).

Parameters

object_name
relation_name.column_name
aggregate_name
constraint_name
function_name
operator_name
policy_name
rule_name
trigger_name

The name of the object to be commented. Names of tables, aggregates, collations, conversions, domains, foreign tables, functions, indexes, operators, operator classes, operator families, sequences, text search objects, types, and views can be schema-qualified. When commenting on a column, *relation_name* must refer to a table, view, composite type, or foreign table.

table_name
domain_name

When creating a comment on a constraint, a trigger, a rule or a policy these parameters specify the name of the table or domain on which that object is defined.

source_type

The name of the source data type of the cast.

target_type

The name of the target data type of the cast.

argmode

The mode of a function or aggregate argument: IN, OUT, INOUT, or VARIADIC. If omitted, the default is IN. Note that COMMENT does not actually pay any attention to OUT arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list the IN, INOUT, and VARIADIC arguments.

argname

The name of a function or aggregate argument. Note that COMMENT does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity.

argtype

The data type of a function or aggregate argument.

large_object_oid

The OID of the large object.

left_type
right_type

The data type(s) of the operator's arguments (optionally schema-qualified). Write NONE for the missing argument of a prefix or postfix operator.

PROCEDURAL

This is a noise word.

type_name

The name of the data type of the transform.

lang_name

The name of the language of the transform.

text

The new comment, written as a string literal; or NULL to drop the comment.

Notes

There is presently no security mechanism for viewing comments: any user connected to a database can see all the comments for objects in that database. For shared objects such as databases, roles, and tablespaces, comments are stored globally so any user connected to any database in the cluster can see all the comments for shared objects. Therefore, don't put security-critical information in comments.

Examples

Attach a comment to the table mytable:

```
COMMENT ON TABLE mytable IS 'This is my table.';
```

Remove it again:

```
COMMENT ON TABLE mytable IS NULL;
```

Some more examples:

```
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Computes sample variance';
COMMENT ON CAST (text AS int4) IS 'Allow casts from text to int4';
COMMENT ON COLLATION "fr_CA" IS 'Canadian French';
COMMENT ON COLUMN my_table.my_column IS 'Employee ID number';
COMMENT ON CONVERSION my_conv IS 'Conversion to UTF8';
COMMENT ON CONSTRAINT bar_col_cons ON bar IS 'Constrains column col';
COMMENT ON CONSTRAINT dom_col_constr ON DOMAIN dom IS 'Constrains col of domain';
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON DOMAIN my_domain IS 'Email Address Domain';
COMMENT ON EXTENSION hstore IS 'implements the hstore data type';
COMMENT ON FOREIGN DATA WRAPPER mywrapper IS 'my foreign data wrapper';
COMMENT ON FOREIGN TABLE my_foreign_table IS 'Employee Information in other database';
COMMENT ON FUNCTION my_function (timestamp) IS 'Returns Roman Numeral';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee ID';
COMMENT ON LANGUAGE plpython IS 'Python support for stored procedures';
COMMENT ON LARGE OBJECT 346344 IS 'Planning document';
COMMENT ON MATERIALIZED VIEW my_matview IS 'Summary of order history';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two texts';
COMMENT ON OPERATOR - (NONE, integer) IS 'Unary minus';
COMMENT ON OPERATOR CLASS int4ops USING btree IS '4 byte integer operators for btrees';
COMMENT ON OPERATOR FAMILY integer_ops USING btree IS 'all integer operators for btrees';
COMMENT ON POLICY my_policy ON mytable IS 'Filter rows by users';
COMMENT ON ROLE my_role IS 'Administration group for finance tables';
COMMENT ON RULE my_rule ON my_table IS 'Logs updates of employee records';
COMMENT ON SCHEMA my_schema IS 'Departmental data';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
COMMENT ON SERVER myserver IS 'my foreign server';
COMMENT ON TABLE my_schema.my_table IS 'Employee Information';
COMMENT ON TABLESPACE my_tablespace IS 'Tablespace for indexes';
COMMENT ON TEXT SEARCH CONFIGURATION my_config IS 'Special word filtering';
COMMENT ON TEXT SEARCH DICTIONARY swedish IS 'Snowball stemmer for Swedish language';
COMMENT ON TEXT SEARCH PARSER my_parser IS 'Splits text into words';
COMMENT ON TEXT SEARCH TEMPLATE snowball IS 'Snowball stemmer';
COMMENT ON TRANSFORM FOR hstore LANGUAGE plpythonu IS 'Transform between hstore and Python
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for RI';
COMMENT ON TYPE complex IS 'Complex number data type';
COMMENT ON VIEW my_view IS 'View of departmental costs';
```

Compatibility

There is no `COMMENT` command in the SQL standard.

COMMIT

Name

COMMIT — commit the current transaction

Synopsis

```
COMMIT [ WORK | TRANSACTION ]
```

Description

`COMMIT` commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs.

Parameters

`WORK`
`TRANSACTION`

Optional key words. They have no effect.

Notes

Use ROLLBACK to abort a transaction.

Issuing `COMMIT` when not inside a transaction does no harm, but it will provoke a warning message.

Examples

To commit the current transaction and make all changes permanent:

```
COMMIT;
```

Compatibility

The SQL standard only specifies the two forms `COMMIT` and `COMMIT WORK`. Otherwise, this command is fully conforming.

See Also

BEGIN, ROLLBACK

COMMIT PREPARED

Name

COMMIT PREPARED — commit a transaction that was earlier prepared for two-phase commit

Synopsis

```
COMMIT PREPARED transaction_id
```

Description

COMMIT PREPARED commits a transaction that is in prepared state.

Parameters

transaction_id

The transaction identifier of the transaction that is to be committed.

Notes

To commit a prepared transaction, you must be either the same user that executed the transaction originally, or a superuser. But you do not have to be in the same session that executed the transaction.

This command cannot be executed inside a transaction block. The prepared transaction is committed immediately.

All currently available prepared transactions are listed in the pg_prepared_xacts system view.

Examples

Commit the transaction identified by the transaction identifier foobar:

```
COMMIT PREPARED 'foobar';
```

Compatibility

COMMIT PREPARED is a PostgreSQL extension. It is intended for use by external transaction management systems, some of which are covered by standards (such as X/Open XA), but the SQL side of those systems is not standardized.

See Also

PREPARE TRANSACTION, ROLLBACK PREPARED

COPY

Name

COPY — copy data between a file and a table

Synopsis

```
COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | PROGRAM 'command' | STDIN }
    [ [ WITH ] ( option [, ...] ) ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | PROGRAM 'command' | STDOUT }
    [ [ WITH ] ( option [, ...] ) ]

where option can be one of:

    FORMAT format_name
    OIDS [ boolean ]
    FREEZE [ boolean ]
    DELIMITER 'delimiter_character'
    NULL 'null_string'
    HEADER [ boolean ]
    QUOTE 'quote_character'
    ESCAPE 'escape_character'
    FORCE_QUOTE { ( column_name [, ...] ) | * }
    FORCE_NOT_NULL ( column_name [, ...] )
    FORCE_NULL ( column_name [, ...] )
    ENCODING 'encoding_name'
```

Description

COPY moves data between PostgreSQL tables and standard file-system files. COPY TO copies the contents of a table *to* a file, while COPY FROM copies data *from* a file to a table (appending the data to whatever is in the table already). COPY TO can also copy the results of a SELECT query.

If a column list is specified, COPY TO copies only the data in the specified columns to the file. For COPY FROM, each field in the file is inserted, in order, into the specified column. Table columns not specified in the COPY FROM column list will receive their default values.

COPY with a file name instructs the PostgreSQL server to directly read from or write to a file. The file must be accessible by the PostgreSQL user (the user ID the server runs as) and the name must be specified from the viewpoint of the server. When PROGRAM is specified, the server executes the given command and reads from the standard output of the program, or writes to the standard input of the program. The command must be specified from the viewpoint of the server, and be executable by the PostgreSQL user. When STDIN or STDOUT is specified, data is transmitted via the connection between the client and the server.

Parameters

table_name

The name (optionally schema-qualified) of an existing table.

column_name

An optional list of columns to be copied. If no column list is specified, all columns of the table will be copied.

query

A SELECT or VALUES command whose results are to be copied. Note that parentheses are required around the query.

filename

The path name of the input or output file. An input file name can be an absolute or relative path, but an output file name must be an absolute path. Windows users might need to use an `E″` string and double any backslashes used in the path name.

PROGRAM

A command to execute. In `COPY FROM`, the input is read from standard output of the command, and in `COPY TO`, the output is written to the standard input of the command.

Note that the command is invoked by the shell, so if you need to pass any arguments to shell command that come from an untrusted source, you must be careful to strip or escape any special characters that might have a special meaning for the shell. For security reasons, it is best to use a fixed command string, or at least avoid passing any user input in it.

STDIN

Specifies that input comes from the client application.

STDOUT

Specifies that output goes to the client application.

boolean

Specifies whether the selected option should be turned on or off. You can write `TRUE`, `ON`, or `1` to enable the option, and `FALSE`, `OFF`, or `0` to disable it. The *boolean* value can also be omitted, in which case `TRUE` is assumed.

FORMAT

Selects the data format to be read or written: `text`, `csv` (Comma Separated Values), or `binary`. The default is `text`.

OIDS

Specifies copying the OID for each row. (An error is raised if `OIDS` is specified for a table that does not have OIDs, or in the case of copying a *query*.)

FREEZE

Requests copying the data with rows already frozen, just as they would be after running the `VACUUM FREEZE` command. This is intended as a performance option for initial data loading. Rows will be

frozen only if the table being loaded has been created or truncated in the current subtransaction, there are no cursors open and there are no older snapshots held by this transaction.

Note that all other sessions will immediately be able to see the data once it has been successfully loaded. This violates the normal rules of MVCC visibility and users specifying should be aware of the potential problems this might cause.

`DELIMITER`

Specifies the character that separates columns within each row (line) of the file. The default is a tab character in text format, a comma in `CSV` format. This must be a single one-byte character. This option is not allowed when using `binary` format.

`NULL`

Specifies the string that represents a null value. The default is `\N` (backslash-N) in text format, and an unquoted empty string in `CSV` format. You might prefer an empty string even in text format for cases where you don't want to distinguish nulls from empty strings. This option is not allowed when using `binary` format.

> **Note:** When using `COPY FROM`, any data item that matches this string will be stored as a null value, so you should make sure that you use the same string as you used with `COPY TO`.

`HEADER`

Specifies that the file contains a header line with the names of each column in the file. On output, the first line contains the column names from the table, and on input, the first line is ignored. This option is allowed only when using `CSV` format.

`QUOTE`

Specifies the quoting character to be used when a data value is quoted. The default is double-quote. This must be a single one-byte character. This option is allowed only when using `CSV` format.

`ESCAPE`

Specifies the character that should appear before a data character that matches the `QUOTE` value. The default is the same as the `QUOTE` value (so that the quoting character is doubled if it appears in the data). This must be a single one-byte character. This option is allowed only when using `CSV` format.

`FORCE_QUOTE`

Forces quoting to be used for all non-`NULL` values in each specified column. `NULL` output is never quoted. If `*` is specified, non-`NULL` values will be quoted in all columns. This option is allowed only in `COPY TO`, and only when using `CSV` format.

`FORCE_NOT_NULL`

Do not match the specified columns' values against the null string. In the default case where the null string is empty, this means that empty values will be read as zero-length strings rather than nulls, even when they are not quoted. This option is allowed only in `COPY FROM`, and only when using `CSV` format.

`FORCE_NULL`

Match the specified columns' values against the null string, even if it has been quoted, and if a match is found set the value to `NULL`. In the default case where the null string is empty, this converts a quoted empty string into NULL. This option is allowed only in `COPY FROM`, and only when using `CSV` format.

`ENCODING`

Specifies that the file is encoded in the *encoding_name*. If this option is omitted, the current client encoding is used. See the Notes below for more details.

Outputs

On successful completion, a `COPY` command returns a command tag of the form

```
COPY count
```

The *count* is the number of rows copied.

> **Note:** psql will print this command tag only if the command was not `COPY ... TO STDOUT`, or the equivalent psql meta-command `\copy ... to stdout`. This is to prevent confusing the command tag with the data that was just printed.

Notes

`COPY` can only be used with plain tables, not with views. However, you can write `COPY (SELECT * FROM viewname) TO ...`.

`COPY` only deals with the specific table named; it does not copy data to or from child tables. Thus for example `COPY table TO` shows the same data as `SELECT * FROM ONLY table`. But `COPY (SELECT * FROM table) TO ...` can be used to dump all of the data in an inheritance hierarchy.

You must have select privilege on the table whose values are read by `COPY TO`, and insert privilege on the table into which values are inserted by `COPY FROM`. It is sufficient to have column privileges on the column(s) listed in the command.

If row-level security is enabled for the table, the relevant `SELECT` policies will apply to `COPY table TO` statements. Currently, `COPY FROM` is not supported for tables with row-level security. Use equivalent `INSERT` statements instead.

Files named in a `COPY` command are read or written directly by the server, not by the client application. Therefore, they must reside on or be accessible to the database server machine, not the client. They must be accessible to and readable or writable by the PostgreSQL user (the user ID the server runs as), not the client. Similarly, the command specified with `PROGRAM` is executed directly by the server, not by the client application, must be executable by the PostgreSQL user. `COPY` naming a file or command is only allowed to database superusers, since it allows reading or writing any file that the server has privileges to access.

Do not confuse `COPY` with the psql instruction `\copy`. `\copy` invokes `COPY FROM STDIN` or `COPY TO STDOUT`, and then fetches/stores the data in a file accessible to the psql client. Thus, file accessibility and access rights depend on the client rather than the server when `\copy` is used.

It is recommended that the file name used in `COPY` always be specified as an absolute path. This is enforced by the server in the case of `COPY TO`, but for `COPY FROM` you do have the option of reading from a file specified by a relative path. The path will be interpreted relative to the working directory of the server process (normally the cluster's data directory), not the client's working directory.

Executing a command with `PROGRAM` might be restricted by the operating system's access control mechanisms, such as SELinux.

`COPY FROM` will invoke any triggers and check constraints on the destination table. However, it will not invoke rules.

`COPY` input and output is affected by `DateStyle`. To ensure portability to other PostgreSQL installations that might use non-default `DateStyle` settings, `DateStyle` should be set to `ISO` before using `COPY TO`. It is also a good idea to avoid dumping data with `IntervalStyle` set to `sql_standard`, because negative interval values might be misinterpreted by a server that has a different setting for `IntervalStyle`.

Input data is interpreted according to `ENCODING` option or the current client encoding, and output data is encoded in `ENCODING` or the current client encoding, even if the data does not pass through the client but is read from or written to a file directly by the server.

`COPY` stops operation at the first error. This should not lead to problems in the event of a `COPY TO`, but the target table will already have received earlier rows in a `COPY FROM`. These rows will not be visible or accessible, but they still occupy disk space. This might amount to a considerable amount of wasted disk space if the failure happened well into a large copy operation. You might wish to invoke `VACUUM` to recover the wasted space.

`FORCE_NULL` and `FORCE_NOT_NULL` can be used simultaneously on the same column. This results in converting quoted null strings to null values and unquoted null strings to empty strings.

File Formats

Text Format

When the `text` format is used, the data read or written is a text file with one line per table row. Columns in a row are separated by the delimiter character. The column values themselves are strings generated by the output function, or acceptable to the input function, of each attribute's data type. The specified null string is used in place of columns that are null. `COPY FROM` will raise an error if any line of the input file contains more or fewer columns than are expected. If `OIDS` is specified, the OID is read or written as the first column, preceding the user data columns.

End of data can be represented by a single line containing just backslash-period (`\.`). An end-of-data marker is not necessary when reading from a file, since the end of file serves perfectly well; it is needed only when copying data to or from client applications using pre-3.0 client protocol.

Backslash characters (`\`) can be used in the `COPY` data to quote data characters that might otherwise be taken as row or column delimiters. In particular, the following characters *must* be preceded by a backslash if they appear as part of a column value: backslash itself, newline, carriage return, and the current delimiter character.

The specified null string is sent by `COPY TO` without adding any backslashes; conversely, `COPY FROM` matches the input against the null string before removing backslashes. Therefore, a null string such as `\N` cannot be confused with the actual data value `\N` (which would be represented as `\\N`).

The following special backslash sequences are recognized by `COPY FROM`:

Sequence	Represents
`\b`	Backspace (ASCII 8)
`\f`	Form feed (ASCII 12)
`\n`	Newline (ASCII 10)
`\r`	Carriage return (ASCII 13)
`\t`	Tab (ASCII 9)
`\v`	Vertical tab (ASCII 11)
`\`*digits*	Backslash followed by one to three octal digits specifies the character with that numeric code
`\x`*digits*	Backslash `x` followed by one or two hex digits specifies the character with that numeric code

Presently, `COPY TO` will never emit an octal or hex-digits backslash sequence, but it does use the other sequences listed above for those control characters.

Any other backslashed character that is not mentioned in the above table will be taken to represent itself. However, beware of adding backslashes unnecessarily, since that might accidentally produce a string matching the end-of-data marker (`\.`) or the null string (`\N` by default). These strings will be recognized before any other backslash processing is done.

It is strongly recommended that applications generating `COPY` data convert data newlines and carriage returns to the `\n` and `\r` sequences respectively. At present it is possible to represent a data carriage return by a backslash and carriage return, and to represent a data newline by a backslash and newline. However, these representations might not be accepted in future releases. They are also highly vulnerable to corruption if the `COPY` file is transferred across different machines (for example, from Unix to Windows or vice versa).

`COPY TO` will terminate each row with a Unix-style newline ("`\n`"). Servers running on Microsoft Windows instead output carriage return/newline ("`\r\n`"), but only for `COPY` to a server file; for consistency across platforms, `COPY TO STDOUT` always sends "`\n`" regardless of server platform. `COPY FROM` can handle lines ending with newlines, carriage returns, or carriage return/newlines. To reduce the risk of error due to un-backslashed newlines or carriage returns that were meant as data, `COPY FROM` will complain if the line endings in the input are not all alike.

CSV Format

This format option is used for importing and exporting the Comma Separated Value (`CSV`) file format used by many other programs, such as spreadsheets. Instead of the escaping rules used by PostgreSQL's standard text format, it produces and recognizes the common CSV escaping mechanism.

The values in each record are separated by the `DELIMITER` character. If the value contains the delimiter character, the `QUOTE` character, the `NULL` string, a carriage return, or line feed character, then the whole value is prefixed and suffixed by the `QUOTE` character, and any occurrence within the value of a `QUOTE`

character or the ESCAPE character is preceded by the escape character. You can also use FORCE_QUOTE to force quotes when outputting non-NULL values in specific columns.

The CSV format has no standard way to distinguish a NULL value from an empty string. PostgreSQL's COPY handles this by quoting. A NULL is output as the NULL parameter string and is not quoted, while a non-NULL value matching the NULL parameter string is quoted. For example, with the default settings, a NULL is written as an unquoted empty string, while an empty string data value is written with double quotes (""). Reading values follows similar rules. You can use FORCE_NOT_NULL to prevent NULL input comparisons for specific columns. You can also use FORCE_NULL to convert quoted null string data values to NULL.

Because backslash is not a special character in the CSV format, \., the end-of-data marker, could also appear as a data value. To avoid any misinterpretation, a \. data value appearing as a lone entry on a line is automatically quoted on output, and on input, if quoted, is not interpreted as the end-of-data marker. If you are loading a file created by another application that has a single unquoted column and might have a value of \., you might need to quote that value in the input file.

Note: In CSV format, all characters are significant. A quoted value surrounded by white space, or any characters other than DELIMITER, will include those characters. This can cause errors if you import data from a system that pads CSV lines with white space out to some fixed width. If such a situation arises you might need to preprocess the CSV file to remove the trailing white space, before importing the data into PostgreSQL.

Note: CSV format will both recognize and produce CSV files with quoted values containing embedded carriage returns and line feeds. Thus the files are not strictly one line per table row like text-format files.

Note: Many programs produce strange and occasionally perverse CSV files, so the file format is more a convention than a standard. Thus you might encounter some files that cannot be imported using this mechanism, and COPY might produce files that other programs cannot process.

Binary Format

The binary format option causes all data to be stored/read as binary format rather than as text. It is somewhat faster than the text and CSV formats, but a binary-format file is less portable across machine architectures and PostgreSQL versions. Also, the binary format is very data type specific; for example it will not work to output binary data from a smallint column and read it into an integer column, even though that would work fine in text format.

The binary file format consists of a file header, zero or more tuples containing the row data, and a file trailer. Headers and data are in network byte order.

Note: PostgreSQL releases before 7.4 used a different binary file format.

File Header

The file header consists of 15 bytes of fixed fields, followed by a variable-length header extension area. The fixed fields are:

Signature

11-byte sequence `PGCOPY\n\377\r\n\0` — note that the zero byte is a required part of the signature. (The signature is designed to allow easy identification of files that have been munged by a non-8-bit-clean transfer. This signature will be changed by end-of-line-translation filters, dropped zero bytes, dropped high bits, or parity changes.)

Flags field

32-bit integer bit mask to denote important aspects of the file format. Bits are numbered from 0 (LSB) to 31 (MSB). Note that this field is stored in network byte order (most significant byte first), as are all the integer fields used in the file format. Bits 16-31 are reserved to denote critical file format issues; a reader should abort if it finds an unexpected bit set in this range. Bits 0-15 are reserved to signal backwards-compatible format issues; a reader should simply ignore any unexpected bits set in this range. Currently only one flag bit is defined, and the rest must be zero:

Bit 16

if 1, OIDs are included in the data; if 0, not

Header extension area length

32-bit integer, length in bytes of remainder of header, not including self. Currently, this is zero, and the first tuple follows immediately. Future changes to the format might allow additional data to be present in the header. A reader should silently skip over any header extension data it does not know what to do with.

The header extension area is envisioned to contain a sequence of self-identifying chunks. The flags field is not intended to tell readers what is in the extension area. Specific design of header extension contents is left for a later release.

This design allows for both backwards-compatible header additions (add header extension chunks, or set low-order flag bits) and non-backwards-compatible changes (set high-order flag bits to signal such changes, and add supporting data to the extension area if needed).

Tuples

Each tuple begins with a 16-bit integer count of the number of fields in the tuple. (Presently, all tuples in a table will have the same count, but that might not always be true.) Then, repeated for each field in the tuple, there is a 32-bit length word followed by that many bytes of field data. (The length word does not include itself, and can be zero.) As a special case, -1 indicates a NULL field value. No value bytes follow in the NULL case.

There is no alignment padding or any other extra data between fields.

Presently, all data values in a binary-format file are assumed to be in binary format (format code one). It is anticipated that a future extension might add a header field that allows per-column format codes to be specified.

To determine the appropriate binary format for the actual tuple data you should consult the PostgreSQL source, in particular the `*send` and `*recv` functions for each column's data type (typically these functions are found in the `src/backend/utils/adt/` directory of the source distribution).

If OIDs are included in the file, the OID field immediately follows the field-count word. It is a normal field except that it's not included in the field-count. In particular it has a length word — this will allow handling of 4-byte vs. 8-byte OIDs without too much pain, and will allow OIDs to be shown as null if that ever proves desirable.

File Trailer

The file trailer consists of a 16-bit integer word containing -1. This is easily distinguished from a tuple's field-count word.

A reader should report an error if a field-count word is neither -1 nor the expected number of columns. This provides an extra check against somehow getting out of sync with the data.

Examples

The following example copies a table to the client using the vertical bar (`|`) as the field delimiter:

```
COPY country TO STDOUT (DELIMITER '|');
```

To copy data from a file into the `country` table:

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

To copy into a file just the countries whose names start with 'A':

```
COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO '/usr1/proj/bray/sql/a_list_co
```

To copy into a compressed file, you can pipe the output through an external compression program:

```
COPY country TO PROGRAM 'gzip > /usr1/proj/bray/sql/country_data.gz';
```

Here is a sample of data suitable for copying into a table from `STDIN`:

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
```

```
ZM      ZAMBIA
ZW      ZIMBABWE
```

Note that the white space on each line is actually a tab character.

The following is the same data, output in binary format. The data is shown after filtering through the Unix utility `od -c`. The table has three columns; the first has type `char(2)`, the second has type `text`, and the third has type `integer`. All the rows have a null value in the third column.

```
0000000   P   G   C   O   P   Y  \n 377  \r  \n  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0 003  \0  \0  \0 002   A   F  \0  \0  \0 013   A
0000040   F   G   H   A   N   I   S   T   A   N 377 377 377 377  \0 003
0000060  \0  \0  \0 002   A   L  \0  \0  \0 007   A   L   B   A   N   I
0000100   A 377 377 377 377  \0 003  \0  \0  \0 002   D   Z  \0  \0  \0
0000120 007   A   L   G   E   R   I   A 377 377 377 377  \0 003  \0  \0
0000140  \0 002   Z   M  \0  \0  \0 006   Z   A   M   B   I   A 377 377
0000160 377 377  \0 003  \0  \0  \0 002   Z   W  \0  \0  \0  \b   Z   I
0000200   M   B   A   B   W   E 377 377 377 377 377 377
```

Compatibility

There is no `COPY` statement in the SQL standard.

The following syntax was used before PostgreSQL version 9.0 and is still supported:

```
COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | STDIN }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'delimiter_character' ]
          [ NULL [ AS ] 'null string' ]
          [ CSV [ HEADER ]
                [ QUOTE [ AS ] 'quote_character' ]
                [ ESCAPE [ AS ] 'escape_character' ]
                [ FORCE NOT NULL column_name [, ...] ] ] ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | STDOUT }
    [ [ WITH ]
          [ BINARY ]
          [ OIDS ]
          [ DELIMITER [ AS ] 'delimiter_character' ]
          [ NULL [ AS ] 'null string' ]
          [ CSV [ HEADER ]
                [ QUOTE [ AS ] 'quote_character' ]
                [ ESCAPE [ AS ] 'escape_character' ]
                [ FORCE QUOTE { column_name [, ...] | * } ] ] ]
```

Note that in this syntax, `BINARY` and `CSV` are treated as independent keywords, not as arguments of a `FORMAT` option.

The following syntax was used before PostgreSQL version 7.3 and is still supported:

```
COPY [ BINARY ] table_name [ WITH OIDS ]
    FROM { 'filename' | STDIN }
    [ [USING] DELIMITERS 'delimiter_character' ]
    [ WITH NULL AS 'null_string' ]

COPY [ BINARY ] table_name [ WITH OIDS ]
    TO { 'filename' | STDOUT }
    [ [USING] DELIMITERS 'delimiter_character' ]
    [ WITH NULL AS 'null_string' ]
```

CREATE AGGREGATE

Name

CREATE AGGREGATE — define a new aggregate function

Synopsis

```
CREATE AGGREGATE name ( [ argmode ] [ argname ] arg_data_type [ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , INITCOND = initial_condition ]
    [ , MSFUNC = msfunc ]
    [ , MINVFUNC = minvfunc ]
    [ , MSTYPE = mstate_data_type ]
    [ , MSSPACE = mstate_data_size ]
    [ , MFINALFUNC = mffunc ]
    [ , MFINALFUNC_EXTRA ]
    [ , MINITCOND = minitial_condition ]
    [ , SORTOP = sort_operator ]
)

CREATE AGGREGATE name ( [ [ argmode ] [ argname ] arg_data_type [ , ... ] ]
                        ORDER BY [ argmode ] [ argname ] arg_data_type [ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , INITCOND = initial_condition ]
    [ , HYPOTHETICAL ]
)

or the old syntax

CREATE AGGREGATE name (
    BASETYPE = base_type,
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , INITCOND = initial_condition ]
    [ , MSFUNC = msfunc ]
    [ , MINVFUNC = minvfunc ]
    [ , MSTYPE = mstate_data_type ]
    [ , MSSPACE = mstate_data_size ]
    [ , MFINALFUNC = mffunc ]
```

```
    [ , MFINALFUNC_EXTRA ]
    [ , MINITCOND = minitial_condition ]
    [ , SORTOP = sort_operator ]
)
```

Description

`CREATE AGGREGATE` defines a new aggregate function. Some basic and commonly-used aggregate functions are included with the distribution; they are documented in Section 9.20. If one defines new types or needs an aggregate function not already provided, then `CREATE AGGREGATE` can be used to provide the desired features.

If a schema name is given (for example, `CREATE AGGREGATE myschema.myagg ...`) then the aggregate function is created in the specified schema. Otherwise it is created in the current schema.

An aggregate function is identified by its name and input data type(s). Two aggregates in the same schema can have the same name if they operate on different input types. The name and input data type(s) of an aggregate must also be distinct from the name and input data type(s) of every ordinary function in the same schema. This behavior is identical to overloading of ordinary function names (see CREATE FUNCTION).

A simple aggregate function is made from one or two ordinary functions: a state transition function *sfunc*, and an optional final calculation function *ffunc*. These are used as follows:

```
sfunc( internal-state, next-data-values ) ---> next-internal-state
ffunc( internal-state ) ---> aggregate-value
```

PostgreSQL creates a temporary variable of data type *stype* to hold the current internal state of the aggregate. At each input row, the aggregate argument value(s) are calculated and the state transition function is invoked with the current state value and the new argument value(s) to calculate a new internal state value. After all the rows have been processed, the final function is invoked once to calculate the aggregate's return value. If there is no final function then the ending state value is returned as-is.

An aggregate function can provide an initial condition, that is, an initial value for the internal state value. This is specified and stored in the database as a value of type `text`, but it must be a valid external representation of a constant of the state value data type. If it is not supplied then the state value starts out null.

If the state transition function is declared "strict", then it cannot be called with null inputs. With such a transition function, aggregate execution behaves as follows. Rows with any null input values are ignored (the function is not called and the previous state value is retained). If the initial state value is null, then at the first row with all-nonnull input values, the first argument value replaces the state value, and the transition function is invoked at each subsequent row with all-nonnull input values. This is handy for implementing aggregates like `max`. Note that this behavior is only available when *state_data_type* is the same as the first *arg_data_type*. When these types are different, you must supply a nonnull initial condition or use a nonstrict transition function.

If the state transition function is not strict, then it will be called unconditionally at each input row, and must deal with null inputs and null state values for itself. This allows the aggregate author to have full control over the aggregate's handling of null values.

If the final function is declared "strict", then it will not be called when the ending state value is null; instead a null result will be returned automatically. (Of course this is just the normal behavior of strict functions.) In any case the final function has the option of returning a null value. For example, the final function for `avg` returns null when it sees there were zero input rows.

Sometimes it is useful to declare the final function as taking not just the state value, but extra parameters corresponding to the aggregate's input values. The main reason for doing this is if the final function is polymorphic and the state value's data type would be inadequate to pin down the result type. These extra parameters are always passed as NULL (and so the final function must not be strict when the `FINALFUNC_EXTRA` option is used), but nonetheless they are valid parameters. The final function could for example make use of `get_fn_expr_argtype` to identify the actual argument type in the current call.

An aggregate can optionally support *moving-aggregate mode*, as described in Section 35.10.1. This requires specifying the `MSFUNC`, `MINVFUNC`, and `MSTYPE` parameters, and optionally the `MSSPACE`, `MFINALFUNC`, `MFINALFUNC_EXTRA`, and `MINITCOND` parameters. Except for `MINVFUNC`, these parameters work like the corresponding simple-aggregate parameters without `M`; they define a separate implementation of the aggregate that includes an inverse transition function.

The syntax with `ORDER BY` in the parameter list creates a special type of aggregate called an *ordered-set aggregate*; or if `HYPOTHETICAL` is specified, then a *hypothetical-set aggregate* is created. These aggregates operate over groups of sorted values in order-dependent ways, so that specification of an input sort order is an essential part of a call. Also, they can have *direct* arguments, which are arguments that are evaluated only once per aggregation rather than once per input row. Hypothetical-set aggregates are a subclass of ordered-set aggregates in which some of the direct arguments are required to match, in number and data types, the aggregated argument columns. This allows the values of those direct arguments to be added to the collection of aggregate-input rows as an additional "hypothetical" row.

Aggregates that behave like `MIN` or `MAX` can sometimes be optimized by looking into an index instead of scanning every input row. If this aggregate can be so optimized, indicate it by specifying a *sort operator*. The basic requirement is that the aggregate must yield the first element in the sort ordering induced by the operator; in other words:

```
SELECT agg(col) FROM tab;
```

must be equivalent to:

```
SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;
```

Further assumptions are that the aggregate ignores null inputs, and that it delivers a null result if and only if there were no non-null inputs. Ordinarily, a data type's $<$ operator is the proper sort operator for `MIN`, and $>$ is the proper sort operator for `MAX`. Note that the optimization will never actually take effect unless the specified operator is the "less than" or "greater than" strategy member of a B-tree index operator class.

To be able to create an aggregate function, you must have `USAGE` privilege on the argument types, the state type(s), and the return type, as well as `EXECUTE` privilege on the transition and final functions.

Parameters

name

The name (optionally schema-qualified) of the aggregate function to create.

argmode

The mode of an argument: `IN` or `VARIADIC`. (Aggregate functions do not support `OUT` arguments.) If omitted, the default is `IN`. Only the last argument can be marked `VARIADIC`.

argname

The name of an argument. This is currently only useful for documentation purposes. If omitted, the argument has no name.

arg_data_type

An input data type on which this aggregate function operates. To create a zero-argument aggregate function, write `*` in place of the list of argument specifications. (An example of such an aggregate is `count(*)`.)

base_type

In the old syntax for `CREATE AGGREGATE`, the input data type is specified by a `basetype` parameter rather than being written next to the aggregate name. Note that this syntax allows only one input parameter. To define a zero-argument aggregate function with this syntax, specify the `basetype` as `"ANY"` (not `*`). Ordered-set aggregates cannot be defined with the old syntax.

sfunc

The name of the state transition function to be called for each input row. For a normal *N*-argument aggregate function, the *sfunc* must take *N*+1 arguments, the first being of type *state_data_type* and the rest matching the declared input data type(s) of the aggregate. The function must return a value of type *state_data_type*. This function takes the current state value and the current input data value(s), and returns the next state value.

For ordered-set (including hypothetical-set) aggregates, the state transition function receives only the current state value and the aggregated arguments, not the direct arguments. Otherwise it is the same.

state_data_type

The data type for the aggregate's state value.

state_data_size

The approximate average size (in bytes) of the aggregate's state value. If this parameter is omitted or is zero, a default estimate is used based on the *state_data_type*. The planner uses this value to estimate the memory required for a grouped aggregate query. The planner will consider using hash aggregation for such a query only if the hash table is estimated to fit in work_mem; therefore, large values of this parameter discourage use of hash aggregation.

ffunc

The name of the final function called to compute the aggregate's result after all input rows have been traversed. For a normal aggregate, this function must take a single argument of type *state_data_type*. The return data type of the aggregate is defined as the return type of this function. If *ffunc* is not specified, then the ending state value is used as the aggregate's result, and the return type is *state_data_type*.

For ordered-set (including hypothetical-set) aggregates, the final function receives not only the final state value, but also the values of all the direct arguments.

If `FINALFUNC_EXTRA` is specified, then in addition to the final state value and any direct arguments, the final function receives extra NULL values corresponding to the aggregate's regular (aggregated)

arguments. This is mainly useful to allow correct resolution of the aggregate result type when a polymorphic aggregate is being defined.

initial_condition

The initial setting for the state value. This must be a string constant in the form accepted for the data type *state_data_type*. If not specified, the state value starts out null.

msfunc

The name of the forward state transition function to be called for each input row in moving-aggregate mode. This is exactly like the regular transition function, except that its first argument and result are of type *mstate_data_type*, which might be different from *state_data_type*.

minvfunc

The name of the inverse state transition function to be used in moving-aggregate mode. This function has the same argument and result types as *msfunc*, but it is used to remove a value from the current aggregate state, rather than add a value to it. The inverse transition function must have the same strictness attribute as the forward state transition function.

mstate_data_type

The data type for the aggregate's state value, when using moving-aggregate mode.

mstate_data_size

The approximate average size (in bytes) of the aggregate's state value, when using moving-aggregate mode. This works the same as *state_data_size*.

mffunc

The name of the final function called to compute the aggregate's result after all input rows have been traversed, when using moving-aggregate mode. This works the same as *ffunc*, except that its first argument's type is *mstate_data_type* and extra dummy arguments are specified by writing MFINALFUNC_EXTRA. The aggregate result type determined by *mffunc* or *mstate_data_type* must match that determined by the aggregate's regular implementation.

minitial_condition

The initial setting for the state value, when using moving-aggregate mode. This works the same as *initial_condition*.

sort_operator

The associated sort operator for a MIN- or MAX-like aggregate. This is just an operator name (possibly schema-qualified). The operator is assumed to have the same input data types as the aggregate (which must be a single-argument normal aggregate).

HYPOTHETICAL

For ordered-set aggregates only, this flag specifies that the aggregate arguments are to be processed according to the requirements for hypothetical-set aggregates: that is, the last few direct arguments must match the data types of the aggregated (WITHIN GROUP) arguments. The HYPOTHETICAL flag has no effect on run-time behavior, only on parse-time resolution of the data types and collations of the aggregate's arguments.

The parameters of CREATE AGGREGATE can be written in any order, not just the order illustrated above.

Notes

In parameters that specify support function names, you can write a schema name if needed, for example `SFUNC = public.sum`. Do not write argument types there, however — the argument types of the support functions are determined from other parameters.

If an aggregate supports moving-aggregate mode, it will improve calculation efficiency when the aggregate is used as a window function for a window with moving frame start (that is, a frame start mode other than `UNBOUNDED PRECEDING`). Conceptually, the forward transition function adds input values to the aggregate's state when they enter the window frame from the bottom, and the inverse transition function removes them again when they leave the frame at the top. So, when values are removed, they are always removed in the same order they were added. Whenever the inverse transition function is invoked, it will thus receive the earliest added but not yet removed argument value(s). The inverse transition function can assume that at least one row will remain in the current state after it removes the oldest row. (When this would not be the case, the window function mechanism simply starts a fresh aggregation, rather than using the inverse transition function.)

The forward transition function for moving-aggregate mode is not allowed to return NULL as the new state value. If the inverse transition function returns NULL, this is taken as an indication that the inverse function cannot reverse the state calculation for this particular input, and so the aggregate calculation will be redone from scratch for the current frame starting position. This convention allows moving-aggregate mode to be used in situations where there are some infrequent cases that are impractical to reverse out of the running state value.

If no moving-aggregate implementation is supplied, the aggregate can still be used with moving frames, but PostgreSQL will recompute the whole aggregation whenever the start of the frame moves. Note that whether or not the aggregate supports moving-aggregate mode, PostgreSQL can handle a moving frame end without recalculation; this is done by continuing to add new values to the aggregate's state. It is assumed that the final function does not damage the aggregate's state value, so that the aggregation can be continued even after an aggregate result value has been obtained for one set of frame boundaries.

The syntax for ordered-set aggregates allows `VARIADIC` to be specified for both the last direct parameter and the last aggregated (`WITHIN GROUP`) parameter. However, the current implementation restricts use of `VARIADIC` in two ways. First, ordered-set aggregates can only use `VARIADIC "any"`, not other variadic array types. Second, if the last direct parameter is `VARIADIC "any"`, then there can be only one aggregated parameter and it must also be `VARIADIC "any"`. (In the representation used in the system catalogs, these two parameters are merged into a single `VARIADIC "any"` item, since `pg_proc` cannot represent functions with more than one `VARIADIC` parameter.) If the aggregate is a hypothetical-set aggregate, the direct arguments that match the `VARIADIC "any"` parameter are the hypothetical ones; any preceding parameters represent additional direct arguments that are not constrained to match the aggregated arguments.

Currently, ordered-set aggregates do not need to support moving-aggregate mode, since they cannot be used as window functions.

Examples

See Section 35.10.

Compatibility

`CREATE AGGREGATE` is a PostgreSQL language extension. The SQL standard does not provide for user-defined aggregate functions.

See Also

ALTER AGGREGATE, DROP AGGREGATE

CREATE CAST

Name

CREATE CAST — define a new cast

Synopsis

```
CREATE CAST (source_type AS target_type)
    WITH FUNCTION function_name (argument_type [, ...])
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
    WITHOUT FUNCTION
    [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
    WITH INOUT
    [ AS ASSIGNMENT | AS IMPLICIT ]
```

Description

`CREATE CAST` defines a new cast. A cast specifies how to perform a conversion between two data types. For example,

```
SELECT CAST(42 AS float8);
```

converts the integer constant 42 to type `float8` by invoking a previously specified function, in this case `float8(int4)`. (If no suitable cast has been defined, the conversion fails.)

Two types can be *binary coercible*, which means that the conversion can be performed "for free" without invoking any function. This requires that corresponding values use the same internal representation. For instance, the types `text` and `varchar` are binary coercible both ways. Binary coercibility is not necessarily a symmetric relationship. For example, the cast from `xml` to `text` can be performed for free in the present implementation, but the reverse direction requires a function that performs at least a syntax check. (Two types that are binary coercible both ways are also referred to as binary compatible.)

You can define a cast as an *I/O conversion cast* by using the `WITH INOUT` syntax. An I/O conversion cast is performed by invoking the output function of the source data type, and passing the resulting string to the input function of the target data type. In many common cases, this feature avoids the need to write a separate cast function for conversion. An I/O conversion cast acts the same as a regular function-based cast; only the implementation is different.

By default, a cast can be invoked only by an explicit cast request, that is an explicit `CAST(`*`x`* `AS` *`typename`*`)` or *`x`*`::`*`typename`* construct.

If the cast is marked `AS ASSIGNMENT` then it can be invoked implicitly when assigning a value to a column of the target data type. For example, supposing that `foo.f1` is a column of type `text`, then:

```
INSERT INTO foo (f1) VALUES (42);
```

will be allowed if the cast from type `integer` to type `text` is marked `AS ASSIGNMENT`, otherwise not. (We generally use the term *assignment cast* to describe this kind of cast.)

If the cast is marked `AS IMPLICIT` then it can be invoked implicitly in any context, whether assignment or internally in an expression. (We generally use the term *implicit cast* to describe this kind of cast.) For example, consider this query:

```
SELECT 2 + 4.0;
```

The parser initially marks the constants as being of type `integer` and `numeric` respectively. There is no `integer + numeric` operator in the system catalogs, but there is a `numeric + numeric` operator. The query will therefore succeed if a cast from `integer` to `numeric` is available and is marked `AS IMPLICIT` — which in fact it is. The parser will apply the implicit cast and resolve the query as if it had been written

```
SELECT CAST ( 2 AS numeric ) + 4.0;
```

Now, the catalogs also provide a cast from `numeric` to `integer`. If that cast were marked `AS IMPLICIT` — which it is not — then the parser would be faced with choosing between the above interpretation and the alternative of casting the `numeric` constant to `integer` and applying the `integer + integer` operator. Lacking any knowledge of which choice to prefer, it would give up and declare the query ambiguous. The fact that only one of the two casts is implicit is the way in which we teach the parser to prefer resolution of a mixed `numeric`-and-`integer` expression as `numeric`; there is no built-in knowledge about that.

It is wise to be conservative about marking casts as implicit. An overabundance of implicit casting paths can cause PostgreSQL to choose surprising interpretations of commands, or to be unable to resolve commands at all because there are multiple possible interpretations. A good rule of thumb is to make a cast implicitly invokable only for information-preserving transformations between types in the same general type category. For example, the cast from `int2` to `int4` can reasonably be implicit, but the cast from `float8` to `int4` should probably be assignment-only. Cross-type-category casts, such as `text` to `int4`, are best made explicit-only.

> **Note:** Sometimes it is necessary for usability or standards-compliance reasons to provide multiple implicit casts among a set of types, resulting in ambiguity that cannot be avoided as above. The parser has a fallback heuristic based on *type categories* and *preferred types* that can help to provide desired behavior in such cases. See CREATE TYPE for more information.

To be able to create a cast, you must own the source or the target data type and have `USAGE` privilege on the other type. To create a binary-coercible cast, you must be superuser. (This restriction is made because an erroneous binary-coercible cast conversion can easily crash the server.)

Parameters

`source_type`

The name of the source data type of the cast.

target_type

The name of the target data type of the cast.

function_name(*argument_type* [, ...])

The function used to perform the cast. The function name can be schema-qualified. If it is not, the function will be looked up in the schema search path. The function's result data type must match the target type of the cast. Its arguments are discussed below.

WITHOUT FUNCTION

Indicates that the source type is binary-coercible to the target type, so no function is required to perform the cast.

WITH INOUT

Indicates that the cast is an I/O conversion cast, performed by invoking the output function of the source data type, and passing the resulting string to the input function of the target data type.

AS ASSIGNMENT

Indicates that the cast can be invoked implicitly in assignment contexts.

AS IMPLICIT

Indicates that the cast can be invoked implicitly in any context.

Cast implementation functions can have one to three arguments. The first argument type must be identical to or binary-coercible from the cast's source type. The second argument, if present, must be type `integer`; it receives the type modifier associated with the destination type, or `-1` if there is none. The third argument, if present, must be type `boolean`; it receives `true` if the cast is an explicit cast, `false` otherwise. (Bizarrely, the SQL standard demands different behaviors for explicit and implicit casts in some cases. This argument is supplied for functions that must implement such casts. It is not recommended that you design your own data types so that this matters.)

The return type of a cast function must be identical to or binary-coercible to the cast's target type.

Ordinarily a cast must have different source and target data types. However, it is allowed to declare a cast with identical source and target types if it has a cast implementation function with more than one argument. This is used to represent type-specific length coercion functions in the system catalogs. The named function is used to coerce a value of the type to the type modifier value given by its second argument.

When a cast has different source and target types and a function that takes more than one argument, it supports converting from one type to another and applying a length coercion in a single step. When no such entry is available, coercion to a type that uses a type modifier involves two cast steps, one to convert between data types and a second to apply the modifier.

A cast to or from a domain type currently has no effect. Casting to or from a domain uses the casts associated with its underlying type.

Notes

Use DROP CAST to remove user-defined casts.

Remember that if you want to be able to convert types both ways you need to declare casts both ways explicitly.

It is normally not necessary to create casts between user-defined types and the standard string types (`text`, `varchar`, and `char(n)`, as well as user-defined types that are defined to be in the string category). PostgreSQL provides automatic I/O conversion casts for that. The automatic casts to string types are treated as assignment casts, while the automatic casts from string types are explicit-only. You can override this behavior by declaring your own cast to replace an automatic cast, but usually the only reason to do so is if you want the conversion to be more easily invokable than the standard assignment-only or explicit-only setting. Another possible reason is that you want the conversion to behave differently from the type's I/O function; but that is sufficiently surprising that you should think twice about whether it's a good idea. (A small number of the built-in types do indeed have different behaviors for conversions, mostly because of requirements of the SQL standard.)

While not required, it is recommended that you continue to follow this old convention of naming cast implementation functions after the target data type. Many users are used to being able to cast data types using a function-style notation, that is *typename*(*x*). This notation is in fact nothing more nor less than a call of the cast implementation function; it is not specially treated as a cast. If your conversion functions are not named to support this convention then you will have surprised users. Since PostgreSQL allows overloading of the same function name with different argument types, there is no difficulty in having multiple conversion functions from different types that all use the target type's name.

> **Note:** Actually the preceding paragraph is an oversimplification: there are two cases in which a function-call construct will be treated as a cast request without having matched it to an actual function. If a function call *name*(*x*) does not exactly match any existing function, but *name* is the name of a data type and `pg_cast` provides a binary-coercible cast to this type from the type of *x*, then the call will be construed as a binary-coercible cast. This exception is made so that binary-coercible casts can be invoked using functional syntax, even though they lack any function. Likewise, if there is no `pg_cast` entry but the cast would be to or from a string type, the call will be construed as an I/O conversion cast. This exception allows I/O conversion casts to be invoked using functional syntax.

> **Note:** There is also an exception to the exception: I/O conversion casts from composite types to string types cannot be invoked using functional syntax, but must be written in explicit cast syntax (either `CAST` or `::` notation). This exception was added because after the introduction of automatically-provided I/O conversion casts, it was found too easy to accidentally invoke such a cast when a function or column reference was intended.

Examples

To create an assignment cast from type `bigint` to type `int4` using the function `int4(bigint)`:

```
CREATE CAST (bigint AS int4) WITH FUNCTION int4(bigint) AS ASSIGNMENT;
```

(This cast is already predefined in the system.)

Compatibility

The `CREATE CAST` command conforms to the SQL standard, except that SQL does not make provisions for binary-coercible types or extra arguments to implementation functions. `AS IMPLICIT` is a PostgreSQL extension, too.

See Also

CREATE FUNCTION, CREATE TYPE, DROP CAST

CREATE COLLATION

Name

CREATE COLLATION — define a new collation

Synopsis

```
CREATE COLLATION name (
    [ LOCALE = locale, ]
    [ LC_COLLATE = lc_collate, ]
    [ LC_CTYPE = lc_ctype ]
)
CREATE COLLATION name FROM existing_collation
```

Description

CREATE COLLATION defines a new collation using the specified operating system locale settings, or by copying an existing collation.

To be able to create a collation, you must have CREATE privilege on the destination schema.

Parameters

name

The name of the collation. The collation name can be schema-qualified. If it is not, the collation is defined in the current schema. The collation name must be unique within that schema. (The system catalogs can contain collations with the same name for other encodings, but these are ignored if the database encoding does not match.)

locale

This is a shortcut for setting LC_COLLATE and LC_CTYPE at once. If you specify this, you cannot specify either of those parameters.

lc_collate

Use the specified operating system locale for the LC_COLLATE locale category. The locale must be applicable to the current database encoding. (See CREATE DATABASE for the precise rules.)

lc_ctype

Use the specified operating system locale for the LC_CTYPE locale category. The locale must be applicable to the current database encoding. (See CREATE DATABASE for the precise rules.)

existing_collation

The name of an existing collation to copy. The new collation will have the same properties as the existing one, but it will be an independent object.

Notes

Use `DROP COLLATION` to remove user-defined collations.

See Section 22.2 for more information about collation support in PostgreSQL.

Examples

To create a collation from the operating system locale `fr_FR.utf8` (assuming the current database encoding is `UTF8`):

```
CREATE COLLATION french (LOCALE = 'fr_FR.utf8');
```

To create a collation from an existing collation:

```
CREATE COLLATION german FROM "de_DE";
```

This can be convenient to be able to use operating-system-independent collation names in applications.

Compatibility

There is a `CREATE COLLATION` statement in the SQL standard, but it is limited to copying an existing collation. The syntax to create a new collation is a PostgreSQL extension.

See Also

ALTER COLLATION, DROP COLLATION

CREATE CONVERSION

Name

CREATE CONVERSION — define a new encoding conversion

Synopsis

```
CREATE [ DEFAULT ] CONVERSION name
    FOR source_encoding TO dest_encoding FROM function_name
```

Description

CREATE CONVERSION defines a new conversion between character set encodings. Also, conversions that are marked DEFAULT can be used for automatic encoding conversion between client and server. For this purpose, two conversions, from encoding A to B *and* from encoding B to A, must be defined.

To be able to create a conversion, you must have EXECUTE privilege on the function and CREATE privilege on the destination schema.

Parameters

DEFAULT

The DEFAULT clause indicates that this conversion is the default for this particular source to destination encoding. There should be only one default encoding in a schema for the encoding pair.

name

The name of the conversion. The conversion name can be schema-qualified. If it is not, the conversion is defined in the current schema. The conversion name must be unique within a schema.

source_encoding

The source encoding name.

dest_encoding

The destination encoding name.

function_name

The function used to perform the conversion. The function name can be schema-qualified. If it is not, the function will be looked up in the path.

The function must have the following signature:

```
conv_proc(
    integer,  -- source encoding ID
    integer,  -- destination encoding ID
    cstring,  -- source string (null terminated C string)
```

```
        internal, -- destination (fill with a null terminated C string)
        integer   -- source string length
    ) RETURNS void;
```

Notes

Use `DROP CONVERSION` to remove user-defined conversions.

The privileges required to create a conversion might be changed in a future release.

Examples

To create a conversion from encoding `UTF8` to `LATIN1` using `myfunc`:

```
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc;
```

Compatibility

`CREATE CONVERSION` is a PostgreSQL extension. There is no `CREATE CONVERSION` statement in the SQL standard, but a `CREATE TRANSLATION` statement that is very similar in purpose and syntax.

See Also

ALTER CONVERSION, CREATE FUNCTION, DROP CONVERSION

CREATE DATABASE

Name

CREATE DATABASE — create a new database

Synopsis

```
CREATE DATABASE name
    [ [ WITH ] [ OWNER [=] user_name ]
           [ TEMPLATE [=] template ]
           [ ENCODING [=] encoding ]
           [ LC_COLLATE [=] lc_collate ]
           [ LC_CTYPE [=] lc_ctype ]
           [ TABLESPACE [=] tablespace_name ]
           [ ALLOW_CONNECTIONS [=] allowconn ]
           [ CONNECTION LIMIT [=] connlimit ]
           [ IS_TEMPLATE [=] istemplate ] ]
```

Description

CREATE DATABASE creates a new PostgreSQL database.

To create a database, you must be a superuser or have the special CREATEDB privilege. See CREATE USER.

By default, the new database will be created by cloning the standard system database template1. A different template can be specified by writing TEMPLATE *name*. In particular, by writing TEMPLATE template0, you can create a virgin database containing only the standard objects predefined by your version of PostgreSQL. This is useful if you wish to avoid copying any installation-local objects that might have been added to template1.

Parameters

name

The name of a database to create.

user_name

The role name of the user who will own the new database, or DEFAULT to use the default (namely, the user executing the command). To create a database owned by another role, you must be a direct or indirect member of that role, or be a superuser.

template

The name of the template from which to create the new database, or DEFAULT to use the default template (template1).

encoding

Character set encoding to use in the new database. Specify a string constant (e.g., 'SQL_ASCII'), or an integer encoding number, or DEFAULT to use the default encoding (namely, the encoding of the template database). The character sets supported by the PostgreSQL server are described in Section 22.3.1. See below for additional restrictions.

lc_collate

Collation order (LC_COLLATE) to use in the new database. This affects the sort order applied to strings, e.g., in queries with ORDER BY, as well as the order used in indexes on text columns. The default is to use the collation order of the template database. See below for additional restrictions.

lc_ctype

Character classification (LC_CTYPE) to use in the new database. This affects the categorization of characters, e.g., lower, upper and digit. The default is to use the character classification of the template database. See below for additional restrictions.

tablespace_name

The name of the tablespace that will be associated with the new database, or DEFAULT to use the template database's tablespace. This tablespace will be the default tablespace used for objects created in this database. See CREATE TABLESPACE for more information.

allowconn

If false then no one can connect to this database. The default is true, allowing connections (except as restricted by other mechanisms, such as GRANT/REVOKE CONNECT).

connlimit

How many concurrent connections can be made to this database. -1 (the default) means no limit.

istemplate

If true, then this database can be cloned by any user with CREATEDB privileges; if false (the default), then only superusers or the owner of the database can clone it.

Optional parameters can be written in any order, not only the order illustrated above.

Notes

CREATE DATABASE cannot be executed inside a transaction block.

Errors along the line of "could not initialize database directory" are most likely related to insufficient permissions on the data directory, a full disk, or other file system problems.

Use DROP DATABASE to remove a database.

The program createdb is a wrapper program around this command, provided for convenience.

Database-level configuration parameters (set via ALTER DATABASE) and database-level permissions (set via GRANT) are not copied from the template database.

Although it is possible to copy a database other than template1 by specifying its name as the template, this is not (yet) intended as a general-purpose "COPY DATABASE" facility. The principal limitation is that no other sessions can be connected to the template database while it is being copied. CREATE DATABASE

will fail if any other connection exists when it starts; otherwise, new connections to the template database are locked out until CREATE DATABASE completes. See Section 21.3 for more information.

The character set encoding specified for the new database must be compatible with the chosen locale settings (LC_COLLATE and LC_CTYPE). If the locale is C (or equivalently POSIX), then all encodings are allowed, but for other locale settings there is only one encoding that will work properly. (On Windows, however, UTF-8 encoding can be used with any locale.) CREATE DATABASE will allow superusers to specify SQL_ASCII encoding regardless of the locale settings, but this choice is deprecated and may result in misbehavior of character-string functions if data that is not encoding-compatible with the locale is stored in the database.

The encoding and locale settings must match those of the template database, except when template0 is used as template. This is because other databases might contain data that does not match the specified encoding, or might contain indexes whose sort ordering is affected by LC_COLLATE and LC_CTYPE. Copying such data would result in a database that is corrupt according to the new settings. template0, however, is known to not contain any data or indexes that would be affected.

The CONNECTION LIMIT option is only enforced approximately; if two new sessions start at about the same time when just one connection "slot" remains for the database, it is possible that both will fail. Also, the limit is not enforced against superusers.

Examples

To create a new database:

```
CREATE DATABASE lusiadas;
```

To create a database sales owned by user salesapp with a default tablespace of salesspace:

```
CREATE DATABASE sales OWNER salesapp TABLESPACE salesspace;
```

To create a database music which supports the ISO-8859-1 character set:

```
CREATE DATABASE music ENCODING 'LATIN1' TEMPLATE template0;
```

In this example, the TEMPLATE template0 clause would only be required if template1's encoding is not ISO-8859-1. Note that changing encoding might require selecting new LC_COLLATE and LC_CTYPE settings as well.

Compatibility

There is no CREATE DATABASE statement in the SQL standard. Databases are equivalent to catalogs, whose creation is implementation-defined.

See Also

ALTER DATABASE, DROP DATABASE

CREATE DOMAIN

Name

CREATE DOMAIN — define a new domain

Synopsis

```
CREATE DOMAIN name [ AS ] data_type
    [ COLLATE collation ]
    [ DEFAULT expression ]
    [ constraint [ ... ] ]

where constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
```

Description

CREATE DOMAIN creates a new domain. A domain is essentially a data type with optional constraints (restrictions on the allowed set of values). The user who defines a domain becomes its owner.

If a schema name is given (for example, CREATE DOMAIN myschema.mydomain ...) then the domain is created in the specified schema. Otherwise it is created in the current schema. The domain name must be unique among the types and domains existing in its schema.

Domains are useful for abstracting common constraints on fields into a single location for maintenance. For example, several tables might contain email address columns, all requiring the same CHECK constraint to verify the address syntax. Define a domain rather than setting up each table's constraint individually.

To be able to create a domain, you must have USAGE privilege on the underlying type.

Parameters

name

The name (optionally schema-qualified) of a domain to be created.

data_type

The underlying data type of the domain. This can include array specifiers.

collation

An optional collation for the domain. If no collation is specified, the underlying data type's default collation is used. The underlying type must be collatable if COLLATE is specified.

`DEFAULT` *`expression`*

The `DEFAULT` clause specifies a default value for columns of the domain data type. The value is any variable-free expression (but subqueries are not allowed). The data type of the default expression must match the data type of the domain. If no default value is specified, then the default value is the null value.

The default expression will be used in any insert operation that does not specify a value for the column. If a default value is defined for a particular column, it overrides any default associated with the domain. In turn, the domain default overrides any default value associated with the underlying data type.

`CONSTRAINT` *`constraint_name`*

An optional name for a constraint. If not specified, the system generates a name.

`NOT NULL`

Values of this domain are prevented from being null (but see notes below).

`NULL`

Values of this domain are allowed to be null. This is the default.

This clause is only intended for compatibility with nonstandard SQL databases. Its use is discouraged in new applications.

`CHECK (`*`expression`*`)`

`CHECK` clauses specify integrity constraints or tests which values of the domain must satisfy. Each constraint must be an expression producing a Boolean result. It should use the key word `VALUE` to refer to the value being tested. Expressions evaluating to TRUE or UNKNOWN succeed. If the expression produces a FALSE result, an error is reported and the value is not allowed to be converted to the domain type.

Currently, `CHECK` expressions cannot contain subqueries nor refer to variables other than `VALUE`.

When a domain has multiple `CHECK` constraints, they will be tested in alphabetical order by name. (PostgreSQL versions before 9.5 did not honor any particular firing order for `CHECK` constraints.)

Notes

Domain constraints, particularly `NOT NULL`, are checked when converting a value to the domain type. It is possible for a column that is nominally of the domain type to read as null despite there being such a constraint. For example, this can happen in an outer-join query, if the domain column is on the nullable side of the outer join. A more subtle example is

```
INSERT INTO tab (domcol) VALUES ((SELECT domcol FROM tab WHERE false));
```

The empty scalar sub-SELECT will produce a null value that is considered to be of the domain type, so no further constraint checking is applied to it, and the insertion will succeed.

It is very difficult to avoid such problems, because of SQL's general assumption that a null value is a valid value of every data type. Best practice therefore is to design a domain's constraints so that a null value is allowed, and then to apply column `NOT NULL` constraints to columns of the domain type as needed, rather than directly to the domain type.

PostgreSQL assumes that `CHECK` constraints' conditions are immutable, that is, they will always give the same result for the same input value. This assumption is what justifies examining `CHECK` constraints only when a value is first converted to be of a domain type, and not at other times. (This is essentially the same as the treatment of table `CHECK` constraints, as described in Section 5.3.1.)

An example of a common way to break this assumption is to reference a user-defined function in a `CHECK` expression, and then change the behavior of that function. PostgreSQL does not disallow that, but it will not notice if there are stored values of the domain type that now violate the `CHECK` constraint. That would cause a subsequent database dump and reload to fail. The recommended way to handle such a change is to drop the constraint (using `ALTER DOMAIN`), adjust the function definition, and re-add the constraint, thereby rechecking it against stored data.

Examples

This example creates the `us_postal_code` data type and then uses the type in a table definition. A regular expression test is used to verify that the value looks like a valid US postal code:

```
CREATE DOMAIN us_postal_code AS TEXT
CHECK(
   VALUE ~ '^\d{5}$'
OR VALUE ~ '^\d{5}-\d{4}$'
);

CREATE TABLE us_snail_addy (
  address_id SERIAL PRIMARY KEY,
  street1 TEXT NOT NULL,
  street2 TEXT,
  street3 TEXT,
  city TEXT NOT NULL,
  postal us_postal_code NOT NULL
);
```

Compatibility

The command `CREATE DOMAIN` conforms to the SQL standard.

See Also

ALTER DOMAIN, DROP DOMAIN

CREATE EVENT TRIGGER

Name

CREATE EVENT TRIGGER — define a new event trigger

Synopsis

```
CREATE EVENT TRIGGER name
    ON event
    [ WHEN filter_variable IN (filter_value [, ... ]) [ AND ... ] ]
    EXECUTE PROCEDURE function_name()
```

Description

CREATE EVENT TRIGGER creates a new event trigger. Whenever the designated event occurs and the WHEN condition associated with the trigger, if any, is satisfied, the trigger function will be executed. For a general introduction to event triggers, see Chapter 37. The user who creates an event trigger becomes its owner.

Parameters

name

The name to give the new trigger. This name must be unique within the database.

event

The name of the event that triggers a call to the given function. See Section 37.1 for more information on event names.

filter_variable

The name of a variable used to filter events. This makes it possible to restrict the firing of the trigger to a subset of the cases in which it is supported. Currently the only supported *filter_variable* is TAG.

filter_value

A list of values for the associated *filter_variable* for which the trigger should fire. For TAG, this means a list of command tags (e.g., 'DROP FUNCTION').

function_name

A user-supplied function that is declared as taking no argument and returning type event_trigger.

Notes

Only superusers can create event triggers.

Event triggers are disabled in single-user mode (see postgres). If an erroneous event trigger disables the database so much that you can't even drop the trigger, restart in single-user mode and you'll be able to do that.

Examples

Forbid the execution of any DDL command:

```
CREATE OR REPLACE FUNCTION abort_any_command()
  RETURNS event_trigger
 LANGUAGE plpgsql
  AS $$
BEGIN
  RAISE EXCEPTION 'command % is disabled', tg_tag;
END;
$$;

CREATE EVENT TRIGGER abort_ddl ON ddl_command_start
   EXECUTE PROCEDURE abort_any_command();
```

Compatibility

There is no `CREATE EVENT TRIGGER` statement in the SQL standard.

See Also

ALTER EVENT TRIGGER, DROP EVENT TRIGGER, CREATE FUNCTION

CREATE EXTENSION

Name

CREATE EXTENSION — install an extension

Synopsis

```
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
    [ WITH ] [ SCHEMA schema_name ]
             [ VERSION version ]
             [ FROM old_version ]
```

Description

CREATE EXTENSION loads a new extension into the current database. There must not be an extension of the same name already loaded.

Loading an extension essentially amounts to running the extension's script file. The script will typically create new SQL objects such as functions, data types, operators and index support methods. CREATE EXTENSION additionally records the identities of all the created objects, so that they can be dropped again if DROP EXTENSION is issued.

Loading an extension requires the same privileges that would be required to create its component objects. For most extensions this means superuser or database owner privileges are needed. The user who runs CREATE EXTENSION becomes the owner of the extension for purposes of later privilege checks, as well as the owner of any objects created by the extension's script.

Parameters

IF NOT EXISTS

Do not throw an error if an extension with the same name already exists. A notice is issued in this case. Note that there is no guarantee that the existing extension is anything like the one that would have been created from the currently-available script file.

extension_name

The name of the extension to be installed. PostgreSQL will create the extension using details from the file SHAREDIR/extension/*extension_name*.control.

schema_name

The name of the schema in which to install the extension's objects, given that the extension allows its contents to be relocated. The named schema must already exist. If not specified, and the extension's control file does not specify a schema either, the current default object creation schema is used.

Remember that the extension itself is not considered to be within any schema: extensions have unqualified names that must be unique database-wide. But objects belonging to the extension can be within schemas.

version

The version of the extension to install. This can be written as either an identifier or a string literal. The default version is whatever is specified in the extension's control file.

old_version

FROM *old_version* must be specified when, and only when, you are attempting to install an extension that replaces an "old style" module that is just a collection of objects not packaged into an extension. This option causes CREATE EXTENSION to run an alternative installation script that absorbs the existing objects into the extension, instead of creating new objects. Be careful that SCHEMA specifies the schema containing these pre-existing objects.

The value to use for *old_version* is determined by the extension's author, and might vary if there is more than one version of the old-style module that can be upgraded into an extension. For the standard additional modules supplied with pre-9.1 PostgreSQL, use unpackaged for *old_version* when updating a module to extension style.

Notes

Before you can use CREATE EXTENSION to load an extension into a database, the extension's supporting files must be installed. Information about installing the extensions supplied with PostgreSQL can be found in Additional Supplied Modules.

The extensions currently available for loading can be identified from the pg_available_extensions or pg_available_extension_versions system views.

Caution

Installing an extension as superuser requires trusting that the extension's author wrote the extension installation script in a secure fashion. It is not terribly difficult for a malicious user to create trojan-horse objects that will compromise later execution of a carelessly-written extension script, allowing that user to acquire superuser privileges. However, trojan-horse objects are only hazardous if they are in the search_path during script execution, meaning that they are in the extension's installation target schema or in the schema of some extension it depends on. Therefore, a good rule of thumb when dealing with extensions whose scripts have not been carefully vetted is to install them only into schemas for which CREATE privilege has not been and will not be granted to any untrusted users. Likewise for any extensions they depend on.

The extensions supplied with PostgreSQL are believed to be secure against installation-time attacks of this sort, except for a few that depend on other extensions. As stated in the documentation for those extensions, they should be installed into secure schemas, or installed into the same schemas as the extensions they depend on, or both.

For information about writing new extensions, see Section 35.15.

Examples

Install the hstore extension into the current database, placing its objects in schema `addons`:

```
CREATE EXTENSION hstore SCHEMA addons;
```

Another way to accomplish the same thing:

```
SET search_path = addons;
CREATE EXTENSION hstore;
```

Update a pre-9.1 installation of `hstore` into extension style:

```
CREATE EXTENSION hstore SCHEMA public FROM unpackaged;
```

Be careful to specify the schema in which you installed the existing `hstore` objects.

Compatibility

`CREATE EXTENSION` is a PostgreSQL extension.

See Also

ALTER EXTENSION, DROP EXTENSION

CREATE FOREIGN DATA WRAPPER

Name

CREATE FOREIGN DATA WRAPPER — define a new foreign-data wrapper

Synopsis

```
CREATE FOREIGN DATA WRAPPER name
    [ HANDLER handler_function | NO HANDLER ]
    [ VALIDATOR validator_function | NO VALIDATOR ]
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

Description

CREATE FOREIGN DATA WRAPPER creates a new foreign-data wrapper. The user who defines a foreign-data wrapper becomes its owner.

The foreign-data wrapper name must be unique within the database.

Only superusers can create foreign-data wrappers.

Parameters

name

The name of the foreign-data wrapper to be created.

HANDLER *handler_function*

handler_function is the name of a previously registered function that will be called to retrieve the execution functions for foreign tables. The handler function must take no arguments, and its return type must be fdw_handler.

It is possible to create a foreign-data wrapper with no handler function, but foreign tables using such a wrapper can only be declared, not accessed.

VALIDATOR *validator_function*

validator_function is the name of a previously registered function that will be called to check the generic options given to the foreign-data wrapper, as well as options for foreign servers, user mappings and foreign tables using the foreign-data wrapper. If no validator function or NO VALIDATOR is specified, then options will not be checked at creation time. (Foreign-data wrappers will possibly ignore or reject invalid option specifications at run time, depending on the implementation.) The validator function must take two arguments: one of type text[], which will contain the array of options as stored in the system catalogs, and one of type oid, which will be the OID of the system catalog containing the options. The return type is ignored; the function should report invalid options using the ereport(ERROR) function.

OPTIONS (*option* '*value*' [, ...])

This clause specifies options for the new foreign-data wrapper. The allowed option names and values are specific to each foreign data wrapper and are validated using the foreign-data wrapper's validator function. Option names must be unique.

Notes

PostgreSQL's foreign-data functionality is still under active development. Optimization of queries is primitive (and mostly left to the wrapper, too). Thus, there is considerable room for future performance improvements.

Examples

Create a useless foreign-data wrapper `dummy`:

```
CREATE FOREIGN DATA WRAPPER dummy;
```

Create a foreign-data wrapper `file` with handler function `file_fdw_handler`:

```
CREATE FOREIGN DATA WRAPPER file HANDLER file_fdw_handler;
```

Create a foreign-data wrapper `mywrapper` with some options:

```
CREATE FOREIGN DATA WRAPPER mywrapper
    OPTIONS (debug 'true');
```

Compatibility

`CREATE FOREIGN DATA WRAPPER` conforms to ISO/IEC 9075-9 (SQL/MED), with the exception that the `HANDLER` and `VALIDATOR` clauses are extensions and the standard clauses `LIBRARY` and `LANGUAGE` are not implemented in PostgreSQL.

Note, however, that the SQL/MED functionality as a whole is not yet conforming.

See Also

ALTER FOREIGN DATA WRAPPER, DROP FOREIGN DATA WRAPPER, CREATE SERVER, CREATE USER MAPPING, CREATE FOREIGN TABLE

CREATE FOREIGN TABLE

Name

CREATE FOREIGN TABLE — define a new foreign table

Synopsis

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name ( [
  { column_name data_type [ OPTIONS ( option 'value' [, ... ] ) ] [ COLLATE collation ] [ columr
    | table_constraint }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
  SERVER server_name
[ OPTIONS ( option 'value' [, ... ] ) ]

where column_constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr }

and table_constraint is:

[ CONSTRAINT constraint_name ]
CHECK ( expression ) [ NO INHERIT ]
```

Description

CREATE FOREIGN TABLE creates a new foreign table in the current database. The table will be owned by the user issuing the command.

If a schema name is given (for example, CREATE FOREIGN TABLE myschema.mytable ...) then the table is created in the specified schema. Otherwise it is created in the current schema. The name of the foreign table must be distinct from the name of any other foreign table, table, sequence, index, view, or materialized view in the same schema.

CREATE FOREIGN TABLE also automatically creates a data type that represents the composite type corresponding to one row of the foreign table. Therefore, foreign tables cannot have the same name as any existing data type in the same schema.

To be able to create a foreign table, you must have USAGE privilege on the foreign server, as well as USAGE privilege on all column types used in the table.

Parameters

`IF NOT EXISTS`

Do not throw an error if a relation with the same name already exists. A notice is issued in this case. Note that there is no guarantee that the existing relation is anything like the one that would have been created.

table_name

The name (optionally schema-qualified) of the table to be created.

column_name

The name of a column to be created in the new table.

data_type

The data type of the column. This can include array specifiers. For more information on the data types supported by PostgreSQL, refer to Chapter 8.

`COLLATE` *collation*

The `COLLATE` clause assigns a collation to the column (which must be of a collatable data type). If not specified, the column data type's default collation is used.

`INHERITS ( parent_table [, ... ] )`

The optional `INHERITS` clause specifies a list of tables from which the new foreign table automatically inherits all columns. Parent tables can be plain tables or foreign tables. See the similar form of CREATE TABLE for more details.

`CONSTRAINT` *constraint_name*

An optional name for a column or table constraint. If the constraint is violated, the constraint name is present in error messages, so constraint names like `col must be positive` can be used to communicate helpful constraint information to client applications. (Double-quotes are needed to specify constraint names that contain spaces.) If a constraint name is not specified, the system generates a name.

`NOT NULL`

The column is not allowed to contain null values.

`NULL`

The column is allowed to contain null values. This is the default.

This clause is only provided for compatibility with non-standard SQL databases. Its use is discouraged in new applications.

`CHECK ( expression ) [ NO INHERIT ]`

The `CHECK` clause specifies an expression producing a Boolean result which each row in the foreign table is expected to satisfy; that is, the expression should produce TRUE or UNKNOWN, never FALSE, for all rows in the foreign table. A check constraint specified as a column constraint should reference that column's value only, while an expression appearing in a table constraint can reference multiple columns.

Currently, `CHECK` expressions cannot contain subqueries nor refer to variables other than columns of the current row. The system column `tableoid` may be referenced, but not any other system column.

A constraint marked with `NO INHERIT` will not propagate to child tables.

`DEFAULT` *default_expr*

The `DEFAULT` clause assigns a default data value for the column whose column definition it appears within. The value is any variable-free expression (subqueries and cross-references to other columns in the current table are not allowed). The data type of the default expression must match the data type of the column.

The default expression will be used in any insert operation that does not specify a value for the column. If there is no default for a column, then the default is null.

server_name

The name of an existing foreign server to use for the foreign table. For details on defining a server, see CREATE SERVER.

`OPTIONS ( option 'value' [, ...] )`

Options to be associated with the new foreign table or one of its columns. The allowed option names and values are specific to each foreign data wrapper and are validated using the foreign-data wrapper's validator function. Duplicate option names are not allowed (although it's OK for a table option and a column option to have the same name).

Notes

Constraints on foreign tables (such as `CHECK` or `NOT NULL` clauses) are not enforced by the core PostgreSQL system, and most foreign data wrappers do not attempt to enforce them either; that is, the constraint is simply assumed to hold true. There would be little point in such enforcement since it would only apply to rows inserted or updated via the foreign table, and not to rows modified by other means, such as directly on the remote server. Instead, a constraint attached to a foreign table should represent a constraint that is being enforced by the remote server.

Some special-purpose foreign data wrappers might be the only access mechanism for the data they access, and in that case it might be appropriate for the foreign data wrapper itself to perform constraint enforcement. But you should not assume that a wrapper does that unless its documentation says so.

Although PostgreSQL does not attempt to enforce constraints on foreign tables, it does assume that they are correct for purposes of query optimization. If there are rows visible in the foreign table that do not satisfy a declared constraint, queries on the table might produce incorrect answers. It is the user's responsibility to ensure that the constraint definition matches reality.

Examples

Create foreign table `films`, which will be accessed through the server `film_server`:

```
CREATE FOREIGN TABLE films (
    code        char(5) NOT NULL,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
```

```
    len         interval hour to minute
)
SERVER film_server;
```

Compatibility

The `CREATE FOREIGN TABLE` command largely conforms to the SQL standard; however, much as with `CREATE TABLE`, `NULL` constraints and zero-column foreign tables are permitted. The ability to specify column default values is also a PostgreSQL extension. Table inheritance, in the form defined by PostgreSQL, is nonstandard.

See Also

ALTER FOREIGN TABLE, DROP FOREIGN TABLE, CREATE TABLE, CREATE SERVER, IMPORT FOREIGN SCHEMA

CREATE FUNCTION

Name

CREATE FUNCTION — define a new function

Synopsis

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | WINDOW
    | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | COST execution_cost
    | ROWS result_rows
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
    [ WITH ( attribute [, ...] ) ]
```

Description

CREATE FUNCTION defines a new function. CREATE OR REPLACE FUNCTION will either create a new function, or replace an existing definition. To be able to define a function, the user must have the USAGE privilege on the language.

If a schema name is included, then the function is created in the specified schema. Otherwise it is created in the current schema. The name of the new function must not match any existing function with the same input argument types in the same schema. However, functions of different argument types can share a name (this is called *overloading*).

To replace the current definition of an existing function, use CREATE OR REPLACE FUNCTION. It is not possible to change the name or argument types of a function this way (if you tried, you would actually be creating a new, distinct function). Also, CREATE OR REPLACE FUNCTION will not let you change the return type of an existing function. To do that, you must drop and recreate the function. (When using OUT parameters, that means you cannot change the types of any OUT parameters except by dropping the function.)

When CREATE OR REPLACE FUNCTION is used to replace an existing function, the ownership and permissions of the function do not change. All other function properties are assigned the values specified or implied in the command. You must own the function to replace it (this includes being a member of the owning role).

If you drop and then recreate a function, the new function is not the same entity as the old; you will have to drop existing rules, views, triggers, etc. that refer to the old function. Use `CREATE OR REPLACE FUNCTION` to change a function definition without breaking objects that refer to the function. Also, `ALTER FUNCTION` can be used to change most of the auxiliary properties of an existing function.

The user that creates the function becomes the owner of the function.

To be able to create a function, you must have `USAGE` privilege on the argument types and the return type.

Parameters

name

The name (optionally schema-qualified) of the function to create.

argmode

The mode of an argument: `IN`, `OUT`, `INOUT`, or `VARIADIC`. If omitted, the default is `IN`. Only `OUT` arguments can follow a `VARIADIC` one. Also, `OUT` and `INOUT` arguments cannot be used together with the `RETURNS TABLE` notation.

argname

The name of an argument. Some languages (including SQL and PL/pgSQL) let you use the name in the function body. For other languages the name of an input argument is just extra documentation, so far as the function itself is concerned; but you can use input argument names when calling a function to improve readability (see Section 4.3). In any case, the name of an output argument is significant, because it defines the column name in the result row type. (If you omit the name for an output argument, the system will choose a default column name.)

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any. The argument types can be base, composite, or domain types, or can reference the type of a table column.

Depending on the implementation language it might also be allowed to specify "pseudotypes" such as `cstring`. Pseudotypes indicate that the actual argument type is either incompletely specified, or outside the set of ordinary SQL data types.

The type of a column is referenced by writing *table_name*`.`*column_name*`%TYPE`. Using this feature can sometimes help make a function independent of changes to the definition of a table.

default_expr

An expression to be used as default value if the parameter is not specified. The expression has to be coercible to the argument type of the parameter. Only input (including `INOUT`) parameters can have a default value. All input parameters following a parameter with a default value must have default values as well.

rettype

The return data type (optionally schema-qualified). The return type can be a base, composite, or domain type, or can reference the type of a table column. Depending on the implementation language it might also be allowed to specify "pseudotypes" such as `cstring`. If the function is not supposed to return a value, specify `void` as the return type.

When there are `OUT` or `INOUT` parameters, the `RETURNS` clause can be omitted. If present, it must agree with the result type implied by the output parameters: `RECORD` if there are multiple output parameters, or the same type as the single output parameter.

The `SETOF` modifier indicates that the function will return a set of items, rather than a single item.

The type of a column is referenced by writing *table_name*.*column_name*`%TYPE`.

column_name

The name of an output column in the `RETURNS TABLE` syntax. This is effectively another way of declaring a named `OUT` parameter, except that `RETURNS TABLE` also implies `RETURNS SETOF`.

column_type

The data type of an output column in the `RETURNS TABLE` syntax.

lang_name

The name of the language that the function is implemented in. It can be `sql`, `c`, `internal`, or the name of a user-defined procedural language, e.g., `plpgsql`. Enclosing the name in single quotes is deprecated and requires matching case.

`TRANSFORM { FOR TYPE` *type_name* `} [, ... ] }`

Lists which transforms a call to the function should apply. Transforms convert between SQL types and language-specific data types; see CREATE TRANSFORM. Procedural language implementations usually have hardcoded knowledge of the built-in types, so those don't need to be listed here. If a procedural language implementation does not know how to handle a type and no transform is supplied, it will fall back to a default behavior for converting data types, but this depends on the implementation.

`WINDOW`

`WINDOW` indicates that the function is a *window function* rather than a plain function. This is currently only useful for functions written in C. The `WINDOW` attribute cannot be changed when replacing an existing function definition.

`IMMUTABLE`
`STABLE`
`VOLATILE`

These attributes inform the query optimizer about the behavior of the function. At most one choice can be specified. If none of these appear, `VOLATILE` is the default assumption.

`IMMUTABLE` indicates that the function cannot modify the database and always returns the same result when given the same argument values; that is, it does not do database lookups or otherwise use information not directly present in its argument list. If this option is given, any call of the function with all-constant arguments can be immediately replaced with the function value.

`STABLE` indicates that the function cannot modify the database, and that within a single table scan it will consistently return the same result for the same argument values, but that its result could change across SQL statements. This is the appropriate selection for functions whose results depend on database lookups, parameter variables (such as the current time zone), etc. (It is inappropriate for `AFTER` triggers that wish to query rows modified by the current command.) Also note that the `current_timestamp` family of functions qualify as stable, since their values do not change within a transaction.

`VOLATILE` indicates that the function value can change even within a single table scan, so no optimizations can be made. Relatively few database functions are volatile in this sense; some examples are `random()`, `currval()`, `timeofday()`. But note that any function that has side-effects must be classified volatile, even if its result is quite predictable, to prevent calls from being optimized away; an example is `setval()`.

For additional details see Section 35.6.

`LEAKPROOF`

`LEAKPROOF` indicates that the function has no side effects. It reveals no information about its arguments other than by its return value. For example, a function which throws an error message for some argument values but not others, or which includes the argument values in any error message, is not leakproof. This affects how the system executes queries against views created with the `security_barrier` option or tables with row level security enabled. The system will enforce conditions from security policies and security barrier views before any user-supplied conditions from the query itself that contain non-leakproof functions, in order to prevent the inadvertent exposure of data. Functions and operators marked as leakproof are assumed to be trustworthy, and may be executed before conditions from security policies and security barrier views. In addition, functions which do not take arguments or which are not passed any arguments from the security barrier view or table do not have to be marked as leakproof to be executed before security conditions. See CREATE VIEW and Section 38.5. This option can only be set by the superuser.

`CALLED ON NULL INPUT`
`RETURNS NULL ON NULL INPUT`
`STRICT`

`CALLED ON NULL INPUT` (the default) indicates that the function will be called normally when some of its arguments are null. It is then the function author's responsibility to check for null values if necessary and respond appropriately.

`RETURNS NULL ON NULL INPUT` or `STRICT` indicates that the function always returns null whenever any of its arguments are null. If this parameter is specified, the function is not executed when there are null arguments; instead a null result is assumed automatically.

`[EXTERNAL] SECURITY INVOKER`
`[EXTERNAL] SECURITY DEFINER`

`SECURITY INVOKER` indicates that the function is to be executed with the privileges of the user that calls it. That is the default. `SECURITY DEFINER` specifies that the function is to be executed with the privileges of the user that created it.

The key word `EXTERNAL` is allowed for SQL conformance, but it is optional since, unlike in SQL, this feature applies to all functions not only external ones.

`execution_cost`

A positive number giving the estimated execution cost for the function, in units of cpu_operator_cost. If the function returns a set, this is the cost per returned row. If the cost is not specified, 1 unit is assumed for C-language and internal functions, and 100 units for functions in all other languages. Larger values cause the planner to try to avoid evaluating the function more often than necessary.

`result_rows`

A positive number giving the estimated number of rows that the planner should expect the function to return. This is only allowed when the function is declared to return a set. The default assumption

is 1000 rows.

configuration_parameter
value

The SET clause causes the specified configuration parameter to be set to the specified value when the function is entered, and then restored to its prior value when the function exits. SET FROM CURRENT saves the value of the parameter that is current when CREATE FUNCTION is executed as the value to be applied when the function is entered.

If a SET clause is attached to a function, then the effects of a SET LOCAL command executed inside the function for the same variable are restricted to the function: the configuration parameter's prior value is still restored at function exit. However, an ordinary SET command (without LOCAL) overrides the SET clause, much as it would do for a previous SET LOCAL command: the effects of such a command will persist after function exit, unless the current transaction is rolled back.

See SET and Chapter 18 for more information about allowed parameter names and values.

definition

A string constant defining the function; the meaning depends on the language. It can be an internal function name, the path to an object file, an SQL command, or text in a procedural language.

It is often helpful to use dollar quoting (see Section 4.1.2.4) to write the function definition string, rather than the normal single quote syntax. Without dollar quoting, any single quotes or backslashes in the function definition must be escaped by doubling them.

obj_file, *link_symbol*

This form of the AS clause is used for dynamically loadable C language functions when the function name in the C language source code is not the same as the name of the SQL function. The string *obj_file* is the name of the file containing the dynamically loadable object, and *link_symbol* is the function's link symbol, that is, the name of the function in the C language source code. If the link symbol is omitted, it is assumed to be the same as the name of the SQL function being defined. The C names of all functions must be different, so you must give overloaded C functions different C names (for example, use the argument types as part of the C names).

When repeated CREATE FUNCTION calls refer to the same object file, the file is only loaded once per session. To unload and reload the file (perhaps during development), start a new session.

attribute

The historical way to specify optional pieces of information about the function. The following attributes can appear here:

isStrict

Equivalent to STRICT or RETURNS NULL ON NULL INPUT.

isCachable

isCachable is an obsolete equivalent of IMMUTABLE; it's still accepted for backwards-compatibility reasons.

Attribute names are not case-sensitive.

Refer to Section 35.3 for further information on writing functions.

Overloading

PostgreSQL allows function *overloading*; that is, the same name can be used for several different functions so long as they have distinct input argument types. Whether or not you use it, this capability entails security precautions when calling functions in databases where some users mistrust other users; see Section 10.3.

Two functions are considered the same if they have the same names and *input* argument types, ignoring any `OUT` parameters. Thus for example these declarations conflict:

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, out text) ...
```

Functions that have different argument type lists will not be considered to conflict at creation time, but if defaults are provided they might conflict in use. For example, consider

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, int default 42) ...
```

A call `foo(10)` will fail due to the ambiguity about which function should be called.

Notes

The full SQL type syntax is allowed for declaring a function's arguments and return value. However, parenthesized type modifiers (e.g., the precision field for type `numeric`) are discarded by `CREATE FUNCTION`. Thus for example `CREATE FUNCTION foo (varchar(10)) ...` is exactly the same as `CREATE FUNCTION foo (varchar) ...`.

When replacing an existing function with `CREATE OR REPLACE FUNCTION`, there are restrictions on changing parameter names. You cannot change the name already assigned to any input parameter (although you can add names to parameters that had none before). If there is more than one output parameter, you cannot change the names of the output parameters, because that would change the column names of the anonymous composite type that describes the function's result. These restrictions are made to ensure that existing calls of the function do not stop working when it is replaced.

If a function is declared `STRICT` with a `VARIADIC` argument, the strictness check tests that the variadic array *as a whole* is non-null. The function will still be called if the array has null elements.

Examples

Here are some trivial examples to help you get started. For more information and examples, see Section 35.3.

```
CREATE FUNCTION add(integer, integer) RETURNS integer
    AS 'select $1 + $2;'
    LANGUAGE SQL
    IMMUTABLE
    RETURNS NULL ON NULL INPUT;
```

Increment an integer, making use of an argument name, in PL/pgSQL:

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$
        BEGIN
                RETURN i + 1;
        END;
$$ LANGUAGE plpgsql;
```

Return a record containing multiple output parameters:

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

You can do the same thing more verbosely with an explicitly named composite type:

```
CREATE TYPE dup_result AS (f1 int, f2 text);

CREATE FUNCTION dup(int) RETURNS dup_result
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

Another way to return multiple columns is to use a `TABLE` function:

```
CREATE FUNCTION dup(int) RETURNS TABLE(f1 int, f2 text)
    AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
    LANGUAGE SQL;

SELECT * FROM dup(42);
```

However, a `TABLE` function is different from the preceding examples, because it actually returns a *set* of records, not just one record.

Writing `SECURITY DEFINER` Functions Safely

Because a `SECURITY DEFINER` function is executed with the privileges of the user that created it, care is needed to ensure that the function cannot be misused. For security, search_path should be set to exclude any schemas writable by untrusted users. This prevents malicious users from creating objects that mask objects used by the function. Particularly important in this regard is the temporary-table schema, which is searched first by default, and is normally writable by anyone. A secure arrangement can be had by forcing the temporary schema to be searched last. To do this, write `pg_temp` as the last entry in `search_path`. This function illustrates safe usage:

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN AS $$
DECLARE passed BOOLEAN;
```

```
BEGIN
        SELECT  (pwd = $2) INTO passed
        FROM    pwds
        WHERE   username = $1;

        RETURN passed;
END;
$$  LANGUAGE plpgsql
    SECURITY DEFINER
    -- Set a secure search_path: trusted schema(s), then 'pg_temp'.
    SET search_path = admin, pg_temp;
```

Before PostgreSQL version 8.3, the `SET` option was not available, and so older functions may contain rather complicated logic to save, set, and restore `search_path`. The `SET` option is far easier to use for this purpose.

Another point to keep in mind is that by default, execute privilege is granted to `PUBLIC` for newly created functions (see GRANT for more information). Frequently you will wish to restrict use of a security definer function to only some users. To do that, you must revoke the default `PUBLIC` privileges and then grant execute privilege selectively. To avoid having a window where the new function is accessible to all, create it and set the privileges within a single transaction. For example:

```
BEGIN;
CREATE FUNCTION check_password(uname TEXT, pass TEXT) ... SECURITY DEFINER;
REVOKE ALL ON FUNCTION check_password(uname TEXT, pass TEXT) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION check_password(uname TEXT, pass TEXT) TO admins;
COMMIT;
```

Compatibility

A `CREATE FUNCTION` command is defined in SQL:1999 and later. The PostgreSQL version is similar but not fully compatible. The attributes are not portable, neither are the different available languages.

For compatibility with some other database systems, *`argmode`* can be written either before or after *`argname`*. But only the first way is standard-compliant.

For parameter defaults, the SQL standard specifies only the syntax with the `DEFAULT` key word. The syntax with = is used in T-SQL and Firebird.

See Also

ALTER FUNCTION, DROP FUNCTION, GRANT, LOAD, REVOKE, createlang

CREATE GROUP

Name

CREATE GROUP — define a new database role

Synopsis

```
CREATE GROUP name [ [ WITH ] option [ ... ] ]

where option can be:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
```

Description

CREATE GROUP is now an alias for CREATE ROLE.

Compatibility

There is no CREATE GROUP statement in the SQL standard.

See Also

CREATE ROLE

CREATE INDEX

Name

CREATE INDEX — define a new index

Synopsis

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON table_name [ USING met
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS
    [ WITH ( storage_parameter [= value] [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

Description

CREATE INDEX constructs an index on the specified column(s) of the specified relation, which can be a table or a materialized view. Indexes are primarily used to enhance database performance (though inappropriate use can result in slower performance).

The key field(s) for the index are specified as column names, or alternatively as expressions written in parentheses. Multiple fields can be specified if the index method supports multicolumn indexes.

An index field can be an expression computed from the values of one or more columns of the table row. This feature can be used to obtain fast access to data based on some transformation of the basic data. For example, an index computed on upper(col) would allow the clause WHERE upper(col) = 'JIM' to use an index.

PostgreSQL provides the index methods B-tree, hash, GiST, SP-GiST, GIN, and BRIN. Users can also define their own index methods, but that is fairly complicated.

When the WHERE clause is present, a *partial index* is created. A partial index is an index that contains entries for only a portion of a table, usually a portion that is more useful for indexing than the rest of the table. For example, if you have a table that contains both billed and unbilled orders where the unbilled orders take up a small fraction of the total table and yet that is an often used section, you can improve performance by creating an index on just that portion. Another possible application is to use WHERE with UNIQUE to enforce uniqueness over a subset of a table. See Section 11.8 for more discussion.

The expression used in the WHERE clause can refer only to columns of the underlying table, but it can use all columns, not just the ones being indexed. Presently, subqueries and aggregate expressions are also forbidden in WHERE. The same restrictions apply to index fields that are expressions.

All functions and operators used in an index definition must be "immutable", that is, their results must depend only on their arguments and never on any outside influence (such as the contents of another table or the current time). This restriction ensures that the behavior of the index is well-defined. To use a user-defined function in an index expression or WHERE clause, remember to mark the function immutable when you create it.

Parameters

`UNIQUE`

Causes the system to check for duplicate values in the table when the index is created (if data already exist) and each time data is added. Attempts to insert or update data which would result in duplicate entries will generate an error.

`CONCURRENTLY`

When this option is used, PostgreSQL will build the index without taking any locks that prevent concurrent inserts, updates, or deletes on the table; whereas a standard index build locks out writes (but not reads) on the table until it's done. There are several caveats to be aware of when using this option — see *Building Indexes Concurrently*.

For temporary tables, `CREATE INDEX` is always non-concurrent, as no other session can access them, and non-concurrent index creation is cheaper.

`IF NOT EXISTS`

Do not throw an error if a relation with the same name already exists. A notice is issued in this case. Note that there is no guarantee that the existing index is anything like the one that would have been created. Index name is required when `IF NOT EXISTS` is specified.

`name`

The name of the index to be created. No schema name can be included here; the index is always created in the same schema as its parent table. If the name is omitted, PostgreSQL chooses a suitable name based on the parent table's name and the indexed column name(s).

`table_name`

The name (possibly schema-qualified) of the table to be indexed.

`method`

The name of the index method to be used. Choices are `btree`, `hash`, `gist`, `spgist`, `gin`, and `brin`. The default method is `btree`.

`column_name`

The name of a column of the table.

`expression`

An expression based on one or more columns of the table. The expression usually must be written with surrounding parentheses, as shown in the syntax. However, the parentheses can be omitted if the expression has the form of a function call.

`collation`

The name of the collation to use for the index. By default, the index uses the collation declared for the column to be indexed or the result collation of the expression to be indexed. Indexes with non-default collations can be useful for queries that involve expressions using non-default collations.

`opclass`

The name of an operator class. See below for details.

`ASC`

Specifies ascending sort order (which is the default).

`DESC`

Specifies descending sort order.

`NULLS FIRST`

Specifies that nulls sort before non-nulls. This is the default when `DESC` is specified.

`NULLS LAST`

Specifies that nulls sort after non-nulls. This is the default when `DESC` is not specified.

`storage_parameter`

The name of an index-method-specific storage parameter. See *Index Storage Parameters* for details.

`tablespace_name`

The tablespace in which to create the index. If not specified, default_tablespace is consulted, or temp_tablespaces for indexes on temporary tables.

`predicate`

The constraint expression for a partial index.

Index Storage Parameters

The optional `WITH` clause specifies *storage parameters* for the index. Each index method has its own set of allowed storage parameters. The B-tree, hash, GiST and SP-GiST index methods all accept this parameter:

`fillfactor`

The fillfactor for an index is a percentage that determines how full the index method will try to pack index pages. For B-trees, leaf pages are filled to this percentage during initial index build, and also when extending the index at the right (adding new largest key values). If pages subsequently become completely full, they will be split, leading to gradual degradation in the index's efficiency. B-trees use a default fillfactor of 90, but any integer value from 10 to 100 can be selected. If the table is static then fillfactor 100 is best to minimize the index's physical size, but for heavily updated tables a smaller fillfactor is better to minimize the need for page splits. The other index methods use fillfactor in different but roughly analogous ways; the default fillfactor varies between methods.

GiST indexes additionally accept this parameter:

`buffering`

Determines whether the buffering build technique described in Section 59.4.1 is used to build the index. With `OFF` it is disabled, with `ON` it is enabled, and with `AUTO` it is initially disabled, but turned on on-the-fly once the index size reaches effective_cache_size. The default is `AUTO`.

GIN indexes accept different parameters:

`fastupdate`

This setting controls usage of the fast update technique described in Section 61.4.1. It is a Boolean parameter: `ON` enables fast update, `OFF` disables it. (Alternative spellings of `ON` and `OFF` are allowed as described in Section 18.1.) The default is `ON`.

> **Note:** Turning `fastupdate` off via `ALTER INDEX` prevents future insertions from going into the list of pending index entries, but does not in itself flush previous entries. You might want to `VACUUM` the table afterward to ensure the pending list is emptied.

`gin_pending_list_limit`

Custom gin_pending_list_limit parameter. This value is specified in kilobytes.

BRIN indexes accept a different parameter:

`pages_per_range`

Defines the number of table blocks that make up one block range for each entry of a BRIN index (see Section 62.1 for more details). The default is `128`.

Building Indexes Concurrently

Creating an index can interfere with regular operation of a database. Normally PostgreSQL locks the table to be indexed against writes and performs the entire index build with a single scan of the table. Other transactions can still read the table, but if they try to insert, update, or delete rows in the table they will block until the index build is finished. This could have a severe effect if the system is a live production database. Very large tables can take many hours to be indexed, and even for smaller tables, an index build can lock out writers for periods that are unacceptably long for a production system.

PostgreSQL supports building indexes without locking out writes. This method is invoked by specifying the `CONCURRENTLY` option of `CREATE INDEX`. When this option is used, PostgreSQL must perform two scans of the table, and in addition it must wait for all existing transactions that could potentially modify or use the index to terminate. Thus this method requires more total work than a standard index build and takes significantly longer to complete. However, since it allows normal operations to continue while the index is built, this method is useful for adding new indexes in a production environment. Of course, the extra CPU and I/O load imposed by the index creation might slow other operations.

In a concurrent index build, the index is actually entered into the system catalogs in one transaction, then two table scans occur in two more transactions. Before each table scan, the index build must wait for existing transactions that have modified the table to terminate. After the second scan, the index build must wait for any transactions that have a snapshot (see Chapter 13) predating the second scan to terminate. Then finally the index can be marked ready for use, and the `CREATE INDEX` command terminates. Even then, however, the index may not be immediately usable for queries: in the worst case, it cannot be used as long as transactions exist that predate the start of the index build.

If a problem arises while scanning the table, such as a deadlock or a uniqueness violation in a unique index, the `CREATE INDEX` command will fail but leave behind an “invalid” index. This index will be ignored for

querying purposes because it might be incomplete; however it will still consume update overhead. The psql `\d` command will report such an index as `INVALID`:

```
postgres=# \d tab
       Table "public.tab"
 Column |  Type   | Modifiers
--------+---------+-----------
 col    | integer |
Indexes:
    "idx" btree (col) INVALID
```

The recommended recovery method in such cases is to drop the index and try again to perform `CREATE INDEX CONCURRENTLY`. (Another possibility is to rebuild the index with `REINDEX`. However, since `REINDEX` does not support concurrent builds, this option is unlikely to seem attractive.)

Another caveat when building a unique index concurrently is that the uniqueness constraint is already being enforced against other transactions when the second table scan begins. This means that constraint violations could be reported in other queries prior to the index becoming available for use, or even in cases where the index build eventually fails. Also, if a failure does occur in the second scan, the "invalid" index continues to enforce its uniqueness constraint afterwards.

Concurrent builds of expression indexes and partial indexes are supported. Errors occurring in the evaluation of these expressions could cause behavior similar to that described above for unique constraint violations.

Regular index builds permit other regular index builds on the same table to occur in parallel, but only one concurrent index build can occur on a table at a time. In both cases, no other types of schema modification on the table are allowed meanwhile. Another difference is that a regular `CREATE INDEX` command can be performed within a transaction block, but `CREATE INDEX CONCURRENTLY` cannot.

Notes

See Chapter 11 for information about when indexes can be used, when they are not used, and in which particular situations they can be useful.

Caution

Hash index operations are not presently WAL-logged, so hash indexes might need to be rebuilt with `REINDEX` after a database crash if there were unwritten changes. Also, changes to hash indexes are not replicated over streaming or file-based replication after the initial base backup, so they give wrong answers to queries that subsequently use them. Hash indexes are also not properly restored during point-in-time recovery. For these reasons, hash index use is presently discouraged.

Currently, only the B-tree, GiST, GIN, and BRIN index methods support multicolumn indexes. Up to 32 fields can be specified by default. (This limit can be altered when building PostgreSQL.) Only B-tree currently supports unique indexes.

An *operator class* can be specified for each column of an index. The operator class identifies the operators to be used by the index for that column. For example, a B-tree index on four-byte integers would use the

`int4_ops` class; this operator class includes comparison functions for four-byte integers. In practice the default operator class for the column's data type is usually sufficient. The main point of having operator classes is that for some data types, there could be more than one meaningful ordering. For example, we might want to sort a complex-number data type either by absolute value or by real part. We could do this by defining two operator classes for the data type and then selecting the proper class when making an index. More information about operator classes is in Section 11.9 and in Section 35.14.

For index methods that support ordered scans (currently, only B-tree), the optional clauses `ASC`, `DESC`, `NULLS FIRST`, and/or `NULLS LAST` can be specified to modify the sort ordering of the index. Since an ordered index can be scanned either forward or backward, it is not normally useful to create a single-column `DESC` index — that sort ordering is already available with a regular index. The value of these options is that multicolumn indexes can be created that match the sort ordering requested by a mixed-ordering query, such as `SELECT ... ORDER BY x ASC, y DESC`. The `NULLS` options are useful if you need to support "nulls sort low" behavior, rather than the default "nulls sort high", in queries that depend on indexes to avoid sorting steps.

For most index methods, the speed of creating an index is dependent on the setting of maintenance_work_mem. Larger values will reduce the time needed for index creation, so long as you don't make it larger than the amount of memory really available, which would drive the machine into swapping.

Use DROP INDEX to remove an index.

Prior releases of PostgreSQL also had an R-tree index method. This method has been removed because it had no significant advantages over the GiST method. If `USING rtree` is specified, `CREATE INDEX` will interpret it as `USING gist`, to simplify conversion of old databases to GiST.

Examples

To create a B-tree index on the column `title` in the table `films`:

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

To create an index on the expression `lower(title)`, allowing efficient case-insensitive searches:

```
CREATE INDEX ON films ((lower(title)));
```

(In this example we have chosen to omit the index name, so the system will choose a name, typically `films_lower_idx`.)

To create an index with non-default collation:

```
CREATE INDEX title_idx_german ON films (title COLLATE "de_DE");
```

To create an index with non-default sort ordering of nulls:

```
CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);
```

To create an index with non-default fill factor:

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH (fillfactor = 70);
```

To create a GIN index with fast updates disabled:

```
CREATE INDEX gin_idx ON documents_table USING GIN (locations) WITH (fastupdate = off);
```

To create an index on the column `code` in the table `films` and have the index reside in the tablespace `indexspace`:

```
CREATE INDEX code_idx ON films (code) TABLESPACE indexspace;
```

To create a GiST index on a point attribute so that we can efficiently use box operators on the result of the conversion function:

```
CREATE INDEX pointloc
    ON points USING gist (box(location,location));
SELECT * FROM points
    WHERE box(location,location) && '(0,0),(1,1)'::box;
```

To create an index without locking out writes to the table:

```
CREATE INDEX CONCURRENTLY sales_quantity_index ON sales_table (quantity);
```

Compatibility

`CREATE INDEX` is a PostgreSQL language extension. There are no provisions for indexes in the SQL standard.

See Also

ALTER INDEX, DROP INDEX

CREATE LANGUAGE

Name

CREATE LANGUAGE — define a new procedural language

Synopsis

```
CREATE [ OR REPLACE ] [ PROCEDURAL ] LANGUAGE name
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ INLINE inline_handler ] [ VALIDATOR valfunction ]
```

Description

CREATE LANGUAGE registers a new procedural language with a PostgreSQL database. Subsequently, functions and trigger procedures can be defined in this new language.

> **Note:** As of PostgreSQL 9.1, most procedural languages have been made into "extensions", and should therefore be installed with CREATE EXTENSION not CREATE LANGUAGE. Direct use of CREATE LANGUAGE should now be confined to extension installation scripts. If you have a "bare" language in your database, perhaps as a result of an upgrade, you can convert it to an extension using CREATE EXTENSION *langname* FROM unpackaged.

CREATE LANGUAGE effectively associates the language name with handler function(s) that are responsible for executing functions written in the language. Refer to Chapter 53 for more information about language handlers.

There are two forms of the CREATE LANGUAGE command. In the first form, the user supplies just the name of the desired language, and the PostgreSQL server consults the pg_pltemplate system catalog to determine the correct parameters. In the second form, the user supplies the language parameters along with the language name. The second form can be used to create a language that is not defined in pg_pltemplate, but this approach is considered obsolescent.

When the server finds an entry in the pg_pltemplate catalog for the given language name, it will use the catalog data even if the command includes language parameters. This behavior simplifies loading of old dump files, which are likely to contain out-of-date information about language support functions.

Ordinarily, the user must have the PostgreSQL superuser privilege to register a new language. However, the owner of a database can register a new language within that database if the language is listed in the pg_pltemplate catalog and is marked as allowed to be created by database owners (tmpldbacreate is true). The default is that trusted languages can be created by database owners, but this can be adjusted by superusers by modifying the contents of pg_pltemplate. The creator of a language becomes its owner and can later drop it, rename it, or assign it to a new owner.

CREATE OR REPLACE LANGUAGE will either create a new language, or replace an existing definition. If the language already exists, its parameters are updated according to the values specified or taken from pg_pltemplate, but the language's ownership and permissions settings do not change, and any existing

functions written in the language are assumed to still be valid. In addition to the normal privilege requirements for creating a language, the user must be superuser or owner of the existing language. The REPLACE case is mainly meant to be used to ensure that the language exists. If the language has a pg_pltemplate entry then REPLACE will not actually change anything about an existing definition, except in the unusual case where the pg_pltemplate entry has been modified since the language was created.

Parameters

TRUSTED

TRUSTED specifies that the language does not grant access to data that the user would not otherwise have. If this key word is omitted when registering the language, only users with the PostgreSQL superuser privilege can use this language to create new functions.

PROCEDURAL

This is a noise word.

name

The name of the new procedural language. The name must be unique among the languages in the database.

For backward compatibility, the name can be enclosed by single quotes.

HANDLER *call_handler*

call_handler is the name of a previously registered function that will be called to execute the procedural language's functions. The call handler for a procedural language must be written in a compiled language such as C with version 1 call convention and registered with PostgreSQL as a function taking no arguments and returning the language_handler type, a placeholder type that is simply used to identify the function as a call handler.

INLINE *inline_handler*

inline_handler is the name of a previously registered function that will be called to execute an anonymous code block (DO command) in this language. If no *inline_handler* function is specified, the language does not support anonymous code blocks. The handler function must take one argument of type internal, which will be the DO command's internal representation, and it will typically return void. The return value of the handler is ignored.

VALIDATOR *valfunction*

valfunction is the name of a previously registered function that will be called when a new function in the language is created, to validate the new function. If no validator function is specified, then a new function will not be checked when it is created. The validator function must take one argument of type oid, which will be the OID of the to-be-created function, and will typically return void.

A validator function would typically inspect the function body for syntactical correctness, but it can also look at other properties of the function, for example if the language cannot handle certain argument types. To signal an error, the validator function should use the ereport() function. The return value of the function is ignored.

The TRUSTED option and the support function name(s) are ignored if the server has an entry for the specified language name in pg_pltemplate.

Notes

The createlang program is a simple wrapper around the CREATE LANGUAGE command. It eases installation of procedural languages from the shell command line.

Use DROP LANGUAGE, or better yet the droplang program, to drop procedural languages.

The system catalog pg_language (see Section 49.28) records information about the currently installed languages. Also, createlang has an option to list the installed languages.

To create functions in a procedural language, a user must have the USAGE privilege for the language. By default, USAGE is granted to PUBLIC (i.e., everyone) for trusted languages. This can be revoked if desired.

Procedural languages are local to individual databases. However, a language can be installed into the template1 database, which will cause it to be available automatically in all subsequently-created databases.

The call handler function, the inline handler function (if any), and the validator function (if any) must already exist if the server does not have an entry for the language in pg_pltemplate. But when there is an entry, the functions need not already exist; they will be automatically defined if not present in the database. (This might result in CREATE LANGUAGE failing, if the shared library that implements the language is not available in the installation.)

In PostgreSQL versions before 7.3, it was necessary to declare handler functions as returning the placeholder type opaque, rather than language_handler. To support loading of old dump files, CREATE LANGUAGE will accept a function declared as returning opaque, but it will issue a notice and change the function's declared return type to language_handler.

Examples

The preferred way of creating any of the standard procedural languages is just:

```
CREATE LANGUAGE plperl;
```

For a language not known in the pg_pltemplate catalog, a sequence such as this is needed:

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

Compatibility

CREATE LANGUAGE is a PostgreSQL extension.

See Also

ALTER LANGUAGE, CREATE FUNCTION, DROP LANGUAGE, GRANT, REVOKE, createlang, droplang

CREATE MATERIALIZED VIEW

Name

CREATE MATERIALIZED VIEW — define a new materialized view

Synopsis

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] table_name
    [ (column_name [, ...] ) ]
    [ WITH ( storage_parameter [= value] [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    AS query
    [ WITH [ NO ] DATA ]
```

Description

CREATE MATERIALIZED VIEW defines a materialized view of a query. The query is executed and used to populate the view at the time the command is issued (unless WITH NO DATA is used) and may be refreshed later using REFRESH MATERIALIZED VIEW.

CREATE MATERIALIZED VIEW is similar to CREATE TABLE AS, except that it also remembers the query used to initialize the view, so that it can be refreshed later upon demand. A materialized view has many of the same properties as a table, but there is no support for temporary materialized views or automatic generation of OIDs.

Parameters

IF NOT EXISTS

Do not throw an error if a materialized view with the same name already exists. A notice is issued in this case. Note that there is no guarantee that the existing materialized view is anything like the one that would have been created.

table_name

The name (optionally schema-qualified) of the materialized view to be created.

column_name

The name of a column in the new materialized view. If column names are not provided, they are taken from the output column names of the query.

WITH (*storage_parameter* [= *value*] [, ...])

This clause specifies optional storage parameters for the new materialized view; see *Storage Parameters* for more information. All parameters supported for CREATE TABLE are also supported for CREATE MATERIALIZED VIEW with the exception of OIDS. See CREATE TABLE for more information.

`TABLESPACE` *`tablespace_name`*

The *`tablespace_name`* is the name of the tablespace in which the new materialized view is to be created. If not specified, default_tablespace is consulted.

`query`

A SELECT, TABLE, or VALUES command. This query will run within a security-restricted operation; in particular, calls to functions that themselves create temporary tables will fail.

`WITH [ NO ] DATA`

This clause specifies whether or not the materialized view should be populated at creation time. If not, the materialized view will be flagged as unscannable and cannot be queried until `REFRESH MATERIALIZED VIEW` is used.

Compatibility

`CREATE MATERIALIZED VIEW` is a PostgreSQL extension.

See Also

ALTER MATERIALIZED VIEW, CREATE TABLE AS, CREATE VIEW, DROP MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

CREATE OPERATOR

Name

CREATE OPERATOR — define a new operator

Synopsis

```
CREATE OPERATOR name (
    PROCEDURE = function_name
    [, LEFTARG = left_type ] [, RIGHTARG = right_type ]
    [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
    [, RESTRICT = res_proc ] [, JOIN = join_proc ]
    [, HASHES ] [, MERGES ]
)
```

Description

CREATE OPERATOR defines a new operator, *name*. The user who defines an operator becomes its owner. If a schema name is given then the operator is created in the specified schema. Otherwise it is created in the current schema.

The operator name is a sequence of up to NAMEDATALEN-1 (63 by default) characters from the following list:

+ - * / < > = ~ ! @ # % ^ & | ` ?

There are a few restrictions on your choice of name:

- -- and /* cannot appear anywhere in an operator name, since they will be taken as the start of a comment.
- A multicharacter operator name cannot end in + or -, unless the name also contains at least one of these characters:

  ~ ! @ # % ^ & | ` ?

  For example, @- is an allowed operator name, but *- is not. This restriction allows PostgreSQL to parse SQL-compliant commands without requiring spaces between tokens.
- The use of => as an operator name is deprecated. It may be disallowed altogether in a future release.

The operator != is mapped to <> on input, so these two names are always equivalent.

At least one of LEFTARG and RIGHTARG must be defined. For binary operators, both must be defined. For right unary operators, only LEFTARG should be defined, while for left unary operators only RIGHTARG should be defined.

Note: Right unary, also called postfix, operators are deprecated and will be removed in PostgreSQL version 14.

The *function_name* procedure must have been previously defined using CREATE FUNCTION and must be defined to accept the correct number of arguments (either one or two) of the indicated types.

The other clauses specify optional operator optimization clauses. Their meaning is detailed in Section 35.13.

To be able to create an operator, you must have USAGE privilege on the argument types and the return type, as well as EXECUTE privilege on the underlying function. If a commutator or negator operator is specified, you must own these operators.

Parameters

name

The name of the operator to be defined. See above for allowable characters. The name can be schema-qualified, for example CREATE OPERATOR myschema.+ (...). If not, then the operator is created in the current schema. Two operators in the same schema can have the same name if they operate on different data types. This is called *overloading*.

function_name

The function used to implement this operator.

left_type

The data type of the operator's left operand, if any. This option would be omitted for a left-unary operator.

right_type

The data type of the operator's right operand, if any. This option would be omitted for a right-unary operator.

com_op

The commutator of this operator.

neg_op

The negator of this operator.

res_proc

The restriction selectivity estimator function for this operator.

join_proc

The join selectivity estimator function for this operator.

HASHES

Indicates this operator can support a hash join.

MERGES

Indicates this operator can support a merge join.

To give a schema-qualified operator name in *com_op* or the other optional arguments, use the OPERATOR() syntax, for example:

```
COMMUTATOR = OPERATOR(myschema.===) ,
```

Notes

Refer to Section 35.12 for further information.

It is not possible to specify an operator's lexical precedence in CREATE OPERATOR, because the parser's precedence behavior is hard-wired. See Section 4.1.6 for precedence details.

The obsolete options SORT1, SORT2, LTCMP, and GTCMP were formerly used to specify the names of sort operators associated with a merge-joinable operator. This is no longer necessary, since information about associated operators is found by looking at B-tree operator families instead. If one of these options is given, it is ignored except for implicitly setting MERGES true.

Use DROP OPERATOR to delete user-defined operators from a database. Use ALTER OPERATOR to modify operators in a database.

Examples

The following command defines a new operator, area-equality, for the data type box:

```
CREATE OPERATOR === (
    LEFTARG = box,
    RIGHTARG = box,
    PROCEDURE = area_equal_procedure,
    COMMUTATOR = ===,
    NEGATOR = !==,
    RESTRICT = area_restriction_procedure,
    JOIN = area_join_procedure,
    HASHES, MERGES
);
```

Compatibility

CREATE OPERATOR is a PostgreSQL extension. There are no provisions for user-defined operators in the SQL standard.

See Also

ALTER OPERATOR, CREATE OPERATOR CLASS, DROP OPERATOR

CREATE OPERATOR CLASS

Name

CREATE OPERATOR CLASS — define a new operator class

Synopsis

```
CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data_type
  USING index_method [ FAMILY family_name ] AS
  {  OPERATOR strategy_number operator_name [ ( op_type, op_type ) ] [ FOR SEARCH | FOR ORDER B
   | FUNCTION support_number [ ( op_type [ , op_type ] ) ] function_name ( argument_type [, ...]
   | STORAGE storage_type
  } [, ... ]
```

Description

CREATE OPERATOR CLASS creates a new operator class. An operator class defines how a particular data type can be used with an index. The operator class specifies that certain operators will fill particular roles or "strategies" for this data type and this index method. The operator class also specifies the support procedures to be used by the index method when the operator class is selected for an index column. All the operators and functions used by an operator class must be defined before the operator class can be created.

If a schema name is given then the operator class is created in the specified schema. Otherwise it is created in the current schema. Two operator classes in the same schema can have the same name only if they are for different index methods.

The user who defines an operator class becomes its owner. Presently, the creating user must be a superuser. (This restriction is made because an erroneous operator class definition could confuse or even crash the server.)

CREATE OPERATOR CLASS does not presently check whether the operator class definition includes all the operators and functions required by the index method, nor whether the operators and functions form a self-consistent set. It is the user's responsibility to define a valid operator class.

Related operator classes can be grouped into *operator families*. To add a new operator class to an existing family, specify the FAMILY option in CREATE OPERATOR CLASS. Without this option, the new class is placed into a family named the same as the new class (creating that family if it doesn't already exist).

Refer to Section 35.14 for further information.

Parameters

name

The name of the operator class to be created. The name can be schema-qualified.

`DEFAULT`

If present, the operator class will become the default operator class for its data type. At most one operator class can be the default for a specific data type and index method.

`data_type`

The column data type that this operator class is for.

`index_method`

The name of the index method this operator class is for.

`family_name`

The name of the existing operator family to add this operator class to. If not specified, a family named the same as the operator class is used (creating it, if it doesn't already exist).

`strategy_number`

The index method's strategy number for an operator associated with the operator class.

`operator_name`

The name (optionally schema-qualified) of an operator associated with the operator class.

`op_type`

In an `OPERATOR` clause, the operand data type(s) of the operator, or `NONE` to signify a left-unary or right-unary operator. The operand data types can be omitted in the normal case where they are the same as the operator class's data type.

In a `FUNCTION` clause, the operand data type(s) the function is intended to support, if different from the input data type(s) of the function (for B-tree comparison functions and hash functions) or the class's data type (for B-tree sort support functions and all functions in GiST, SP-GiST, GIN and BRIN operator classes). These defaults are correct, and so *`op_type`* need not be specified in `FUNCTION` clauses, except for the case of a B-tree sort support function that is meant to support cross-data-type comparisons.

`sort_family_name`

The name (optionally schema-qualified) of an existing `btree` operator family that describes the sort ordering associated with an ordering operator.

If neither `FOR SEARCH` nor `FOR ORDER BY` is specified, `FOR SEARCH` is the default.

`support_number`

The index method's support procedure number for a function associated with the operator class.

`function_name`

The name (optionally schema-qualified) of a function that is an index method support procedure for the operator class.

`argument_type`

The parameter data type(s) of the function.

storage_type

The data type actually stored in the index. Normally this is the same as the column data type, but some index methods (currently GiST, GIN and BRIN) allow it to be different. The STORAGE clause must be omitted unless the index method allows a different type to be used.

The OPERATOR, FUNCTION, and STORAGE clauses can appear in any order.

Notes

Because the index machinery does not check access permissions on functions before using them, including a function or operator in an operator class is tantamount to granting public execute permission on it. This is usually not an issue for the sorts of functions that are useful in an operator class.

The operators should not be defined by SQL functions. A SQL function is likely to be inlined into the calling query, which will prevent the optimizer from recognizing that the query matches an index.

Before PostgreSQL 8.4, the OPERATOR clause could include a RECHECK option. This is no longer supported because whether an index operator is "lossy" is now determined on-the-fly at run time. This allows efficient handling of cases where an operator might or might not be lossy.

Examples

The following example command defines a GiST index operator class for the data type _int4 (array of int4). See the intarray module for the complete example.

```
CREATE OPERATOR CLASS gist__int_ops
    DEFAULT FOR TYPE _int4 USING gist AS
        OPERATOR        3       &&,
        OPERATOR        6       = (anyarray, anyarray),
        OPERATOR        7       @>,
        OPERATOR        8       <@,
        OPERATOR        20      @@ (_int4, query_int),
        FUNCTION        1       g_int_consistent (internal, _int4, int, oid, internal),
        FUNCTION        2       g_int_union (internal, internal),
        FUNCTION        3       g_int_compress (internal),
        FUNCTION        4       g_int_decompress (internal),
        FUNCTION        5       g_int_penalty (internal, internal, internal),
        FUNCTION        6       g_int_picksplit (internal, internal),
        FUNCTION        7       g_int_same (_int4, _int4, internal);
```

Compatibility

CREATE OPERATOR CLASS is a PostgreSQL extension. There is no CREATE OPERATOR CLASS statement in the SQL standard.

See Also

ALTER OPERATOR CLASS, DROP OPERATOR CLASS, CREATE OPERATOR FAMILY, ALTER OPERATOR FAMILY

CREATE OPERATOR FAMILY

Name

CREATE OPERATOR FAMILY — define a new operator family

Synopsis

```
CREATE OPERATOR FAMILY name USING index_method
```

Description

CREATE OPERATOR FAMILY creates a new operator family. An operator family defines a collection of related operator classes, and perhaps some additional operators and support functions that are compatible with these operator classes but not essential for the functioning of any individual index. (Operators and functions that are essential to indexes should be grouped within the relevant operator class, rather than being "loose" in the operator family. Typically, single-data-type operators are bound to operator classes, while cross-data-type operators can be loose in an operator family containing operator classes for both data types.)

The new operator family is initially empty. It should be populated by issuing subsequent CREATE OPERATOR CLASS commands to add contained operator classes, and optionally ALTER OPERATOR FAMILY commands to add "loose" operators and their corresponding support functions.

If a schema name is given then the operator family is created in the specified schema. Otherwise it is created in the current schema. Two operator families in the same schema can have the same name only if they are for different index methods.

The user who defines an operator family becomes its owner. Presently, the creating user must be a superuser. (This restriction is made because an erroneous operator family definition could confuse or even crash the server.)

Refer to Section 35.14 for further information.

Parameters

name

The name of the operator family to be created. The name can be schema-qualified.

index_method

The name of the index method this operator family is for.

Compatibility

`CREATE OPERATOR FAMILY` is a PostgreSQL extension. There is no `CREATE OPERATOR FAMILY` statement in the SQL standard.

See Also

ALTER OPERATOR FAMILY, DROP OPERATOR FAMILY, CREATE OPERATOR CLASS, ALTER OPERATOR CLASS, DROP OPERATOR CLASS

CREATE POLICY

Name

CREATE POLICY — define a new row level security policy for a table

Synopsis

```
CREATE POLICY name ON table_name
    [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
    [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
    [ USING ( using_expression ) ]
    [ WITH CHECK ( check_expression ) ]
```

Description

The CREATE POLICY command defines a new row-level security policy for a table. Note that row-level security must be enabled on the table (using ALTER TABLE ... ENABLE ROW LEVEL SECURITY) in order for created policies to be applied.

A policy grants the permission to select, insert, update, or delete rows that match the relevant policy expression. Existing table rows are checked against the expression specified in USING, while new rows that would be created via INSERT or UPDATE are checked against the expression specified in WITH CHECK. When a USING expression returns true for a given row then that row is visible to the user, while if false or null is returned then the row is not visible. When a WITH CHECK expression returns true for a row then that row is inserted or updated, while if false or null is returned then an error occurs.

For INSERT and UPDATE statements, WITH CHECK expressions are enforced after BEFORE triggers are fired, and before any actual data modifications are made. Thus a BEFORE ROW trigger may modify the data to be inserted, affecting the result of the security policy check. WITH CHECK expressions are enforced before any other constraints.

Policy names are per-table. Therefore, one policy name can be used for many different tables and have a definition for each table which is appropriate to that table.

Policies can be applied for specific commands or for specific roles. The default for newly created policies is that they apply for all commands and roles, unless otherwise specified. Multiple policies may apply to a single command; see below for more details. Table 1 summarizes how the different types of policy apply to specific commands.

For policies that can have both USING and WITH CHECK expressions (ALL and UPDATE), if no WITH CHECK expression is defined, then the USING expression will be used both to determine which rows are visible (normal USING case) and which new rows will be allowed to be added (WITH CHECK case).

If row-level security is enabled for a table, but no applicable policies exist, a “default deny” policy is assumed, so that no rows will be visible or updatable.

Parameters

name

The name of the policy to be created. This must be distinct from the name of any other policy for the table.

table_name

The name (optionally schema-qualified) of the table the policy applies to.

command

The command to which the policy applies. Valid options are ALL, SELECT, INSERT, UPDATE, and DELETE. ALL is the default. See below for specifics regarding how these are applied.

role_name

The role(s) to which the policy is to be applied. The default is PUBLIC, which will apply the policy to all roles.

using_expression

Any SQL conditional expression (returning boolean). The conditional expression cannot contain any aggregate or window functions. This expression will be added to queries that refer to the table if row level security is enabled. Rows for which the expression returns true will be visible. Any rows for which the expression returns false or null will not be visible to the user (in a SELECT), and will not be available for modification (in an UPDATE or DELETE). Such rows are silently suppressed; no error is reported.

check_expression

Any SQL conditional expression (returning boolean). The conditional expression cannot contain any aggregate or window functions. This expression will be used in INSERT and UPDATE queries against the table if row level security is enabled. Only rows for which the expression evaluates to true will be allowed. An error will be thrown if the expression evaluates to false or null for any of the records inserted or any of the records that result from the update. Note that the *check_expression* is evaluated against the proposed new contents of the row, not the original contents.

Per-Command Policies

ALL

Using ALL for a policy means that it will apply to all commands, regardless of the type of command. If an ALL policy exists and more specific policies exist, then both the ALL policy and the more specific policy (or policies) will be applied. Additionally, ALL policies will be applied to both the selection side of a query and the modification side, using the USING expression for both cases if only a USING expression has been defined.

As an example, if an UPDATE is issued, then the ALL policy will be applicable both to what the UPDATE will be able to select as rows to be updated (applying the USING expression), and to the resulting updated rows, to check if they are permitted to be added to the table (applying the WITH CHECK expression, if defined, and the USING expression otherwise). If an INSERT or UPDATE command attempts to add rows to the table that do not pass the ALL policy's WITH CHECK expression, the entire command will be aborted.

`SELECT`

Using `SELECT` for a policy means that it will apply to `SELECT` queries and whenever `SELECT` permissions are required on the relation the policy is defined for. The result is that only those records from the relation that pass the `SELECT` policy will be returned during a `SELECT` query, and that queries that require `SELECT` permissions, such as `UPDATE`, will also only see those records that are allowed by the `SELECT` policy. A `SELECT` policy cannot have a `WITH CHECK` expression, as it only applies in cases where records are being retrieved from the relation.

`INSERT`

Using `INSERT` for a policy means that it will apply to `INSERT` commands. Rows being inserted that do not pass this policy will result in a policy violation error, and the entire `INSERT` command will be aborted. An `INSERT` policy cannot have a `USING` expression, as it only applies in cases where records are being added to the relation.

Note that `INSERT` with `ON CONFLICT DO UPDATE` checks `INSERT` policies' `WITH CHECK` expressions only for rows appended to the relation by the `INSERT` path.

`UPDATE`

Using `UPDATE` for a policy means that it will apply to `UPDATE`, `SELECT FOR UPDATE` and `SELECT FOR SHARE` commands, as well as auxiliary `ON CONFLICT DO UPDATE` clauses of `INSERT` commands. Since `UPDATE` involves pulling an existing record and replacing it with a new modified record, `UPDATE` policies accept both a `USING` expression and a `WITH CHECK` expression. The `USING` expression determines which records the `UPDATE` command will see to operate against, while the `WITH CHECK` expression defines which modified rows are allowed to be stored back into the relation.

Any rows whose updated values do not pass the `WITH CHECK` expression will cause an error, and the entire command will be aborted. If only a `USING` clause is specified, then that clause will be used for both `USING` and `WITH CHECK` cases.

Typically an `UPDATE` command also needs to read data from columns in the relation being updated (e.g., in a `WHERE` clause or a `RETURNING` clause, or in an expression on the right hand side of the `SET` clause). In this case, `SELECT` rights are also required on the relation being updated, and the appropriate `SELECT` or `ALL` policies will be applied in addition to the `UPDATE` policies. Thus the user must have access to the row(s) being updated through a `SELECT` or `ALL` policy in addition to being granted permission to update the row(s) via an `UPDATE` or `ALL` policy.

When an `INSERT` command has an auxiliary `ON CONFLICT DO UPDATE` clause, if the `UPDATE` path is taken, the row to be updated is first checked against the `USING` expressions of any `UPDATE` policies, and then the new updated row is checked against the `WITH CHECK` expressions. Note, however, that unlike a standalone `UPDATE` command, if the existing row does not pass the `USING` expressions, an error will be thrown (the `UPDATE` path will *never* be silently avoided).

`DELETE`

Using `DELETE` for a policy means that it will apply to `DELETE` commands. Only rows that pass this policy will be seen by a `DELETE` command. There can be rows that are visible through a `SELECT` that are not available for deletion, if they do not pass the `USING` expression for the `DELETE` policy.

In most cases a `DELETE` command also needs to read data from columns in the relation that it is deleting from (e.g., in a `WHERE` clause or a `RETURNING` clause). In this case, `SELECT` rights are also required on the relation, and the appropriate `SELECT` or `ALL` policies will be applied in addition to

the DELETE policies. Thus the user must have access to the row(s) being deleted through a SELECT or ALL policy in addition to being granted permission to delete the row(s) via a DELETE or ALL policy.

A DELETE policy cannot have a WITH CHECK expression, as it only applies in cases where records are being deleted from the relation, so that there is no new row to check.

Table 1. Policies Applied by Command Type

Command	SELECT/ALL policy	INSERT/ALL policy	UPDATE/ALL policy		DELETE/ALL policy
	USING expression	WITH CHECK expression	USING expression	WITH CHECK expression	USING expression
SELECT	Existing row	—	—	—	—
SELECT FOR UPDATE/SHARE	Existing row	—	Existing row	—	—
INSERT	—	New row	—	—	—
INSERT ... RETURNING	New row [a]	New row	—	—	—
UPDATE	Existing & new rows [a]	—	Existing row	New row	—
DELETE	Existing row [a]	—	—	—	Existing row
ON CONFLICT DO UPDATE	Existing & new rows	—	Existing row	New row	—

Notes:
a. If read access is required to the existing or new row (for example, a WHERE or RETURNING clause that refers to columns from the relation).

Application of Multiple Policies

When multiple policies of different command types apply to the same command (for example, SELECT and UPDATE policies applied to an UPDATE command), then the user must have both types of permissions (for example, permission to select rows from the relation as well as permission to update them). Thus the expressions for one type of policy are combined with the expressions for the other type of policy using the AND operator.

When multiple policies of the same command type apply to the same command, then at least one of the policies must grant access to the relation. Thus the expressions from all the policies of that type are combined using the OR operator. If there are no applicable policies, then access is denied.

Note that, for the purposes of combining multiple policies, ALL policies are treated as having the same type as whichever other type of policy is being applied.

For example, in an UPDATE command requiring both SELECT and UPDATE permissions, if there are multiple applicable policies of each type, they will be combined as follows:

```
(
  expression from SELECT/ALL policy 1
```

```
  OR
  expression from SELECT/ALL policy 2
  OR
  ...
)
AND
(
  expression from UPDATE/ALL policy 1
  OR
  expression from UPDATE/ALL policy 2
  OR
  ...
)
```

Notes

You must be the owner of a table to create or change policies for it.

While policies will be applied for explicit queries against tables in the database, they are not applied when the system is performing internal referential integrity checks or validating constraints. This means there are indirect ways to determine that a given value exists. An example of this is attempting to insert a duplicate value into a column that is a primary key or has a unique constraint. If the insert fails then the user can infer that the value already exists. (This example assumes that the user is permitted by policy to insert records which they are not allowed to see.) Another example is where a user is allowed to insert into a table which references another, otherwise hidden table. Existence can be determined by the user inserting values into the referencing table, where success would indicate that the value exists in the referenced table. These issues can be addressed by carefully crafting policies to prevent users from being able to insert, delete, or update records at all which might possibly indicate a value they are not otherwise able to see, or by using generated values (e.g., surrogate keys) instead of keys with external meanings.

Generally, the system will enforce filter conditions imposed using security policies prior to qualifications that appear in user queries, in order to prevent inadvertent exposure of the protected data to user-defined functions which might not be trustworthy. However, functions and operators marked by the system (or the system administrator) as `LEAKPROOF` may be evaluated before policy expressions, as they are assumed to be trustworthy.

Since policy expressions are added to the user's query directly, they will be run with the rights of the user running the overall query. Therefore, users who are using a given policy must be able to access any tables or functions referenced in the expression or they will simply receive a permission denied error when attempting to query the table that has row-level security enabled. This does not change how views work, however. As with normal queries and views, permission checks and policies for the tables which are referenced by a view will use the view owner's rights and any policies which apply to the view owner.

Additional discussion and practical examples can be found in Section 5.7.

Compatibility

`CREATE POLICY` is a PostgreSQL extension.

See Also

ALTER POLICY, DROP POLICY, ALTER TABLE

CREATE ROLE

Name

CREATE ROLE — define a new database role

Synopsis

```
CREATE ROLE name [ [ WITH ] option [ ... ] ]

where option can be:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
```

Description

CREATE ROLE adds a new role to a PostgreSQL database cluster. A role is an entity that can own database objects and have database privileges; a role can be considered a “user”, a “group”, or both depending on how it is used. Refer to Chapter 20 and Chapter 19 for information about managing users and authentication. You must have CREATEROLE privilege or be a database superuser to use this command.

Note that roles are defined at the database cluster level, and so are valid in all databases in the cluster.

Parameters

name

The name of the new role.

`SUPERUSER`
`NOSUPERUSER`

These clauses determine whether the new role is a "superuser", who can override all access restrictions within the database. Superuser status is dangerous and should be used only when really needed. You must yourself be a superuser to create a new superuser. If not specified, `NOSUPERUSER` is the default.

`CREATEDB`
`NOCREATEDB`

These clauses define a role's ability to create databases. If `CREATEDB` is specified, the role being defined will be allowed to create new databases. Specifying `NOCREATEDB` will deny a role the ability to create databases. If not specified, `NOCREATEDB` is the default.

`CREATEROLE`
`NOCREATEROLE`

These clauses determine whether a role will be permitted to create new roles (that is, execute `CREATE ROLE`). A role with `CREATEROLE` privilege can also alter and drop other roles. If not specified, `NOCREATEROLE` is the default.

`CREATEUSER`
`NOCREATEUSER`

These clauses are an obsolete, but still accepted, spelling of `SUPERUSER` and `NOSUPERUSER`. Note that they are *not* equivalent to `CREATEROLE` as one might naively expect!

`INHERIT`
`NOINHERIT`

These clauses determine whether a role "inherits" the privileges of roles it is a member of. A role with the `INHERIT` attribute can automatically use whatever database privileges have been granted to all roles it is directly or indirectly a member of. Without `INHERIT`, membership in another role only grants the ability to `SET ROLE` to that other role; the privileges of the other role are only available after having done so. If not specified, `INHERIT` is the default.

`LOGIN`
`NOLOGIN`

These clauses determine whether a role is allowed to log in; that is, whether the role can be given as the initial session authorization name during client connection. A role having the `LOGIN` attribute can be thought of as a user. Roles without this attribute are useful for managing database privileges, but are not users in the usual sense of the word. If not specified, `NOLOGIN` is the default, except when `CREATE ROLE` is invoked through its alternative spelling CREATE USER.

`REPLICATION`
`NOREPLICATION`

These clauses determine whether a role is allowed to initiate streaming replication or put the system in and out of backup mode. A role having the `REPLICATION` attribute is a very highly privileged role, and should only be used on roles actually used for replication. If not specified, `NOREPLICATION` is the default. You must be a superuser to create a new role having the `REPLICATION` attribute.

`BYPASSRLS`
`NOBYPASSRLS`

These clauses determine whether a role bypasses every row-level security (RLS) policy. `NOBYPASSRLS` is the default. You must be a superuser to create a new role having the `BYPASSRLS` attribute.

Note that pg_dump will set `row_security` to `OFF` by default, to ensure all contents of a table are dumped out. If the user running pg_dump does not have appropriate permissions, an error will be returned. However, superusers and the owner of the table being dumped always bypass RLS.

`CONNECTION LIMIT` *connlimit*

If role can log in, this specifies how many concurrent connections the role can make. -1 (the default) means no limit.

`PASSWORD` *password*

Sets the role's password. (A password is only of use for roles having the `LOGIN` attribute, but you can nonetheless define one for roles without it.) If you do not plan to use password authentication you can omit this option. If no password is specified, the password will be set to null and password authentication will always fail for that user. A null password can optionally be written explicitly as `PASSWORD NULL`.

`ENCRYPTED`
`UNENCRYPTED`

These key words control whether the password is stored encrypted in the system catalogs. (If neither is specified, the default behavior is determined by the configuration parameter password_encryption.) If the presented password string is already in MD5-encrypted format, then it is stored encrypted as-is, regardless of whether `ENCRYPTED` or `UNENCRYPTED` is specified (since the system cannot decrypt the specified encrypted password string). This allows reloading of encrypted passwords during dump/restore.

`VALID UNTIL` '*timestamp*'

The `VALID UNTIL` clause sets a date and time after which the role's password is no longer valid. If this clause is omitted the password will be valid for all time.

`IN ROLE` *role_name*

The `IN ROLE` clause lists one or more existing roles to which the new role will be immediately added as a new member. (Note that there is no option to add the new role as an administrator; use a separate `GRANT` command to do that.)

`IN GROUP` *role_name*

`IN GROUP` is an obsolete spelling of `IN ROLE`.

`ROLE` *role_name*

The `ROLE` clause lists one or more existing roles which are automatically added as members of the new role. (This in effect makes the new role a "group".)

`ADMIN` *role_name*

The `ADMIN` clause is like `ROLE`, but the named roles are added to the new role `WITH ADMIN OPTION`, giving them the right to grant membership in this role to others.

`USER` *role_name*

The `USER` clause is an obsolete spelling of the `ROLE` clause.

`SYSID` *uid*

The `SYSID` clause is ignored, but is accepted for backwards compatibility.

Notes

Use ALTER ROLE to change the attributes of a role, and DROP ROLE to remove a role. All the attributes specified by `CREATE ROLE` can be modified by later `ALTER ROLE` commands.

The preferred way to add and remove members of roles that are being used as groups is to use GRANT and REVOKE.

The `VALID UNTIL` clause defines an expiration time for a password only, not for the role *per se*. In particular, the expiration time is not enforced when logging in using a non-password-based authentication method.

The `INHERIT` attribute governs inheritance of grantable privileges (that is, access privileges for database objects and role memberships). It does not apply to the special role attributes set by `CREATE ROLE` and `ALTER ROLE`. For example, being a member of a role with `CREATEDB` privilege does not immediately grant the ability to create databases, even if `INHERIT` is set; it would be necessary to become that role via SET ROLE before creating a database.

The `INHERIT` attribute is the default for reasons of backwards compatibility: in prior releases of PostgreSQL, users always had access to all privileges of groups they were members of. However, `NOINHERIT` provides a closer match to the semantics specified in the SQL standard.

Be careful with the `CREATEROLE` privilege. There is no concept of inheritance for the privileges of a `CREATEROLE`-role. That means that even if a role does not have a certain privilege but is allowed to create other roles, it can easily create another role with different privileges than its own (except for creating roles with superuser privileges). For example, if the role "user" has the `CREATEROLE` privilege but not the `CREATEDB` privilege, nonetheless it can create a new role with the `CREATEDB` privilege. Therefore, regard roles that have the `CREATEROLE` privilege as almost-superuser-roles.

PostgreSQL includes a program createuser that has the same functionality as `CREATE ROLE` (in fact, it calls this command) but can be run from the command shell.

The `CONNECTION LIMIT` option is only enforced approximately; if two new sessions start at about the same time when just one connection "slot" remains for the role, it is possible that both will fail. Also, the limit is never enforced for superusers.

Caution must be exercised when specifying an unencrypted password with this command. The password will be transmitted to the server in cleartext, and it might also be logged in the client's command history or the server log. The command createuser, however, transmits the password encrypted. Also, psql contains a command `\password` that can be used to safely change the password later.

Examples

Create a role that can log in, but don't give it a password:

```
CREATE ROLE jonathan LOGIN;
```

Create a role with a password:

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

(`CREATE USER` is the same as `CREATE ROLE` except that it implies `LOGIN`.)

Create a role with a password that is valid until the end of 2004. After one second has ticked in 2005, the password is no longer valid.

```
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

Create a role that can create databases and manage roles:

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

Compatibility

The `CREATE ROLE` statement is in the SQL standard, but the standard only requires the syntax

```
CREATE ROLE name [ WITH ADMIN role_name ]
```

Multiple initial administrators, and all the other options of `CREATE ROLE`, are PostgreSQL extensions.

The SQL standard defines the concepts of users and roles, but it regards them as distinct concepts and leaves all commands defining users to be specified by each database implementation. In PostgreSQL we have chosen to unify users and roles into a single kind of entity. Roles therefore have many more optional attributes than they do in the standard.

The behavior specified by the SQL standard is most closely approximated by giving users the `NOINHERIT` attribute, while roles are given the `INHERIT` attribute.

See Also

SET ROLE, ALTER ROLE, DROP ROLE, GRANT, REVOKE, createuser

CREATE RULE

Name

CREATE RULE — define a new rewrite rule

Synopsis

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table_name [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }

where event can be one of:

    SELECT | INSERT | UPDATE | DELETE
```

Description

CREATE RULE defines a new rule applying to a specified table or view. CREATE OR REPLACE RULE will either create a new rule, or replace an existing rule of the same name for the same table.

The PostgreSQL rule system allows one to define an alternative action to be performed on insertions, updates, or deletions in database tables. Roughly speaking, a rule causes additional commands to be executed when a given command on a given table is executed. Alternatively, an INSTEAD rule can replace a given command by another, or cause a command not to be executed at all. Rules are used to implement SQL views as well. It is important to realize that a rule is really a command transformation mechanism, or command macro. The transformation happens before the execution of the command starts. If you actually want an operation that fires independently for each physical row, you probably want to use a trigger, not a rule. More information about the rules system is in Chapter 38.

Presently, ON SELECT rules must be unconditional INSTEAD rules and must have actions that consist of a single SELECT command. Thus, an ON SELECT rule effectively turns the table into a view, whose visible contents are the rows returned by the rule's SELECT command rather than whatever had been stored in the table (if anything). It is considered better style to write a CREATE VIEW command than to create a real table and define an ON SELECT rule for it.

You can create the illusion of an updatable view by defining ON INSERT, ON UPDATE, and ON DELETE rules (or any subset of those that's sufficient for your purposes) to replace update actions on the view with appropriate updates on other tables. If you want to support INSERT RETURNING and so on, then be sure to put a suitable RETURNING clause into each of these rules.

There is a catch if you try to use conditional rules for complex view updates: there *must* be an unconditional INSTEAD rule for each action you wish to allow on the view. If the rule is conditional, or is not INSTEAD, then the system will still reject attempts to perform the update action, because it thinks it might end up trying to perform the action on the dummy table of the view in some cases. If you want to handle all the useful cases in conditional rules, add an unconditional DO INSTEAD NOTHING rule to ensure that the system understands it will never be called on to update the dummy table. Then make the conditional rules

non-INSTEAD; in the cases where they are applied, they add to the default INSTEAD NOTHING action. (This method does not currently work to support RETURNING queries, however.)

> **Note:** A view that is simple enough to be automatically updatable (see CREATE VIEW) does not require a user-created rule in order to be updatable. While you can create an explicit rule anyway, the automatic update transformation will generally outperform an explicit rule.
>
> Another alternative worth considering is to use INSTEAD OF triggers (see CREATE TRIGGER) in place of rules.

Parameters

name

The name of a rule to create. This must be distinct from the name of any other rule for the same table. Multiple rules on the same table and same event type are applied in alphabetical name order.

event

The event is one of SELECT, INSERT, UPDATE, or DELETE. Note that an INSERT containing an ON CONFLICT clause cannot be used on tables that have either INSERT or UPDATE rules. Consider using an updatable view instead.

table_name

The name (optionally schema-qualified) of the table or view the rule applies to.

condition

Any SQL conditional expression (returning boolean). The condition expression cannot refer to any tables except NEW and OLD, and cannot contain aggregate functions.

INSTEAD

INSTEAD indicates that the commands should be executed *instead of* the original command.

ALSO

ALSO indicates that the commands should be executed *in addition to* the original command.

If neither ALSO nor INSTEAD is specified, ALSO is the default.

command

The command or commands that make up the rule action. Valid commands are SELECT, INSERT, UPDATE, DELETE, or NOTIFY.

Within *condition* and *command*, the special table names NEW and OLD can be used to refer to values in the referenced table. NEW is valid in ON INSERT and ON UPDATE rules to refer to the new row being inserted or updated. OLD is valid in ON UPDATE and ON DELETE rules to refer to the existing row being updated or deleted.

Notes

You must be the owner of a table to create or change rules for it.

In a rule for INSERT, UPDATE, or DELETE on a view, you can add a RETURNING clause that emits the view's columns. This clause will be used to compute the outputs if the rule is triggered by an INSERT RETURNING, UPDATE RETURNING, or DELETE RETURNING command respectively. When the rule is triggered by a command without RETURNING, the rule's RETURNING clause will be ignored. The current implementation allows only unconditional INSTEAD rules to contain RETURNING; furthermore there can be at most one RETURNING clause among all the rules for the same event. (This ensures that there is only one candidate RETURNING clause to be used to compute the results.) RETURNING queries on the view will be rejected if there is no RETURNING clause in any available rule.

It is very important to take care to avoid circular rules. For example, though each of the following two rule definitions are accepted by PostgreSQL, the SELECT command would cause PostgreSQL to report an error because of recursive expansion of a rule:

```
CREATE RULE "_RETURN" AS
    ON SELECT TO t1
    DO INSTEAD
        SELECT * FROM t2;

CREATE RULE "_RETURN" AS
    ON SELECT TO t2
    DO INSTEAD
        SELECT * FROM t1;

SELECT * FROM t1;
```

Presently, if a rule action contains a NOTIFY command, the NOTIFY command will be executed unconditionally, that is, the NOTIFY will be issued even if there are not any rows that the rule should apply to. For example, in:

```
CREATE RULE notify_me AS ON UPDATE TO mytable DO ALSO NOTIFY mytable;

UPDATE mytable SET name = 'foo' WHERE id = 42;
```

one NOTIFY event will be sent during the UPDATE, whether or not there are any rows that match the condition `id = 42`. This is an implementation restriction that might be fixed in future releases.

Compatibility

CREATE RULE is a PostgreSQL language extension, as is the entire query rewrite system.

See Also

ALTER RULE, DROP RULE

CREATE SCHEMA

Name

CREATE SCHEMA — define a new schema

Synopsis

```
CREATE SCHEMA schema_name [ AUTHORIZATION role_specification ] [ schema_element [ ... ] ]
CREATE SCHEMA AUTHORIZATION role_specification [ schema_element [ ... ] ]
CREATE SCHEMA IF NOT EXISTS schema_name [ AUTHORIZATION role_specification ]
CREATE SCHEMA IF NOT EXISTS AUTHORIZATION role_specification

where role_specification can be:

    user_name
  | CURRENT_USER
  | SESSION_USER
```

Description

CREATE SCHEMA enters a new schema into the current database. The schema name must be distinct from the name of any existing schema in the current database.

A schema is essentially a namespace: it contains named objects (tables, data types, functions, and operators) whose names can duplicate those of other objects existing in other schemas. Named objects are accessed either by "qualifying" their names with the schema name as a prefix, or by setting a search path that includes the desired schema(s). A CREATE command specifying an unqualified object name creates the object in the current schema (the one at the front of the search path, which can be determined with the function current_schema).

Optionally, CREATE SCHEMA can include subcommands to create objects within the new schema. The subcommands are treated essentially the same as separate commands issued after creating the schema, except that if the AUTHORIZATION clause is used, all the created objects will be owned by that user.

Parameters

schema_name

The name of a schema to be created. If this is omitted, the *user_name* is used as the schema name. The name cannot begin with pg_, as such names are reserved for system schemas.

user_name

The role name of the user who will own the new schema. If omitted, defaults to the user executing the command. To create a schema owned by another role, you must be a direct or indirect member of that role, or be a superuser.

schema_element

An SQL statement defining an object to be created within the schema. Currently, only `CREATE TABLE`, `CREATE VIEW`, `CREATE INDEX`, `CREATE SEQUENCE`, `CREATE TRIGGER` and `GRANT` are accepted as clauses within `CREATE SCHEMA`. Other kinds of objects may be created in separate commands after the schema is created.

`IF NOT EXISTS`

Do nothing (except issuing a notice) if a schema with the same name already exists. *schema_element* subcommands cannot be included when this option is used.

Notes

To create a schema, the invoking user must have the `CREATE` privilege for the current database. (Of course, superusers bypass this check.)

Examples

Create a schema:

```
CREATE SCHEMA myschema;
```

Create a schema for user `joe`; the schema will also be named `joe`:

```
CREATE SCHEMA AUTHORIZATION joe;
```

Create a schema named `test` that will be owned by user `joe`, unless there already is a schema named `test`. (It does not matter whether `joe` owns the pre-existing schema.)

```
CREATE SCHEMA IF NOT EXISTS test AUTHORIZATION joe;
```

Create a schema and create a table and view within it:

```
CREATE SCHEMA hollywood
    CREATE TABLE films (title text, release date, awards text[])
    CREATE VIEW winners AS
        SELECT title, release FROM films WHERE awards IS NOT NULL;
```

Notice that the individual subcommands do not end with semicolons.

The following is an equivalent way of accomplishing the same result:

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.films (title text, release date, awards text[]);
CREATE VIEW hollywood.winners AS
    SELECT title, release FROM hollywood.films WHERE awards IS NOT NULL;
```

Compatibility

The SQL standard allows a `DEFAULT CHARACTER SET` clause in `CREATE SCHEMA`, as well as more subcommand types than are presently accepted by PostgreSQL.

The SQL standard specifies that the subcommands in `CREATE SCHEMA` can appear in any order. The present PostgreSQL implementation does not handle all cases of forward references in subcommands; it might sometimes be necessary to reorder the subcommands in order to avoid forward references.

According to the SQL standard, the owner of a schema always owns all objects within it. PostgreSQL allows schemas to contain objects owned by users other than the schema owner. This can happen only if the schema owner grants the `CREATE` privilege on his schema to someone else, or a superuser chooses to create objects in it.

The `IF NOT EXISTS` option is a PostgreSQL extension.

See Also

ALTER SCHEMA, DROP SCHEMA

CREATE SEQUENCE

Name

CREATE SEQUENCE — define a new sequence generator

Synopsis

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE [ IF NOT EXISTS ] name [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
```

Description

CREATE SEQUENCE creates a new sequence number generator. This involves creating and initializing a new special single-row table with the name *name*. The generator will be owned by the user issuing the command.

If a schema name is given then the sequence is created in the specified schema. Otherwise it is created in the current schema. Temporary sequences exist in a special schema, so a schema name cannot be given when creating a temporary sequence. The sequence name must be distinct from the name of any other sequence, table, index, view, or foreign table in the same schema.

After a sequence is created, you use the functions nextval, currval, and setval to operate on the sequence. These functions are documented in Section 9.16.

Although you cannot update a sequence directly, you can use a query like:

```
SELECT * FROM name;
```

to examine the parameters and current state of a sequence. In particular, the last_value field of the sequence shows the last value allocated by any session. (Of course, this value might be obsolete by the time it's printed, if other sessions are actively doing nextval calls.)

Parameters

TEMPORARY or TEMP

If specified, the sequence object is created only for this session, and is automatically dropped on session exit. Existing permanent sequences with the same name are not visible (in this session) while the temporary sequence exists, unless they are referenced with schema-qualified names.

IF NOT EXISTS

Do not throw an error if a relation with the same name already exists. A notice is issued in this case. Note that there is no guarantee that the existing relation is anything like the sequence that would have been created - it might not even be a sequence.

name

The name (optionally schema-qualified) of the sequence to be created.

increment

The optional clause `INCREMENT BY` *`increment`* specifies which value is added to the current sequence value to create a new value. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is 1.

minvalue
`NO MINVALUE`

The optional clause `MINVALUE` *`minvalue`* determines the minimum value a sequence can generate. If this clause is not supplied or `NO MINVALUE` is specified, then defaults will be used. The defaults are 1 and -2^{63}-1 for ascending and descending sequences, respectively.

maxvalue
`NO MAXVALUE`

The optional clause `MAXVALUE` *`maxvalue`* determines the maximum value for the sequence. If this clause is not supplied or `NO MAXVALUE` is specified, then default values will be used. The defaults are 2^{63}-1 and -1 for ascending and descending sequences, respectively.

start

The optional clause `START WITH` *`start`* allows the sequence to begin anywhere. The default starting value is *`minvalue`* for ascending sequences and *`maxvalue`* for descending ones.

cache

The optional clause `CACHE` *`cache`* specifies how many sequence numbers are to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e., no cache), and this is also the default.

`CYCLE`
`NO CYCLE`

The `CYCLE` option allows the sequence to wrap around when the *`maxvalue`* or *`minvalue`* has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be the *`minvalue`* or *`maxvalue`*, respectively.

If `NO CYCLE` is specified, any calls to `nextval` after the sequence has reached its maximum value will return an error. If neither `CYCLE` or `NO CYCLE` are specified, `NO CYCLE` is the default.

`OWNED BY` *`table_name.column_name`*
`OWNED BY NONE`

The `OWNED BY` option causes the sequence to be associated with a specific table column, such that if that column (or its whole table) is dropped, the sequence will be automatically dropped as well. The specified table must have the same owner and be in the same schema as the sequence. `OWNED BY NONE`, the default, specifies that there is no such association.

Notes

Use `DROP SEQUENCE` to remove a sequence.

Sequences are based on `bigint` arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807).

Because `nextval` and `setval` calls are never rolled back, sequence objects cannot be used if "gapless" assignment of sequence numbers is needed. It is possible to build gapless assignment by using exclusive locking of a table containing a counter; but this solution is much more expensive than sequence objects, especially if many transactions need sequence numbers concurrently.

Unexpected results might be obtained if a *`cache`* setting greater than one is used for a sequence object that will be used concurrently by multiple sessions. Each session will allocate and cache successive sequence values during one access to the sequence object and increase the sequence object's `last_value` accordingly. Then, the next *`cache`*-1 uses of `nextval` within that session simply return the preallocated values without touching the sequence object. So, any numbers allocated but not used within a session will be lost when that session ends, resulting in "holes" in the sequence.

Furthermore, although multiple sessions are guaranteed to allocate distinct sequence values, the values might be generated out of sequence when all the sessions are considered. For example, with a *`cache`* setting of 10, session A might reserve values 1..10 and return `nextval`=1, then session B might reserve values 11..20 and return `nextval`=11 before session A has generated `nextval`=2. Thus, with a *`cache`* setting of one it is safe to assume that `nextval` values are generated sequentially; with a *`cache`* setting greater than one you should only assume that the `nextval` values are all distinct, not that they are generated purely sequentially. Also, `last_value` will reflect the latest value reserved by any session, whether or not it has yet been returned by `nextval`.

Another consideration is that a `setval` executed on such a sequence will not be noticed by other sessions until they have used up any preallocated values they have cached.

Examples

Create an ascending sequence called `serial`, starting at 101:

```
CREATE SEQUENCE serial START 101;
```

Select the next number from this sequence:

```
SELECT nextval('serial');

 nextval
---------
     101
```

Select the next number from this sequence:

```
SELECT nextval('serial');

 nextval
---------
     102
```

Use this sequence in an `INSERT` command:

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

Update the sequence value after a `COPY FROM`:

```
BEGIN;
COPY distributors FROM 'input_file';
SELECT setval('serial', max(id)) FROM distributors;
END;
```

Compatibility

`CREATE SEQUENCE` conforms to the SQL standard, with the following exceptions:

- The standard's `AS <data type>` expression is not supported.
- Obtaining the next value is done using the `nextval()` function instead of the standard's `NEXT VALUE FOR` expression.
- The `OWNED BY` clause is a PostgreSQL extension.

See Also

ALTER SEQUENCE, DROP SEQUENCE

CREATE SERVER

Name

CREATE SERVER — define a new foreign server

Synopsis

```
CREATE SERVER server_name [ TYPE 'server_type' ] [ VERSION 'server_version' ]
    FOREIGN DATA WRAPPER fdw_name
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

Description

CREATE SERVER defines a new foreign server. The user who defines the server becomes its owner.

A foreign server typically encapsulates connection information that a foreign-data wrapper uses to access an external data resource. Additional user-specific connection information may be specified by means of user mappings.

The server name must be unique within the database.

Creating a server requires USAGE privilege on the foreign-data wrapper being used.

Parameters

server_name

The name of the foreign server to be created.

server_type

Optional server type, potentially useful to foreign-data wrappers.

server_version

Optional server version, potentially useful to foreign-data wrappers.

fdw_name

The name of the foreign-data wrapper that manages the server.

OPTIONS (*option* '*value*' [, ...])

This clause specifies the options for the server. The options typically define the connection details of the server, but the actual names and values are dependent on the server's foreign-data wrapper.

Notes

When using the dblink module, a foreign server's name can be used as an argument of the dblink_connect function to indicate the connection parameters. It is necessary to have the `USAGE` privilege on the foreign server to be able to use it in this way.

Examples

Create a server `myserver` that uses the foreign-data wrapper `postgres_fdw`:

```
CREATE SERVER myserver FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'foo', dbname 'foodb
```

See postgres_fdw for more details.

Compatibility

`CREATE SERVER` conforms to ISO/IEC 9075-9 (SQL/MED).

See Also

ALTER SERVER, DROP SERVER, CREATE FOREIGN DATA WRAPPER, CREATE FOREIGN TABLE, CREATE USER MAPPING

CREATE TABLE

Name

CREATE TABLE — define a new table

Synopsis

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_
  { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_
    OF type_name [ (
  { column_name WITH OPTIONS [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]

where column_constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

and table_constraint is:

[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  EXCLUDE [ USING index_method ] ( exclude_element WITH operator [, ... ] ) index_parameters [ W
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ]
```

```
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

and *like_option* is:

```
{ INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES | STORAGE | COMMENTS | ALL }
```

index_parameters in UNIQUE, PRIMARY KEY, and EXCLUDE constraints are:

```
[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]
```

exclude_element in an EXCLUDE constraint is:

```
{ column_name | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

Description

CREATE TABLE will create a new, initially empty table in the current database. The table will be owned by the user issuing the command.

If a schema name is given (for example, CREATE TABLE myschema.mytable ...) then the table is created in the specified schema. Otherwise it is created in the current schema. Temporary tables exist in a special schema, so a schema name cannot be given when creating a temporary table. The name of the table must be distinct from the name of any other table, sequence, index, view, or foreign table in the same schema.

CREATE TABLE also automatically creates a data type that represents the composite type corresponding to one row of the table. Therefore, tables cannot have the same name as any existing data type in the same schema.

The optional constraint clauses specify constraints (tests) that new or updated rows must satisfy for an insert or update operation to succeed. A constraint is an SQL object that helps define the set of valid values in the table in various ways.

There are two ways to define constraints: table constraints and column constraints. A column constraint is defined as part of a column definition. A table constraint definition is not tied to a particular column, and it can encompass more than one column. Every column constraint can also be written as a table constraint; a column constraint is only a notational convenience for use when the constraint only affects one column.

To be able to create a table, you must have USAGE privilege on all column types or the type in the OF clause, respectively.

Parameters

TEMPORARY or TEMP

If specified, the table is created as a temporary table. Temporary tables are automatically dropped at the end of a session, or optionally at the end of the current transaction (see ON COMMIT below). Existing permanent tables with the same name are not visible to the current session while the tempo-

rary table exists, unless they are referenced with schema-qualified names. Any indexes created on a temporary table are automatically temporary as well.

The autovacuum daemon cannot access and therefore cannot vacuum or analyze temporary tables. For this reason, appropriate vacuum and analyze operations should be performed via session SQL commands. For example, if a temporary table is going to be used in complex queries, it is wise to run `ANALYZE` on the temporary table after it is populated.

Optionally, `GLOBAL` or `LOCAL` can be written before `TEMPORARY` or `TEMP`. This presently makes no difference in PostgreSQL and is deprecated; see *Compatibility*.

`UNLOGGED`

If specified, the table is created as an unlogged table. Data written to unlogged tables is not written to the write-ahead log (see Chapter 29), which makes them considerably faster than ordinary tables. However, they are not crash-safe: an unlogged table is automatically truncated after a crash or unclean shutdown. The contents of an unlogged table are also not replicated to standby servers. Any indexes created on an unlogged table are automatically unlogged as well.

`IF NOT EXISTS`

Do not throw an error if a relation with the same name already exists. A notice is issued in this case. Note that there is no guarantee that the existing relation is anything like the one that would have been created.

`table_name`

The name (optionally schema-qualified) of the table to be created.

`OF` *`type_name`*

Creates a *typed table*, which takes its structure from the specified composite type (name optionally schema-qualified). A typed table is tied to its type; for example the table will be dropped if the type is dropped (with `DROP TYPE ... CASCADE`).

When a typed table is created, then the data types of the columns are determined by the underlying composite type and are not specified by the `CREATE TABLE` command. But the `CREATE TABLE` command can add defaults and constraints to the table and can specify storage parameters.

`column_name`

The name of a column to be created in the new table.

`data_type`

The data type of the column. This can include array specifiers. For more information on the data types supported by PostgreSQL, refer to Chapter 8.

`COLLATE` *`collation`*

The `COLLATE` clause assigns a collation to the column (which must be of a collatable data type). If not specified, the column data type's default collation is used.

`INHERITS ( parent_table [, ... ] )`

The optional `INHERITS` clause specifies a list of tables from which the new table automatically inherits all columns. Parent tables can be plain tables or foreign tables.

Use of INHERITS creates a persistent relationship between the new child table and its parent table(s). Schema modifications to the parent(s) normally propagate to children as well, and by default the data of the child table is included in scans of the parent(s).

If the same column name exists in more than one parent table, an error is reported unless the data types of the columns match in each of the parent tables. If there is no conflict, then the duplicate columns are merged to form a single column in the new table. If the column name list of the new table contains a column name that is also inherited, the data type must likewise match the inherited column(s), and the column definitions are merged into one. If the new table explicitly specifies a default value for the column, this default overrides any defaults from inherited declarations of the column. Otherwise, any parents that specify default values for the column must all specify the same default, or an error will be reported.

CHECK constraints are merged in essentially the same way as columns: if multiple parent tables and/or the new table definition contain identically-named CHECK constraints, these constraints must all have the same check expression, or an error will be reported. Constraints having the same name and expression will be merged into one copy. A constraint marked NO INHERIT in a parent will not be considered. Notice that an unnamed CHECK constraint in the new table will never be merged, since a unique name will always be chosen for it.

Column STORAGE settings are also copied from parent tables.

LIKE *source_table* [*like_option* ...]

The LIKE clause specifies a table from which the new table automatically copies all column names, their data types, and their not-null constraints.

Unlike INHERITS, the new table and original table are completely decoupled after creation is complete. Changes to the original table will not be applied to the new table, and it is not possible to include data of the new table in scans of the original table.

Default expressions for the copied column definitions will be copied only if INCLUDING DEFAULTS is specified. The default behavior is to exclude default expressions, resulting in the copied columns in the new table having null defaults. Note that copying defaults that call database-modification functions, such as nextval, may create a functional linkage between the original and new tables.

Not-null constraints are always copied to the new table. CHECK constraints will be copied only if INCLUDING CONSTRAINTS is specified. No distinction is made between column constraints and table constraints.

Indexes, PRIMARY KEY, UNIQUE, and EXCLUDE constraints on the original table will be created on the new table only if INCLUDING INDEXES is specified. Names for the new indexes and constraints are chosen according to the default rules, regardless of how the originals were named. (This behavior avoids possible duplicate-name failures for the new indexes.)

STORAGE settings for the copied column definitions will be copied only if INCLUDING STORAGE is specified. The default behavior is to exclude STORAGE settings, resulting in the copied columns in the new table having type-specific default settings. For more on STORAGE settings, see Section 63.2.

Comments for the copied columns, constraints, and indexes will be copied only if INCLUDING COMMENTS is specified. The default behavior is to exclude comments, resulting in the copied columns and constraints in the new table having no comments.

INCLUDING ALL is an abbreviated form of INCLUDING DEFAULTS INCLUDING CONSTRAINTS INCLUDING INDEXES INCLUDING STORAGE INCLUDING COMMENTS.

Note that unlike `INHERITS`, columns and constraints copied by `LIKE` are not merged with similarly named columns and constraints. If the same name is specified explicitly or in another `LIKE` clause, an error is signaled.

The `LIKE` clause can also be used to copy column definitions from views, foreign tables, or composite types. Inapplicable options (e.g., `INCLUDING INDEXES` from a view) are ignored.

`CONSTRAINT` *`constraint_name`*

An optional name for a column or table constraint. If the constraint is violated, the constraint name is present in error messages, so constraint names like `col must be positive` can be used to communicate helpful constraint information to client applications. (Double-quotes are needed to specify constraint names that contain spaces.) If a constraint name is not specified, the system generates a name.

`NOT NULL`

The column is not allowed to contain null values.

`NULL`

The column is allowed to contain null values. This is the default.

This clause is only provided for compatibility with non-standard SQL databases. Its use is discouraged in new applications.

`CHECK (` *`expression`* `) [ NO INHERIT ]`

The `CHECK` clause specifies an expression producing a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to TRUE or UNKNOWN succeed. Should any row of an insert or update operation produce a FALSE result, an error exception is raised and the insert or update does not alter the database. A check constraint specified as a column constraint should reference that column's value only, while an expression appearing in a table constraint can reference multiple columns.

Currently, `CHECK` expressions cannot contain subqueries nor refer to variables other than columns of the current row (see Section 5.3.1). The system column `tableoid` may be referenced, but not any other system column.

A constraint marked with `NO INHERIT` will not propagate to child tables.

When a table has multiple `CHECK` constraints, they will be tested for each row in alphabetical order by name, after checking `NOT NULL` constraints. (PostgreSQL versions before 9.5 did not honor any particular firing order for `CHECK` constraints.)

`DEFAULT` *`default_expr`*

The `DEFAULT` clause assigns a default data value for the column whose column definition it appears within. The value is any variable-free expression (subqueries and cross-references to other columns in the current table are not allowed). The data type of the default expression must match the data type of the column.

The default expression will be used in any insert operation that does not specify a value for the column. If there is no default for a column, then the default is null.

`UNIQUE` (column constraint)
`UNIQUE ( column_name [, ... ] )` (table constraint)

The `UNIQUE` constraint specifies that a group of one or more columns of a table can contain only unique values. The behavior of the unique table constraint is the same as that for column constraints, with the additional capability to span multiple columns.

For the purpose of a unique constraint, null values are not considered equal.

Each unique table constraint must name a set of columns that is different from the set of columns named by any other unique or primary key constraint defined for the table. (Otherwise it would just be the same constraint listed twice.)

`PRIMARY KEY` (column constraint)
`PRIMARY KEY ( column_name [, ... ] )` (table constraint)

The `PRIMARY KEY` constraint specifies that a column or columns of a table can contain only unique (non-duplicate), nonnull values. Only one primary key can be specified for a table, whether as a column constraint or a table constraint.

The primary key constraint should name a set of columns that is different from the set of columns named by any unique constraint defined for the same table. (Otherwise, the unique constraint is redundant and will be discarded.)

`PRIMARY KEY` enforces the same data constraints as a combination of `UNIQUE` and `NOT NULL`, but identifying a set of columns as the primary key also provides metadata about the design of the schema, since a primary key implies that other tables can rely on this set of columns as a unique identifier for rows.

`EXCLUDE [ USING index_method ] ( exclude_element WITH operator [, ... ] )`
`index_parameters [ WHERE ( predicate ) ]`

The `EXCLUDE` clause defines an exclusion constraint, which guarantees that if any two rows are compared on the specified column(s) or expression(s) using the specified operator(s), not all of these comparisons will return `TRUE`. If all of the specified operators test for equality, this is equivalent to a `UNIQUE` constraint, although an ordinary unique constraint will be faster. However, exclusion constraints can specify constraints that are more general than simple equality. For example, you can specify a constraint that no two rows in the table contain overlapping circles (see Section 8.8) by using the `&&` operator.

Exclusion constraints are implemented using an index, so each specified operator must be associated with an appropriate operator class (see Section 11.9) for the index access method *index_method*. The operators are required to be commutative. Each *exclude_element* can optionally specify an operator class and/or ordering options; these are described fully under CREATE INDEX.

The access method must support `amgettuple` (see Chapter 58); at present this means GIN cannot be used. Although it's allowed, there is little point in using B-tree or hash indexes with an exclusion constraint, because this does nothing that an ordinary unique constraint doesn't do better. So in practice the access method will always be GiST or SP-GiST.

The *predicate* allows you to specify an exclusion constraint on a subset of the table; internally this creates a partial index. Note that parentheses are required around the predicate.

`REFERENCES` *`reftable`* `[ ( ` *`refcolumn`* ` ) ] [ MATCH` *`matchtype`* `] [ ON DELETE` *`action`* `] [ ON UPDATE` *`action`* `]` (column constraint)
`FOREIGN KEY ( ` *`column_name`* `[, ... ] ) REFERENCES` *`reftable`* `[ ( ` *`refcolumn`* `[, ... ] ) ] [ MATCH` *`matchtype`* `] [ ON DELETE` *`action`* `] [ ON UPDATE` *`action`* `]` (table constraint)

These clauses specify a foreign key constraint, which requires that a group of one or more columns of the new table must only contain values that match values in the referenced column(s) of some row of the referenced table. If the *`refcolumn`* list is omitted, the primary key of the *`reftable`* is used. The referenced columns must be the columns of a non-deferrable unique or primary key constraint in the referenced table. Note that foreign key constraints cannot be defined between temporary tables and permanent tables.

A value inserted into the referencing column(s) is matched against the values of the referenced table and referenced columns using the given match type. There are three match types: `MATCH FULL`, `MATCH PARTIAL`, and `MATCH SIMPLE` (which is the default). `MATCH FULL` will not allow one column of a multicolumn foreign key to be null unless all foreign key columns are null; if they are all null, the row is not required to have a match in the referenced table. `MATCH SIMPLE` allows any of the foreign key columns to be null; if any of them are null, the row is not required to have a match in the referenced table. `MATCH PARTIAL` is not yet implemented. (Of course, `NOT NULL` constraints can be applied to the referencing column(s) to prevent these cases from arising.)

In addition, when the data in the referenced columns is changed, certain actions are performed on the data in this table's columns. The `ON DELETE` clause specifies the action to perform when a referenced row in the referenced table is being deleted. Likewise, the `ON UPDATE` clause specifies the action to perform when a referenced column in the referenced table is being updated to a new value. If the row is updated, but the referenced column is not actually changed, no action is done. Referential actions other than the `NO ACTION` check cannot be deferred, even if the constraint is declared deferrable. There are the following possible actions for each clause:

`NO ACTION`

Produce an error indicating that the deletion or update would create a foreign key constraint violation. If the constraint is deferred, this error will be produced at constraint check time if there still exist any referencing rows. This is the default action.

`RESTRICT`

Produce an error indicating that the deletion or update would create a foreign key constraint violation. This is the same as `NO ACTION` except that the check is not deferrable.

`CASCADE`

Delete any rows referencing the deleted row, or update the values of the referencing column(s) to the new values of the referenced columns, respectively.

`SET NULL`

Set the referencing column(s) to null.

`SET DEFAULT`

Set the referencing column(s) to their default values. (There must be a row in the referenced table matching the default values, if they are not null, or the operation will fail.)

If the referenced column(s) are changed frequently, it might be wise to add an index to the referencing column(s) so that referential actions associated with the foreign key constraint can be performed more efficiently.

`DEFERRABLE`
`NOT DEFERRABLE`

This controls whether the constraint can be deferred. A constraint that is not deferrable will be checked immediately after every command. Checking of constraints that are deferrable can be postponed until the end of the transaction (using the SET CONSTRAINTS command). `NOT DEFERRABLE` is the default. Currently, only `UNIQUE`, `PRIMARY KEY`, `EXCLUDE`, and `REFERENCES` (foreign key) constraints accept this clause. `NOT NULL` and `CHECK` constraints are not deferrable. Note that deferrable constraints cannot be used as conflict arbitrators in an `INSERT` statement that includes an `ON CONFLICT DO UPDATE` clause.

`INITIALLY IMMEDIATE`
`INITIALLY DEFERRED`

If a constraint is deferrable, this clause specifies the default time to check the constraint. If the constraint is `INITIALLY IMMEDIATE`, it is checked after each statement. This is the default. If the constraint is `INITIALLY DEFERRED`, it is checked only at the end of the transaction. The constraint check time can be altered with the SET CONSTRAINTS command.

`WITH ( storage_parameter [= value] [, ... ] )`

This clause specifies optional storage parameters for a table or index; see *Storage Parameters* for more information. The `WITH` clause for a table can also include `OIDS=TRUE` (or just `OIDS`) to specify that rows of the new table should have OIDs (object identifiers) assigned to them, or `OIDS=FALSE` to specify that the rows should not have OIDs. If `OIDS` is not specified, the default setting depends upon the default_with_oids configuration parameter. (If the new table inherits from any tables that have OIDs, then `OIDS=TRUE` is forced even if the command says `OIDS=FALSE`.)

If `OIDS=FALSE` is specified or implied, the new table does not store OIDs and no OID will be assigned for a row inserted into it. This is generally considered worthwhile, since it will reduce OID consumption and thereby postpone the wraparound of the 32-bit OID counter. Once the counter wraps around, OIDs can no longer be assumed to be unique, which makes them considerably less useful. In addition, excluding OIDs from a table reduces the space required to store the table on disk by 4 bytes per row (on most machines), slightly improving performance.

To remove OIDs from a table after it has been created, use ALTER TABLE.

`WITH OIDS`
`WITHOUT OIDS`

These are obsolescent syntaxes equivalent to `WITH (OIDS)` and `WITH (OIDS=FALSE)`, respectively. If you wish to give both an `OIDS` setting and storage parameters, you must use the `WITH ( ... )` syntax; see above.

`ON COMMIT`

The behavior of temporary tables at the end of a transaction block can be controlled using `ON COMMIT`. The three options are:

`PRESERVE ROWS`

No special action is taken at the ends of transactions. This is the default behavior.

`DELETE ROWS`

All rows in the temporary table will be deleted at the end of each transaction block. Essentially, an automatic TRUNCATE is done at each commit.

`DROP`

The temporary table will be dropped at the end of the current transaction block.

`TABLESPACE` *`tablespace_name`*

The *`tablespace_name`* is the name of the tablespace in which the new table is to be created. If not specified, default_tablespace is consulted, or temp_tablespaces if the table is temporary.

`USING INDEX TABLESPACE` *`tablespace_name`*

This clause allows selection of the tablespace in which the index associated with a `UNIQUE`, `PRIMARY KEY`, or `EXCLUDE` constraint will be created. If not specified, default_tablespace is consulted, or temp_tablespaces if the table is temporary.

Storage Parameters

The `WITH` clause can specify *storage parameters* for tables, and for indexes associated with a `UNIQUE`, `PRIMARY KEY`, or `EXCLUDE` constraint. Storage parameters for indexes are documented in CREATE INDEX. The storage parameters currently available for tables are listed below. For many of these parameters, as shown, there is an additional parameter with the same name prefixed with `toast.`, which controls the behavior of the table's secondary TOAST table, if any (see Section 63.2 for more information about TOAST). If a table parameter value is set and the equivalent `toast.` parameter is not, the TOAST table will use the table's parameter value.

`fillfactor` (`integer`)

The fillfactor for a table is a percentage between 10 and 100. 100 (complete packing) is the default. When a smaller fillfactor is specified, `INSERT` operations pack table pages only to the indicated percentage; the remaining space on each page is reserved for updating rows on that page. This gives `UPDATE` a chance to place the updated copy of a row on the same page as the original, which is more efficient than placing it on a different page. For a table whose entries are never updated, complete packing is the best choice, but in heavily updated tables smaller fillfactors are appropriate. This parameter cannot be set for TOAST tables.

`autovacuum_enabled`, `toast.autovacuum_enabled` (`boolean`)

Enables or disables the autovacuum daemon for a particular table. If true, the autovacuum daemon will perform automatic `VACUUM` and/or `ANALYZE` operations on this table following the rules discussed in Section 23.1.6. If false, this table will not be autovacuumed, except to prevent transaction ID wraparound. See Section 23.1.5 for more about wraparound prevention. Note that the autovacuum daemon does not run at all (except to prevent transaction ID wraparound) if the autovacuum parameter is false; setting individual tables' storage parameters does not override that. Therefore there is seldom much point in explicitly setting this storage parameter to `true`, only to `false`.

`autovacuum_vacuum_threshold`, `toast.autovacuum_vacuum_threshold` (`integer`)

Per-table value for autovacuum_vacuum_threshold parameter.

`autovacuum_vacuum_scale_factor`, `toast.autovacuum_vacuum_scale_factor` (`floating point`)

Per-table value for autovacuum_vacuum_scale_factor parameter.

`autovacuum_analyze_threshold` (`integer`)

Per-table value for autovacuum_analyze_threshold parameter.

`autovacuum_analyze_scale_factor` (`floating point`)

Per-table value for autovacuum_analyze_scale_factor parameter.

`autovacuum_vacuum_cost_delay`, `toast.autovacuum_vacuum_cost_delay` (`integer`)

Per-table value for autovacuum_vacuum_cost_delay parameter.

`autovacuum_vacuum_cost_limit`, `toast.autovacuum_vacuum_cost_limit` (`integer`)

Per-table value for autovacuum_vacuum_cost_limit parameter.

`autovacuum_freeze_min_age`, `toast.autovacuum_freeze_min_age` (`integer`)

Per-table value for vacuum_freeze_min_age parameter. Note that autovacuum will ignore per-table `autovacuum_freeze_min_age` parameters that are larger than half the system-wide autovacuum_freeze_max_age setting.

`autovacuum_freeze_max_age`, `toast.autovacuum_freeze_max_age` (`integer`)

Per-table value for autovacuum_freeze_max_age parameter. Note that autovacuum will ignore per-table `autovacuum_freeze_max_age` parameters that are larger than the system-wide setting (it can only be set smaller).

`autovacuum_freeze_table_age`, `toast.autovacuum_freeze_table_age` (`integer`)

Per-table value for vacuum_freeze_table_age parameter.

`autovacuum_multixact_freeze_min_age`, `toast.autovacuum_multixact_freeze_min_age` (`integer`)

Per-table value for vacuum_multixact_freeze_min_age parameter. Note that autovacuum will ignore per-table `autovacuum_multixact_freeze_min_age` parameters that are larger than half the system-wide autovacuum_multixact_freeze_max_age setting.

`autovacuum_multixact_freeze_max_age`, `toast.autovacuum_multixact_freeze_max_age` (`integer`)

Per-table value for autovacuum_multixact_freeze_max_age parameter. Note that autovacuum will ignore per-table `autovacuum_multixact_freeze_max_age` parameters that are larger than the system-wide setting (it can only be set smaller).

`autovacuum_multixact_freeze_table_age`, `toast.autovacuum_multixact_freeze_table_age` (`integer`)

Per-table value for vacuum_multixact_freeze_table_age parameter.

`log_autovacuum_min_duration`, `toast.log_autovacuum_min_duration` (`integer`)

Per-table value for log_autovacuum_min_duration parameter.

`user_catalog_table` (`boolean`)

Declare the table as an additional catalog table for purposes of logical replication. See Section 46.6.2 for details. This parameter cannot be set for TOAST tables.

Notes

Using OIDs in new applications is not recommended: where possible, using a `SERIAL` or other sequence generator as the table's primary key is preferred. However, if your application does make use of OIDs to identify specific rows of a table, it is recommended to create a unique constraint on the `oid` column of that table, to ensure that OIDs in the table will indeed uniquely identify rows even after counter wraparound. Avoid assuming that OIDs are unique across tables; if you need a database-wide unique identifier, use the combination of `tableoid` and row OID for the purpose.

Tip: The use of `OIDS=FALSE` is not recommended for tables with no primary key, since without either an OID or a unique data key, it is difficult to identify specific rows.

PostgreSQL automatically creates an index for each unique constraint and primary key constraint to enforce uniqueness. Thus, it is not necessary to create an index explicitly for primary key columns. (See CREATE INDEX for more information.)

Unique constraints and primary keys are not inherited in the current implementation. This makes the combination of inheritance and unique constraints rather dysfunctional.

A table cannot have more than 1600 columns. (In practice, the effective limit is usually lower because of tuple-length constraints.)

Examples

Create table `films` and table `distributors`:

```
CREATE TABLE films (
    code        char(5) CONSTRAINT firstkey PRIMARY KEY,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
);

CREATE TABLE distributors (
     did    integer PRIMARY KEY DEFAULT nextval('serial'),
     name   varchar(40) NOT NULL CHECK (name <> '')
);
```

Create a table with a 2-dimensional array:

```
CREATE TABLE array_int (
    vector  int[][]
);
```

Define a unique table constraint for the table `films`. Unique table constraints can be defined on one or more columns of the table:

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT production UNIQUE(date_prod)
);
```

Define a check column constraint:

```
CREATE TABLE distributors (
    did     integer CHECK (did > 100),
    name    varchar(40)
);
```

Define a check table constraint:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    CONSTRAINT con1 CHECK (did > 100 AND name <> '')
);
```

Define a primary key table constraint for the table `films`:

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

Define a primary key constraint for table `distributors`. The following two examples are equivalent, the first using the table constraint syntax, the second the column constraint syntax:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    PRIMARY KEY(did)
);

CREATE TABLE distributors (
    did     integer PRIMARY KEY,
    name    varchar(40)
);
```

Assign a literal constant default value for the column `name`, arrange for the default value of column `did` to be generated by selecting the next value of a sequence object, and make the default value of `modtime` be the time at which the row is inserted:

```
CREATE TABLE distributors (
    name      varchar(40) DEFAULT 'Luso Films',
    did       integer DEFAULT nextval('distributors_serial'),
    modtime   timestamp DEFAULT current_timestamp
);
```

Define two `NOT NULL` column constraints on the table `distributors`, one of which is explicitly given a name:

```
CREATE TABLE distributors (
    did     integer CONSTRAINT no_null NOT NULL,
    name    varchar(40) NOT NULL
);
```

Define a unique constraint for the `name` column:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40) UNIQUE
);
```

The same, specified as a table constraint:

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name)
);
```

Create the same table, specifying 70% fill factor for both the table and its unique index:

```
CREATE TABLE distributors (
```

```
    did     integer,
    name    varchar(40),
    UNIQUE(name) WITH (fillfactor=70)
)
WITH (fillfactor=70);
```

Create table `circles` with an exclusion constraint that prevents any two circles from overlapping:

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

Create table `cinemas` in tablespace `diskvol1`:

```
CREATE TABLE cinemas (
        id serial,
        name text,
        location text
) TABLESPACE diskvol1;
```

Create a composite type and a typed table:

```
CREATE TYPE employee_type AS (name text, salary numeric);

CREATE TABLE employees OF employee_type (
    PRIMARY KEY (name),
    salary WITH OPTIONS DEFAULT 1000
);
```

Compatibility

The `CREATE TABLE` command conforms to the SQL standard, with exceptions listed below.

Temporary Tables

Although the syntax of `CREATE TEMPORARY TABLE` resembles that of the SQL standard, the effect is not the same. In the standard, temporary tables are defined just once and automatically exist (starting with empty contents) in every session that needs them. PostgreSQL instead requires each session to issue its own `CREATE TEMPORARY TABLE` command for each temporary table to be used. This allows different sessions to use the same temporary table name for different purposes, whereas the standard's approach constrains all instances of a given temporary table name to have the same table structure.

The standard's definition of the behavior of temporary tables is widely ignored. PostgreSQL's behavior on this point is similar to that of several other SQL databases.

The SQL standard also distinguishes between global and local temporary tables, where a local temporary table has a separate set of contents for each SQL module within each session, though its definition is still shared across sessions. Since PostgreSQL does not support SQL modules, this distinction is not relevant in PostgreSQL.

For compatibility's sake, PostgreSQL will accept the `GLOBAL` and `LOCAL` keywords in a temporary table declaration, but they currently have no effect. Use of these keywords is discouraged, since future versions of PostgreSQL might adopt a more standard-compliant interpretation of their meaning.

The `ON COMMIT` clause for temporary tables also resembles the SQL standard, but has some differences. If the `ON COMMIT` clause is omitted, SQL specifies that the default behavior is `ON COMMIT DELETE ROWS`. However, the default behavior in PostgreSQL is `ON COMMIT PRESERVE ROWS`. The `ON COMMIT DROP` option does not exist in SQL.

Non-deferred Uniqueness Constraints

When a `UNIQUE` or `PRIMARY KEY` constraint is not deferrable, PostgreSQL checks for uniqueness immediately whenever a row is inserted or modified. The SQL standard says that uniqueness should be enforced only at the end of the statement; this makes a difference when, for example, a single command updates multiple key values. To obtain standard-compliant behavior, declare the constraint as `DEFERRABLE` but not deferred (i.e., `INITIALLY IMMEDIATE`). Be aware that this can be significantly slower than immediate uniqueness checking.

Column Check Constraints

The SQL standard says that `CHECK` column constraints can only refer to the column they apply to; only `CHECK` table constraints can refer to multiple columns. PostgreSQL does not enforce this restriction; it treats column and table check constraints alike.

`EXCLUDE` Constraint

The `EXCLUDE` constraint type is a PostgreSQL extension.

`NULL` "Constraint"

The `NULL` "constraint" (actually a non-constraint) is a PostgreSQL extension to the SQL standard that is included for compatibility with some other database systems (and for symmetry with the `NOT NULL` constraint). Since it is the default for any column, its presence is simply noise.

Inheritance

Multiple inheritance via the `INHERITS` clause is a PostgreSQL language extension. SQL:1999 and later define single inheritance using a different syntax and different semantics. SQL:1999-style inheritance is not yet supported by PostgreSQL.

Zero-column Tables

PostgreSQL allows a table of no columns to be created (for example, `CREATE TABLE foo();`). This is an extension from the SQL standard, which does not allow zero-column tables. Zero-column tables are not in themselves very useful, but disallowing them creates odd special cases for `ALTER TABLE DROP COLUMN`, so it seems cleaner to ignore this spec restriction.

`LIKE` Clause

While a `LIKE` clause exists in the SQL standard, many of the options that PostgreSQL accepts for it are not in the standard, and some of the standard's options are not implemented by PostgreSQL.

`WITH` Clause

The `WITH` clause is a PostgreSQL extension; neither storage parameters nor OIDs are in the standard.

Tablespaces

The PostgreSQL concept of tablespaces is not part of the standard. Hence, the clauses `TABLESPACE` and `USING INDEX TABLESPACE` are extensions.

Typed Tables

Typed tables implement a subset of the SQL standard. According to the standard, a typed table has columns corresponding to the underlying composite type as well as one other column that is the "self-referencing column". PostgreSQL does not support these self-referencing columns explicitly, but the same effect can be had using the OID feature.

See Also

ALTER TABLE, DROP TABLE, CREATE TABLE AS, CREATE TABLESPACE, CREATE TYPE

CREATE TABLE AS

Name

CREATE TABLE AS — define a new table from the results of a query

Synopsis

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_
    [ (column_name [, ...] ) ]
    [ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
    [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
    [ TABLESPACE tablespace_name ]
    AS query
    [ WITH [ NO ] DATA ]
```

Description

CREATE TABLE AS creates a table and fills it with data computed by a SELECT command. The table columns have the names and data types associated with the output columns of the SELECT (except that you can override the column names by giving an explicit list of new column names).

CREATE TABLE AS bears some resemblance to creating a view, but it is really quite different: it creates a new table and evaluates the query just once to fill the new table initially. The new table will not track subsequent changes to the source tables of the query. In contrast, a view re-evaluates its defining SELECT statement whenever it is queried.

Parameters

GLOBAL or LOCAL

Ignored for compatibility. Use of these keywords is deprecated; refer to CREATE TABLE for details.

TEMPORARY or TEMP

If specified, the table is created as a temporary table. Refer to CREATE TABLE for details.

UNLOGGED

If specified, the table is created as an unlogged table. Refer to CREATE TABLE for details.

IF NOT EXISTS

Do not throw an error if a relation with the same name already exists. A notice is issued in this case. Refer to CREATE TABLE for details.

table_name

The name (optionally schema-qualified) of the table to be created.

column_name

The name of a column in the new table. If column names are not provided, they are taken from the output column names of the query.

WITH (*storage_parameter* [= *value*] [, ...])

This clause specifies optional storage parameters for the new table; see *Storage Parameters* for more information. The WITH clause can also include OIDS=TRUE (or just OIDS) to specify that rows of the new table should have OIDs (object identifiers) assigned to them, or OIDS=FALSE to specify that the rows should not have OIDs. See CREATE TABLE for more information.

WITH OIDS
WITHOUT OIDS

These are obsolescent syntaxes equivalent to WITH (OIDS) and WITH (OIDS=FALSE), respectively. If you wish to give both an OIDS setting and storage parameters, you must use the WITH (...) syntax; see above.

ON COMMIT

The behavior of temporary tables at the end of a transaction block can be controlled using ON COMMIT. The three options are:

PRESERVE ROWS

No special action is taken at the ends of transactions. This is the default behavior.

DELETE ROWS

All rows in the temporary table will be deleted at the end of each transaction block. Essentially, an automatic TRUNCATE is done at each commit.

DROP

The temporary table will be dropped at the end of the current transaction block.

TABLESPACE *tablespace_name*

The *tablespace_name* is the name of the tablespace in which the new table is to be created. If not specified, default_tablespace is consulted, or temp_tablespaces if the table is temporary.

query

A SELECT, TABLE, or VALUES command, or an EXECUTE command that runs a prepared SELECT, TABLE, or VALUES query.

WITH [NO] DATA

This clause specifies whether or not the data produced by the query should be copied into the new table. If not, only the table structure is copied. The default is to copy the data.

Notes

This command is functionally similar to SELECT INTO, but it is preferred since it is less likely to be confused with other uses of the SELECT INTO syntax. Furthermore, CREATE TABLE AS offers a superset of the functionality offered by SELECT INTO.

The `CREATE TABLE AS` command allows the user to explicitly specify whether OIDs should be included. If the presence of OIDs is not explicitly specified, the default_with_oids configuration variable is used.

Examples

Create a new table `films_recent` consisting of only recent entries from the table `films`:

```
CREATE TABLE films_recent AS
  SELECT * FROM films WHERE date_prod >= '2002-01-01';
```

To copy a table completely, the short form using the `TABLE` command can also be used:

```
CREATE TABLE films2 AS
  TABLE films;
```

Create a new temporary table `films_recent`, consisting of only recent entries from the table `films`, using a prepared statement. The new table has OIDs and will be dropped at commit:

```
PREPARE recentfilms(date) AS
  SELECT * FROM films WHERE date_prod > $1;
CREATE TEMP TABLE films_recent WITH (OIDS) ON COMMIT DROP AS
  EXECUTE recentfilms('2002-01-01');
```

Compatibility

`CREATE TABLE AS` conforms to the SQL standard. The following are nonstandard extensions:

- The standard requires parentheses around the subquery clause; in PostgreSQL, these parentheses are optional.
- In the standard, the `WITH [ NO ] DATA` clause is required; in PostgreSQL it is optional.
- PostgreSQL handles temporary tables in a way rather different from the standard; see CREATE TABLE for details.
- The `WITH` clause is a PostgreSQL extension; neither storage parameters nor OIDs are in the standard.
- The PostgreSQL concept of tablespaces is not part of the standard. Hence, the clause `TABLESPACE` is an extension.

See Also

CREATE MATERIALIZED VIEW, CREATE TABLE, EXECUTE, SELECT, SELECT INTO, VALUES

CREATE TABLESPACE

Name

CREATE TABLESPACE — define a new tablespace

Synopsis

```
CREATE TABLESPACE tablespace_name
    [ OWNER { new_owner | CURRENT_USER | SESSION_USER } ]
    LOCATION 'directory'
    [ WITH ( tablespace_option = value [, ... ] ) ]
```

Description

CREATE TABLESPACE registers a new cluster-wide tablespace. The tablespace name must be distinct from the name of any existing tablespace in the database cluster.

A tablespace allows superusers to define an alternative location on the file system where the data files containing database objects (such as tables and indexes) can reside.

A user with appropriate privileges can pass *tablespace_name* to CREATE DATABASE, CREATE TABLE, CREATE INDEX or ADD CONSTRAINT to have the data files for these objects stored within the specified tablespace.

> **Warning**
>
> A tablespace cannot be used independently of the cluster in which it is defined; see Section 21.6.

Parameters

tablespace_name

The name of a tablespace to be created. The name cannot begin with pg_, as such names are reserved for system tablespaces.

user_name

The name of the user who will own the tablespace. If omitted, defaults to the user executing the command. Only superusers can create tablespaces, but they can assign ownership of tablespaces to non-superusers.

directory

The directory that will be used for the tablespace. The directory should be empty and must be owned by the PostgreSQL system user. The directory must be specified by an absolute path name.

tablespace_option

A tablespace parameter to be set or reset. Currently, the only available parameters are `seq_page_cost` and `random_page_cost`. Setting either value for a particular tablespace will override the planner's usual estimate of the cost of reading pages from tables in that tablespace, as established by the configuration parameters of the same name (see seq_page_cost, random_page_cost). This may be useful if one tablespace is located on a disk which is faster or slower than the remainder of the I/O subsystem.

Notes

Tablespaces are only supported on systems that support symbolic links.

`CREATE TABLESPACE` cannot be executed inside a transaction block.

Examples

Create a tablespace `dbspace` at `/data/dbs`:

```
CREATE TABLESPACE dbspace LOCATION '/data/dbs';
```

Create a tablespace `indexspace` at `/data/indexes` owned by user `genevieve`:

```
CREATE TABLESPACE indexspace OWNER genevieve LOCATION '/data/indexes';
```

Compatibility

`CREATE TABLESPACE` is a PostgreSQL extension.

See Also

CREATE DATABASE, CREATE TABLE, CREATE INDEX, DROP TABLESPACE, ALTER TABLESPACE

CREATE TEXT SEARCH CONFIGURATION

Name

CREATE TEXT SEARCH CONFIGURATION — define a new text search configuration

Synopsis

```
CREATE TEXT SEARCH CONFIGURATION name (
    PARSER = parser_name |
    COPY = source_config
)
```

Description

CREATE TEXT SEARCH CONFIGURATION creates a new text search configuration. A text search configuration specifies a text search parser that can divide a string into tokens, plus dictionaries that can be used to determine which tokens are of interest for searching.

If only the parser is specified, then the new text search configuration initially has no mappings from token types to dictionaries, and therefore will ignore all words. Subsequent ALTER TEXT SEARCH CONFIGURATION commands must be used to create mappings to make the configuration useful. Alternatively, an existing text search configuration can be copied.

If a schema name is given then the text search configuration is created in the specified schema. Otherwise it is created in the current schema.

The user who defines a text search configuration becomes its owner.

Refer to Chapter 12 for further information.

Parameters

name

The name of the text search configuration to be created. The name can be schema-qualified.

parser_name

The name of the text search parser to use for this configuration.

source_config

The name of an existing text search configuration to copy.

Notes

The `PARSER` and `COPY` options are mutually exclusive, because when an existing configuration is copied, its parser selection is copied too.

Compatibility

There is no `CREATE TEXT SEARCH CONFIGURATION` statement in the SQL standard.

See Also

ALTER TEXT SEARCH CONFIGURATION, DROP TEXT SEARCH CONFIGURATION

CREATE TEXT SEARCH DICTIONARY

Name

CREATE TEXT SEARCH DICTIONARY — define a new text search dictionary

Synopsis

```
CREATE TEXT SEARCH DICTIONARY name (
    TEMPLATE = template
    [, option = value [, ... ]]
)
```

Description

CREATE TEXT SEARCH DICTIONARY creates a new text search dictionary. A text search dictionary specifies a way of recognizing interesting or uninteresting words for searching. A dictionary depends on a text search template, which specifies the functions that actually perform the work. Typically the dictionary provides some options that control the detailed behavior of the template's functions.

If a schema name is given then the text search dictionary is created in the specified schema. Otherwise it is created in the current schema.

The user who defines a text search dictionary becomes its owner.

Refer to Chapter 12 for further information.

Parameters

name

The name of the text search dictionary to be created. The name can be schema-qualified.

template

The name of the text search template that will define the basic behavior of this dictionary.

option

The name of a template-specific option to be set for this dictionary.

value

The value to use for a template-specific option. If the value is not a simple identifier or number, it must be quoted (but you can always quote it, if you wish).

The options can appear in any order.

Examples

The following example command creates a Snowball-based dictionary with a nonstandard list of stop words.

```
CREATE TEXT SEARCH DICTIONARY my_russian (
    template = snowball,
    language = russian,
    stopwords = myrussian
);
```

Compatibility

There is no `CREATE TEXT SEARCH DICTIONARY` statement in the SQL standard.

See Also

ALTER TEXT SEARCH DICTIONARY, DROP TEXT SEARCH DICTIONARY

CREATE TEXT SEARCH PARSER

Name

CREATE TEXT SEARCH PARSER — define a new text search parser

Synopsis

```
CREATE TEXT SEARCH PARSER name (
    START = start_function ,
    GETTOKEN = gettoken_function ,
    END = end_function ,
    LEXTYPES = lextypes_function
    [, HEADLINE = headline_function ]
)
```

Description

CREATE TEXT SEARCH PARSER creates a new text search parser. A text search parser defines a method for splitting a text string into tokens and assigning types (categories) to the tokens. A parser is not particularly useful by itself, but must be bound into a text search configuration along with some text search dictionaries to be used for searching.

If a schema name is given then the text search parser is created in the specified schema. Otherwise it is created in the current schema.

You must be a superuser to use CREATE TEXT SEARCH PARSER. (This restriction is made because an erroneous text search parser definition could confuse or even crash the server.)

Refer to Chapter 12 for further information.

Parameters

name

The name of the text search parser to be created. The name can be schema-qualified.

start_function

The name of the start function for the parser.

gettoken_function

The name of the get-next-token function for the parser.

end_function

The name of the end function for the parser.

lextypes_function

The name of the lextypes function for the parser (a function that returns information about the set of token types it produces).

headline_function

The name of the headline function for the parser (a function that summarizes a set of tokens).

The function names can be schema-qualified if necessary. Argument types are not given, since the argument list for each type of function is predetermined. All except the headline function are required.

The arguments can appear in any order, not only the one shown above.

Compatibility

There is no `CREATE TEXT SEARCH PARSER` statement in the SQL standard.

See Also

ALTER TEXT SEARCH PARSER, DROP TEXT SEARCH PARSER

CREATE TEXT SEARCH TEMPLATE

Name

CREATE TEXT SEARCH TEMPLATE — define a new text search template

Synopsis

```
CREATE TEXT SEARCH TEMPLATE name (
    [ INIT = init_function , ]
    LEXIZE = lexize_function
)
```

Description

CREATE TEXT SEARCH TEMPLATE creates a new text search template. Text search templates define the functions that implement text search dictionaries. A template is not useful by itself, but must be instantiated as a dictionary to be used. The dictionary typically specifies parameters to be given to the template functions.

If a schema name is given then the text search template is created in the specified schema. Otherwise it is created in the current schema.

You must be a superuser to use CREATE TEXT SEARCH TEMPLATE. This restriction is made because an erroneous text search template definition could confuse or even crash the server. The reason for separating templates from dictionaries is that a template encapsulates the "unsafe" aspects of defining a dictionary. The parameters that can be set when defining a dictionary are safe for unprivileged users to set, and so creating a dictionary need not be a privileged operation.

Refer to Chapter 12 for further information.

Parameters

name

The name of the text search template to be created. The name can be schema-qualified.

init_function

The name of the init function for the template.

lexize_function

The name of the lexize function for the template.

The function names can be schema-qualified if necessary. Argument types are not given, since the argument list for each type of function is predetermined. The lexize function is required, but the init function is optional.

The arguments can appear in any order, not only the one shown above.

Compatibility

There is no `CREATE TEXT SEARCH TEMPLATE` statement in the SQL standard.

See Also

ALTER TEXT SEARCH TEMPLATE, DROP TEXT SEARCH TEMPLATE

CREATE TRANSFORM

Name

CREATE TRANSFORM — define a new transform

Synopsis

```
CREATE [ OR REPLACE ] TRANSFORM FOR type_name LANGUAGE lang_name (
    FROM SQL WITH FUNCTION from_sql_function_name (argument_type [, ...]),
    TO SQL WITH FUNCTION to_sql_function_name (argument_type [, ...])
);
```

Description

CREATE TRANSFORM defines a new transform. CREATE OR REPLACE TRANSFORM will either create a new transform, or replace an existing definition.

A transform specifies how to adapt a data type to a procedural language. For example, when writing a function in PL/Python using the hstore type, PL/Python has no prior knowledge how to present hstore values in the Python environment. Language implementations usually default to using the text representation, but that is inconvenient when, for example, an associative array or a list would be more appropriate.

A transform specifies two functions:

- A "from SQL" function that converts the type from the SQL environment to the language. This function will be invoked on the arguments of a function written in the language.
- A "to SQL" function that converts the type from the language to the SQL environment. This function will be invoked on the return value of a function written in the language.

It is not necessary to provide both of these functions. If one is not specified, the language-specific default behavior will be used if necessary. (To prevent a transformation in a certain direction from happening at all, you could also write a transform function that always errors out.)

To be able to create a transform, you must own and have USAGE privilege on the type, have USAGE privilege on the language, and own and have EXECUTE privilege on the from-SQL and to-SQL functions, if specified.

Parameters

type_name

The name of the data type of the transform.

lang_name

The name of the language of the transform.

from_sql_function_name(*argument_type* [, ...])

The name of the function for converting the type from the SQL environment to the language. It must take one argument of type `internal` and return type `internal`. The actual argument will be of the type for the transform, and the function should be coded as if it were. (But it is not allowed to declare an SQL-level function returning `internal` without at least one argument of type `internal`.) The actual return value will be something specific to the language implementation.

to_sql_function_name(*argument_type* [, ...])

The name of the function for converting the type from the language to the SQL environment. It must take one argument of type `internal` and return the type that is the type for the transform. The actual argument value will be something specific to the language implementation.

Notes

Use DROP TRANSFORM to remove transforms.

Examples

To create a transform for type `hstore` and language `plpythonu`, first set up the type and the language:

```
CREATE TYPE hstore ...;

CREATE LANGUAGE plpythonu ...;
```

Then create the necessary functions:

```
CREATE FUNCTION hstore_to_plpython(val internal) RETURNS internal
LANGUAGE C STRICT IMMUTABLE
AS ...;

CREATE FUNCTION plpython_to_hstore(val internal) RETURNS hstore
LANGUAGE C STRICT IMMUTABLE
AS ...;
```

And finally create the transform to connect them all together:

```
CREATE TRANSFORM FOR hstore LANGUAGE plpythonu (
    FROM SQL WITH FUNCTION hstore_to_plpython(internal),
    TO SQL WITH FUNCTION plpython_to_hstore(internal)
);
```

In practice, these commands would be wrapped up in extensions.

The `contrib` section contains a number of extensions that provide transforms, which can serve as real-world examples.

Compatibility

This form of `CREATE TRANSFORM` is a PostgreSQL extension. There is a `CREATE TRANSFORM` command in the SQL standard, but it is for adapting data types to client languages. That usage is not supported by PostgreSQL.

See Also

CREATE FUNCTION, CREATE LANGUAGE, CREATE TYPE, DROP TRANSFORM

CREATE TRIGGER

Name

CREATE TRIGGER — define a new trigger

Synopsis

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments )

where event can be one of:

    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

Description

CREATE TRIGGER creates a new trigger. The trigger will be associated with the specified table, view, or foreign table and will execute the specified function *function_name* when certain events occur.

The trigger can be specified to fire before the operation is attempted on a row (before constraints are checked and the INSERT, UPDATE, or DELETE is attempted); or after the operation has completed (after constraints are checked and the INSERT, UPDATE, or DELETE has completed); or instead of the operation (in the case of inserts, updates or deletes on a view). If the trigger fires before or instead of the event, the trigger can skip the operation for the current row, or change the row being inserted (for INSERT and UPDATE operations only). If the trigger fires after the event, all changes, including the effects of other triggers, are "visible" to the trigger.

A trigger that is marked FOR EACH ROW is called once for every row that the operation modifies. For example, a DELETE that affects 10 rows will cause any ON DELETE triggers on the target relation to be called 10 separate times, once for each deleted row. In contrast, a trigger that is marked FOR EACH STATEMENT only executes once for any given operation, regardless of how many rows it modifies (in particular, an operation that modifies zero rows will still result in the execution of any applicable FOR EACH STATEMENT triggers). Note that with an INSERT with an ON CONFLICT DO UPDATE clause, both INSERT and UPDATE statement level trigger will be fired.

Triggers that are specified to fire INSTEAD OF the trigger event must be marked FOR EACH ROW, and can only be defined on views. BEFORE and AFTER triggers on a view must be marked as FOR EACH STATEMENT.

In addition, triggers may be defined to fire for TRUNCATE, though only FOR EACH STATEMENT.

The following table summarizes which types of triggers may be used on tables, views, and foreign tables:

When	Event	Row-level	Statement-level
BEFORE	INSERT/UPDATE/DELETE	Tables and foreign tables	Tables, views, and foreign tables
	TRUNCATE	—	Tables
AFTER	INSERT/UPDATE/DELETE	Tables and foreign tables	Tables, views, and foreign tables
	TRUNCATE	—	Tables
INSTEAD OF	INSERT/UPDATE/DELETE	Views	—
	TRUNCATE	—	—

Also, a trigger definition can specify a Boolean WHEN condition, which will be tested to see whether the trigger should be fired. In row-level triggers the WHEN condition can examine the old and/or new values of columns of the row. Statement-level triggers can also have WHEN conditions, although the feature is not so useful for them since the condition cannot refer to any values in the table.

If multiple triggers of the same kind are defined for the same event, they will be fired in alphabetical order by name.

When the CONSTRAINT option is specified, this command creates a *constraint trigger*. This is the same as a regular trigger except that the timing of the trigger firing can be adjusted using SET CONSTRAINTS. Constraint triggers must be AFTER ROW triggers on tables. They can be fired either at the end of the statement causing the triggering event, or at the end of the containing transaction; in the latter case they are said to be *deferred*. A pending deferred-trigger firing can also be forced to happen immediately by using SET CONSTRAINTS. Constraint triggers are expected to raise an exception when the constraints they implement are violated.

SELECT does not modify any rows so you cannot create SELECT triggers. Rules and views are more appropriate in such cases.

Refer to Chapter 36 for more information about triggers.

Parameters

name

The name to give the new trigger. This must be distinct from the name of any other trigger for the same table. The name cannot be schema-qualified — the trigger inherits the schema of its table. For a constraint trigger, this is also the name to use when modifying the trigger's behavior using SET CONSTRAINTS.

`BEFORE`
`AFTER`
`INSTEAD OF`

Determines whether the function is called before, after, or instead of the event. A constraint trigger can only be specified as `AFTER`.

`event`

One of `INSERT`, `UPDATE`, `DELETE`, or `TRUNCATE`; this specifies the event that will fire the trigger. Multiple events can be specified using `OR`.

For `UPDATE` events, it is possible to specify a list of columns using this syntax:

`UPDATE OF` *`column_name1`* `[,` *`column_name2`* `... ]`
The trigger will only fire if at least one of the listed columns is mentioned as a target of the `UPDATE` command.

`INSTEAD OF UPDATE` events do not support lists of columns.

`table_name`

The name (optionally schema-qualified) of the table, view, or foreign table the trigger is for.

`referenced_table_name`

The (possibly schema-qualified) name of another table referenced by the constraint. This option is used for foreign-key constraints and is not recommended for general use. This can only be specified for constraint triggers.

`DEFERRABLE`
`NOT DEFERRABLE`
`INITIALLY IMMEDIATE`
`INITIALLY DEFERRED`

The default timing of the trigger. See the CREATE TABLE documentation for details of these constraint options. This can only be specified for constraint triggers.

`FOR EACH ROW`
`FOR EACH STATEMENT`

This specifies whether the trigger procedure should be fired once for every row affected by the trigger event, or just once per SQL statement. If neither is specified, `FOR EACH STATEMENT` is the default. Constraint triggers can only be specified `FOR EACH ROW`.

`condition`

A Boolean expression that determines whether the trigger function will actually be executed. If `WHEN` is specified, the function will only be called if the *`condition`* returns `true`. In `FOR EACH ROW` triggers, the `WHEN` condition can refer to columns of the old and/or new row values by writing `OLD.`*`column_name`* or `NEW.`*`column_name`* respectively. Of course, `INSERT` triggers cannot refer to `OLD` and `DELETE` triggers cannot refer to `NEW`.

`INSTEAD OF` triggers do not support `WHEN` conditions.

Currently, `WHEN` expressions cannot contain subqueries.

Note that for constraint triggers, evaluation of the `WHEN` condition is not deferred, but occurs immediately after the row update operation is performed. If the condition does not evaluate to true then the trigger is not queued for deferred execution.

function_name

A user-supplied function that is declared as taking no arguments and returning type `trigger`, which is executed when the trigger fires.

arguments

An optional comma-separated list of arguments to be provided to the function when the trigger is executed. The arguments are literal string constants. Simple names and numeric constants can be written here, too, but they will all be converted to strings. Please check the description of the implementation language of the trigger function to find out how these arguments can be accessed within the function; it might be different from normal function arguments.

Notes

To create a trigger on a table, the user must have the `TRIGGER` privilege on the table. The user must also have `EXECUTE` privilege on the trigger function.

Use DROP TRIGGER to remove a trigger.

A column-specific trigger (one defined using the `UPDATE OF` *column_name* syntax) will fire when any of its columns are listed as targets in the `UPDATE` command's `SET` list. It is possible for a column's value to change even when the trigger is not fired, because changes made to the row's contents by `BEFORE UPDATE` triggers are not considered. Conversely, a command such as `UPDATE ... SET x = x ...` will fire a trigger on column `x`, even though the column's value did not change.

There are a few built-in trigger functions that can be used to solve common problems without having to write your own trigger code; see Section 9.27.

In a `BEFORE` trigger, the `WHEN` condition is evaluated just before the function is or would be executed, so using `WHEN` is not materially different from testing the same condition at the beginning of the trigger function. Note in particular that the `NEW` row seen by the condition is the current value, as possibly modified by earlier triggers. Also, a `BEFORE` trigger's `WHEN` condition is not allowed to examine the system columns of the `NEW` row (such as `oid`), because those won't have been set yet.

In an `AFTER` trigger, the `WHEN` condition is evaluated just after the row update occurs, and it determines whether an event is queued to fire the trigger at the end of statement. So when an `AFTER` trigger's `WHEN` condition does not return true, it is not necessary to queue an event nor to re-fetch the row at end of statement. This can result in significant speedups in statements that modify many rows, if the trigger only needs to be fired for a few of the rows.

Statement-level triggers on a view are fired only if the action on the view is handled by a row-level `INSTEAD OF` trigger. If the action is handled by an `INSTEAD` rule, then whatever statements are emitted by the rule are executed in place of the original statement naming the view, so that the triggers that will be fired are those on tables named in the replacement statements. Similarly, if the view is automatically updatable, then the action is handled by automatically rewriting the statement into an action on the view's base table, so that the base table's statement-level triggers are the ones that are fired.

In PostgreSQL versions before 7.3, it was necessary to declare trigger functions as returning the placeholder type `opaque`, rather than `trigger`. To support loading of old dump files, `CREATE TRIGGER` will accept a function declared as returning `opaque`, but it will issue a notice and change the function's declared return type to `trigger`.

Examples

Execute the function `check_account_update` whenever a row of the table `accounts` is about to be updated:

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    EXECUTE PROCEDURE check_account_update();
```

The same, but only execute the function if column `balance` is specified as a target in the `UPDATE` command:

```
CREATE TRIGGER check_update
    BEFORE UPDATE OF balance ON accounts
    FOR EACH ROW
    EXECUTE PROCEDURE check_account_update();
```

This form only executes the function if column `balance` has in fact changed value:

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
    EXECUTE PROCEDURE check_account_update();
```

Call a function to log updates of `accounts`, but only if something changed:

```
CREATE TRIGGER log_update
    AFTER UPDATE ON accounts
    FOR EACH ROW
    WHEN (OLD.* IS DISTINCT FROM NEW.*)
    EXECUTE PROCEDURE log_account_update();
```

Execute the function `view_insert_row` for each row to insert rows into the tables underlying a view:

```
CREATE TRIGGER view_insert
    INSTEAD OF INSERT ON my_view
    FOR EACH ROW
    EXECUTE PROCEDURE view_insert_row();
```

Section 36.4 contains a complete example of a trigger function written in C.

Compatibility

The `CREATE TRIGGER` statement in PostgreSQL implements a subset of the SQL standard. The following functionalities are currently missing:

- SQL allows you to define aliases for the “old” and “new” rows or tables for use in the definition of the triggered action (e.g., `CREATE TRIGGER ... ON tablename REFERENCING OLD ROW AS`

`somename NEW ROW AS othername ...`). Since PostgreSQL allows trigger procedures to be written in any number of user-defined languages, access to the data is handled in a language-specific way.

- PostgreSQL does not allow the old and new tables to be referenced in statement-level triggers, i.e., the tables that contain all the old and/or new rows, which are referred to by the `OLD TABLE` and `NEW TABLE` clauses in the SQL standard.
- PostgreSQL only allows the execution of a user-defined function for the triggered action. The standard allows the execution of a number of other SQL commands, such as `CREATE TABLE`, as the triggered action. This limitation is not hard to work around by creating a user-defined function that executes the desired commands.

SQL specifies that multiple triggers should be fired in time-of-creation order. PostgreSQL uses name order, which was judged to be more convenient.

SQL specifies that `BEFORE DELETE` triggers on cascaded deletes fire *after* the cascaded `DELETE` completes. The PostgreSQL behavior is for `BEFORE DELETE` to always fire before the delete action, even a cascading one. This is considered more consistent. There is also nonstandard behavior if `BEFORE` triggers modify rows or prevent updates during an update that is caused by a referential action. This can lead to constraint violations or stored data that does not honor the referential constraint.

The ability to specify multiple actions for a single trigger using `OR` is a PostgreSQL extension of the SQL standard.

The ability to fire triggers for `TRUNCATE` is a PostgreSQL extension of the SQL standard, as is the ability to define statement-level triggers on views.

`CREATE CONSTRAINT TRIGGER` is a PostgreSQL extension of the SQL standard.

See Also

ALTER TRIGGER, DROP TRIGGER, CREATE FUNCTION, SET CONSTRAINTS

CREATE TYPE

Name

CREATE TYPE — define a new data type

Synopsis

```
CREATE TYPE name AS
    ( [ attribute_name data_type [ COLLATE collation ] [, ... ] ] )

CREATE TYPE name AS ENUM
    ( [ 'label' [, ... ] ] )

CREATE TYPE name AS RANGE (
    SUBTYPE = subtype
    [ , SUBTYPE_OPCLASS = subtype_operator_class ]
    [ , COLLATION = collation ]
    [ , CANONICAL = canonical_function ]
    [ , SUBTYPE_DIFF = subtype_diff_function ]
)

CREATE TYPE name (
    INPUT = input_function,
    OUTPUT = output_function
    [ , RECEIVE = receive_function ]
    [ , SEND = send_function ]
    [ , TYPMOD_IN = type_modifier_input_function ]
    [ , TYPMOD_OUT = type_modifier_output_function ]
    [ , ANALYZE = analyze_function ]
    [ , INTERNALLENGTH = { internallength | VARIABLE } ]
    [ , PASSEDBYVALUE ]
    [ , ALIGNMENT = alignment ]
    [ , STORAGE = storage ]
    [ , LIKE = like_type ]
    [ , CATEGORY = category ]
    [ , PREFERRED = preferred ]
    [ , DEFAULT = default ]
    [ , ELEMENT = element ]
    [ , DELIMITER = delimiter ]
    [ , COLLATABLE = collatable ]
)

CREATE TYPE name
```

Description

CREATE TYPE registers a new data type for use in the current database. The user who defines a type becomes its owner.

If a schema name is given then the type is created in the specified schema. Otherwise it is created in the current schema. The type name must be distinct from the name of any existing type or domain in the same schema. (Because tables have associated data types, the type name must also be distinct from the name of any existing table in the same schema.)

There are five forms of CREATE TYPE, as shown in the syntax synopsis above. They respectively create a *composite type*, an *enum type*, a *range type*, a *base type*, or a *shell type*. The first four of these are discussed in turn below. A shell type is simply a placeholder for a type to be defined later; it is created by issuing CREATE TYPE with no parameters except for the type name. Shell types are needed as forward references when creating range types and base types, as discussed in those sections.

Composite Types

The first form of CREATE TYPE creates a composite type. The composite type is specified by a list of attribute names and data types. An attribute's collation can be specified too, if its data type is collatable. A composite type is essentially the same as the row type of a table, but using CREATE TYPE avoids the need to create an actual table when all that is wanted is to define a type. A stand-alone composite type is useful, for example, as the argument or return type of a function.

To be able to create a composite type, you must have USAGE privilege on all attribute types.

Enumerated Types

The second form of CREATE TYPE creates an enumerated (enum) type, as described in Section 8.7. Enum types take a list of quoted labels, each of which must be less than NAMEDATALEN bytes long (64 bytes in a standard PostgreSQL build). (It is possible to create an enumerated type with zero labels, but such a type cannot be used to hold values before at least one label is added using ALTER TYPE.)

Range Types

The third form of CREATE TYPE creates a new range type, as described in Section 8.17.

The range type's *subtype* can be any type with an associated b-tree operator class (to determine the ordering of values for the range type). Normally the subtype's default b-tree operator class is used to determine ordering; to use a non-default operator class, specify its name with *subtype_opclass*. If the subtype is collatable, and you want to use a non-default collation in the range's ordering, specify the desired collation with the *collation* option.

The optional *canonical* function must take one argument of the range type being defined, and return a value of the same type. This is used to convert range values to a canonical form, when applicable. See Section 8.17.8 for more information. Creating a *canonical* function is a bit tricky, since it must be defined before the range type can be declared. To do this, you must first create a shell type, which is a placeholder type that has no properties except a name and an owner. This is done by issuing the command CREATE TYPE *name*, with no additional parameters. Then the function can be declared using

the shell type as argument and result, and finally the range type can be declared using the same name. This automatically replaces the shell type entry with a valid range type.

The optional *subtype_diff* function must take two values of the *subtype* type as argument, and return a `double precision` value representing the difference between the two given values. While this is optional, providing it allows much greater efficiency of GiST indexes on columns of the range type. See Section 8.17.8 for more information.

Base Types

The fourth form of `CREATE TYPE` creates a new base type (scalar type). To create a new base type, you must be a superuser. (This restriction is made because an erroneous type definition could confuse or even crash the server.)

The parameters can appear in any order, not only that illustrated above, and most are optional. You must register two or more functions (using `CREATE FUNCTION`) before defining the type. The support functions *input_function* and *output_function* are required, while the functions *receive_function*, *send_function*, *type_modifier_input_function*, *type_modifier_output_function* and *analyze_function* are optional. Generally these functions have to be coded in C or another low-level language.

The *input_function* converts the type's external textual representation to the internal representation used by the operators and functions defined for the type. *output_function* performs the reverse transformation. The input function can be declared as taking one argument of type `cstring`, or as taking three arguments of types `cstring`, `oid`, `integer`. The first argument is the input text as a C string, the second argument is the type's own OID (except for array types, which instead receive their element type's OID), and the third is the `typmod` of the destination column, if known (-1 will be passed if not). The input function must return a value of the data type itself. Usually, an input function should be declared STRICT; if it is not, it will be called with a NULL first parameter when reading a NULL input value. The function must still return NULL in this case, unless it raises an error. (This case is mainly meant to support domain input functions, which might need to reject NULL inputs.) The output function must be declared as taking one argument of the new data type. The output function must return type `cstring`. Output functions are not invoked for NULL values.

The optional *receive_function* converts the type's external binary representation to the internal representation. If this function is not supplied, the type cannot participate in binary input. The binary representation should be chosen to be cheap to convert to internal form, while being reasonably portable. (For example, the standard integer data types use network byte order as the external binary representation, while the internal representation is in the machine's native byte order.) The receive function should perform adequate checking to ensure that the value is valid. The receive function can be declared as taking one argument of type `internal`, or as taking three arguments of types `internal`, `oid`, `integer`. The first argument is a pointer to a `StringInfo` buffer holding the received byte string; the optional arguments are the same as for the text input function. The receive function must return a value of the data type itself. Usually, a receive function should be declared STRICT; if it is not, it will be called with a NULL first parameter when reading a NULL input value. The function must still return NULL in this case, unless it raises an error. (This case is mainly meant to support domain receive functions, which might need to reject NULL inputs.) Similarly, the optional *send_function* converts from the internal representation to the external binary representation. If this function is not supplied, the type cannot participate in binary output. The send function must be declared as taking one argument of the new data type. The send function must return type `bytea`. Send functions are not invoked for NULL values.

You should at this point be wondering how the input and output functions can be declared to have results or arguments of the new type, when they have to be created before the new type can be created. The answer is that the type should first be defined as a *shell type*, which is a placeholder type that has no properties except a name and an owner. This is done by issuing the command `CREATE TYPE` *`name`*, with no additional parameters. Then the C I/O functions can be defined referencing the shell type. Finally, `CREATE TYPE` with a full definition replaces the shell entry with a complete, valid type definition, after which the new type can be used normally.

The optional *`type_modifier_input_function`* and *`type_modifier_output_function`* are needed if the type supports modifiers, that is optional constraints attached to a type declaration, such as `char(5)` or `numeric(30,2)`. PostgreSQL allows user-defined types to take one or more simple constants or identifiers as modifiers. However, this information must be capable of being packed into a single non-negative integer value for storage in the system catalogs. The *`type_modifier_input_function`* is passed the declared modifier(s) in the form of a `cstring` array. It must check the values for validity (throwing an error if they are wrong), and if they are correct, return a single non-negative `integer` value that will be stored as the column "typmod". Type modifiers will be rejected if the type does not have a *`type_modifier_input_function`*. The *`type_modifier_output_function`* converts the internal integer typmod value back to the correct form for user display. It must return a `cstring` value that is the exact string to append to the type name; for example `numeric`'s function might return `(30,2)`. It is allowed to omit the *`type_modifier_output_function`*, in which case the default display format is just the stored typmod integer value enclosed in parentheses.

The optional *`analyze_function`* performs type-specific statistics collection for columns of the data type. By default, `ANALYZE` will attempt to gather statistics using the type's "equals" and "less-than" operators, if there is a default b-tree operator class for the type. For non-scalar types this behavior is likely to be unsuitable, so it can be overridden by specifying a custom analysis function. The analysis function must be declared to take a single argument of type `internal`, and return a `boolean` result. The detailed API for analysis functions appears in `src/include/commands/vacuum.h`.

While the details of the new type's internal representation are only known to the I/O functions and other functions you create to work with the type, there are several properties of the internal representation that must be declared to PostgreSQL. Foremost of these is *`internallength`*. Base data types can be fixed-length, in which case *`internallength`* is a positive integer, or variable length, indicated by setting *`internallength`* to `VARIABLE`. (Internally, this is represented by setting `typlen` to -1.) The internal representation of all variable-length types must start with a 4-byte integer giving the total length of this value of the type. (Note that the length field is often encoded, as described in Section 63.2; it's unwise to access it directly.)

The optional flag `PASSEDBYVALUE` indicates that values of this data type are passed by value, rather than by reference. Types passed by value must be fixed-length, and their internal representation cannot be larger than the size of the `Datum` type (4 bytes on some machines, 8 bytes on others).

The *`alignment`* parameter specifies the storage alignment required for the data type. The allowed values equate to alignment on 1, 2, 4, or 8 byte boundaries. Note that variable-length types must have an alignment of at least 4, since they necessarily contain an `int4` as their first component.

The *`storage`* parameter allows selection of storage strategies for variable-length data types. (Only `plain` is allowed for fixed-length types.) `plain` specifies that data of the type will always be stored in-line and not compressed. `extended` specifies that the system will first try to compress a long data value, and will move the value out of the main table row if it's still too long. `external` allows the value to be moved out

of the main table, but the system will not try to compress it. `main` allows compression, but discourages moving the value out of the main table. (Data items with this storage strategy might still be moved out of the main table if there is no other way to make a row fit, but they will be kept in the main table preferentially over `extended` and `external` items.)

All *`storage`* values other than `plain` imply that the functions of the data type can handle values that have been *toasted*, as described in Section 63.2 and Section 35.11.1. The specific other value given merely determines the default TOAST storage strategy for columns of a toastable data type; users can pick other strategies for individual columns using `ALTER TABLE SET STORAGE`.

The *`like_type`* parameter provides an alternative method for specifying the basic representation properties of a data type: copy them from some existing type. The values of *`internallength`*, *`passedbyvalue`*, *`alignment`*, and *`storage`* are copied from the named type. (It is possible, though usually undesirable, to override some of these values by specifying them along with the `LIKE` clause.) Specifying representation this way is especially useful when the low-level implementation of the new type "piggybacks" on an existing type in some fashion.

The *`category`* and *`preferred`* parameters can be used to help control which implicit cast will be applied in ambiguous situations. Each data type belongs to a category named by a single ASCII character, and each type is either "preferred" or not within its category. The parser will prefer casting to preferred types (but only from other types within the same category) when this rule is helpful in resolving overloaded functions or operators. For more details see Chapter 10. For types that have no implicit casts to or from any other types, it is sufficient to leave these settings at the defaults. However, for a group of related types that have implicit casts, it is often helpful to mark them all as belonging to a category and select one or two of the "most general" types as being preferred within the category. The *`category`* parameter is especially useful when adding a user-defined type to an existing built-in category, such as the numeric or string types. However, it is also possible to create new entirely-user-defined type categories. Select any ASCII character other than an upper-case letter to name such a category.

A default value can be specified, in case a user wants columns of the data type to default to something other than the null value. Specify the default with the `DEFAULT` key word. (Such a default can be overridden by an explicit `DEFAULT` clause attached to a particular column.)

To indicate that a type is an array, specify the type of the array elements using the `ELEMENT` key word. For example, to define an array of 4-byte integers (`int4`), specify `ELEMENT = int4`. More details about array types appear below.

To indicate the delimiter to be used between values in the external representation of arrays of this type, *`delimiter`* can be set to a specific character. The default delimiter is the comma (`,`). Note that the delimiter is associated with the array element type, not the array type itself.

If the optional Boolean parameter *`collatable`* is true, column definitions and expressions of the type may carry collation information through use of the `COLLATE` clause. It is up to the implementations of the functions operating on the type to actually make use of the collation information; this does not happen automatically merely by marking the type collatable.

Array Types

Whenever a user-defined type is created, PostgreSQL automatically creates an associated array type, whose name consists of the element type's name prepended with an underscore, and truncated if necessary to keep it less than `NAMEDATALEN` bytes long. (If the name so generated collides with an existing

type name, the process is repeated until a non-colliding name is found.) This implicitly-created array type is variable length and uses the built-in input and output functions `array_in` and `array_out`. The array type tracks any changes in its element type's owner or schema, and is dropped if the element type is.

You might reasonably ask why there is an `ELEMENT` option, if the system makes the correct array type automatically. The only case where it's useful to use `ELEMENT` is when you are making a fixed-length type that happens to be internally an array of a number of identical things, and you want to allow these things to be accessed directly by subscripting, in addition to whatever operations you plan to provide for the type as a whole. For example, type `point` is represented as just two floating-point numbers, which can be accessed using `point[0]` and `point[1]`. Note that this facility only works for fixed-length types whose internal form is exactly a sequence of identical fixed-length fields. A subscriptable variable-length type must have the generalized internal representation used by `array_in` and `array_out`. For historical reasons (i.e., this is clearly wrong but it's far too late to change it), subscripting of fixed-length array types starts from zero, rather than from one as for variable-length arrays.

Parameters

name

The name (optionally schema-qualified) of a type to be created.

attribute_name

The name of an attribute (column) for the composite type.

data_type

The name of an existing data type to become a column of the composite type.

collation

The name of an existing collation to be associated with a column of a composite type, or with a range type.

label

A string literal representing the textual label associated with one value of an enum type.

subtype

The name of the element type that the range type will represent ranges of.

subtype_operator_class

The name of a b-tree operator class for the subtype.

canonical_function

The name of the canonicalization function for the range type.

subtype_diff_function

The name of a difference function for the subtype.

input_function

The name of a function that converts data from the type's external textual form to its internal form.

output_function

The name of a function that converts data from the type's internal form to its external textual form.

receive_function

The name of a function that converts data from the type's external binary form to its internal form.

send_function

The name of a function that converts data from the type's internal form to its external binary form.

type_modifier_input_function

The name of a function that converts an array of modifier(s) for the type into internal form.

type_modifier_output_function

The name of a function that converts the internal form of the type's modifier(s) to external textual form.

analyze_function

The name of a function that performs statistical analysis for the data type.

internallength

A numeric constant that specifies the length in bytes of the new type's internal representation. The default assumption is that it is variable-length.

alignment

The storage alignment requirement of the data type. If specified, it must be `char`, `int2`, `int4`, or `double`; the default is `int4`.

storage

The storage strategy for the data type. If specified, must be `plain`, `external`, `extended`, or `main`; the default is `plain`.

like_type

The name of an existing data type that the new type will have the same representation as. The values of *internallength*, *passedbyvalue*, *alignment*, and *storage* are copied from that type, unless overridden by explicit specification elsewhere in this `CREATE TYPE` command.

category

The category code (a single ASCII character) for this type. The default is `'U'` for "user-defined type". Other standard category codes can be found in Table 49-55. You may also choose other ASCII characters in order to create custom categories.

preferred

True if this type is a preferred type within its type category, else false. The default is false. Be very careful about creating a new preferred type within an existing type category, as this could cause surprising changes in behavior.

default

The default value for the data type. If this is omitted, the default is null.

element

The type being created is an array; this specifies the type of the array elements.

delimiter

The delimiter character to be used between values in arrays made of this type.

collatable

True if this type's operations can use collation information. The default is false.

Notes

Because there are no restrictions on use of a data type once it's been created, creating a base type or range type is tantamount to granting public execute permission on the functions mentioned in the type definition. This is usually not an issue for the sorts of functions that are useful in a type definition. But you might want to think twice before designing a type in a way that would require "secret" information to be used while converting it to or from external form.

Before PostgreSQL version 8.3, the name of a generated array type was always exactly the element type's name with one underscore character (_) prepended. (Type names were therefore restricted in length to one less character than other names.) While this is still usually the case, the array type name may vary from this in case of maximum-length names or collisions with user type names that begin with underscore. Writing code that depends on this convention is therefore deprecated. Instead, use `pg_type.typarray` to locate the array type associated with a given type.

It may be advisable to avoid using type and table names that begin with underscore. While the server will change generated array type names to avoid collisions with user-given names, there is still risk of confusion, particularly with old client software that may assume that type names beginning with underscores always represent arrays.

Before PostgreSQL version 8.2, the shell-type creation syntax `CREATE TYPE` *name* did not exist. The way to create a new base type was to create its input function first. In this approach, PostgreSQL will first see the name of the new data type as the return type of the input function. The shell type is implicitly created in this situation, and then it can be referenced in the definitions of the remaining I/O functions. This approach still works, but is deprecated and might be disallowed in some future release. Also, to avoid accidentally cluttering the catalogs with shell types as a result of simple typos in function definitions, a shell type will only be made this way when the input function is written in C.

In PostgreSQL versions before 7.3, it was customary to avoid creating a shell type at all, by replacing the functions' forward references to the type name with the placeholder pseudotype `opaque`. The `cstring` arguments and results also had to be declared as `opaque` before 7.3. To support loading of old dump files, `CREATE TYPE` will accept I/O functions declared using `opaque`, but it will issue a notice and change the function declarations to use the correct types.

Examples

This example creates a composite type and uses it in a function definition:

```
CREATE TYPE compfoo AS (f1 int, f2 text);
```

```
CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
    SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

This example creates an enumerated type and uses it in a table definition:

```
CREATE TYPE bug_status AS ENUM ('new', 'open', 'closed');

CREATE TABLE bug (
    id serial,
    description text,
    status bug_status
);
```

This example creates a range type:

```
CREATE TYPE float8_range AS RANGE (subtype = float8, subtype_diff = float8mi);
```

This example creates the base data type `box` and then uses the type in a table definition:

```
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS ... ;
CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS ... ;

CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

If the internal structure of `box` were an array of four `float4` elements, we might instead use:

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

which would allow a box value's component numbers to be accessed by subscripting. Otherwise the type behaves the same as before.

This example creates a large object type and uses it in a table definition:

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE
);
CREATE TABLE big_objs (
    id integer,
    obj bigobj
);
```

More examples, including suitable input and output functions, are in Section 35.11.

Compatibility

The first form of the `CREATE TYPE` command, which creates a composite type, conforms to the SQL standard. The other forms are PostgreSQL extensions. The `CREATE TYPE` statement in the SQL standard also defines other forms that are not implemented in PostgreSQL.

The ability to create a composite type with zero attributes is a PostgreSQL-specific deviation from the standard (analogous to the same case in `CREATE TABLE`).

See Also

ALTER TYPE, CREATE DOMAIN, CREATE FUNCTION, DROP TYPE

CREATE USER

Name

CREATE USER — define a new database role

Synopsis

```
CREATE USER name [ [ WITH ] option [ ... ] ]

where option can be:

      SUPERUSER | NOSUPERUSER
    | CREATEDB | NOCREATEDB
    | CREATEROLE | NOCREATEROLE
    | CREATEUSER | NOCREATEUSER
    | INHERIT | NOINHERIT
    | LOGIN | NOLOGIN
    | REPLICATION | NOREPLICATION
    | BYPASSRLS | NOBYPASSRLS
    | CONNECTION LIMIT connlimit
    | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
    | VALID UNTIL 'timestamp'
    | IN ROLE role_name [, ...]
    | IN GROUP role_name [, ...]
    | ROLE role_name [, ...]
    | ADMIN role_name [, ...]
    | USER role_name [, ...]
    | SYSID uid
```

Description

CREATE USER is now an alias for CREATE ROLE. The only difference is that when the command is spelled CREATE USER, LOGIN is assumed by default, whereas NOLOGIN is assumed when the command is spelled CREATE ROLE.

Compatibility

The CREATE USER statement is a PostgreSQL extension. The SQL standard leaves the definition of users to the implementation.

See Also

CREATE ROLE

CREATE USER MAPPING

Name

CREATE USER MAPPING — define a new mapping of a user to a foreign server

Synopsis

```
CREATE USER MAPPING FOR { user_name | USER | CURRENT_USER | PUBLIC }
    SERVER server_name
    [ OPTIONS ( option 'value' [ , ... ] ) ]
```

Description

CREATE USER MAPPING defines a mapping of a user to a foreign server. A user mapping typically encapsulates connection information that a foreign-data wrapper uses together with the information encapsulated by a foreign server to access an external data resource.

The owner of a foreign server can create user mappings for that server for any user. Also, a user can create a user mapping for his own user name if USAGE privilege on the server has been granted to the user.

Parameters

user_name

The name of an existing user that is mapped to foreign server. CURRENT_USER and USER match the name of the current user. When PUBLIC is specified, a so-called public mapping is created that is used when no user-specific mapping is applicable.

server_name

The name of an existing server for which the user mapping is to be created.

OPTIONS (*option* '*value*' [, ...])

This clause specifies the options of the user mapping. The options typically define the actual user name and password of the mapping. Option names must be unique. The allowed option names and values are specific to the server's foreign-data wrapper.

Examples

Create a user mapping for user bob, server foo:

```
CREATE USER MAPPING FOR bob SERVER foo OPTIONS (user 'bob', password 'secret');
```

Compatibility

`CREATE USER MAPPING` conforms to ISO/IEC 9075-9 (SQL/MED).

See Also

ALTER USER MAPPING, DROP USER MAPPING, CREATE FOREIGN DATA WRAPPER, CREATE SERVER

CREATE VIEW

Name

CREATE VIEW — define a new view

Synopsis

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name [ ( column_name [, ...] )
    [ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
    AS query
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Description

CREATE VIEW defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.

CREATE OR REPLACE VIEW is similar, but if a view of the same name already exists, it is replaced. The new query must generate the same columns that were generated by the existing view query (that is, the same column names in the same order and with the same data types), but it may add additional columns to the end of the list. The calculations giving rise to the output columns may be completely different.

If a schema name is given (for example, CREATE VIEW myschema.myview ...) then the view is created in the specified schema. Otherwise it is created in the current schema. Temporary views exist in a special schema, so a schema name cannot be given when creating a temporary view. The name of the view must be distinct from the name of any other view, table, sequence, index or foreign table in the same schema.

Parameters

TEMPORARY or TEMP

If specified, the view is created as a temporary view. Temporary views are automatically dropped at the end of the current session. Existing permanent relations with the same name are not visible to the current session while the temporary view exists, unless they are referenced with schema-qualified names.

If any of the tables referenced by the view are temporary, the view is created as a temporary view (whether TEMPORARY is specified or not).

RECURSIVE

Creates a recursive view. The syntax

CREATE RECURSIVE VIEW [*schema* .] *view_name* (*column_names*) AS SELECT ...;
is equivalent to

CREATE VIEW [*schema* .] *view_name* AS WITH RECURSIVE *view_name* (*column_names*) AS (SELECT ..

A view column name list must be specified for a recursive view.

name

The name (optionally schema-qualified) of a view to be created.

column_name

An optional list of names to be used for columns of the view. If not given, the column names are deduced from the query.

WITH (*view_option_name* [= *view_option_value*] [, ...])

This clause specifies optional parameters for a view; the following parameters are supported:

check_option (string)

This parameter may be either local or cascaded, and is equivalent to specifying WITH [CASCADED | LOCAL] CHECK OPTION (see below). This option can be changed on existing views using ALTER VIEW.

security_barrier (boolean)

This should be used if the view is intended to provide row-level security. See Section 38.5 for full details.

query

A SELECT or VALUES command which will provide the columns and rows of the view.

WITH [CASCADED | LOCAL] CHECK OPTION

This option controls the behavior of automatically updatable views. When this option is specified, INSERT and UPDATE commands on the view will be checked to ensure that new rows satisfy the view-defining condition (that is, the new rows are checked to ensure that they are visible through the view). If they are not, the update will be rejected. If the CHECK OPTION is not specified, INSERT and UPDATE commands on the view are allowed to create rows that are not visible through the view. The following check options are supported:

LOCAL

New rows are only checked against the conditions defined directly in the view itself. Any conditions defined on underlying base views are not checked (unless they also specify the CHECK OPTION).

CASCADED

New rows are checked against the conditions of the view and all underlying base views. If the CHECK OPTION is specified, and neither LOCAL nor CASCADED is specified, then CASCADED is assumed.

The CHECK OPTION may not be used with RECURSIVE views.

Note that the CHECK OPTION is only supported on views that are automatically updatable, and do not have INSTEAD OF triggers or INSTEAD rules. If an automatically updatable view is defined on top of a base view that has INSTEAD OF triggers, then the LOCAL CHECK OPTION may be used to check the conditions on the automatically updatable view, but the conditions on the base view

with `INSTEAD OF` triggers will not be checked (a cascaded check option will not cascade down to a trigger-updatable view, and any check options defined directly on a trigger-updatable view will be ignored). If the view or any of its base relations has an `INSTEAD` rule that causes the `INSERT` or `UPDATE` command to be rewritten, then all check options will be ignored in the rewritten query, including any checks from automatically updatable views defined on top of the relation with the `INSTEAD` rule.

Notes

Use the DROP VIEW statement to drop views.

Be careful that the names and types of the view's columns will be assigned the way you want. For example:

```
CREATE VIEW vista AS SELECT 'Hello World';
```

is bad form in two ways: the column name defaults to `?column?`, and the column data type defaults to `unknown`. If you want a string literal in a view's result, use something like:

```
CREATE VIEW vista AS SELECT text 'Hello World' AS hello;
```

Access to tables referenced in the view is determined by permissions of the view owner. In some cases, this can be used to provide secure but restricted access to the underlying tables. However, not all views are secure against tampering; see Section 38.5 for details. Functions called in the view are treated the same as if they had been called directly from the query using the view. Therefore the user of a view must have permissions to call all functions used by the view.

When `CREATE OR REPLACE VIEW` is used on an existing view, only the view's defining SELECT rule is changed. Other view properties, including ownership, permissions, and non-SELECT rules, remain unchanged. You must own the view to replace it (this includes being a member of the owning role).

Updatable Views

Simple views are automatically updatable: the system will allow `INSERT`, `UPDATE` and `DELETE` statements to be used on the view in the same way as on a regular table. A view is automatically updatable if it satisfies all of the following conditions:

- The view must have exactly one entry in its `FROM` list, which must be a table or another updatable view.
- The view definition must not contain `WITH`, `DISTINCT`, `GROUP BY`, `HAVING`, `LIMIT`, or `OFFSET` clauses at the top level.
- The view definition must not contain set operations (`UNION`, `INTERSECT` or `EXCEPT`) at the top level.
- The view's select list must not contain any aggregates, window functions or set-returning functions.

An automatically updatable view may contain a mix of updatable and non-updatable columns. A column is updatable if it is a simple reference to an updatable column of the underlying base relation; otherwise

the column is read-only, and an error will be raised if an INSERT or UPDATE statement attempts to assign a value to it.

If the view is automatically updatable the system will convert any INSERT, UPDATE or DELETE statement on the view into the corresponding statement on the underlying base relation. INSERT statements that have an ON CONFLICT UPDATE clause are fully supported.

If an automatically updatable view contains a WHERE condition, the condition restricts which rows of the base relation are available to be modified by UPDATE and DELETE statements on the view. However, an UPDATE is allowed to change a row so that it no longer satisfies the WHERE condition, and thus is no longer visible through the view. Similarly, an INSERT command can potentially insert base-relation rows that do not satisfy the WHERE condition and thus are not visible through the view (ON CONFLICT UPDATE may similarly affect an existing row not visible through the view). The CHECK OPTION may be used to prevent INSERT and UPDATE commands from creating such rows that are not visible through the view.

If an automatically updatable view is marked with the security_barrier property then all the view's WHERE conditions (and any conditions using operators which are marked as LEAKPROOF) will always be evaluated before any conditions that a user of the view has added. See Section 38.5 for full details. Note that, due to this, rows which are not ultimately returned (because they do not pass the user's WHERE conditions) may still end up being locked. EXPLAIN can be used to see which conditions are applied at the relation level (and therefore do not lock rows) and which are not.

A more complex view that does not satisfy all these conditions is read-only by default: the system will not allow an insert, update, or delete on the view. You can get the effect of an updatable view by creating INSTEAD OF triggers on the view, which must convert attempted inserts, etc. on the view into appropriate actions on other tables. For more information see CREATE TRIGGER. Another possibility is to create rules (see CREATE RULE), but in practice triggers are easier to understand and use correctly.

Note that the user performing the insert, update or delete on the view must have the corresponding insert, update or delete privilege on the view. In addition the view's owner must have the relevant privileges on the underlying base relations, but the user performing the update does not need any permissions on the underlying base relations (see Section 38.5).

Examples

Create a view consisting of all comedy films:

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';
```

This will create a view containing the columns that are in the film table at the time of view creation. Though * was used to create the view, columns added later to the table will not be part of the view.

Create a view with LOCAL CHECK OPTION:

```
CREATE VIEW universal_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'U'
```

```
    WITH LOCAL CHECK OPTION;
```

This will create a view based on the `comedies` view, showing only films with `kind = 'Comedy'` and `classification = 'U'`. Any attempt to `INSERT` or `UPDATE` a row in the view will be rejected if the new row doesn't have `classification = 'U'`, but the film `kind` will not be checked.

Create a view with `CASCADED CHECK OPTION`:

```
CREATE VIEW pg_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'PG'
    WITH CASCADED CHECK OPTION;
```

This will create a view that checks both the `kind` and `classification` of new rows.

Create a view with a mix of updatable and non-updatable columns:

```
CREATE VIEW comedies AS
    SELECT f.*,
           country_code_to_name(f.country_code) AS country,
           (SELECT avg(r.rating)
            FROM user_ratings r
            WHERE r.film_id = f.id) AS avg_rating
    FROM films f
    WHERE f.kind = 'Comedy';
```

This view will support `INSERT`, `UPDATE` and `DELETE`. All the columns from the `films` table will be updatable, whereas the computed columns `country` and `avg_rating` will be read-only.

Create a recursive view consisting of the numbers from 1 to 100:

```
CREATE RECURSIVE VIEW public.nums_1_100 (n) AS
    VALUES (1)
UNION ALL
    SELECT n+1 FROM nums_1_100 WHERE n < 100;
```

Notice that although the recursive view's name is schema-qualified in this `CREATE`, its internal self-reference is not schema-qualified. This is because the implicitly-created CTE's name cannot be schema-qualified.

Compatibility

`CREATE OR REPLACE VIEW` is a PostgreSQL language extension. So is the concept of a temporary view. The `WITH ( ... )` clause is an extension as well.

See Also

ALTER VIEW, DROP VIEW, CREATE MATERIALIZED VIEW

DEALLOCATE

Name

`DEALLOCATE` — deallocate a prepared statement

Synopsis

```
DEALLOCATE [ PREPARE ] { name | ALL }
```

Description

`DEALLOCATE` is used to deallocate a previously prepared SQL statement. If you do not explicitly deallocate a prepared statement, it is deallocated when the session ends.

For more information on prepared statements, see PREPARE.

Parameters

`PREPARE`

This key word is ignored.

`name`

The name of the prepared statement to deallocate.

`ALL`

Deallocate all prepared statements.

Compatibility

The SQL standard includes a `DEALLOCATE` statement, but it is only for use in embedded SQL.

See Also

EXECUTE, PREPARE

DECLARE

Name

DECLARE — define a cursor

Synopsis

```
DECLARE name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
    CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
```

Description

DECLARE allows a user to create cursors, which can be used to retrieve a small number of rows at a time out of a larger query. After the cursor is created, rows are fetched from it using FETCH.

> **Note:** This page describes usage of cursors at the SQL command level. If you are trying to use cursors inside a PL/pgSQL function, the rules are different — see Section 40.7.

Parameters

name

The name of the cursor to be created.

BINARY

Causes the cursor to return data in binary rather than in text format.

INSENSITIVE

Indicates that data retrieved from the cursor should be unaffected by updates to the table(s) underlying the cursor that occur after the cursor is created. In PostgreSQL, this is the default behavior; so this key word has no effect and is only accepted for compatibility with the SQL standard.

SCROLL

NO SCROLL

SCROLL specifies that the cursor can be used to retrieve rows in a nonsequential fashion (e.g., backward). Depending upon the complexity of the query's execution plan, specifying SCROLL might impose a performance penalty on the query's execution time. NO SCROLL specifies that the cursor cannot be used to retrieve rows in a nonsequential fashion. The default is to allow scrolling in some cases; this is not the same as specifying SCROLL. See *Notes* for details.

`WITH HOLD`
`WITHOUT HOLD`

`WITH HOLD` specifies that the cursor can continue to be used after the transaction that created it successfully commits. `WITHOUT HOLD` specifies that the cursor cannot be used outside of the transaction that created it. If neither `WITHOUT HOLD` nor `WITH HOLD` is specified, `WITHOUT HOLD` is the default.

query

A SELECT or VALUES command which will provide the rows to be returned by the cursor.

The key words `BINARY`, `INSENSITIVE`, and `SCROLL` can appear in any order.

Notes

Normal cursors return data in text format, the same as a `SELECT` would produce. The `BINARY` option specifies that the cursor should return data in binary format. This reduces conversion effort for both the server and client, at the cost of more programmer effort to deal with platform-dependent binary data formats. As an example, if a query returns a value of one from an integer column, you would get a string of `1` with a default cursor, whereas with a binary cursor you would get a 4-byte field containing the internal representation of the value (in big-endian byte order).

Binary cursors should be used carefully. Many applications, including psql, are not prepared to handle binary cursors and expect data to come back in the text format.

Note: When the client application uses the "extended query" protocol to issue a `FETCH` command, the Bind protocol message specifies whether data is to be retrieved in text or binary format. This choice overrides the way that the cursor is defined. The concept of a binary cursor as such is thus obsolete when using extended query protocol — any cursor can be treated as either text or binary.

Unless `WITH HOLD` is specified, the cursor created by this command can only be used within the current transaction. Thus, `DECLARE` without `WITH HOLD` is useless outside a transaction block: the cursor would survive only to the completion of the statement. Therefore PostgreSQL reports an error if such a command is used outside a transaction block. Use BEGIN and COMMIT (or ROLLBACK) to define a transaction block.

If `WITH HOLD` is specified and the transaction that created the cursor successfully commits, the cursor can continue to be accessed by subsequent transactions in the same session. (But if the creating transaction is aborted, the cursor is removed.) A cursor created with `WITH HOLD` is closed when an explicit `CLOSE` command is issued on it, or the session ends. In the current implementation, the rows represented by a held cursor are copied into a temporary file or memory area so that they remain available for subsequent transactions.

`WITH HOLD` may not be specified when the query includes `FOR UPDATE` or `FOR SHARE`.

The `SCROLL` option should be specified when defining a cursor that will be used to fetch backwards. This is required by the SQL standard. However, for compatibility with earlier versions, PostgreSQL will allow backward fetches without `SCROLL`, if the cursor's query plan is simple enough that no extra overhead is needed to support it. However, application developers are advised not to rely on using backward fetches from a cursor that has not been created with `SCROLL`. If `NO SCROLL` is specified, then backward fetches are disallowed in any case.

Backward fetches are also disallowed when the query includes `FOR UPDATE` or `FOR SHARE`; therefore `SCROLL` may not be specified in this case.

Caution

Scrollable and `WITH HOLD` cursors may give unexpected results if they invoke any volatile functions (see Section 35.6). When a previously fetched row is re-fetched, the functions might be re-executed, perhaps leading to results different from the first time. One workaround for such cases is to declare the cursor `WITH HOLD` and commit the transaction before reading any rows from it. This will force the entire output of the cursor to be materialized in temporary storage, so that volatile functions are executed exactly once for each row.

If the cursor's query includes `FOR UPDATE` or `FOR SHARE`, then returned rows are locked at the time they are first fetched, in the same way as for a regular SELECT command with these options. In addition, the returned rows will be the most up-to-date versions; therefore these options provide the equivalent of what the SQL standard calls a "sensitive cursor". (Specifying `INSENSITIVE` together with `FOR UPDATE` or `FOR SHARE` is an error.)

Caution

It is generally recommended to use `FOR UPDATE` if the cursor is intended to be used with `UPDATE ... WHERE CURRENT OF` or `DELETE ... WHERE CURRENT OF`. Using `FOR UPDATE` prevents other sessions from changing the rows between the time they are fetched and the time they are updated. Without `FOR UPDATE`, a subsequent `WHERE CURRENT OF` command will have no effect if the row was changed since the cursor was created.

Another reason to use `FOR UPDATE` is that without it, a subsequent `WHERE CURRENT OF` might fail if the cursor query does not meet the SQL standard's rules for being "simply updatable" (in particular, the cursor must reference just one table and not use grouping or `ORDER BY`). Cursors that are not simply updatable might work, or might not, depending on plan choice details; so in the worst case, an application might work in testing and then fail in production. If `FOR UPDATE` is specified, the cursor is guaranteed to be updatable.

The main reason not to use `FOR UPDATE` with `WHERE CURRENT OF` is if you need the cursor to be scrollable, or to be insensitive to the subsequent updates (that is, continue to show the old data). If this is a requirement, pay close heed to the caveats shown above.

The SQL standard only makes provisions for cursors in embedded SQL. The PostgreSQL server does not implement an `OPEN` statement for cursors; a cursor is considered to be open when it is declared. However, ECPG, the embedded SQL preprocessor for PostgreSQL, supports the standard SQL cursor conventions, including those involving `DECLARE` and `OPEN` statements.

You can see all available cursors by querying the `pg_cursors` system view.

Examples

To declare a cursor:

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

See FETCH for more examples of cursor usage.

Compatibility

The SQL standard says that it is implementation-dependent whether cursors are sensitive to concurrent updates of the underlying data by default. In PostgreSQL, cursors are insensitive by default, and can be made sensitive by specifying `FOR UPDATE`. Other products may work differently.

The SQL standard allows cursors only in embedded SQL and in modules. PostgreSQL permits cursors to be used interactively.

Binary cursors are a PostgreSQL extension.

See Also

CLOSE, FETCH, MOVE

DELETE

Name

DELETE — delete rows of a table

Synopsis

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING from_item [, ...] ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

Description

DELETE deletes rows that satisfy the WHERE clause from the specified table. If the WHERE clause is absent, the effect is to delete all rows in the table. The result is a valid, but empty table.

> **Tip:** TRUNCATE is a PostgreSQL extension that provides a faster mechanism to remove all rows from a table.

There are two ways to delete rows in a table using information contained in other tables in the database: using sub-selects, or specifying additional tables in the USING clause. Which technique is more appropriate depends on the specific circumstances.

The optional RETURNING clause causes DELETE to compute and return value(s) based on each row actually deleted. Any expression using the table's columns, and/or columns of other tables mentioned in USING, can be computed. The syntax of the RETURNING list is identical to that of the output list of SELECT.

You must have the DELETE privilege on the table to delete from it, as well as the SELECT privilege for any table in the USING clause or whose values are read in the *condition*.

Parameters

with_query

The WITH clause allows you to specify one or more subqueries that can be referenced by name in the DELETE query. See Section 7.8 and SELECT for details.

table_name

The name (optionally schema-qualified) of the table to delete rows from. If ONLY is specified before the table name, matching rows are deleted from the named table only. If ONLY is not specified, matching rows are also deleted from any tables inheriting from the named table. Optionally, * can be specified after the table name to explicitly indicate that descendant tables are included.

alias

A substitute name for the target table. When an alias is provided, it completely hides the actual name of the table. For example, given `DELETE FROM foo AS f`, the remainder of the `DELETE` statement must refer to this table as `f` not `foo`.

from_item

A table expression allowing columns from other tables to appear in the `WHERE` condition. This uses the same syntax as the ***FROM Clause*** of a `SELECT` statement; for example, an alias for the table name can be specified. Do not repeat the target table as a *`from_item`* unless you wish to set up a self-join (in which case it must appear with an alias in the *`from_item`*).

condition

An expression that returns a value of type `boolean`. Only rows for which this expression returns `true` will be deleted.

cursor_name

The name of the cursor to use in a `WHERE CURRENT OF` condition. The row to be deleted is the one most recently fetched from this cursor. The cursor must be a non-grouping query on the `DELETE`'s target table. Note that `WHERE CURRENT OF` cannot be specified together with a Boolean condition. See DECLARE for more information about using cursors with `WHERE CURRENT OF`.

output_expression

An expression to be computed and returned by the `DELETE` command after each row is deleted. The expression can use any column names of the table named by *`table_name`* or table(s) listed in `USING`. Write `*` to return all columns.

output_name

A name to use for a returned column.

Outputs

On successful completion, a `DELETE` command returns a command tag of the form

`DELETE` *`count`*

The *`count`* is the number of rows deleted. Note that the number may be less than the number of rows that matched the *`condition`* when deletes were suppressed by a `BEFORE DELETE` trigger. If *`count`* is 0, no rows were deleted by the query (this is not considered an error).

If the `DELETE` command contains a `RETURNING` clause, the result will be similar to that of a `SELECT` statement containing the columns and values defined in the `RETURNING` list, computed over the row(s) deleted by the command.

Notes

PostgreSQL lets you reference columns of other tables in the WHERE condition by specifying the other tables in the USING clause. For example, to delete all films produced by a given producer, one can do:

```
DELETE FROM films USING producers
  WHERE producer_id = producers.id AND producers.name = 'foo';
```

What is essentially happening here is a join between films and producers, with all successfully joined films rows being marked for deletion. This syntax is not standard. A more standard way to do it is:

```
DELETE FROM films
  WHERE producer_id IN (SELECT id FROM producers WHERE name = 'foo');
```

In some cases the join style is easier to write or faster to execute than the sub-select style.

Examples

Delete all films but musicals:

```
DELETE FROM films WHERE kind <> 'Musical';
```

Clear the table films:

```
DELETE FROM films;
```

Delete completed tasks, returning full details of the deleted rows:

```
DELETE FROM tasks WHERE status = 'DONE' RETURNING *;
```

Delete the row of tasks on which the cursor c_tasks is currently positioned:

```
DELETE FROM tasks WHERE CURRENT OF c_tasks;
```

Compatibility

This command conforms to the SQL standard, except that the USING and RETURNING clauses are PostgreSQL extensions, as is the ability to use WITH with DELETE.

DISCARD

Name

DISCARD — discard session state

Synopsis

```
DISCARD { ALL | PLANS | SEQUENCES | TEMPORARY | TEMP }
```

Description

DISCARD releases internal resources associated with a database session. This command is useful for partially or fully resetting the session's state. There are several subcommands to release different types of resources; the DISCARD ALL variant subsumes all the others, and also resets additional state.

Parameters

PLANS

Releases all cached query plans, forcing re-planning to occur the next time the associated prepared statement is used.

SEQUENCES

Discards all cached sequence-related state, including currval()/lastval() information and any preallocated sequence values that have not yet been returned by nextval(). (See CREATE SEQUENCE for a description of preallocated sequence values.)

TEMPORARY or TEMP

Drops all temporary tables created in the current session.

ALL

Releases all temporary resources associated with the current session and resets the session to its initial state. Currently, this has the same effect as executing the following sequence of statements:

```
SET SESSION AUTHORIZATION DEFAULT;
RESET ALL;
DEALLOCATE ALL;
CLOSE ALL;
UNLISTEN *;
SELECT pg_advisory_unlock_all();
DISCARD PLANS;
DISCARD SEQUENCES;
DISCARD TEMP;
```

Notes

`DISCARD ALL` cannot be executed inside a transaction block.

Compatibility

`DISCARD` is a PostgreSQL extension.

DO

Name

DO — execute an anonymous code block

Synopsis

```
DO [ LANGUAGE lang_name ] code
```

Description

DO executes an anonymous code block, or in other words a transient anonymous function in a procedural language.

The code block is treated as though it were the body of a function with no parameters, returning void. It is parsed and executed a single time.

The optional LANGUAGE clause can be written either before or after the code block.

Parameters

code

The procedural language code to be executed. This must be specified as a string literal, just as in CREATE FUNCTION. Use of a dollar-quoted literal is recommended.

lang_name

The name of the procedural language the code is written in. If omitted, the default is plpgsql.

Notes

The procedural language to be used must already have been installed into the current database by means of CREATE LANGUAGE. plpgsql is installed by default, but other languages are not.

The user must have USAGE privilege for the procedural language, or must be a superuser if the language is untrusted. This is the same privilege requirement as for creating a function in the language.

Examples

Grant all privileges on all views in schema public to role webuser:

```
DO $$DECLARE r record;
BEGIN
    FOR r IN SELECT table_schema, table_name FROM information_schema.tables
```

```
            WHERE table_type = 'VIEW' AND table_schema = 'public'
    LOOP
        EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' || quote_ident(r.tabl
    END LOOP;
END$$;
```

Compatibility

There is no `DO` statement in the SQL standard.

See Also

CREATE LANGUAGE

DROP AGGREGATE

Name

DROP AGGREGATE — remove an aggregate function

Synopsis

```
DROP AGGREGATE [ IF EXISTS ] name ( aggregate_signature ) [ CASCADE | RESTRICT ]

where aggregate_signature is:

* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ] argtype [ , ...
```

Description

DROP AGGREGATE removes an existing aggregate function. To execute this command the current user must be the owner of the aggregate function.

Parameters

IF EXISTS

Do not throw an error if the aggregate does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing aggregate function.

argmode

The mode of an argument: IN or VARIADIC. If omitted, the default is IN.

argname

The name of an argument. Note that DROP AGGREGATE does not actually pay any attention to argument names, since only the argument data types are needed to determine the aggregate function's identity.

argtype

An input data type on which the aggregate function operates. To reference a zero-argument aggregate function, write * in place of the list of argument specifications. To reference an ordered-set aggregate function, write ORDER BY between the direct and aggregated argument specifications.

CASCADE

Automatically drop objects that depend on the aggregate function.

`RESTRICT`

Refuse to drop the aggregate function if any objects depend on it. This is the default.

Notes

Alternative syntaxes for referencing ordered-set aggregates are described under ALTER AGGREGATE.

Examples

To remove the aggregate function `myavg` for type `integer`:

```
DROP AGGREGATE myavg(integer);
```

To remove the hypothetical-set aggregate function `myrank`, which takes an arbitrary list of ordering columns and a matching list of direct arguments:

```
DROP AGGREGATE myrank(VARIADIC "any" ORDER BY VARIADIC "any");
```

Compatibility

There is no `DROP AGGREGATE` statement in the SQL standard.

See Also

ALTER AGGREGATE, CREATE AGGREGATE

DROP CAST

Name

`DROP CAST` — remove a cast

Synopsis

```
DROP CAST [ IF EXISTS ] (source_type AS target_type) [ CASCADE | RESTRICT ]
```

Description

`DROP CAST` removes a previously defined cast.

To be able to drop a cast, you must own the source or the target data type. These are the same privileges that are required to create a cast.

Parameters

`IF EXISTS`

Do not throw an error if the cast does not exist. A notice is issued in this case.

`source_type`

The name of the source data type of the cast.

`target_type`

The name of the target data type of the cast.

`CASCADE`
`RESTRICT`

These key words do not have any effect, since there are no dependencies on casts.

Examples

To drop the cast from type `text` to type `int`:

```
DROP CAST (text AS int);
```

Compatibility

The `DROP CAST` command conforms to the SQL standard.

See Also

CREATE CAST

DROP COLLATION

Name

DROP COLLATION — remove a collation

Synopsis

```
DROP COLLATION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Description

DROP COLLATION removes a previously defined collation. To be able to drop a collation, you must own the collation.

Parameters

IF EXISTS

Do not throw an error if the collation does not exist. A notice is issued in this case.

name

The name of the collation. The collation name can be schema-qualified.

CASCADE

Automatically drop objects that depend on the collation.

RESTRICT

Refuse to drop the collation if any objects depend on it. This is the default.

Examples

To drop the collation named german:

```
DROP COLLATION german;
```

Compatibility

The DROP COLLATION command conforms to the SQL standard, apart from the IF EXISTS option, which is a PostgreSQL extension.

See Also

ALTER COLLATION, CREATE COLLATION

DROP CONVERSION

Name

DROP CONVERSION — remove a conversion

Synopsis

```
DROP CONVERSION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Description

DROP CONVERSION removes a previously defined conversion. To be able to drop a conversion, you must own the conversion.

Parameters

IF EXISTS

Do not throw an error if the conversion does not exist. A notice is issued in this case.

name

The name of the conversion. The conversion name can be schema-qualified.

CASCADE
RESTRICT

These key words do not have any effect, since there are no dependencies on conversions.

Examples

To drop the conversion named myname:

```
DROP CONVERSION myname;
```

Compatibility

There is no DROP CONVERSION statement in the SQL standard, but a DROP TRANSLATION statement that goes along with the CREATE TRANSLATION statement that is similar to the CREATE CONVERSION statement in PostgreSQL.

See Also

ALTER CONVERSION, CREATE CONVERSION

DROP DATABASE

Name

`DROP DATABASE` — remove a database

Synopsis

```
DROP DATABASE [ IF EXISTS ] name
```

Description

`DROP DATABASE` drops a database. It removes the catalog entries for the database and deletes the directory containing the data. It can only be executed by the database owner. Also, it cannot be executed while you or anyone else are connected to the target database. (Connect to `postgres` or any other database to issue this command.)

`DROP DATABASE` cannot be undone. Use it with care!

Parameters

`IF EXISTS`

Do not throw an error if the database does not exist. A notice is issued in this case.

`name`

The name of the database to remove.

Notes

`DROP DATABASE` cannot be executed inside a transaction block.

This command cannot be executed while connected to the target database. Thus, it might be more convenient to use the program dropdb instead, which is a wrapper around this command.

Compatibility

There is no `DROP DATABASE` statement in the SQL standard.

See Also

CREATE DATABASE

DROP DOMAIN

Name

DROP DOMAIN — remove a domain

Synopsis

```
DROP DOMAIN [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP DOMAIN removes a domain. Only the owner of a domain can remove it.

Parameters

IF EXISTS

Do not throw an error if the domain does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing domain.

CASCADE

Automatically drop objects that depend on the domain (such as table columns).

RESTRICT

Refuse to drop the domain if any objects depend on it. This is the default.

Examples

To remove the domain box:

```
DROP DOMAIN box;
```

Compatibility

This command conforms to the SQL standard, except for the IF EXISTS option, which is a PostgreSQL extension.

See Also

CREATE DOMAIN, ALTER DOMAIN

DROP EVENT TRIGGER

Name

DROP EVENT TRIGGER — remove an event trigger

Synopsis

```
DROP EVENT TRIGGER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Description

`DROP EVENT TRIGGER` removes an existing event trigger. To execute this command, the current user must be the owner of the event trigger.

Parameters

`IF EXISTS`

Do not throw an error if the event trigger does not exist. A notice is issued in this case.

`name`

The name of the event trigger to remove.

`CASCADE`

Automatically drop objects that depend on the trigger.

`RESTRICT`

Refuse to drop the trigger if any objects depend on it. This is the default.

Examples

Destroy the trigger `snitch`:

```
DROP EVENT TRIGGER snitch;
```

Compatibility

There is no `DROP EVENT TRIGGER` statement in the SQL standard.

See Also

CREATE EVENT TRIGGER, ALTER EVENT TRIGGER

DROP EXTENSION

Name

DROP EXTENSION — remove an extension

Synopsis

```
DROP EXTENSION [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP EXTENSION removes extensions from the database. Dropping an extension causes its component objects to be dropped as well.

You must own the extension to use DROP EXTENSION.

Parameters

IF EXISTS

Do not throw an error if the extension does not exist. A notice is issued in this case.

name

The name of an installed extension.

CASCADE

Automatically drop objects that depend on the extension.

RESTRICT

Refuse to drop the extension if any objects depend on it (other than its own member objects and other extensions listed in the same DROP command). This is the default.

Examples

To remove the extension hstore from the current database:

```
DROP EXTENSION hstore;
```

This command will fail if any of hstore's objects are in use in the database, for example if any tables have columns of the hstore type. Add the CASCADE option to forcibly remove those dependent objects as well.

Compatibility

`DROP EXTENSION` is a PostgreSQL extension.

See Also

CREATE EXTENSION, ALTER EXTENSION

DROP FOREIGN DATA WRAPPER

Name

`DROP FOREIGN DATA WRAPPER` — remove a foreign-data wrapper

Synopsis

```
DROP FOREIGN DATA WRAPPER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Description

`DROP FOREIGN DATA WRAPPER` removes an existing foreign-data wrapper. To execute this command, the current user must be the owner of the foreign-data wrapper.

Parameters

`IF EXISTS`

Do not throw an error if the foreign-data wrapper does not exist. A notice is issued in this case.

`name`

The name of an existing foreign-data wrapper.

`CASCADE`

Automatically drop objects that depend on the foreign-data wrapper (such as servers).

`RESTRICT`

Refuse to drop the foreign-data wrappers if any objects depend on it. This is the default.

Examples

Drop the foreign-data wrapper `dbi`:

```
DROP FOREIGN DATA WRAPPER dbi;
```

Compatibility

`DROP FOREIGN DATA WRAPPER` conforms to ISO/IEC 9075-9 (SQL/MED). The `IF EXISTS` clause is a PostgreSQL extension.

See Also

CREATE FOREIGN DATA WRAPPER, ALTER FOREIGN DATA WRAPPER

DROP FOREIGN TABLE

Name

`DROP FOREIGN TABLE` — remove a foreign table

Synopsis

```
DROP FOREIGN TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP FOREIGN TABLE` removes a foreign table. Only the owner of a foreign table can remove it.

Parameters

`IF EXISTS`

Do not throw an error if the foreign table does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of the foreign table to drop.

`CASCADE`

Automatically drop objects that depend on the foreign table (such as views).

`RESTRICT`

Refuse to drop the foreign table if any objects depend on it. This is the default.

Examples

To destroy two foreign tables, `films` and `distributors`:

```
DROP FOREIGN TABLE films, distributors;
```

Compatibility

This command conforms to the ISO/IEC 9075-9 (SQL/MED), except that the standard only allows one foreign table to be dropped per command, and apart from the `IF EXISTS` option, which is a PostgreSQL extension.

See Also

ALTER FOREIGN TABLE, CREATE FOREIGN TABLE

DROP FUNCTION

Name

DROP FUNCTION — remove a function

Synopsis

```
DROP FUNCTION [ IF EXISTS ] name ( [ [ argmode ] [ argname ] argtype [, ...] ] )
    [ CASCADE | RESTRICT ]
```

Description

DROP FUNCTION removes the definition of an existing function. To execute this command the user must be the owner of the function. The argument types to the function must be specified, since several different functions can exist with the same name and different argument lists.

Parameters

IF EXISTS

Do not throw an error if the function does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing function.

argmode

The mode of an argument: IN, OUT, INOUT, or VARIADIC. If omitted, the default is IN. Note that DROP FUNCTION does not actually pay any attention to OUT arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list the IN, INOUT, and VARIADIC arguments.

argname

The name of an argument. Note that DROP FUNCTION does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity.

argtype

The data type(s) of the function's arguments (optionally schema-qualified), if any.

CASCADE

Automatically drop objects that depend on the function (such as operators or triggers).

RESTRICT

Refuse to drop the function if any objects depend on it. This is the default.

Examples

This command removes the square root function:

```
DROP FUNCTION sqrt(integer);
```

Compatibility

A `DROP FUNCTION` statement is defined in the SQL standard, but it is not compatible with this command.

See Also

CREATE FUNCTION, ALTER FUNCTION

DROP GROUP

Name

`DROP GROUP` — remove a database role

Synopsis

```
DROP GROUP [ IF EXISTS ] name [, ...]
```

Description

`DROP GROUP` is now an alias for DROP ROLE.

Compatibility

There is no `DROP GROUP` statement in the SQL standard.

See Also

DROP ROLE

DROP INDEX

Name

DROP INDEX — remove an index

Synopsis

```
DROP INDEX [ CONCURRENTLY ] [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP INDEX drops an existing index from the database system. To execute this command you must be the owner of the index.

Parameters

CONCURRENTLY

Drop the index without locking out concurrent selects, inserts, updates, and deletes on the index's table. A normal DROP INDEX acquires exclusive lock on the table, blocking other accesses until the index drop can be completed. With this option, the command instead waits until conflicting transactions have completed.

There are several caveats to be aware of when using this option. Only one index name can be specified, and the CASCADE option is not supported. (Thus, an index that supports a UNIQUE or PRIMARY KEY constraint cannot be dropped this way.) Also, regular DROP INDEX commands can be performed within a transaction block, but DROP INDEX CONCURRENTLY cannot.

For temporary tables, DROP INDEX is always non-concurrent, as no other session can access them, and non-concurrent index drop is cheaper.

IF EXISTS

Do not throw an error if the index does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an index to remove.

CASCADE

Automatically drop objects that depend on the index.

RESTRICT

Refuse to drop the index if any objects depend on it. This is the default.

Examples

This command will remove the index `title_idx`:

```
DROP INDEX title_idx;
```

Compatibility

`DROP INDEX` is a PostgreSQL language extension. There are no provisions for indexes in the SQL standard.

See Also

CREATE INDEX

DROP LANGUAGE

Name

DROP LANGUAGE — remove a procedural language

Synopsis

```
DROP [ PROCEDURAL ] LANGUAGE [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Description

`DROP LANGUAGE` removes the definition of a previously registered procedural language. You must be a superuser or the owner of the language to use `DROP LANGUAGE`.

> **Note:** As of PostgreSQL 9.1, most procedural languages have been made into "extensions", and should therefore be removed with DROP EXTENSION not `DROP LANGUAGE`.

Parameters

`IF EXISTS`

Do not throw an error if the language does not exist. A notice is issued in this case.

name

The name of an existing procedural language. For backward compatibility, the name can be enclosed by single quotes.

`CASCADE`

Automatically drop objects that depend on the language (such as functions in the language).

`RESTRICT`

Refuse to drop the language if any objects depend on it. This is the default.

Examples

This command removes the procedural language `plsample`:

```
DROP LANGUAGE plsample;
```

Compatibility

There is no `DROP LANGUAGE` statement in the SQL standard.

See Also

ALTER LANGUAGE, CREATE LANGUAGE, droplang

DROP MATERIALIZED VIEW

Name

DROP MATERIALIZED VIEW — remove a materialized view

Synopsis

```
DROP MATERIALIZED VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP MATERIALIZED VIEW drops an existing materialized view. To execute this command you must be the owner of the materialized view.

Parameters

IF EXISTS

Do not throw an error if the materialized view does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of the materialized view to remove.

CASCADE

Automatically drop objects that depend on the materialized view (such as other materialized views, or regular views).

RESTRICT

Refuse to drop the materialized view if any objects depend on it. This is the default.

Examples

This command will remove the materialized view called order_summary:

```
DROP MATERIALIZED VIEW order_summary;
```

Compatibility

DROP MATERIALIZED VIEW is a PostgreSQL extension.

See Also

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, REFRESH MATERIALIZED VIEW

DROP OPERATOR

Name

DROP OPERATOR — remove an operator

Synopsis

```
DROP OPERATOR [ IF EXISTS ] name ( { left_type | NONE } , { right_type | NONE } ) [ CASCADE |
```

Description

DROP OPERATOR drops an existing operator from the database system. To execute this command you must be the owner of the operator.

Parameters

IF EXISTS

Do not throw an error if the operator does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing operator.

left_type

The data type of the operator's left operand; write NONE if the operator has no left operand.

right_type

The data type of the operator's right operand; write NONE if the operator has no right operand.

CASCADE

Automatically drop objects that depend on the operator.

RESTRICT

Refuse to drop the operator if any objects depend on it. This is the default.

Examples

Remove the power operator a^b for type integer:

```
DROP OPERATOR ^ (integer, integer);
```

Remove the left unary bitwise complement operator ~b for type bit:

```
DROP OPERATOR ~ (none, bit);
```

Remove the right unary factorial operator `x!` for type `bigint`:

```
DROP OPERATOR ! (bigint, none);
```

Compatibility

There is no `DROP OPERATOR` statement in the SQL standard.

See Also

CREATE OPERATOR, ALTER OPERATOR

DROP OPERATOR CLASS

Name

DROP OPERATOR CLASS — remove an operator class

Synopsis

```
DROP OPERATOR CLASS [ IF EXISTS ] name USING index_method [ CASCADE | RESTRICT ]
```

Description

DROP OPERATOR CLASS drops an existing operator class. To execute this command you must be the owner of the operator class.

DROP OPERATOR CLASS does not drop any of the operators or functions referenced by the class. If there are any indexes depending on the operator class, you will need to specify CASCADE for the drop to complete.

Parameters

IF EXISTS

Do not throw an error if the operator class does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing operator class.

index_method

The name of the index access method the operator class is for.

CASCADE

Automatically drop objects that depend on the operator class.

RESTRICT

Refuse to drop the operator class if any objects depend on it. This is the default.

Notes

DROP OPERATOR CLASS will not drop the operator family containing the class, even if there is nothing else left in the family (in particular, in the case where the family was implicitly created by CREATE OPERATOR CLASS). An empty operator family is harmless, but for the sake of tidiness you might wish to remove the family with DROP OPERATOR FAMILY; or perhaps better, use DROP OPERATOR FAMILY in the first place.

Examples

Remove the B-tree operator class `widget_ops`:

```
DROP OPERATOR CLASS widget_ops USING btree;
```

This command will not succeed if there are any existing indexes that use the operator class. Add `CASCADE` to drop such indexes along with the operator class.

Compatibility

There is no `DROP OPERATOR CLASS` statement in the SQL standard.

See Also

ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR FAMILY

DROP OPERATOR FAMILY

Name

DROP OPERATOR FAMILY — remove an operator family

Synopsis

```
DROP OPERATOR FAMILY [ IF EXISTS ] name USING index_method [ CASCADE | RESTRICT ]
```

Description

`DROP OPERATOR FAMILY` drops an existing operator family. To execute this command you must be the owner of the operator family.

`DROP OPERATOR FAMILY` includes dropping any operator classes contained in the family, but it does not drop any of the operators or functions referenced by the family. If there are any indexes depending on operator classes within the family, you will need to specify `CASCADE` for the drop to complete.

Parameters

`IF EXISTS`

Do not throw an error if the operator family does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing operator family.

`index_method`

The name of the index access method the operator family is for.

`CASCADE`

Automatically drop objects that depend on the operator family.

`RESTRICT`

Refuse to drop the operator family if any objects depend on it. This is the default.

Examples

Remove the B-tree operator family `float_ops`:

```
DROP OPERATOR FAMILY float_ops USING btree;
```

This command will not succeed if there are any existing indexes that use operator classes within the family. Add `CASCADE` to drop such indexes along with the operator family.

Compatibility

There is no `DROP OPERATOR FAMILY` statement in the SQL standard.

See Also

ALTER OPERATOR FAMILY, CREATE OPERATOR FAMILY, ALTER OPERATOR CLASS, CREATE OPERATOR CLASS, DROP OPERATOR CLASS

DROP OWNED

Name

DROP OWNED — remove database objects owned by a database role

Synopsis

```
DROP OWNED BY { name | CURRENT_USER | SESSION_USER } [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP OWNED drops all the objects within the current database that are owned by one of the specified roles. Any privileges granted to the given roles on objects in the current database or on shared objects (databases, tablespaces) will also be revoked.

Parameters

name

The name of a role whose objects will be dropped, and whose privileges will be revoked.

CASCADE

Automatically drop objects that depend on the affected objects.

RESTRICT

Refuse to drop the objects owned by a role if any other database objects depend on one of the affected objects. This is the default.

Notes

DROP OWNED is often used to prepare for the removal of one or more roles. Because DROP OWNED only affects the objects in the current database, it is usually necessary to execute this command in each database that contains objects owned by a role that is to be removed.

Using the CASCADE option might make the command recurse to objects owned by other users.

The REASSIGN OWNED command is an alternative that reassigns the ownership of all the database objects owned by one or more roles. However, REASSIGN OWNED does not deal with privileges for other objects.

Databases and tablespaces owned by the role(s) will not be removed.

See Section 20.4 for more discussion.

Compatibility

The `DROP OWNED` command is a PostgreSQL extension.

See Also

REASSIGN OWNED, DROP ROLE

DROP POLICY

Name

DROP POLICY — remove a row level security policy from a table

Synopsis

```
DROP POLICY [ IF EXISTS ] name ON table_name
```

Description

`DROP POLICY` removes the specified policy from the table. Note that if the last policy is removed for a table and the table still has row level security enabled via `ALTER TABLE`, then the default-deny policy will be used. `ALTER TABLE ... DISABLE ROW LEVEL SECURITY` can be used to disable row level security for a table, whether policies for the table exist or not.

Parameters

`IF EXISTS`

Do not throw an error if the policy does not exist. A notice is issued in this case.

`name`

The name of the policy to drop.

`table_name`

The name (optionally schema-qualified) of the table that the policy is on.

Examples

To drop the policy called `p1` on the table named `my_table`:

```
DROP POLICY p1 ON my_table;
```

Compatibility

`DROP POLICY` is a PostgreSQL extension.

See Also

CREATE POLICY, ALTER POLICY

DROP ROLE

Name

DROP ROLE — remove a database role

Synopsis

```
DROP ROLE [ IF EXISTS ] name [, ...]
```

Description

DROP ROLE removes the specified role(s). To drop a superuser role, you must be a superuser yourself; to drop non-superuser roles, you must have CREATEROLE privilege.

A role cannot be removed if it is still referenced in any database of the cluster; an error will be raised if so. Before dropping the role, you must drop all the objects it owns (or reassign their ownership) and revoke any privileges the role has been granted on other objects. The REASSIGN OWNED and DROP OWNED commands can be useful for this purpose; see Section 20.4 for more discussion.

However, it is not necessary to remove role memberships involving the role; DROP ROLE automatically revokes any memberships of the target role in other roles, and of other roles in the target role. The other roles are not dropped nor otherwise affected.

Parameters

IF EXISTS

Do not throw an error if the role does not exist. A notice is issued in this case.

name

The name of the role to remove.

Notes

PostgreSQL includes a program dropuser that has the same functionality as this command (in fact, it calls this command) but can be run from the command shell.

Examples

To drop a role:

```
DROP ROLE jonathan;
```

Compatibility

The SQL standard defines `DROP ROLE`, but it allows only one role to be dropped at a time, and it specifies different privilege requirements than PostgreSQL uses.

See Also

CREATE ROLE, ALTER ROLE, SET ROLE

DROP RULE

Name

DROP RULE — remove a rewrite rule

Synopsis

```
DROP RULE [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

Description

DROP RULE drops a rewrite rule.

Parameters

IF EXISTS

Do not throw an error if the rule does not exist. A notice is issued in this case.

name

The name of the rule to drop.

table_name

The name (optionally schema-qualified) of the table or view that the rule applies to.

CASCADE

Automatically drop objects that depend on the rule.

RESTRICT

Refuse to drop the rule if any objects depend on it. This is the default.

Examples

To drop the rewrite rule newrule:

```
DROP RULE newrule ON mytable;
```

Compatibility

DROP RULE is a PostgreSQL language extension, as is the entire query rewrite system.

See Also

CREATE RULE, ALTER RULE

DROP SCHEMA

Name

`DROP SCHEMA` — remove a schema

Synopsis

```
DROP SCHEMA [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP SCHEMA` removes schemas from the database.

A schema can only be dropped by its owner or a superuser. Note that the owner can drop the schema (and thereby all contained objects) even if he does not own some of the objects within the schema.

Parameters

`IF EXISTS`

Do not throw an error if the schema does not exist. A notice is issued in this case.

`name`

The name of a schema.

`CASCADE`

Automatically drop objects (tables, functions, etc.) that are contained in the schema.

`RESTRICT`

Refuse to drop the schema if it contains any objects. This is the default.

Examples

To remove schema `mystuff` from the database, along with everything it contains:

```
DROP SCHEMA mystuff CASCADE;
```

Compatibility

`DROP SCHEMA` is fully conforming with the SQL standard, except that the standard only allows one schema to be dropped per command, and apart from the `IF EXISTS` option, which is a PostgreSQL extension.

See Also

ALTER SCHEMA, CREATE SCHEMA

DROP SEQUENCE

Name

DROP SEQUENCE — remove a sequence

Synopsis

```
DROP SEQUENCE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP SEQUENCE removes sequence number generators. A sequence can only be dropped by its owner or a superuser.

Parameters

IF EXISTS

Do not throw an error if the sequence does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of a sequence.

CASCADE

Automatically drop objects that depend on the sequence.

RESTRICT

Refuse to drop the sequence if any objects depend on it. This is the default.

Examples

To remove the sequence serial:

```
DROP SEQUENCE serial;
```

Compatibility

DROP SEQUENCE conforms to the SQL standard, except that the standard only allows one sequence to be dropped per command, and apart from the IF EXISTS option, which is a PostgreSQL extension.

See Also

CREATE SEQUENCE, ALTER SEQUENCE

DROP SERVER

Name

DROP SERVER — remove a foreign server descriptor

Synopsis

```
DROP SERVER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Description

`DROP SERVER` removes an existing foreign server descriptor. To execute this command, the current user must be the owner of the server.

Parameters

`IF EXISTS`

Do not throw an error if the server does not exist. A notice is issued in this case.

`name`

The name of an existing server.

`CASCADE`

Automatically drop objects that depend on the server (such as user mappings).

`RESTRICT`

Refuse to drop the server if any objects depend on it. This is the default.

Examples

Drop a server `foo` if it exists:

```
DROP SERVER IF EXISTS foo;
```

Compatibility

`DROP SERVER` conforms to ISO/IEC 9075-9 (SQL/MED). The `IF EXISTS` clause is a PostgreSQL extension.

See Also

CREATE SERVER, ALTER SERVER

DROP TABLE

Name

DROP TABLE — remove a table

Synopsis

```
DROP TABLE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP TABLE removes tables from the database. Only the table owner, the schema owner, and superuser can drop a table. To empty a table of rows without destroying the table, use DELETE or TRUNCATE.

DROP TABLE always removes any indexes, rules, triggers, and constraints that exist for the target table. However, to drop a table that is referenced by a view or a foreign-key constraint of another table, CASCADE must be specified. (CASCADE will remove a dependent view entirely, but in the foreign-key case it will only remove the foreign-key constraint, not the other table entirely.)

Parameters

IF EXISTS

Do not throw an error if the table does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of the table to drop.

CASCADE

Automatically drop objects that depend on the table (such as views).

RESTRICT

Refuse to drop the table if any objects depend on it. This is the default.

Examples

To destroy two tables, films and distributors:

```
DROP TABLE films, distributors;
```

Compatibility

This command conforms to the SQL standard, except that the standard only allows one table to be dropped per command, and apart from the `IF EXISTS` option, which is a PostgreSQL extension.

See Also

ALTER TABLE, CREATE TABLE

DROP TABLESPACE

Name

DROP TABLESPACE — remove a tablespace

Synopsis

```
DROP TABLESPACE [ IF EXISTS ] name
```

Description

DROP TABLESPACE removes a tablespace from the system.

A tablespace can only be dropped by its owner or a superuser. The tablespace must be empty of all database objects before it can be dropped. It is possible that objects in other databases might still reside in the tablespace even if no objects in the current database are using the tablespace. Also, if the tablespace is listed in the temp_tablespaces setting of any active session, the DROP might fail due to temporary files residing in the tablespace.

Parameters

IF EXISTS

Do not throw an error if the tablespace does not exist. A notice is issued in this case.

name

The name of a tablespace.

Notes

DROP TABLESPACE cannot be executed inside a transaction block.

Examples

To remove tablespace mystuff from the system:

```
DROP TABLESPACE mystuff;
```

Compatibility

`DROP TABLESPACE` is a PostgreSQL extension.

See Also

CREATE TABLESPACE, ALTER TABLESPACE

DROP TEXT SEARCH CONFIGURATION

Name

DROP TEXT SEARCH CONFIGURATION — remove a text search configuration

Synopsis

```
DROP TEXT SEARCH CONFIGURATION [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Description

DROP TEXT SEARCH CONFIGURATION drops an existing text search configuration. To execute this command you must be the owner of the configuration.

Parameters

IF EXISTS

Do not throw an error if the text search configuration does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing text search configuration.

CASCADE

Automatically drop objects that depend on the text search configuration.

RESTRICT

Refuse to drop the text search configuration if any objects depend on it. This is the default.

Examples

Remove the text search configuration my_english:

```
DROP TEXT SEARCH CONFIGURATION my_english;
```

This command will not succeed if there are any existing indexes that reference the configuration in to_tsvector calls. Add CASCADE to drop such indexes along with the text search configuration.

Compatibility

There is no DROP TEXT SEARCH CONFIGURATION statement in the SQL standard.

See Also

ALTER TEXT SEARCH CONFIGURATION, CREATE TEXT SEARCH CONFIGURATION

DROP TEXT SEARCH DICTIONARY

Name

DROP TEXT SEARCH DICTIONARY — remove a text search dictionary

Synopsis

```
DROP TEXT SEARCH DICTIONARY [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Description

`DROP TEXT SEARCH DICTIONARY` drops an existing text search dictionary. To execute this command you must be the owner of the dictionary.

Parameters

`IF EXISTS`

Do not throw an error if the text search dictionary does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing text search dictionary.

`CASCADE`

Automatically drop objects that depend on the text search dictionary.

`RESTRICT`

Refuse to drop the text search dictionary if any objects depend on it. This is the default.

Examples

Remove the text search dictionary `english`:

```
DROP TEXT SEARCH DICTIONARY english;
```

This command will not succeed if there are any existing text search configurations that use the dictionary. Add `CASCADE` to drop such configurations along with the dictionary.

Compatibility

There is no `DROP TEXT SEARCH DICTIONARY` statement in the SQL standard.

See Also

ALTER TEXT SEARCH DICTIONARY, CREATE TEXT SEARCH DICTIONARY

DROP TEXT SEARCH PARSER

Name

DROP TEXT SEARCH PARSER — remove a text search parser

Synopsis

```
DROP TEXT SEARCH PARSER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Description

DROP TEXT SEARCH PARSER drops an existing text search parser. You must be a superuser to use this command.

Parameters

IF EXISTS

Do not throw an error if the text search parser does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of an existing text search parser.

CASCADE

Automatically drop objects that depend on the text search parser.

RESTRICT

Refuse to drop the text search parser if any objects depend on it. This is the default.

Examples

Remove the text search parser my_parser:

```
DROP TEXT SEARCH PARSER my_parser;
```

This command will not succeed if there are any existing text search configurations that use the parser. Add CASCADE to drop such configurations along with the parser.

Compatibility

There is no DROP TEXT SEARCH PARSER statement in the SQL standard.

See Also

ALTER TEXT SEARCH PARSER, CREATE TEXT SEARCH PARSER

DROP TEXT SEARCH TEMPLATE

Name

DROP TEXT SEARCH TEMPLATE — remove a text search template

Synopsis

```
DROP TEXT SEARCH TEMPLATE [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

Description

`DROP TEXT SEARCH TEMPLATE` drops an existing text search template. You must be a superuser to use this command.

Parameters

`IF EXISTS`

Do not throw an error if the text search template does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of an existing text search template.

`CASCADE`

Automatically drop objects that depend on the text search template.

`RESTRICT`

Refuse to drop the text search template if any objects depend on it. This is the default.

Examples

Remove the text search template `thesaurus`:

```
DROP TEXT SEARCH TEMPLATE thesaurus;
```

This command will not succeed if there are any existing text search dictionaries that use the template. Add `CASCADE` to drop such dictionaries along with the template.

Compatibility

There is no `DROP TEXT SEARCH TEMPLATE` statement in the SQL standard.

See Also

ALTER TEXT SEARCH TEMPLATE, CREATE TEXT SEARCH TEMPLATE

DROP TRANSFORM

Name

`DROP TRANSFORM` — remove a transform

Synopsis

```
DROP TRANSFORM [ IF EXISTS ] FOR type_name LANGUAGE lang_name
```

Description

`DROP TRANSFORM` removes a previously defined transform.

To be able to drop a transform, you must own the type and the language. These are the same privileges that are required to create a transform.

Parameters

`IF EXISTS`

Do not throw an error if the transform does not exist. A notice is issued in this case.

`type_name`

The name of the data type of the transform.

`lang_name`

The name of the language of the transform.

`CASCADE`

Automatically drop objects that depend on the transform.

`RESTRICT`

Refuse to drop the transform if any objects depend on it. This is the default.

Examples

To drop the transform for type `hstore` and language `plpythonu`:

```
DROP TRANSFORM FOR hstore LANGUAGE plpythonu;
```

Compatibility

This form of `DROP TRANSFORM` is a PostgreSQL extension. See CREATE TRANSFORM for details.

See Also

CREATE TRANSFORM

DROP TRIGGER

Name

`DROP TRIGGER` — remove a trigger

Synopsis

```
DROP TRIGGER [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

Description

`DROP TRIGGER` removes an existing trigger definition. To execute this command, the current user must be the owner of the table for which the trigger is defined.

Parameters

`IF EXISTS`

Do not throw an error if the trigger does not exist. A notice is issued in this case.

`name`

The name of the trigger to remove.

`table_name`

The name (optionally schema-qualified) of the table for which the trigger is defined.

`CASCADE`

Automatically drop objects that depend on the trigger.

`RESTRICT`

Refuse to drop the trigger if any objects depend on it. This is the default.

Examples

Destroy the trigger `if_dist_exists` on the table `films`:

```
DROP TRIGGER if_dist_exists ON films;
```

Compatibility

The `DROP TRIGGER` statement in PostgreSQL is incompatible with the SQL standard. In the SQL standard, trigger names are not local to tables, so the command is simply `DROP TRIGGER` *`name`*.

See Also

CREATE TRIGGER

DROP TYPE

Name

DROP TYPE — remove a data type

Synopsis

```
DROP TYPE [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

DROP TYPE removes a user-defined data type. Only the owner of a type can remove it.

Parameters

IF EXISTS

Do not throw an error if the type does not exist. A notice is issued in this case.

name

The name (optionally schema-qualified) of the data type to remove.

CASCADE

Automatically drop objects that depend on the type (such as table columns, functions, operators).

RESTRICT

Refuse to drop the type if any objects depend on it. This is the default.

Examples

To remove the data type box:

```
DROP TYPE box;
```

Compatibility

This command is similar to the corresponding command in the SQL standard, apart from the IF EXISTS option, which is a PostgreSQL extension. But note that much of the CREATE TYPE command and the data type extension mechanisms in PostgreSQL differ from the SQL standard.

See Also

ALTER TYPE, CREATE TYPE

DROP USER

Name

`DROP USER` — remove a database role

Synopsis

```
DROP USER [ IF EXISTS ] name [, ...]
```

Description

`DROP USER` is simply an alternate spelling of DROP ROLE.

Compatibility

The `DROP USER` statement is a PostgreSQL extension. The SQL standard leaves the definition of users to the implementation.

See Also

DROP ROLE

DROP USER MAPPING

Name

DROP USER MAPPING — remove a user mapping for a foreign server

Synopsis

```
DROP USER MAPPING [ IF EXISTS ] FOR { user_name | USER | CURRENT_USER | PUBLIC } SERVER serve
```

Description

DROP USER MAPPING removes an existing user mapping from foreign server.

The owner of a foreign server can drop user mappings for that server for any user. Also, a user can drop a user mapping for his own user name if USAGE privilege on the server has been granted to the user.

Parameters

IF EXISTS

Do not throw an error if the user mapping does not exist. A notice is issued in this case.

user_name

User name of the mapping. CURRENT_USER and USER match the name of the current user. PUBLIC is used to match all present and future user names in the system.

server_name

Server name of the user mapping.

Examples

Drop a user mapping bob, server foo if it exists:

```
DROP USER MAPPING IF EXISTS FOR bob SERVER foo;
```

Compatibility

DROP USER MAPPING conforms to ISO/IEC 9075-9 (SQL/MED). The IF EXISTS clause is a PostgreSQL extension.

See Also

CREATE USER MAPPING, ALTER USER MAPPING

DROP VIEW

Name

DROP VIEW — remove a view

Synopsis

```
DROP VIEW [ IF EXISTS ] name [, ...] [ CASCADE | RESTRICT ]
```

Description

`DROP VIEW` drops an existing view. To execute this command you must be the owner of the view.

Parameters

`IF EXISTS`

Do not throw an error if the view does not exist. A notice is issued in this case.

`name`

The name (optionally schema-qualified) of the view to remove.

`CASCADE`

Automatically drop objects that depend on the view (such as other views).

`RESTRICT`

Refuse to drop the view if any objects depend on it. This is the default.

Examples

This command will remove the view called `kinds`:

```
DROP VIEW kinds;
```

Compatibility

This command conforms to the SQL standard, except that the standard only allows one view to be dropped per command, and apart from the `IF EXISTS` option, which is a PostgreSQL extension.

See Also

ALTER VIEW, CREATE VIEW

END

Name

END — commit the current transaction

Synopsis

```
END [ WORK | TRANSACTION ]
```

Description

END commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs. This command is a PostgreSQL extension that is equivalent to COMMIT.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

Notes

Use ROLLBACK to abort a transaction.

Issuing END when not inside a transaction does no harm, but it will provoke a warning message.

Examples

To commit the current transaction and make all changes permanent:

```
END;
```

Compatibility

END is a PostgreSQL extension that provides functionality equivalent to COMMIT, which is specified in the SQL standard.

See Also

BEGIN, COMMIT, ROLLBACK

EXECUTE

Name

EXECUTE — execute a prepared statement

Synopsis

```
EXECUTE name [ ( parameter [, ...] ) ]
```

Description

EXECUTE is used to execute a previously prepared statement. Since prepared statements only exist for the duration of a session, the prepared statement must have been created by a PREPARE statement executed earlier in the current session.

If the PREPARE statement that created the statement specified some parameters, a compatible set of parameters must be passed to the EXECUTE statement, or else an error is raised. Note that (unlike functions) prepared statements are not overloaded based on the type or number of their parameters; the name of a prepared statement must be unique within a database session.

For more information on the creation and usage of prepared statements, see PREPARE.

Parameters

name

The name of the prepared statement to execute.

parameter

The actual value of a parameter to the prepared statement. This must be an expression yielding a value that is compatible with the data type of this parameter, as was determined when the prepared statement was created.

Outputs

The command tag returned by EXECUTE is that of the prepared statement, and not EXECUTE.

Examples

Examples are given in the *Examples* section of the PREPARE documentation.

Compatibility

The SQL standard includes an `EXECUTE` statement, but it is only for use in embedded SQL. This version of the `EXECUTE` statement also uses a somewhat different syntax.

See Also

DEALLOCATE, PREPARE

EXPLAIN

Name

EXPLAIN — show the execution plan of a statement

Synopsis

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement

where option can be one of:

    ANALYZE [ boolean ]
    VERBOSE [ boolean ]
    COSTS [ boolean ]
    BUFFERS [ boolean ]
    TIMING [ boolean ]
    FORMAT { TEXT | XML | JSON | YAML }
```

Description

This command displays the execution plan that the PostgreSQL planner generates for the supplied statement. The execution plan shows how the table(s) referenced by the statement will be scanned — by plain sequential scan, index scan, etc. — and if multiple tables are referenced, what join algorithms will be used to bring together the required rows from each input table.

The most critical part of the display is the estimated statement execution cost, which is the planner's guess at how long it will take to run the statement (measured in cost units that are arbitrary, but conventionally mean disk page fetches). Actually two numbers are shown: the start-up cost before the first row can be returned, and the total cost to return all the rows. For most queries the total cost is what matters, but in contexts such as a subquery in EXISTS, the planner will choose the smallest start-up cost instead of the smallest total cost (since the executor will stop after getting one row, anyway). Also, if you limit the number of rows to return with a LIMIT clause, the planner makes an appropriate interpolation between the endpoint costs to estimate which plan is really the cheapest.

The ANALYZE option causes the statement to be actually executed, not only planned. Then actual run time statistics are added to the display, including the total elapsed time expended within each plan node (in milliseconds) and the total number of rows it actually returned. This is useful for seeing whether the planner's estimates are close to reality.

> **Important:** Keep in mind that the statement is actually executed when the ANALYZE option is used. Although EXPLAIN will discard any output that a SELECT would return, other side effects of the statement will happen as usual. If you wish to use EXPLAIN ANALYZE on an INSERT, UPDATE, DELETE, CREATE TABLE AS, or EXECUTE statement without letting the command affect your data, use this approach:
>
> ```
> BEGIN;
> EXPLAIN ANALYZE ...;
> ```

```
ROLLBACK;
```

Only the ANALYZE and VERBOSE options can be specified, and only in that order, without surrounding the option list in parentheses. Prior to PostgreSQL 9.0, the unparenthesized syntax was the only one supported. It is expected that all new options will be supported only in the parenthesized syntax.

Parameters

ANALYZE

Carry out the command and show actual run times and other statistics. This parameter defaults to FALSE.

VERBOSE

Display additional information regarding the plan. Specifically, include the output column list for each node in the plan tree, schema-qualify table and function names, always label variables in expressions with their range table alias, and always print the name of each trigger for which statistics are displayed. This parameter defaults to FALSE.

COSTS

Include information on the estimated startup and total cost of each plan node, as well as the estimated number of rows and the estimated width of each row. This parameter defaults to TRUE.

BUFFERS

Include information on buffer usage. Specifically, include the number of shared blocks hit, read, dirtied, and written, the number of local blocks hit, read, dirtied, and written, and the number of temp blocks read and written. A *hit* means that a read was avoided because the block was found already in cache when needed. Shared blocks contain data from regular tables and indexes; local blocks contain data from temporary tables and indexes; while temp blocks contain short-term working data used in sorts, hashes, Materialize plan nodes, and similar cases. The number of blocks *dirtied* indicates the number of previously unmodified blocks that were changed by this query; while the number of blocks *written* indicates the number of previously-dirtied blocks evicted from cache by this backend during query processing. The number of blocks shown for an upper-level node includes those used by all its child nodes. In text format, only non-zero values are printed. This parameter may only be used when ANALYZE is also enabled. It defaults to FALSE.

TIMING

Include actual startup time and time spent in each node in the output. The overhead of repeatedly reading the system clock can slow down the query significantly on some systems, so it may be useful to set this parameter to FALSE when only actual row counts, and not exact times, are needed. Run time of the entire statement is always measured, even when node-level timing is turned off with this option. This parameter may only be used when ANALYZE is also enabled. It defaults to TRUE.

`FORMAT`

Specify the output format, which can be TEXT, XML, JSON, or YAML. Non-text output contains the same information as the text output format, but is easier for programs to parse. This parameter defaults to `TEXT`.

`boolean`

Specifies whether the selected option should be turned on or off. You can write `TRUE`, `ON`, or `1` to enable the option, and `FALSE`, `OFF`, or `0` to disable it. The *`boolean`* value can also be omitted, in which case `TRUE` is assumed.

`statement`

Any `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `VALUES`, `EXECUTE`, `DECLARE`, `CREATE TABLE AS`, or `CREATE MATERIALIZED VIEW AS` statement, whose execution plan you wish to see.

Outputs

The command's result is a textual description of the plan selected for the *`statement`*, optionally annotated with execution statistics. Section 14.1 describes the information provided.

Notes

In order to allow the PostgreSQL query planner to make reasonably informed decisions when optimizing queries, the `pg_statistic` data should be up-to-date for all tables used in the query. Normally the autovacuum daemon will take care of that automatically. But if a table has recently had substantial changes in its contents, you might need to do a manual ANALYZE rather than wait for autovacuum to catch up with the changes.

In order to measure the run-time cost of each node in the execution plan, the current implementation of `EXPLAIN ANALYZE` adds profiling overhead to query execution. As a result, running `EXPLAIN ANALYZE` on a query can sometimes take significantly longer than executing the query normally. The amount of overhead depends on the nature of the query, as well as the platform being used. The worst case occurs for plan nodes that in themselves require very little time per execution, and on machines that have relatively slow operating system calls for obtaining the time of day.

Examples

To show the plan for a simple query on a table with a single `integer` column and 10000 rows:

```
EXPLAIN SELECT * FROM foo;

                       QUERY PLAN
-------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

Here is the same query, with JSON output formatting:

```
EXPLAIN (FORMAT JSON) SELECT * FROM foo;
           QUERY PLAN
--------------------------------
 [                              +
   {                            +
     "Plan": {                  +
       "Node Type": "Seq Scan",+
       "Relation Name": "foo", +
       "Alias": "foo",         +
       "Startup Cost": 0.00,   +
       "Total Cost": 155.00,   +
       "Plan Rows": 10000,     +
       "Plan Width": 4         +
     }                          +
   }                            +
 ]
(1 row)
```

If there is an index and we use a query with an indexable `WHERE` condition, `EXPLAIN` might show a different plan:

```
EXPLAIN SELECT * FROM foo WHERE i = 4;

                         QUERY PLAN
-------------------------------------------------------------
 Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
   Index Cond: (i = 4)
(2 rows)
```

Here is the same query, but in YAML format:

```
EXPLAIN (FORMAT YAML) SELECT * FROM foo WHERE i='4';
          QUERY PLAN
-------------------------------
 - Plan:                      +
     Node Type: "Index Scan"  +
     Scan Direction: "Forward"+
     Index Name: "fi"         +
     Relation Name: "foo"     +
     Alias: "foo"             +
     Startup Cost: 0.00       +
     Total Cost: 5.98         +
     Plan Rows: 1             +
     Plan Width: 4            +
     Index Cond: "(i = 4)"
(1 row)
```

XML format is left as an exercise for the reader.

Here is the same plan with cost estimates suppressed:

```
EXPLAIN (COSTS FALSE) SELECT * FROM foo WHERE i = 4;

        QUERY PLAN
----------------------------
 Index Scan using fi on foo
   Index Cond: (i = 4)
(2 rows)
```

Here is an example of a query plan for a query using an aggregate function:

```
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;

                              QUERY PLAN
-----------------------------------------------------------------------
 Aggregate  (cost=23.93..23.93 rows=1 width=4)
   ->  Index Scan using fi on foo  (cost=0.00..23.92 rows=6 width=4)
         Index Cond: (i < 10)
(3 rows)
```

Here is an example of using `EXPLAIN EXECUTE` to display the execution plan for a prepared query:

```
PREPARE query(int, int) AS SELECT sum(bar) FROM test
    WHERE id > $1 AND id < $2
    GROUP BY foo;

EXPLAIN ANALYZE EXECUTE query(100, 200);

                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=9.54..9.54 rows=1 width=8) (actual time=0.156..0.161 rows=11 loops=1)
   Group Key: foo
   ->  Index Scan using test_pkey on test  (cost=0.29..9.29 rows=50 width=8) (actual time=0
         Index Cond: ((id > $1) AND (id < $2))
 Planning time: 0.197 ms
 Execution time: 0.225 ms
(6 rows)
```

Of course, the specific numbers shown here depend on the actual contents of the tables involved. Also note that the numbers, and even the selected query strategy, might vary between PostgreSQL releases due to planner improvements. In addition, the `ANALYZE` command uses random sampling to estimate data statistics; therefore, it is possible for cost estimates to change after a fresh run of `ANALYZE`, even if the actual distribution of data in the table has not changed.

Compatibility

There is no `EXPLAIN` statement defined in the SQL standard.

See Also

ANALYZE

FETCH

Name

FETCH — retrieve rows from a query using a cursor

Synopsis

```
FETCH [ direction [ FROM | IN ] ] cursor_name

where direction can be empty or one of:

    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE count
    RELATIVE count
    count
    ALL
    FORWARD
    FORWARD count
    FORWARD ALL
    BACKWARD
    BACKWARD count
    BACKWARD ALL
```

Description

FETCH retrieves rows using a previously-created cursor.

A cursor has an associated position, which is used by FETCH. The cursor position can be before the first row of the query result, on any particular row of the result, or after the last row of the result. When created, a cursor is positioned before the first row. After fetching some rows, the cursor is positioned on the row most recently retrieved. If FETCH runs off the end of the available rows then the cursor is left positioned after the last row, or before the first row if fetching backward. FETCH ALL or FETCH BACKWARD ALL will always leave the cursor positioned after the last row or before the first row.

The forms NEXT, PRIOR, FIRST, LAST, ABSOLUTE, RELATIVE fetch a single row after moving the cursor appropriately. If there is no such row, an empty result is returned, and the cursor is left positioned before the first row or after the last row as appropriate.

The forms using FORWARD and BACKWARD retrieve the indicated number of rows moving in the forward or backward direction, leaving the cursor positioned on the last-returned row (or after/before all rows, if the *count* exceeds the number of rows available).

RELATIVE 0, FORWARD 0, and BACKWARD 0 all request fetching the current row without moving the cursor, that is, re-fetching the most recently fetched row. This will succeed unless the cursor is positioned before the first row or after the last row; in which case, no row is returned.

Note: This page describes usage of cursors at the SQL command level. If you are trying to use cursors inside a PL/pgSQL function, the rules are different — see Section 40.7.3.

Parameters

direction

direction defines the fetch direction and number of rows to fetch. It can be one of the following:

NEXT

Fetch the next row. This is the default if *direction* is omitted.

PRIOR

Fetch the prior row.

FIRST

Fetch the first row of the query (same as ABSOLUTE 1).

LAST

Fetch the last row of the query (same as ABSOLUTE -1).

ABSOLUTE *count*

Fetch the *count*'th row of the query, or the abs(*count*)'th row from the end if *count* is negative. Position before first row or after last row if *count* is out of range; in particular, ABSOLUTE 0 positions before the first row.

RELATIVE *count*

Fetch the *count*'th succeeding row, or the abs(*count*)'th prior row if *count* is negative. RELATIVE 0 re-fetches the current row, if any.

count

Fetch the next *count* rows (same as FORWARD *count*).

ALL

Fetch all remaining rows (same as FORWARD ALL).

FORWARD

Fetch the next row (same as NEXT).

FORWARD *count*

Fetch the next *count* rows. FORWARD 0 re-fetches the current row.

FORWARD ALL

Fetch all remaining rows.

BACKWARD

Fetch the prior row (same as PRIOR).

`BACKWARD` *count*

Fetch the prior *count* rows (scanning backwards). `BACKWARD 0` re-fetches the current row.

`BACKWARD ALL`

Fetch all prior rows (scanning backwards).

count

count is a possibly-signed integer constant, determining the location or number of rows to fetch. For `FORWARD` and `BACKWARD` cases, specifying a negative *count* is equivalent to changing the sense of `FORWARD` and `BACKWARD`.

cursor_name

An open cursor's name.

Outputs

On successful completion, a `FETCH` command returns a command tag of the form

`FETCH` *count*

The *count* is the number of rows fetched (possibly zero). Note that in psql, the command tag will not actually be displayed, since psql displays the fetched rows instead.

Notes

The cursor should be declared with the `SCROLL` option if one intends to use any variants of `FETCH` other than `FETCH NEXT` or `FETCH FORWARD` with a positive count. For simple queries PostgreSQL will allow backwards fetch from cursors not declared with `SCROLL`, but this behavior is best not relied on. If the cursor is declared with `NO SCROLL`, no backward fetches are allowed.

`ABSOLUTE` fetches are not any faster than navigating to the desired row with a relative move: the underlying implementation must traverse all the intermediate rows anyway. Negative absolute fetches are even worse: the query must be read to the end to find the last row, and then traversed backward from there. However, rewinding to the start of the query (as with `FETCH ABSOLUTE 0`) is fast.

DECLARE is used to define a cursor. Use MOVE to change cursor position without retrieving data.

Examples

The following example traverses a table using a cursor:

```
BEGIN WORK;

-- Set up a cursor:
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;
```

```
-- Fetch the first 5 rows in the cursor liahona:
FETCH FORWARD 5 FROM liahona;

 code  |          title          | did | date_prod  |   kind   |  len
-------+-------------------------+-----+------------+----------+-------
 BL101 | The Third Man           | 101 | 1949-12-23 | Drama    | 01:44
 BL102 | The African Queen       | 101 | 1951-08-11 | Romantic | 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic | 01:25
 P_301 | Vertigo                 | 103 | 1958-11-14 | Action   | 02:08
 P_302 | Becket                  | 103 | 1964-02-03 | Drama    | 02:28

-- Fetch the previous row:
FETCH PRIOR FROM liahona;

 code  |  title  | did | date_prod  |  kind  |  len
-------+---------+-----+------------+--------+-------
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08

-- Close the cursor and end the transaction:
CLOSE liahona;
COMMIT WORK;
```

Compatibility

The SQL standard defines `FETCH` for use in embedded SQL only. The variant of `FETCH` described here returns the data as if it were a `SELECT` result rather than placing it in host variables. Other than this point, `FETCH` is fully upward-compatible with the SQL standard.

The `FETCH` forms involving `FORWARD` and `BACKWARD`, as well as the forms `FETCH` *`count`* and `FETCH ALL`, in which `FORWARD` is implicit, are PostgreSQL extensions.

The SQL standard allows only `FROM` preceding the cursor name; the option to use `IN`, or to leave them out altogether, is an extension.

See Also

CLOSE, DECLARE, MOVE

GRANT

Name

GRANT — define access privileges

Synopsis

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE sequence_name [, ...]
         | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE database_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN domain_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER fdw_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER server_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTION function_name ( [ [ argmode ] [ arg_name ] arg_type [, ...] ] ) [, ...]
         | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE lang_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
```

```
    ON LARGE OBJECT loid [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE type_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT role_name [, ...] TO role_specification [, ...]
    [ WITH ADMIN OPTION ]
    [ GRANTED BY role_specification ]

where role_specification can be:

    [ GROUP ] role_name
  | PUBLIC
  | CURRENT_USER
  | SESSION_USER
```

Description

The GRANT command has two basic variants: one that grants privileges on a database object (table, column, view, foreign table, sequence, database, foreign-data wrapper, foreign server, function, procedural language, schema, or tablespace), and one that grants membership in a role. These variants are similar in many ways, but they are different enough to be described separately.

GRANT on Database Objects

This variant of the GRANT command gives specific privileges on a database object to one or more roles. These privileges are added to those already granted, if any.

There is also an option to grant privileges on all objects of the same type within one or more schemas. This functionality is currently supported only for tables, sequences, and functions (but note that ALL TABLES is considered to include views and foreign tables).

The key word PUBLIC indicates that the privileges are to be granted to all roles, including those that might be created later. PUBLIC can be thought of as an implicitly defined group that always includes all roles. Any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to PUBLIC.

If WITH GRANT OPTION is specified, the recipient of the privilege can in turn grant it to others. Without a grant option, the recipient cannot do that. Grant options cannot be granted to PUBLIC.

There is no need to grant privileges to the owner of an object (usually the user that created it), as the owner has all privileges by default. (The owner could, however, choose to revoke some of his own privileges for safety.)

The right to drop an object, or to alter its definition in any way, is not treated as a grantable privilege; it is inherent in the owner, and cannot be granted or revoked. (However, a similar effect can be obtained by granting or revoking membership in the role that owns the object; see below.) The owner implicitly has all grant options for the object, too.

PostgreSQL grants default privileges on some types of objects to `PUBLIC`. No privileges are granted to `PUBLIC` by default on tables, table columns, sequences, foreign data wrappers, foreign servers, large objects, schemas, or tablespaces. For other types of objects, the default privileges granted to `PUBLIC` are as follows: `CONNECT` and `TEMPORARY` (create temporary tables) privileges for databases; `EXECUTE` privilege for functions; and `USAGE` privilege for languages and data types (including domains). The object owner can, of course, `REVOKE` both default and expressly granted privileges. (For maximum security, issue the `REVOKE` in the same transaction that creates the object; then there is no window in which another user can use the object.) Also, these initial default privilege settings can be changed using the ALTER DEFAULT PRIVILEGES command.

The possible privileges are:

SELECT

Allows SELECT from any column, or the specific columns listed, of the specified table, view, or sequence. Also allows the use of COPY TO. This privilege is also needed to reference existing column values in UPDATE or DELETE. For sequences, this privilege also allows the use of the `currval` function. For large objects, this privilege allows the object to be read.

INSERT

Allows INSERT of a new row into the specified table. If specific columns are listed, only those columns may be assigned to in the `INSERT` command (other columns will therefore receive default values). Also allows COPY FROM.

UPDATE

Allows UPDATE of any column, or the specific columns listed, of the specified table. (In practice, any nontrivial `UPDATE` command will require `SELECT` privilege as well, since it must reference table columns to determine which rows to update, and/or to compute new values for columns.) `SELECT ... FOR UPDATE` and `SELECT ... FOR SHARE` also require this privilege on at least one column, in addition to the `SELECT` privilege. For sequences, this privilege allows the use of the `nextval` and `setval` functions. For large objects, this privilege allows writing or truncating the object.

DELETE

Allows DELETE of a row from the specified table. (In practice, any nontrivial `DELETE` command will require `SELECT` privilege as well, since it must reference table columns to determine which rows to delete.)

TRUNCATE

Allows TRUNCATE on the specified table.

REFERENCES

To create a foreign key constraint, it is necessary to have this privilege on both the referencing and referenced columns. The privilege may be granted for all columns of a table, or just specific columns.

TRIGGER

Allows the creation of a trigger on the specified table. (See the CREATE TRIGGER statement.)

CREATE

For databases, allows new schemas to be created within the database.

For schemas, allows new objects to be created within the schema. To rename an existing object, you must own the object *and* have this privilege for the containing schema.

For tablespaces, allows tables, indexes, and temporary files to be created within the tablespace, and allows databases to be created that have the tablespace as their default tablespace. (Note that revoking this privilege will not alter the placement of existing objects.)

CONNECT

Allows the user to connect to the specified database. This privilege is checked at connection startup (in addition to checking any restrictions imposed by `pg_hba.conf`).

TEMPORARY
TEMP

Allows temporary tables to be created while using the specified database.

EXECUTE

Allows the use of the specified function and the use of any operators that are implemented on top of the function. This is the only type of privilege that is applicable to functions. (This syntax works for aggregate functions, as well.)

USAGE

For procedural languages, allows the use of the specified language for the creation of functions in that language. This is the only type of privilege that is applicable to procedural languages.

For schemas, allows access to objects contained in the specified schema (assuming that the objects' own privilege requirements are also met). Essentially this allows the grantee to "look up" objects within the schema. Without this permission, it is still possible to see the object names, e.g., by querying the system tables. Also, after revoking this permission, existing backends might have statements that have previously performed this lookup, so this is not a completely secure way to prevent object access.

For sequences, this privilege allows the use of the `currval` and `nextval` functions.

For types and domains, this privilege allow the use of the type or domain in the creation of tables, functions, and other schema objects. (Note that it does not control general "usage" of the type, such as values of the type appearing in queries. It only prevents objects from being created that depend on the type. The main purpose of the privilege is controlling which users create dependencies on a type, which could prevent the owner from changing the type later.)

For foreign-data wrappers, this privilege enables the grantee to create new servers using that foreign-data wrapper.

For servers, this privilege enables the grantee to create foreign tables using the server, and also to create, alter, or drop his own user's user mappings associated with that server.

ALL PRIVILEGES

Grant all of the available privileges at once. The `PRIVILEGES` key word is optional in PostgreSQL, though it is required by strict SQL.

The privileges required by other commands are listed on the reference page of the respective command.

GRANT on Roles

This variant of the `GRANT` command grants membership in a role to one or more other roles. Membership in a role is significant because it conveys the privileges granted to a role to each of its members.

If `WITH ADMIN OPTION` is specified, the member can in turn grant membership in the role to others, and revoke membership in the role as well. Without the admin option, ordinary users cannot do that. A role is not considered to hold `WITH ADMIN OPTION` on itself, but it may grant or revoke membership in itself from a database session where the session user matches the role. Database superusers can grant or revoke membership in any role to anyone. Roles having `CREATEROLE` privilege can grant or revoke membership in any role that is not a superuser.

If `GRANTED BY` is specified, the grant is recorded as having been done by the specified role. Only database superusers may use this option, except when it names the same role executing the command.

Unlike the case with privileges, membership in a role cannot be granted to `PUBLIC`. Note also that this form of the command does not allow the noise word `GROUP` in *`role_specification`*.

Notes

The REVOKE command is used to revoke access privileges.

Since PostgreSQL 8.1, the concepts of users and groups have been unified into a single kind of entity called a role. It is therefore no longer necessary to use the keyword `GROUP` to identify whether a grantee is a user or a group. `GROUP` is still allowed in the command, but it is a noise word.

A user may perform `SELECT`, `INSERT`, etc. on a column if he holds that privilege for either the specific column or its whole table. Granting the privilege at the table level and then revoking it for one column will not do what you might wish: the table-level grant is unaffected by a column-level operation.

When a non-owner of an object attempts to `GRANT` privileges on the object, the command will fail outright if the user has no privileges whatsoever on the object. As long as some privilege is available, the command will proceed, but it will grant only those privileges for which the user has grant options. The `GRANT ALL PRIVILEGES` forms will issue a warning message if no grant options are held, while the other forms will issue a warning if grant options for any of the privileges specifically named in the command are not held. (In principle these statements apply to the object owner as well, but since the owner is always treated as holding all grant options, the cases can never occur.)

It should be noted that database superusers can access all objects regardless of object privilege settings. This is comparable to the rights of `root` in a Unix system. As with `root`, it's unwise to operate as a superuser except when absolutely necessary.

If a superuser chooses to issue a `GRANT` or `REVOKE` command, the command is performed as though it were issued by the owner of the affected object. In particular, privileges granted via such a command will

appear to have been granted by the object owner. (For role membership, the membership appears to have been granted by the containing role itself.)

`GRANT` and `REVOKE` can also be done by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges `WITH GRANT OPTION` on the object. In this case the privileges will be recorded as having been granted by the role that actually owns the object or holds the privileges `WITH GRANT OPTION`. For example, if table `t1` is owned by role `g1`, of which role `u1` is a member, then `u1` can grant privileges on `t1` to `u2`, but those privileges will appear to have been granted directly by `g1`. Any other member of role `g1` could revoke them later.

If the role executing `GRANT` holds the required privileges indirectly via more than one role membership path, it is unspecified which containing role will be recorded as having done the grant. In such cases it is best practice to use `SET ROLE` to become the specific role you want to do the `GRANT` as.

Granting permission on a table does not automatically extend permissions to any sequences used by the table, including sequences tied to `SERIAL` columns. Permissions on sequences must be set separately.

Use psql's `\dp` command to obtain information about existing privileges for tables and columns. For example:

```
=> \dp mytable
                               Access privileges
 Schema |  Name   | Type  |   Access privileges   | Column access privileges
--------+---------+-------+-----------------------+--------------------------
 public | mytable | table | miriam=arwdDxt/miriam | col1:
                          : =r/miriam             :   miriam_rw=rw/miriam
                          : admin=arw/miriam
(1 row)
```

The entries shown by `\dp` are interpreted thus:

```
rolename=xxxx -- privileges granted to a role
        =xxxx -- privileges granted to PUBLIC

            r -- SELECT ("read")
            w -- UPDATE ("write")
            a -- INSERT ("append")
            d -- DELETE
            D -- TRUNCATE
            x -- REFERENCES
            t -- TRIGGER
            X -- EXECUTE
            U -- USAGE
            C -- CREATE
            c -- CONNECT
            T -- TEMPORARY
      arwdDxt -- ALL PRIVILEGES (for tables, varies for other objects)
            * -- grant option for preceding privilege

        /yyyy -- role that granted this privilege
```

The above example display would be seen by user `miriam` after creating table `mytable` and doing:

```
GRANT SELECT ON mytable TO PUBLIC;
```

```
GRANT SELECT, UPDATE, INSERT ON mytable TO admin;
GRANT SELECT (col1), UPDATE (col1) ON mytable TO miriam_rw;
```

For non-table objects there are other `\d` commands that can display their privileges.

If the "Access privileges" column is empty for a given object, it means the object has default privileges (that is, its privileges column is null). Default privileges always include all privileges for the owner, and can include some privileges for `PUBLIC` depending on the object type, as explained above. The first `GRANT` or `REVOKE` on an object will instantiate the default privileges (producing, for example, `{miriam=arwdDxt/miriam}`) and then modify them per the specified request. Similarly, entries are shown in "Column access privileges" only for columns with nondefault privileges. (Note: for this purpose, "default privileges" always means the built-in default privileges for the object's type. An object whose privileges have been affected by an `ALTER DEFAULT PRIVILEGES` command will always be shown with an explicit privilege entry that includes the effects of the `ALTER`.)

Notice that the owner's implicit grant options are not marked in the access privileges display. A `*` will appear only when grant options have been explicitly granted to someone.

Examples

Grant insert privilege to all users on table `films`:

```
GRANT INSERT ON films TO PUBLIC;
```

Grant all available privileges to user `manuel` on view `kinds`:

```
GRANT ALL PRIVILEGES ON kinds TO manuel;
```

Note that while the above will indeed grant all privileges if executed by a superuser or the owner of `kinds`, when executed by someone else it will only grant those permissions for which the someone else has grant options.

Grant membership in role `admins` to user `joe`:

```
GRANT admins TO joe;
```

Compatibility

According to the SQL standard, the `PRIVILEGES` key word in `ALL PRIVILEGES` is required. The SQL standard does not support setting the privileges on more than one object per command.

PostgreSQL allows an object owner to revoke his own ordinary privileges: for example, a table owner can make the table read-only to himself by revoking his own `INSERT`, `UPDATE`, `DELETE`, and `TRUNCATE` privileges. This is not possible according to the SQL standard. The reason is that PostgreSQL treats the owner's privileges as having been granted by the owner to himself; therefore he can revoke them too.

In the SQL standard, the owner's privileges are granted by an assumed entity "_SYSTEM". Not being "_SYSTEM", the owner cannot revoke these rights.

According to the SQL standard, grant options can be granted to `PUBLIC`; PostgreSQL only supports granting grant options to roles.

The SQL standard allows the `GRANTED BY` option to be used in all forms of `GRANT`. PostgreSQL only supports it when granting role membership, and even then only superusers may use it in nontrivial ways.

The SQL standard provides for a `USAGE` privilege on other kinds of objects: character sets, collations, translations.

In the SQL standard, sequences only have a `USAGE` privilege, which controls the use of the `NEXT VALUE FOR` expression, which is equivalent to the function `nextval` in PostgreSQL. The sequence privileges `SELECT` and `UPDATE` are PostgreSQL extensions. The application of the sequence `USAGE` privilege to the `currval` function is also a PostgreSQL extension (as is the function itself).

Privileges on databases, tablespaces, schemas, and languages are PostgreSQL extensions.

See Also

REVOKE, ALTER DEFAULT PRIVILEGES

IMPORT FOREIGN SCHEMA

Name

IMPORT FOREIGN SCHEMA — import table definitions from a foreign server

Synopsis

```
IMPORT FOREIGN SCHEMA remote_schema
    [ { LIMIT TO | EXCEPT } ( table_name [, ...] ) ]
    FROM SERVER server_name
    INTO local_schema
    [ OPTIONS ( option 'value' [, ... ] ) ]
```

Description

IMPORT FOREIGN SCHEMA creates foreign tables that represent tables existing on a foreign server. The new foreign tables will be owned by the user issuing the command and are created with the correct column definitions and options to match the remote tables.

By default, all tables and views existing in a particular schema on the foreign server are imported. Optionally, the list of tables can be limited to a specified subset, or specific tables can be excluded. The new foreign tables are all created in the target schema, which must already exist.

To use IMPORT FOREIGN SCHEMA, the user must have USAGE privilege on the foreign server, as well as CREATE privilege on the target schema.

Parameters

remote_schema

The remote schema to import from. The specific meaning of a remote schema depends on the foreign data wrapper in use.

LIMIT TO (*table_name* [, ...])

Import only foreign tables matching one of the given table names. Other tables existing in the foreign schema will be ignored.

EXCEPT (*table_name* [, ...])

Exclude specified foreign tables from the import. All tables existing in the foreign schema will be imported except the ones listed here.

server_name

The foreign server to import from.

local_schema

The schema in which the imported foreign tables will be created.

`OPTIONS ( *option* '*value*' [, ...] )`

Options to be used during the import. The allowed option names and values are specific to each foreign data wrapper.

Examples

Import table definitions from a remote schema `foreign_films` on server `film_server`, creating the foreign tables in local schema `films`:

```
IMPORT FOREIGN SCHEMA foreign_films
    FROM SERVER film_server INTO films;
```

As above, but import only the two tables `actors` and `directors` (if they exist):

```
IMPORT FOREIGN SCHEMA foreign_films LIMIT TO (actors, directors)
    FROM SERVER film_server INTO films;
```

Compatibility

The `IMPORT FOREIGN SCHEMA` command conforms to the SQL standard, except that the `OPTIONS` clause is a PostgreSQL extension.

See Also

CREATE FOREIGN TABLE, CREATE SERVER

INSERT

Name

INSERT — create new rows in a table

Synopsis

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
    [ ON CONFLICT [ conflict_target ] conflict_action ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]

where conflict_target can be one of:

    ( { index_column_name | ( index_expression ) } [ COLLATE collation ] [ opclass ] [, ...] ) [
    ON CONSTRAINT constraint_name

and conflict_action is one of:

    DO NOTHING
    DO UPDATE SET { column_name = { expression | DEFAULT } |
                    ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] ) |
                    ( column_name [, ...] ) = ( sub-SELECT )
                  } [, ...]
              [ WHERE condition ]
```

Description

INSERT inserts new rows into a table. One can insert one or more rows specified by value expressions, or zero or more rows resulting from a query.

The target column names can be listed in any order. If no list of column names is given at all, the default is all the columns of the table in their declared order; or the first *N* column names, if there are only *N* columns supplied by the VALUES clause or *query*. The values supplied by the VALUES clause or *query* are associated with the explicit or implicit column list left-to-right.

Each column not present in the explicit or implicit column list will be filled with a default value, either its declared default value or null if there is none.

If the expression for any column is not of the correct data type, automatic type conversion will be attempted.

ON CONFLICT can be used to specify an alternative action to raising a unique constraint or exclusion constraint violation error. (See *ON CONFLICT Clause* below.)

The optional RETURNING clause causes INSERT to compute and return value(s) based on each row actually inserted (or updated, if an ON CONFLICT DO UPDATE clause was used). This is primarily useful

for obtaining values that were supplied by defaults, such as a serial sequence number. However, any expression using the table's columns is allowed. The syntax of the RETURNING list is identical to that of the output list of SELECT. Only rows that were successfully inserted or updated will be returned. For example, if a row was locked but not updated because an ON CONFLICT DO UPDATE ... WHERE clause *condition* was not satisfied, the row will not be returned.

You must have INSERT privilege on a table in order to insert into it. If ON CONFLICT DO UPDATE is present, UPDATE privilege on the table is also required.

If a column list is specified, you only need INSERT privilege on the listed columns. Similarly, when ON CONFLICT DO UPDATE is specified, you only need UPDATE privilege on the column(s) that are listed to be updated. However, ON CONFLICT DO UPDATE also requires SELECT privilege on any column whose values are read in the ON CONFLICT DO UPDATE expressions or *condition*.

Use of the RETURNING clause requires SELECT privilege on all columns mentioned in RETURNING. If you use the *query* clause to insert rows from a query, you of course need to have SELECT privilege on any table or column used in the query.

Parameters

Inserting

This section covers parameters that may be used when only inserting new rows. Parameters *exclusively* used with the ON CONFLICT clause are described separately.

with_query

The WITH clause allows you to specify one or more subqueries that can be referenced by name in the INSERT query. See Section 7.8 and SELECT for details.

It is possible for the *query* (SELECT statement) to also contain a WITH clause. In such a case both sets of *with_query* can be referenced within the *query*, but the second one takes precedence since it is more closely nested.

table_name

The name (optionally schema-qualified) of an existing table.

alias

A substitute name for *table_name*. When an alias is provided, it completely hides the actual name of the table. This is particularly useful when ON CONFLICT DO UPDATE targets a table named excluded, since that's also the name of the special table representing rows proposed for insertion.

column_name

The name of a column in the table named by *table_name*. The column name can be qualified with a subfield name or array subscript, if needed. (Inserting into only some fields of a composite column leaves the other fields null.) When referencing a column with ON CONFLICT DO UPDATE, do not include the table's name in the specification of a target column. For example, INSERT INTO table_name ... ON CONFLICT DO UPDATE SET table_name.col = 1 is invalid (this follows the general behavior for UPDATE).

`DEFAULT VALUES`

All columns will be filled with their default values.

`expression`

An expression or value to assign to the corresponding column.

`DEFAULT`

The corresponding column will be filled with its default value.

`query`

A query (`SELECT` statement) that supplies the rows to be inserted. Refer to the SELECT statement for a description of the syntax.

`output_expression`

An expression to be computed and returned by the `INSERT` command after each row is inserted or updated. The expression can use any column names of the table named by *`table_name`*. Write `*` to return all columns of the inserted or updated row(s).

`output_name`

A name to use for a returned column.

`ON CONFLICT` Clause

The optional `ON CONFLICT` clause specifies an alternative action to raising a unique violation or exclusion constraint violation error. For each individual row proposed for insertion, either the insertion proceeds, or, if an *arbiter* constraint or index specified by `conflict_target` is violated, the alternative `conflict_action` is taken. `ON CONFLICT DO NOTHING` simply avoids inserting a row as its alternative action. `ON CONFLICT DO UPDATE` updates the existing row that conflicts with the row proposed for insertion as its alternative action.

`conflict_target` can perform *unique index inference*. When performing inference, it consists of one or more *`index_column_name`* columns and/or *`index_expression`* expressions, and an optional *`index_predicate`*. All *`table_name`* unique indexes that, without regard to order, contain exactly the `conflict_target`-specified columns/expressions are inferred (chosen) as arbiter indexes. If an *`index_predicate`* is specified, it must, as a further requirement for inference, satisfy arbiter indexes. Note that this means a non-partial unique index (a unique index without a predicate) will be inferred (and thus used by `ON CONFLICT`) if such an index satisfying every other criteria is available. If an attempt at inference is unsuccessful, an error is raised.

`ON CONFLICT DO UPDATE` guarantees an atomic `INSERT` or `UPDATE` outcome; provided there is no independent error, one of those two outcomes is guaranteed, even under high concurrency. This is also known as *UPSERT* — "UPDATE or INSERT".

`conflict_target`

Specifies which conflicts `ON CONFLICT` takes the alternative action on by choosing *arbiter indexes*. Either performs *unique index inference*, or names a constraint explicitly. For `ON CONFLICT DO NOTHING`, it is optional to specify a `conflict_target`; when omitted, conflicts with all usable constraints (and unique indexes) are handled. For `ON CONFLICT DO UPDATE`, a `conflict_target` *must* be provided.

conflict_action

`conflict_action` specifies an alternative `ON CONFLICT` action. It can be either `DO NOTHING`, or a `DO UPDATE` clause specifying the exact details of the `UPDATE` action to be performed in case of a conflict. The `SET` and `WHERE` clauses in `ON CONFLICT DO UPDATE` have access to the existing row using the table's name (or an alias), and to rows proposed for insertion using the special `excluded` table. `SELECT` privilege is required on any column in the target table where corresponding `excluded` columns are read.

Note that the effects of all per-row `BEFORE INSERT` triggers are reflected in `excluded` values, since those effects may have contributed to the row being excluded from insertion.

index_column_name

The name of a *table_name* column. Used to infer arbiter indexes. Follows `CREATE INDEX` format. `SELECT` privilege on *index_column_name* is required.

index_expression

Similar to *index_column_name*, but used to infer expressions on *table_name* columns appearing within index definitions (not simple columns). Follows `CREATE INDEX` format. `SELECT` privilege on any column appearing within *index_expression* is required.

collation

When specified, mandates that corresponding *index_column_name* or *index_expression* use a particular collation in order to be matched during inference. Typically this is omitted, as collations usually do not affect whether or not a constraint violation occurs. Follows `CREATE INDEX` format.

opclass

When specified, mandates that corresponding *index_column_name* or *index_expression* use particular operator class in order to be matched during inference. Typically this is omitted, as the *equality* semantics are often equivalent across a type's operator classes anyway, or because it's sufficient to trust that the defined unique indexes have the pertinent definition of equality. Follows `CREATE INDEX` format.

index_predicate

Used to allow inference of partial unique indexes. Any indexes that satisfy the predicate (which need not actually be partial indexes) can be inferred. Follows `CREATE INDEX` format. `SELECT` privilege on any column appearing within *index_predicate* is required.

constraint_name

Explicitly specifies an arbiter *constraint* by name, rather than inferring a constraint or index.

condition

An expression that returns a value of type `boolean`. Only rows for which this expression returns `true` will be updated, although all rows will be locked when the `ON CONFLICT DO UPDATE` action is taken. Note that *condition* is evaluated last, after a conflict has been identified as a candidate to update.

Note that exclusion constraints are not supported as arbiters with `ON CONFLICT DO UPDATE`. In all cases, only `NOT DEFERRABLE` constraints and unique indexes are supported as arbiters.

`INSERT` with an `ON CONFLICT DO UPDATE` clause is a "deterministic" statement. This means that the command will not be allowed to affect any single existing row more than once; a cardinality violation error

will be raised when this situation arises. Rows proposed for insertion should not duplicate each other in terms of attributes constrained by an arbiter index or constraint.

> **Tip:** It is often preferable to use unique index inference rather than naming a constraint directly using `ON CONFLICT ON CONSTRAINT` *`constraint_name`*. Inference will continue to work correctly when the underlying index is replaced by another more or less equivalent index in an overlapping way, for example when using `CREATE UNIQUE INDEX ... CONCURRENTLY` before dropping the index being replaced.

Outputs

On successful completion, an `INSERT` command returns a command tag of the form

```
INSERT oid count
```

The *`count`* is the number of rows inserted or updated. If *`count`* is exactly one, and the target table has OIDs, then *`oid`* is the OID assigned to the inserted row. The single row must have been inserted rather than updated. Otherwise *`oid`* is zero.

If the `INSERT` command contains a `RETURNING` clause, the result will be similar to that of a `SELECT` statement containing the columns and values defined in the `RETURNING` list, computed over the row(s) inserted or updated by the command.

Examples

Insert a single row into table `films`:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
```

In this example, the `len` column is omitted and therefore it will have the default value:

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

This example uses the `DEFAULT` clause for the date columns rather than specifying a value:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes');
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');
```

To insert a row consisting entirely of default values:

```
INSERT INTO films DEFAULT VALUES;
```

To insert multiple rows using the multirow `VALUES` syntax:

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
    ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
    ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

This example inserts some rows into table `films` from a table `tmp_films` with the same column layout as `films`:

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

This example inserts into array columns:

```
-- Create an empty 3x3 gameboard for noughts-and-crosses
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1, '{{" "," "," "},{" "," "," "},{" "," "," "}}');
-- The subscripts in the above example aren't really needed
INSERT INTO tictactoe (game, board)
    VALUES (2, '{{X," "," "},{" ",O," "},{" ",X," "}}');
```

Insert a single row into table `distributors`, returning the sequence number generated by the `DEFAULT` clause:

```
INSERT INTO distributors (did, dname) VALUES (DEFAULT, 'XYZ Widgets')
   RETURNING did;
```

Increment the sales count of the salesperson who manages the account for Acme Corporation, and record the whole updated row along with current time in a log table:

```
WITH upd AS (
  UPDATE employees SET sales_count = sales_count + 1 WHERE id =
    (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation')
    RETURNING *
)
INSERT INTO employees_log SELECT *, current_timestamp FROM upd;
```

Insert or update new distributors as appropriate. Assumes a unique index has been defined that constrains values appearing in the `did` column. Note that the special `excluded` table is used to reference values originally proposed for insertion:

```
INSERT INTO distributors (did, dname)
    VALUES (5, 'Gizmo Transglobal'), (6, 'Associated Computing, Inc')
```

```
    ON CONFLICT (did) DO UPDATE SET dname = EXCLUDED.dname;
```

Insert a distributor, or do nothing for rows proposed for insertion when an existing, excluded row (a row with a matching constrained column or columns after before row insert triggers fire) exists. Example assumes a unique index has been defined that constrains values appearing in the `did` column:

```
INSERT INTO distributors (did, dname) VALUES (7, 'Redline GmbH')
    ON CONFLICT (did) DO NOTHING;
```

Insert or update new distributors as appropriate. Example assumes a unique index has been defined that constrains values appearing in the `did` column. `WHERE` clause is used to limit the rows actually updated (any existing row not updated will still be locked, though):

```
-- Don't update existing distributors based in a certain ZIP code
INSERT INTO distributors AS d (did, dname) VALUES (8, 'Anvil Distribution')
    ON CONFLICT (did) DO UPDATE
    SET dname = EXCLUDED.dname || ' (formerly ' || d.dname || ')'
    WHERE d.zipcode <> '21201';

-- Name a constraint directly in the statement (uses associated
-- index to arbitrate taking the DO NOTHING action)
INSERT INTO distributors (did, dname) VALUES (9, 'Antwerp Design')
    ON CONFLICT ON CONSTRAINT distributors_pkey DO NOTHING;
```

Insert new distributor if possible; otherwise `DO NOTHING`. Example assumes a unique index has been defined that constrains values appearing in the `did` column on a subset of rows where the `is_active` Boolean column evaluates to `true`:

```
-- This statement could infer a partial unique index on "did"
-- with a predicate of "WHERE is_active", but it could also
-- just use a regular unique constraint on "did"
INSERT INTO distributors (did, dname) VALUES (10, 'Conrad International')
    ON CONFLICT (did) WHERE is_active DO NOTHING;
```

Compatibility

`INSERT` conforms to the SQL standard, except that the `RETURNING` clause is a PostgreSQL extension, as is the ability to use `WITH` with `INSERT`, and the ability to specify an alternative action with `ON CONFLICT`. Also, the case in which a column name list is omitted, but not all the columns are filled from the `VALUES` clause or *`query`*, is disallowed by the standard.

Possible limitations of the *`query`* clause are documented under SELECT.

LISTEN

Name

LISTEN — listen for a notification

Synopsis

```
LISTEN channel
```

Description

LISTEN registers the current session as a listener on the notification channel named *channel*. If the current session is already registered as a listener for this notification channel, nothing is done.

Whenever the command NOTIFY *channel* is invoked, either by this session or another one connected to the same database, all the sessions currently listening on that notification channel are notified, and each will in turn notify its connected client application.

A session can be unregistered for a given notification channel with the UNLISTEN command. A session's listen registrations are automatically cleared when the session ends.

The method a client application must use to detect notification events depends on which PostgreSQL application programming interface it uses. With the libpq library, the application issues LISTEN as an ordinary SQL command, and then must periodically call the function PQnotifies to find out whether any notification events have been received. Other interfaces such as libpgtcl provide higher-level methods for handling notify events; indeed, with libpgtcl the application programmer should not even issue LISTEN or UNLISTEN directly. See the documentation for the interface you are using for more details.

NOTIFY contains a more extensive discussion of the use of LISTEN and NOTIFY.

Parameters

channel

Name of a notification channel (any identifier).

Notes

LISTEN takes effect at transaction commit. If LISTEN or UNLISTEN is executed within a transaction that later rolls back, the set of notification channels being listened to is unchanged.

A transaction that has executed LISTEN cannot be prepared for two-phase commit.

Examples

Configure and execute a listen/notify sequence from psql:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

Compatibility

There is no `LISTEN` statement in the SQL standard.

See Also

NOTIFY, UNLISTEN

LOAD

Name

LOAD — load a shared library file

Synopsis

```
LOAD 'filename'
```

Description

This command loads a shared library file into the PostgreSQL server's address space. If the file has been loaded already, the command does nothing. Shared library files that contain C functions are automatically loaded whenever one of their functions is called. Therefore, an explicit LOAD is usually only needed to load a library that modifies the server's behavior through "hooks" rather than providing a set of functions.

The file name is specified in the same way as for shared library names in CREATE FUNCTION; in particular, one can rely on a search path and automatic addition of the system's standard shared library file name extension. See Section 35.9 for more information on this topic.

Non-superusers can only apply LOAD to library files located in `$libdir/plugins/` — the specified *filename* must begin with exactly that string. (It is the database administrator's responsibility to ensure that only "safe" libraries are installed there.)

Compatibility

LOAD is a PostgreSQL extension.

See Also

CREATE FUNCTION

LOCK

Name

LOCK — lock a table

Synopsis

```
LOCK [ TABLE ] [ ONLY ] name [ * ] [, ...] [ IN lockmode MODE ] [ NOWAIT ]

where lockmode is one of:

    ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
    | SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE
```

Description

LOCK TABLE obtains a table-level lock, waiting if necessary for any conflicting locks to be released. If NOWAIT is specified, LOCK TABLE does not wait to acquire the desired lock: if it cannot be acquired immediately, the command is aborted and an error is emitted. Once obtained, the lock is held for the remainder of the current transaction. (There is no UNLOCK TABLE command; locks are always released at transaction end.)

When acquiring locks automatically for commands that reference tables, PostgreSQL always uses the least restrictive lock mode possible. LOCK TABLE provides for cases when you might need more restrictive locking. For example, suppose an application runs a transaction at the READ COMMITTED isolation level and needs to ensure that data in a table remains stable for the duration of the transaction. To achieve this you could obtain SHARE lock mode over the table before querying. This will prevent concurrent data changes and ensure subsequent reads of the table see a stable view of committed data, because SHARE lock mode conflicts with the ROW EXCLUSIVE lock acquired by writers, and your LOCK TABLE *name* IN SHARE MODE statement will wait until any concurrent holders of ROW EXCLUSIVE mode locks commit or roll back. Thus, once you obtain the lock, there are no uncommitted writes outstanding; furthermore none can begin until you release the lock.

To achieve a similar effect when running a transaction at the REPEATABLE READ or SERIALIZABLE isolation level, you have to execute the LOCK TABLE statement before executing any SELECT or data modification statement. A REPEATABLE READ or SERIALIZABLE transaction's view of data will be frozen when its first SELECT or data modification statement begins. A LOCK TABLE later in the transaction will still prevent concurrent writes — but it won't ensure that what the transaction reads corresponds to the latest committed values.

If a transaction of this sort is going to change the data in the table, then it should use SHARE ROW EXCLUSIVE lock mode instead of SHARE mode. This ensures that only one transaction of this type runs at a time. Without this, a deadlock is possible: two transactions might both acquire SHARE mode, and then be unable to also acquire ROW EXCLUSIVE mode to actually perform their updates. (Note that a transaction's own locks never conflict, so a transaction can acquire ROW EXCLUSIVE mode when it holds SHARE mode — but not if anyone else holds SHARE mode.) To avoid deadlocks, make sure all transactions acquire locks

on the same objects in the same order, and if multiple lock modes are involved for a single object, then transactions should always acquire the most restrictive mode first.

More information about the lock modes and locking strategies can be found in Section 13.3.

Parameters

name

The name (optionally schema-qualified) of an existing table to lock. If ONLY is specified before the table name, only that table is locked. If ONLY is not specified, the table and all its descendant tables (if any) are locked. Optionally, * can be specified after the table name to explicitly indicate that descendant tables are included.

The command LOCK TABLE a, b; is equivalent to LOCK TABLE a; LOCK TABLE b;. The tables are locked one-by-one in the order specified in the LOCK TABLE command.

lockmode

The lock mode specifies which locks this lock conflicts with. Lock modes are described in Section 13.3.

If no lock mode is specified, then ACCESS EXCLUSIVE, the most restrictive mode, is used.

NOWAIT

Specifies that LOCK TABLE should not wait for any conflicting locks to be released: if the specified lock(s) cannot be acquired immediately without waiting, the transaction is aborted.

Notes

LOCK TABLE ... IN ACCESS SHARE MODE requires SELECT privileges on the target table. LOCK TABLE ... IN ROW EXCLUSIVE MODE requires INSERT, UPDATE, DELETE, or TRUNCATE privileges on the target table. All other forms of LOCK require table-level UPDATE, DELETE, or TRUNCATE privileges.

LOCK TABLE is useless outside a transaction block: the lock would remain held only to the completion of the statement. Therefore PostgreSQL reports an error if LOCK is used outside a transaction block. Use BEGIN and COMMIT (or ROLLBACK) to define a transaction block.

LOCK TABLE only deals with table-level locks, and so the mode names involving ROW are all misnomers. These mode names should generally be read as indicating the intention of the user to acquire row-level locks within the locked table. Also, ROW EXCLUSIVE mode is a shareable table lock. Keep in mind that all the lock modes have identical semantics so far as LOCK TABLE is concerned, differing only in the rules about which modes conflict with which. For information on how to acquire an actual row-level lock, see Section 13.3.2 and the *The Locking Clause* in the SELECT reference documentation.

Examples

Obtain a `SHARE` lock on a primary key table when going to perform inserts into a foreign key table:

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- Do ROLLBACK if record was not returned
INSERT INTO films_user_comments VALUES
    (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

Take a `SHARE ROW EXCLUSIVE` lock on a primary key table when going to perform a delete operation:

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
    (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

Compatibility

There is no `LOCK TABLE` in the SQL standard, which instead uses `SET TRANSACTION` to specify concurrency levels on transactions. PostgreSQL supports that too; see SET TRANSACTION for details.

Except for `ACCESS SHARE`, `ACCESS EXCLUSIVE`, and `SHARE UPDATE EXCLUSIVE` lock modes, the PostgreSQL lock modes and the `LOCK TABLE` syntax are compatible with those present in Oracle.

MOVE

Name

MOVE — position a cursor

Synopsis

```
MOVE [ direction [ FROM | IN ] ] cursor_name
```

where *direction* can be empty or one of:

```
    NEXT
    PRIOR
    FIRST
    LAST
    ABSOLUTE count
    RELATIVE count
    count
    ALL
    FORWARD
    FORWARD count
    FORWARD ALL
    BACKWARD
    BACKWARD count
    BACKWARD ALL
```

Description

MOVE repositions a cursor without retrieving any data. MOVE works exactly like the FETCH command, except it only positions the cursor and does not return rows.

The parameters for the MOVE command are identical to those of the FETCH command; refer to FETCH for details on syntax and usage.

Outputs

On successful completion, a MOVE command returns a command tag of the form

```
MOVE count
```

The *count* is the number of rows that a FETCH command with the same parameters would have returned (possibly zero).

Examples

```
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;

-- Skip the first 5 rows:
MOVE FORWARD 5 IN liahona;
MOVE 5

-- Fetch the 6th row from the cursor liahona:
FETCH 1 FROM liahona;
 code  | title  | did | date_prod  |  kind  |  len
-------+--------+-----+------------+--------+-------
 P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)

-- Close the cursor liahona and end the transaction:
CLOSE liahona;
COMMIT WORK;
```

Compatibility

There is no `MOVE` statement in the SQL standard.

See Also

CLOSE, DECLARE, FETCH

NOTIFY

Name

NOTIFY — generate a notification

Synopsis

```
NOTIFY channel [ , payload ]
```

Description

The NOTIFY command sends a notification event together with an optional "payload" string to each client application that has previously executed LISTEN *channel* for the specified channel name in the current database. Notifications are visible to all users.

NOTIFY provides a simple interprocess communication mechanism for a collection of processes accessing the same PostgreSQL database. A payload string can be sent along with the notification, and higher-level mechanisms for passing structured data can be built by using tables in the database to pass additional data from notifier to listener(s).

The information passed to the client for a notification event includes the notification channel name, the notifying session's server process PID, and the payload string, which is an empty string if it has not been specified.

It is up to the database designer to define the channel names that will be used in a given database and what each one means. Commonly, the channel name is the same as the name of some table in the database, and the notify event essentially means, "I changed this table, take a look at it to see what's new". But no such association is enforced by the NOTIFY and LISTEN commands. For example, a database designer could use several different channel names to signal different sorts of changes to a single table. Alternatively, the payload string could be used to differentiate various cases.

When NOTIFY is used to signal the occurrence of changes to a particular table, a useful programming technique is to put the NOTIFY in a rule that is triggered by table updates. In this way, notification happens automatically when the table is changed, and the application programmer cannot accidentally forget to do it.

NOTIFY interacts with SQL transactions in some important ways. Firstly, if a NOTIFY is executed inside a transaction, the notify events are not delivered until and unless the transaction is committed. This is appropriate, since if the transaction is aborted, all the commands within it have had no effect, including NOTIFY. But it can be disconcerting if one is expecting the notification events to be delivered immediately. Secondly, if a listening session receives a notification signal while it is within a transaction, the notification event will not be delivered to its connected client until just after the transaction is completed (either committed or aborted). Again, the reasoning is that if a notification were delivered within a transaction that was later aborted, one would want the notification to be undone somehow — but the server cannot "take back" a notification once it has sent it to the client. So notification events are only delivered between transactions. The upshot of this is that applications using NOTIFY for real-time signaling should try to keep their transactions short.

If the same channel name is signaled multiple times from the same transaction with identical payload strings, the database server can decide to deliver a single notification only. On the other hand, notifications with distinct payload strings will always be delivered as distinct notifications. Similarly, notifications from different transactions will never get folded into one notification. Except for dropping later instances of duplicate notifications, `NOTIFY` guarantees that notifications from the same transaction get delivered in the order they were sent. It is also guaranteed that messages from different transactions are delivered in the order in which the transactions committed.

It is common for a client that executes `NOTIFY` to be listening on the same notification channel itself. In that case it will get back a notification event, just like all the other listening sessions. Depending on the application logic, this could result in useless work, for example, reading a database table to find the same updates that that session just wrote out. It is possible to avoid such extra work by noticing whether the notifying session's server process PID (supplied in the notification event message) is the same as one's own session's PID (available from libpq). When they are the same, the notification event is one's own work bouncing back, and can be ignored.

Parameters

`channel`

Name of the notification channel to be signaled (any identifier).

`payload`

The "payload" string to be communicated along with the notification. This must be specified as a simple string literal. In the default configuration it must be shorter than 8000 bytes. (If binary data or large amounts of information need to be communicated, it's best to put it in a database table and send the key of the record.)

Notes

There is a queue that holds notifications that have been sent but not yet processed by all listening sessions. If this queue becomes full, transactions calling `NOTIFY` will fail at commit. The queue is quite large (8GB in a standard installation) and should be sufficiently sized for almost every use case. However, no cleanup can take place if a session executes `LISTEN` and then enters a transaction for a very long time. Once the queue is half full you will see warnings in the log file pointing you to the session that is preventing cleanup. In this case you should make sure that this session ends its current transaction so that cleanup can proceed.

A transaction that has executed `NOTIFY` cannot be prepared for two-phase commit.

pg_notify

To send a notification you can also use the function `pg_notify(text, text)`. The function takes the channel name as the first argument and the payload as the second. The function is much easier to use than the `NOTIFY` command if you need to work with non-constant channel names and payloads.

Examples

Configure and execute a listen/notify sequence from psql:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
NOTIFY virtual, 'This is the payload';
Asynchronous notification "virtual" with payload "This is the payload" received from server

LISTEN foo;
SELECT pg_notify('fo' || 'o', 'pay' || 'load');
Asynchronous notification "foo" with payload "payload" received from server process with PI
```

Compatibility

There is no `NOTIFY` statement in the SQL standard.

See Also

LISTEN, UNLISTEN

PREPARE

Name

PREPARE — prepare a statement for execution

Synopsis

```
PREPARE name [ ( data_type [, ...] ) ] AS statement
```

Description

PREPARE creates a prepared statement. A prepared statement is a server-side object that can be used to optimize performance. When the PREPARE statement is executed, the specified statement is parsed, analyzed, and rewritten. When an EXECUTE command is subsequently issued, the prepared statement is planned and executed. This division of labor avoids repetitive parse analysis work, while allowing the execution plan to depend on the specific parameter values supplied.

Prepared statements can take parameters: values that are substituted into the statement when it is executed. When creating the prepared statement, refer to parameters by position, using $1, $2, etc. A corresponding list of parameter data types can optionally be specified. When a parameter's data type is not specified or is declared as unknown, the type is inferred from the context in which the parameter is first used (if possible). When executing the statement, specify the actual values for these parameters in the EXECUTE statement. Refer to EXECUTE for more information about that.

Prepared statements only last for the duration of the current database session. When the session ends, the prepared statement is forgotten, so it must be recreated before being used again. This also means that a single prepared statement cannot be used by multiple simultaneous database clients; however, each client can create their own prepared statement to use. Prepared statements can be manually cleaned up using the DEALLOCATE command.

Prepared statements have the largest performance advantage when a single session is being used to execute a large number of similar statements. The performance difference will be particularly significant if the statements are complex to plan or rewrite, for example, if the query involves a join of many tables or requires the application of several rules. If the statement is relatively simple to plan and rewrite but relatively expensive to execute, the performance advantage of prepared statements will be less noticeable.

Parameters

name

An arbitrary name given to this particular prepared statement. It must be unique within a single session and is subsequently used to execute or deallocate a previously prepared statement.

data_type

The data type of a parameter to the prepared statement. If the data type of a particular parameter is unspecified or is specified as `unknown`, it will be inferred from the context in which the parameter is first used. To refer to the parameters in the prepared statement itself, use `$1`, `$2`, etc.

statement

Any `SELECT`, `INSERT`, `UPDATE`, `DELETE`, or `VALUES` statement.

Notes

If a prepared statement is executed enough times, the server may eventually decide to save and re-use a generic plan rather than re-planning each time. This will occur immediately if the prepared statement has no parameters; otherwise it occurs only if the generic plan appears to be not much more expensive than a plan that depends on specific parameter values. Typically, a generic plan will be selected only if the query's performance is estimated to be fairly insensitive to the specific parameter values supplied.

To examine the query plan PostgreSQL is using for a prepared statement, use EXPLAIN. If a generic plan is in use, it will contain parameter symbols `$n`, while a custom plan will have the current actual parameter values substituted into it.

For more information on query planning and the statistics collected by PostgreSQL for that purpose, see the ANALYZE documentation.

Although the main point of a prepared statement is to avoid repeated parse analysis and planning of the statement, PostgreSQL will force re-analysis and re-planning of the statement before using it whenever database objects used in the statement have undergone definitional (DDL) changes since the previous use of the prepared statement. Also, if the value of search_path changes from one use to the next, the statement will be re-parsed using the new `search_path`. (This latter behavior is new as of PostgreSQL 9.3.) These rules make use of a prepared statement semantically almost equivalent to re-submitting the same query text over and over, but with a performance benefit if no object definitions are changed, especially if the best plan remains the same across uses. An example of a case where the semantic equivalence is not perfect is that if the statement refers to a table by an unqualified name, and then a new table of the same name is created in a schema appearing earlier in the `search_path`, no automatic re-parse will occur since no object used in the statement changed. However, if some other change forces a re-parse, the new table will be referenced in subsequent uses.

You can see all prepared statements available in the session by querying the `pg_prepared_statements` system view.

Examples

Create a prepared statement for an `INSERT` statement, and then execute it:

```
PREPARE fooplan (int, text, bool, numeric) AS
    INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

Create a prepared statement for a `SELECT` statement, and then execute it:

```
PREPARE usrrptplan (int) AS
    SELECT * FROM users u, logs l WHERE u.usrid=$1 AND u.usrid=l.usrid
    AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

Note that the data type of the second parameter is not specified, so it is inferred from the context in which `$2` is used.

Compatibility

The SQL standard includes a `PREPARE` statement, but it is only for use in embedded SQL. This version of the `PREPARE` statement also uses a somewhat different syntax.

See Also

DEALLOCATE, EXECUTE

PREPARE TRANSACTION

Name

PREPARE TRANSACTION — prepare the current transaction for two-phase commit

Synopsis

```
PREPARE TRANSACTION transaction_id
```

Description

PREPARE TRANSACTION prepares the current transaction for two-phase commit. After this command, the transaction is no longer associated with the current session; instead, its state is fully stored on disk, and there is a very high probability that it can be committed successfully, even if a database crash occurs before the commit is requested.

Once prepared, a transaction can later be committed or rolled back with COMMIT PREPARED or ROLLBACK PREPARED, respectively. Those commands can be issued from any session, not only the one that executed the original transaction.

From the point of view of the issuing session, PREPARE TRANSACTION is not unlike a ROLLBACK command: after executing it, there is no active current transaction, and the effects of the prepared transaction are no longer visible. (The effects will become visible again if the transaction is committed.)

If the PREPARE TRANSACTION command fails for any reason, it becomes a ROLLBACK: the current transaction is canceled.

Parameters

transaction_id

An arbitrary identifier that later identifies this transaction for COMMIT PREPARED or ROLLBACK PREPARED. The identifier must be written as a string literal, and must be less than 200 bytes long. It must not be the same as the identifier used for any currently prepared transaction.

Notes

PREPARE TRANSACTION is not intended for use in applications or interactive sessions. Its purpose is to allow an external transaction manager to perform atomic global transactions across multiple databases or other transactional resources. Unless you're writing a transaction manager, you probably shouldn't be using PREPARE TRANSACTION.

This command must be used inside a transaction block. Use BEGIN to start one.

It is not currently allowed to PREPARE a transaction that has executed any operations involving temporary tables, created any cursors WITH HOLD, or executed LISTEN, UNLISTEN, or NOTIFY. Those features are too tightly tied to the current session to be useful in a transaction to be prepared.

If the transaction modified any run-time parameters with SET (without the LOCAL option), those effects persist after PREPARE TRANSACTION, and will not be affected by any later COMMIT PREPARED or ROLLBACK PREPARED. Thus, in this one respect PREPARE TRANSACTION acts more like COMMIT than ROLLBACK.

All currently available prepared transactions are listed in the pg_prepared_xacts system view.

Caution

It is unwise to leave transactions in the prepared state for a long time. This will interfere with the ability of VACUUM to reclaim storage, and in extreme cases could cause the database to shut down to prevent transaction ID wraparound (see Section 23.1.5). Keep in mind also that the transaction continues to hold whatever locks it held. The intended usage of the feature is that a prepared transaction will normally be committed or rolled back as soon as an external transaction manager has verified that other databases are also prepared to commit.

If you have not set up an external transaction manager to track prepared transactions and ensure they get closed out promptly, it is best to keep the prepared-transaction feature disabled by setting max_prepared_transactions to zero. This will prevent accidental creation of prepared transactions that might then be forgotten and eventually cause problems.

Examples

Prepare the current transaction for two-phase commit, using foobar as the transaction identifier:

```
PREPARE TRANSACTION 'foobar';
```

Compatibility

PREPARE TRANSACTION is a PostgreSQL extension. It is intended for use by external transaction management systems, some of which are covered by standards (such as X/Open XA), but the SQL side of those systems is not standardized.

See Also

COMMIT PREPARED, ROLLBACK PREPARED

REASSIGN OWNED

Name

REASSIGN OWNED — change the ownership of database objects owned by a database role

Synopsis

```
REASSIGN OWNED BY { old_role | CURRENT_USER | SESSION_USER } [, ...]
               TO { new_role | CURRENT_USER | SESSION_USER }
```

Description

REASSIGN OWNED instructs the system to change the ownership of database objects owned by any of the *old_roles* to *new_role*.

Parameters

old_role

The name of a role. The ownership of all the objects within the current database, and of all shared objects (databases, tablespaces), owned by this role will be reassigned to *new_role*.

new_role

The name of the role that will be made the new owner of the affected objects.

Notes

REASSIGN OWNED is often used to prepare for the removal of one or more roles. Because REASSIGN OWNED does not affect objects within other databases, it is usually necessary to execute this command in each database that contains objects owned by a role that is to be removed.

REASSIGN OWNED requires privileges on both the source role(s) and the target role.

The DROP OWNED command is an alternative that simply drops all the database objects owned by one or more roles.

The REASSIGN OWNED command does not affect any privileges granted to the *old_roles* on objects that are not owned by them. Likewise, it does not affect default privileges created with ALTER DEFAULT PRIVILEGES. Use DROP OWNED to revoke such privileges.

See Section 20.4 for more discussion.

Compatibility

The `REASSIGN OWNED` command is a PostgreSQL extension.

See Also

DROP OWNED, DROP ROLE, ALTER DATABASE

REFRESH MATERIALIZED VIEW

Name

REFRESH MATERIALIZED VIEW — replace the contents of a materialized view

Synopsis

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] name
    [ WITH [ NO ] DATA ]
```

Description

REFRESH MATERIALIZED VIEW completely replaces the contents of a materialized view. To execute this command you must be the owner of the materialized view. The old contents are discarded. If WITH DATA is specified (or defaults) the backing query is executed to provide the new data, and the materialized view is left in a scannable state. If WITH NO DATA is specified no new data is generated and the materialized view is left in an unscannable state.

CONCURRENTLY and WITH NO DATA may not be specified together.

Parameters

CONCURRENTLY

Refresh the materialized view without locking out concurrent selects on the materialized view. Without this option a refresh which affects a lot of rows will tend to use fewer resources and complete more quickly, but could block other connections which are trying to read from the materialized view. This option may be faster in cases where a small number of rows are affected.

This option is only allowed if there is at least one UNIQUE index on the materialized view which uses only column names and includes all rows; that is, it must not be an expression index or include a WHERE clause.

This option may not be used when the materialized view is not already populated.

Even with this option only one REFRESH at a time may run against any one materialized view.

name

The name (optionally schema-qualified) of the materialized view to refresh.

Notes

While the default index for future CLUSTER operations is retained, REFRESH MATERIALIZED VIEW does not order the generated rows based on this property. If you want the data to be ordered upon generation, you must use an ORDER BY clause in the backing query.

Examples

This command will replace the contents of the materialized view called `order_summary` using the query from the materialized view's definition, and leave it in a scannable state:

```
REFRESH MATERIALIZED VIEW order_summary;
```

This command will free storage associated with the materialized view `annual_statistics_basis` and leave it in an unscannable state:

```
REFRESH MATERIALIZED VIEW annual_statistics_basis WITH NO DATA;
```

Compatibility

`REFRESH MATERIALIZED VIEW` is a PostgreSQL extension.

See Also

CREATE MATERIALIZED VIEW, ALTER MATERIALIZED VIEW, DROP MATERIALIZED VIEW

REINDEX

Name

REINDEX — rebuild indexes

Synopsis

```
REINDEX [ ( VERBOSE ) ] { INDEX | TABLE | SCHEMA | DATABASE | SYSTEM } name
```

Description

REINDEX rebuilds an index using the data stored in the index's table, replacing the old copy of the index. There are several scenarios in which to use REINDEX:

- An index has become corrupted, and no longer contains valid data. Although in theory this should never happen, in practice indexes can become corrupted due to software bugs or hardware failures. REINDEX provides a recovery method.
- An index has become "bloated", that is it contains many empty or nearly-empty pages. This can occur with B-tree indexes in PostgreSQL under certain uncommon access patterns. REINDEX provides a way to reduce the space consumption of the index by writing a new version of the index without the dead pages. See Section 23.2 for more information.
- You have altered a storage parameter (such as fillfactor) for an index, and wish to ensure that the change has taken full effect.
- An index build with the CONCURRENTLY option failed, leaving an "invalid" index. Such indexes are useless but it can be convenient to use REINDEX to rebuild them. Note that REINDEX will not perform a concurrent build. To build the index without interfering with production you should drop the index and reissue the CREATE INDEX CONCURRENTLY command.

Parameters

INDEX

Recreate the specified index.

TABLE

Recreate all indexes of the specified table. If the table has a secondary "TOAST" table, that is reindexed as well.

`SCHEMA`

Recreate all indexes of the specified schema. If a table of this schema has a secondary "TOAST" table, that is reindexed as well. Indexes on shared system catalogs are also processed. This form of `REINDEX` cannot be executed inside a transaction block.

`DATABASE`

Recreate all indexes within the current database. Indexes on shared system catalogs are also processed. This form of `REINDEX` cannot be executed inside a transaction block.

`SYSTEM`

Recreate all indexes on system catalogs within the current database. Indexes on shared system catalogs are included. Indexes on user tables are not processed. This form of `REINDEX` cannot be executed inside a transaction block.

`name`

The name of the specific index, table, or database to be reindexed. Index and table names can be schema-qualified. Presently, `REINDEX DATABASE` and `REINDEX SYSTEM` can only reindex the current database, so their parameter must match the current database's name.

`VERBOSE`

Prints a progress report as each index is reindexed.

Notes

If you suspect corruption of an index on a user table, you can simply rebuild that index, or all indexes on the table, using `REINDEX INDEX` or `REINDEX TABLE`.

Things are more difficult if you need to recover from corruption of an index on a system table. In this case it's important for the system to not have used any of the suspect indexes itself. (Indeed, in this sort of scenario you might find that server processes are crashing immediately at start-up, due to reliance on the corrupted indexes.) To recover safely, the server must be started with the `-P` option, which prevents it from using indexes for system catalog lookups.

One way to do this is to shut down the server and start a single-user PostgreSQL server with the `-P` option included on its command line. Then, `REINDEX DATABASE`, `REINDEX SYSTEM`, `REINDEX TABLE`, or `REINDEX INDEX` can be issued, depending on how much you want to reconstruct. If in doubt, use `REINDEX SYSTEM` to select reconstruction of all system indexes in the database. Then quit the single-user server session and restart the regular server. See the postgres reference page for more information about how to interact with the single-user server interface.

Alternatively, a regular server session can be started with `-P` included in its command line options. The method for doing this varies across clients, but in all libpq-based clients, it is possible to set the `PGOPTIONS` environment variable to `-P` before starting the client. Note that while this method does not require locking out other clients, it might still be wise to prevent other users from connecting to the damaged database until repairs have been completed.

`REINDEX` is similar to a drop and recreate of the index in that the index contents are rebuilt from scratch. However, the locking considerations are rather different. `REINDEX` locks out writes but not reads of the index's parent table. It also takes an exclusive lock on the specific index being processed, which will block reads that attempt to use that index. In contrast, `DROP INDEX` momentarily takes an exclusive lock on the

parent table, blocking both writes and reads. The subsequent `CREATE INDEX` locks out writes but not reads; since the index is not there, no read will attempt to use it, meaning that there will be no blocking but reads might be forced into expensive sequential scans.

Reindexing a single index or table requires being the owner of that index or table. Reindexing a database requires being the owner of the database (note that the owner can therefore rebuild indexes of tables owned by other users). Of course, superusers can always reindex anything.

Examples

Rebuild a single index:

```
REINDEX INDEX my_index;
```

Rebuild all the indexes on the table `my_table`:

```
REINDEX TABLE my_table;
```

Rebuild all indexes in a particular database, without trusting the system indexes to be valid already:

```
$ export PGOPTIONS="-P"
$ psql broken_db
...
broken_db=> REINDEX DATABASE broken_db;
broken_db=> \q
```

Compatibility

There is no `REINDEX` command in the SQL standard.

RELEASE SAVEPOINT

Name

`RELEASE SAVEPOINT` — destroy a previously defined savepoint

Synopsis

```
RELEASE [ SAVEPOINT ] savepoint_name
```

Description

`RELEASE SAVEPOINT` destroys a savepoint previously defined in the current transaction.

Destroying a savepoint makes it unavailable as a rollback point, but it has no other user visible behavior. It does not undo the effects of commands executed after the savepoint was established. (To do that, see ROLLBACK TO SAVEPOINT.) Destroying a savepoint when it is no longer needed allows the system to reclaim some resources earlier than transaction end.

`RELEASE SAVEPOINT` also destroys all savepoints that were established after the named savepoint was established.

Parameters

`savepoint_name`

The name of the savepoint to destroy.

Notes

Specifying a savepoint name that was not previously defined is an error.

It is not possible to release a savepoint when the transaction is in an aborted state.

If multiple savepoints have the same name, only the one that was most recently defined is released.

Examples

To establish and later destroy a savepoint:

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
```

```
COMMIT;
```

The above transaction will insert both 3 and 4.

Compatibility

This command conforms to the SQL standard. The standard specifies that the key word `SAVEPOINT` is mandatory, but PostgreSQL allows it to be omitted.

See Also

BEGIN, COMMIT, ROLLBACK, ROLLBACK TO SAVEPOINT, SAVEPOINT

RESET

Name

RESET — restore the value of a run-time parameter to the default value

Synopsis

```
RESET configuration_parameter
RESET ALL
```

Description

RESET restores run-time parameters to their default values. RESET is an alternative spelling for

```
SET configuration_parameter TO DEFAULT
```

Refer to SET for details.

The default value is defined as the value that the parameter would have had, if no SET had ever been issued for it in the current session. The actual source of this value might be a compiled-in default, the configuration file, command-line options, or per-database or per-user default settings. This is subtly different from defining it as "the value that the parameter had at session start", because if the value came from the configuration file, it will be reset to whatever is specified by the configuration file now. See Chapter 18 for details.

The transactional behavior of RESET is the same as SET: its effects will be undone by transaction rollback.

Parameters

configuration_parameter

Name of a settable run-time parameter. Available parameters are documented in Chapter 18 and on the SET reference page.

ALL

Resets all settable run-time parameters to default values.

Examples

Set the timezone configuration variable to its default value:

```
RESET timezone;
```

Compatibility

`RESET` is a PostgreSQL extension.

See Also

SET, SHOW

REVOKE

Name

REVOKE — remove access privileges

Synopsis

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE sequence_name [, ...]
         | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE database_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN domain_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER fdw_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
```

```
    ON FOREIGN SERVER server_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { FUNCTION function_name ( [ [ argmode ] [ arg_name ] arg_type [, ...] ] ) [, ...]
         | ALL FUNCTIONS IN SCHEMA schema_name [, ...] }
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE lang_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE type_name [, ...]
    FROM role_specification [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ ADMIN OPTION FOR ]
    role_name [, ...] FROM role_specification [, ...]
    [ GRANTED BY role_specification ]
    [ CASCADE | RESTRICT ]
```

where *role_specification* can be:

```
    [ GROUP ] role_name
  | PUBLIC
  | CURRENT_USER
  | SESSION_USER
```

Description

The `REVOKE` command revokes previously granted privileges from one or more roles. The key word `PUBLIC` refers to the implicitly defined group of all roles.

See the description of the GRANT command for the meaning of the privilege types.

Note that any particular role will have the sum of privileges granted directly to it, privileges granted to any role it is presently a member of, and privileges granted to `PUBLIC`. Thus, for example, revoking `SELECT` privilege from `PUBLIC` does not necessarily mean that all roles have lost `SELECT` privilege on the object: those who have it granted directly or via another role will still have it. Similarly, revoking `SELECT` from a user might not prevent that user from using `SELECT` if `PUBLIC` or another membership role still has `SELECT` rights.

If `GRANT OPTION FOR` is specified, only the grant option for the privilege is revoked, not the privilege itself. Otherwise, both the privilege and the grant option are revoked.

If a user holds a privilege with grant option and has granted it to other users then the privileges held by those other users are called dependent privileges. If the privilege or the grant option held by the first user is being revoked and dependent privileges exist, those dependent privileges are also revoked if `CASCADE` is specified; if it is not, the revoke action will fail. This recursive revocation only affects privileges that were granted through a chain of users that is traceable to the user that is the subject of this `REVOKE` command. Thus, the affected users might effectively keep the privilege if it was also granted through other users.

When revoking privileges on a table, the corresponding column privileges (if any) are automatically revoked on each column of the table, as well. On the other hand, if a role has been granted privileges on a table, then revoking the same privileges from individual columns will have no effect.

When revoking membership in a role, `GRANT OPTION` is instead called `ADMIN OPTION`, but the behavior is similar. This form of the command also allows a `GRANTED BY` option, but that option is currently ignored (except for checking the existence of the named role). Note also that this form of the command does not allow the noise word `GROUP` in *`role_specification`*.

Notes

Use psql's `\dp` command to display the privileges granted on existing tables and columns. See GRANT for information about the format. For non-table objects there are other `\d` commands that can display their privileges.

A user can only revoke privileges that were granted directly by that user. If, for example, user A has granted a privilege with grant option to user B, and user B has in turn granted it to user C, then user A cannot revoke the privilege directly from C. Instead, user A could revoke the grant option from user B and use the `CASCADE` option so that the privilege is in turn revoked from user C. For another example, if both A and B have granted the same privilege to C, A can revoke his own grant but not B's grant, so C will still effectively have the privilege.

When a non-owner of an object attempts to `REVOKE` privileges on the object, the command will fail outright if the user has no privileges whatsoever on the object. As long as some privilege is available, the command will proceed, but it will revoke only those privileges for which the user has grant options. The `REVOKE ALL PRIVILEGES` forms will issue a warning message if no grant options are held, while the other forms will issue a warning if grant options for any of the privileges specifically named in the

command are not held. (In principle these statements apply to the object owner as well, but since the owner is always treated as holding all grant options, the cases can never occur.)

If a superuser chooses to issue a `GRANT` or `REVOKE` command, the command is performed as though it were issued by the owner of the affected object. Since all privileges ultimately come from the object owner (possibly indirectly via chains of grant options), it is possible for a superuser to revoke all privileges, but this might require use of `CASCADE` as stated above.

`REVOKE` can also be done by a role that is not the owner of the affected object, but is a member of the role that owns the object, or is a member of a role that holds privileges `WITH GRANT OPTION` on the object. In this case the command is performed as though it were issued by the containing role that actually owns the object or holds the privileges `WITH GRANT OPTION`. For example, if table `t1` is owned by role `g1`, of which role `u1` is a member, then `u1` can revoke privileges on `t1` that are recorded as being granted by `g1`. This would include grants made by `u1` as well as by other members of role `g1`.

If the role executing `REVOKE` holds privileges indirectly via more than one role membership path, it is unspecified which containing role will be used to perform the command. In such cases it is best practice to use `SET ROLE` to become the specific role you want to do the `REVOKE` as. Failure to do so might lead to revoking privileges other than the ones you intended, or not revoking anything at all.

Examples

Revoke insert privilege for the public on table `films`:

```
REVOKE INSERT ON films FROM PUBLIC;
```

Revoke all privileges from user `manuel` on view `kinds`:

```
REVOKE ALL PRIVILEGES ON kinds FROM manuel;
```

Note that this actually means “revoke all privileges that I granted”.

Revoke membership in role `admins` from user `joe`:

```
REVOKE admins FROM joe;
```

Compatibility

The compatibility notes of the GRANT command apply analogously to `REVOKE`. The keyword `RESTRICT` or `CASCADE` is required according to the standard, but PostgreSQL assumes `RESTRICT` by default.

See Also

GRANT

ROLLBACK

Name

ROLLBACK — abort the current transaction

Synopsis

```
ROLLBACK [ WORK | TRANSACTION ]
```

Description

ROLLBACK rolls back the current transaction and causes all the updates made by the transaction to be discarded.

Parameters

WORK
TRANSACTION

Optional key words. They have no effect.

Notes

Use COMMIT to successfully terminate a transaction.

Issuing ROLLBACK outside of a transaction block emits a warning and otherwise has no effect.

Examples

To abort all changes:

```
ROLLBACK;
```

Compatibility

The SQL standard only specifies the two forms ROLLBACK and ROLLBACK WORK. Otherwise, this command is fully conforming.

See Also

BEGIN, COMMIT, ROLLBACK TO SAVEPOINT

ROLLBACK PREPARED

Name

`ROLLBACK PREPARED` — cancel a transaction that was earlier prepared for two-phase commit

Synopsis

```
ROLLBACK PREPARED transaction_id
```

Description

`ROLLBACK PREPARED` rolls back a transaction that is in prepared state.

Parameters

`transaction_id`

The transaction identifier of the transaction that is to be rolled back.

Notes

To roll back a prepared transaction, you must be either the same user that executed the transaction originally, or a superuser. But you do not have to be in the same session that executed the transaction.

This command cannot be executed inside a transaction block. The prepared transaction is rolled back immediately.

All currently available prepared transactions are listed in the `pg_prepared_xacts` system view.

Examples

Roll back the transaction identified by the transaction identifier `foobar`:

```
ROLLBACK PREPARED 'foobar';
```

Compatibility

`ROLLBACK PREPARED` is a PostgreSQL extension. It is intended for use by external transaction management systems, some of which are covered by standards (such as X/Open XA), but the SQL side of those systems is not standardized.

See Also

PREPARE TRANSACTION, COMMIT PREPARED

ROLLBACK TO SAVEPOINT

Name

`ROLLBACK TO SAVEPOINT` — roll back to a savepoint

Synopsis

```
ROLLBACK [ WORK | TRANSACTION ] TO [ SAVEPOINT ] savepoint_name
```

Description

Roll back all commands that were executed after the savepoint was established. The savepoint remains valid and can be rolled back to again later, if needed.

`ROLLBACK TO SAVEPOINT` implicitly destroys all savepoints that were established after the named savepoint.

Parameters

savepoint_name

The savepoint to roll back to.

Notes

Use RELEASE SAVEPOINT to destroy a savepoint without discarding the effects of commands executed after it was established.

Specifying a savepoint name that has not been established is an error.

Cursors have somewhat non-transactional behavior with respect to savepoints. Any cursor that is opened inside a savepoint will be closed when the savepoint is rolled back. If a previously opened cursor is affected by a `FETCH` or `MOVE` command inside a savepoint that is later rolled back, the cursor remains at the position that `FETCH` left it pointing to (that is, the cursor motion caused by `FETCH` is not rolled back). Closing a cursor is not undone by rolling back, either. However, other side-effects caused by the cursor's query (such as side-effects of volatile functions called by the query) *are* rolled back if they occur during a savepoint that is later rolled back. A cursor whose execution causes a transaction to abort is put in a cannot-execute state, so while the transaction can be restored using `ROLLBACK TO SAVEPOINT`, the cursor can no longer be used.

Examples

To undo the effects of the commands executed after `my_savepoint` was established:

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

Cursor positions are not affected by savepoint rollback:

```
BEGIN;

DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;

SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        1

ROLLBACK TO SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        2

COMMIT;
```

Compatibility

The SQL standard specifies that the key word `SAVEPOINT` is mandatory, but PostgreSQL and Oracle allow it to be omitted. SQL allows only `WORK`, not `TRANSACTION`, as a noise word after `ROLLBACK`. Also, SQL has an optional clause `AND [ NO ] CHAIN` which is not currently supported by PostgreSQL. Otherwise, this command conforms to the SQL standard.

See Also

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, SAVEPOINT

SAVEPOINT

Name

SAVEPOINT — define a new savepoint within the current transaction

Synopsis

```
SAVEPOINT savepoint_name
```

Description

SAVEPOINT establishes a new savepoint within the current transaction.

A savepoint is a special mark inside a transaction that allows all commands that are executed after it was established to be rolled back, restoring the transaction state to what it was at the time of the savepoint.

Parameters

savepoint_name

The name to give to the new savepoint.

Notes

Use ROLLBACK TO SAVEPOINT to rollback to a savepoint. Use RELEASE SAVEPOINT to destroy a savepoint, keeping the effects of commands executed after it was established.

Savepoints can only be established when inside a transaction block. There can be multiple savepoints defined within a transaction.

Examples

To establish a savepoint and later undo the effects of all commands executed after it was established:

```
BEGIN;
    INSERT INTO table1 VALUES (1);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (2);
    ROLLBACK TO SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (3);
COMMIT;
```

The above transaction will insert the values 1 and 3, but not 2.

To establish and later destroy a savepoint:

```
BEGIN;
    INSERT INTO table1 VALUES (3);
    SAVEPOINT my_savepoint;
    INSERT INTO table1 VALUES (4);
    RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

The above transaction will insert both 3 and 4.

Compatibility

SQL requires a savepoint to be destroyed automatically when another savepoint with the same name is established. In PostgreSQL, the old savepoint is kept, though only the more recent one will be used when rolling back or releasing. (Releasing the newer savepoint with `RELEASE SAVEPOINT` will cause the older one to again become accessible to `ROLLBACK TO SAVEPOINT` and `RELEASE SAVEPOINT`.) Otherwise, `SAVEPOINT` is fully SQL conforming.

See Also

BEGIN, COMMIT, RELEASE SAVEPOINT, ROLLBACK, ROLLBACK TO SAVEPOINT

SECURITY LABEL

Name

SECURITY LABEL — define or change a security label applied to an object

Synopsis

```
SECURITY LABEL [ FOR provider ] ON
{
  TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE aggregate_name ( aggregate_signature ) |
  DATABASE object_name |
  DOMAIN object_name |
  EVENT TRIGGER object_name |
  FOREIGN TABLE object_name
  FUNCTION function_name ( [ [ argmode ] [ argname ] argtype [, ...] ] ) |
  LARGE OBJECT large_object_oid |
  MATERIALIZED VIEW object_name |
  [ PROCEDURAL ] LANGUAGE object_name |
  ROLE object_name |
  SCHEMA object_name |
  SEQUENCE object_name |
  TABLESPACE object_name |
  TYPE object_name |
  VIEW object_name
} IS 'label'

where aggregate_signature is:

* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ] argtype [ , ... ]
```

Description

SECURITY LABEL applies a security label to a database object. An arbitrary number of security labels, one per label provider, can be associated with a given database object. Label providers are loadable modules which register themselves by using the function register_label_provider.

> **Note:** register_label_provider is not an SQL function; it can only be called from C code loaded into the backend.

The label provider determines whether a given label is valid and whether it is permissible to assign that label to a given object. The meaning of a given label is likewise at the discretion of the label provider.

PostgreSQL places no restrictions on whether or how a label provider must interpret security labels; it merely provides a mechanism for storing them. In practice, this facility is intended to allow integration with label-based mandatory access control (MAC) systems such as SELinux. Such systems make all access control decisions based on object labels, rather than traditional discretionary access control (DAC) concepts such as users and groups.

Parameters

object_name
table_name.column_name
aggregate_name
function_name

The name of the object to be labeled. Names of tables, aggregates, domains, foreign tables, functions, sequences, types, and views can be schema-qualified.

provider

The name of the provider with which this label is to be associated. The named provider must be loaded and must consent to the proposed labeling operation. If exactly one provider is loaded, the provider name may be omitted for brevity.

argmode

The mode of a function or aggregate argument: `IN`, `OUT`, `INOUT`, or `VARIADIC`. If omitted, the default is `IN`. Note that `SECURITY LABEL` does not actually pay any attention to `OUT` arguments, since only the input arguments are needed to determine the function's identity. So it is sufficient to list the `IN`, `INOUT`, and `VARIADIC` arguments.

argname

The name of a function or aggregate argument. Note that `SECURITY LABEL` does not actually pay any attention to argument names, since only the argument data types are needed to determine the function's identity.

argtype

The data type of a function or aggregate argument.

large_object_oid

The OID of the large object.

`PROCEDURAL`

This is a noise word.

label

The new security label, written as a string literal; or `NULL` to drop the security label.

Examples

The following example shows how the security label of a table might be changed.

```
SECURITY LABEL FOR selinux ON TABLE mytable IS 'system_u:object_r:sepgsql_table_t:s0';
```

Compatibility

There is no `SECURITY LABEL` command in the SQL standard.

See Also

sepgsql, `src/test/modules/dummy_seclabel`

SELECT

Name

SELECT, TABLE, WITH — retrieve rows from a table or view

Synopsis

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ * | expression [ [ AS ] output_name ] [, ...] ]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY grouping_element [, ...] ]
    [ HAVING condition ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF table_name [, ...] ] [ NOWAIT

where from_item can be one of:

    [ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
                [ TABLESAMPLE sampling_method ( argument [, ...] ) [ REPEATABLE ( seed ) ] ]
    [ LATERAL ] ( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
    with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] )
                [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] ) [ AS ] alias ( column_definition [, ...]
    [ LATERAL ] function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
    [ LATERAL ] ROWS FROM( function_name ( [ argument [, ...] ] ) [ AS ( column_definition [, .
                [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    from_item [ NATURAL ] join_type from_item [ ON join_condition | USING ( join_column [, ...]

and grouping_element can be one of:

    ( )
    expression
    ( expression [, ...] )
    ROLLUP ( { expression | ( expression [, ...] ) } [, ...] )
    CUBE ( { expression | ( expression [, ...] ) } [, ...] )
    GROUPING SETS ( grouping_element [, ...] )

and with_query is:

    with_query_name [ ( column_name [, ...] ) ] AS ( select | values | insert | update | delete )

TABLE [ ONLY ] table_name [ * ]
```

Description

`SELECT` retrieves rows from zero or more tables. The general processing of `SELECT` is as follows:

1. All queries in the `WITH` list are computed. These effectively serve as temporary tables that can be referenced in the `FROM` list. A `WITH` query that is referenced more than once in `FROM` is computed only once. (See *WITH Clause* below.)
2. All elements in the `FROM` list are computed. (Each element in the `FROM` list is a real or virtual table.) If more than one element is specified in the `FROM` list, they are cross-joined together. (See *FROM Clause* below.)
3. If the `WHERE` clause is specified, all rows that do not satisfy the condition are eliminated from the output. (See *WHERE Clause* below.)
4. If the `GROUP BY` clause is specified, or if there are aggregate function calls, the output is combined into groups of rows that match on one or more values, and the results of aggregate functions are computed. If the `HAVING` clause is present, it eliminates groups that do not satisfy the given condition. (See *GROUP BY Clause* and *HAVING Clause* below.)
5. The actual output rows are computed using the `SELECT` output expressions for each selected row or row group. (See *SELECT List* below.)
6. `SELECT DISTINCT` eliminates duplicate rows from the result. `SELECT DISTINCT ON` eliminates rows that match on all the specified expressions. `SELECT ALL` (the default) will return all candidate rows, including duplicates. (See *DISTINCT Clause* below.)
7. Using the operators `UNION`, `INTERSECT`, and `EXCEPT`, the output of more than one `SELECT` statement can be combined to form a single result set. The `UNION` operator returns all rows that are in one or both of the result sets. The `INTERSECT` operator returns all rows that are strictly in both result sets. The `EXCEPT` operator returns the rows that are in the first result set but not in the second. In all three cases, duplicate rows are eliminated unless `ALL` is specified. The noise word `DISTINCT` can be added to explicitly specify eliminating duplicate rows. Notice that `DISTINCT` is the default behavior here, even though `ALL` is the default for `SELECT` itself. (See *UNION Clause*, *INTERSECT Clause*, and *EXCEPT Clause* below.)
8. If the `ORDER BY` clause is specified, the returned rows are sorted in the specified order. If `ORDER BY` is not given, the rows are returned in whatever order the system finds fastest to produce. (See *ORDER BY Clause* below.)
9. If the `LIMIT` (or `FETCH FIRST`) or `OFFSET` clause is specified, the `SELECT` statement only returns a subset of the result rows. (See *LIMIT Clause* below.)
10. If `FOR UPDATE`, `FOR NO KEY UPDATE`, `FOR SHARE` or `FOR KEY SHARE` is specified, the `SELECT` statement locks the selected rows against concurrent updates. (See *The Locking Clause* below.)

You must have `SELECT` privilege on each column used in a `SELECT` command. The use of `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` or `FOR KEY SHARE` requires `UPDATE` privilege as well (for at least one column of each table so selected).

Parameters

WITH Clause

The WITH clause allows you to specify one or more subqueries that can be referenced by name in the primary query. The subqueries effectively act as temporary tables or views for the duration of the primary query. Each subquery can be a SELECT, TABLE, VALUES, INSERT, UPDATE or DELETE statement. When writing a data-modifying statement (INSERT, UPDATE or DELETE) in WITH, it is usual to include a RETURNING clause. It is the output of RETURNING, *not* the underlying table that the statement modifies, that forms the temporary table that is read by the primary query. If RETURNING is omitted, the statement is still executed, but it produces no output so it cannot be referenced as a table by the primary query.

A name (without schema qualification) must be specified for each WITH query. Optionally, a list of column names can be specified; if this is omitted, the column names are inferred from the subquery.

If RECURSIVE is specified, it allows a SELECT subquery to reference itself by name. Such a subquery must have the form

```
non_recursive_term UNION [ ALL | DISTINCT ] recursive_term
```

where the recursive self-reference must appear on the right-hand side of the UNION. Only one recursive self-reference is permitted per query. Recursive data-modifying statements are not supported, but you can use the results of a recursive SELECT query in a data-modifying statement. See Section 7.8 for an example.

Another effect of RECURSIVE is that WITH queries need not be ordered: a query can reference another one that is later in the list. (However, circular references, or mutual recursion, are not implemented.) Without RECURSIVE, WITH queries can only reference sibling WITH queries that are earlier in the WITH list.

A key property of WITH queries is that they are evaluated only once per execution of the primary query, even if the primary query refers to them more than once. In particular, data-modifying statements are guaranteed to be executed once and only once, regardless of whether the primary query reads all or any of their output.

When there are multiple queries in the WITH clause, RECURSIVE should be written only once, immediately after WITH. It applies to all queries in the WITH clause, though it has no effect on queries that do not use recursion or forward references.

The primary query and the WITH queries are all (notionally) executed at the same time. This implies that the effects of a data-modifying statement in WITH cannot be seen from other parts of the query, other than by reading its RETURNING output. If two such data-modifying statements attempt to modify the same row, the results are unspecified.

See Section 7.8 for additional information.

FROM Clause

The FROM clause specifies one or more source tables for the SELECT. If multiple sources are specified, the result is the Cartesian product (cross join) of all the sources. But usually qualification conditions are added (via WHERE) to restrict the returned rows to a small subset of the Cartesian product.

The FROM clause can contain the following elements:

table_name

The name (optionally schema-qualified) of an existing table or view. If ONLY is specified before the table name, only that table is scanned. If ONLY is not specified, the table and all its descendant tables (if any) are scanned. Optionally, * can be specified after the table name to explicitly indicate that descendant tables are included.

alias

A substitute name for the FROM item containing the alias. An alias is used for brevity or to eliminate ambiguity for self-joins (where the same table is scanned multiple times). When an alias is provided, it completely hides the actual name of the table or function; for example given FROM foo AS f, the remainder of the SELECT must refer to this FROM item as f not foo. If an alias is written, a column alias list can also be written to provide substitute names for one or more columns of the table.

TABLESAMPLE *sampling_method* (*argument* [, ...]) [REPEATABLE (*seed*)]

A TABLESAMPLE clause after a *table_name* indicates that the specified *sampling_method* should be used to retrieve a subset of the rows in that table. This sampling precedes the application of any other filters such as WHERE clauses. The standard PostgreSQL distribution includes two sampling methods, BERNOULLI and SYSTEM, and other sampling methods can be installed in the database via extensions.

The BERNOULLI and SYSTEM sampling methods each accept a single *argument* which is the fraction of the table to sample, expressed as a percentage between 0 and 100. This argument can be any real-valued expression. (Other sampling methods might accept more or different arguments.) These two methods each return a randomly-chosen sample of the table that will contain approximately the specified percentage of the table's rows. The BERNOULLI method scans the whole table and selects or ignores individual rows independently with the specified probability. The SYSTEM method does block-level sampling with each block having the specified chance of being selected; all rows in each selected block are returned. The SYSTEM method is significantly faster than the BERNOULLI method when small sampling percentages are specified, but it may return a less-random sample of the table as a result of clustering effects.

The optional REPEATABLE clause specifies a *seed* number or expression to use for generating random numbers within the sampling method. The seed value can be any non-null floating-point value. Two queries that specify the same seed and *argument* values will select the same sample of the table, if the table has not been changed meanwhile. But different seed values will usually produce different samples. If REPEATABLE is not given then a new random seed is selected for each query. Note that some add-on sampling methods do not accept REPEATABLE, and will always produce new samples on each use.

select

A sub-SELECT can appear in the FROM clause. This acts as though its output were created as a temporary table for the duration of this single SELECT command. Note that the sub-SELECT must be surrounded by parentheses, and an alias *must* be provided for it. A VALUES command can also be used here.

with_query_name

A WITH query is referenced by writing its name, just as though the query's name were a table name. (In fact, the WITH query hides any real table of the same name for the purposes of the primary query. If necessary, you can refer to a real table of the same name by schema-qualifying the table's name.) An alias can be provided in the same way as for a table.

function_name

Function calls can appear in the FROM clause. (This is especially useful for functions that return result sets, but any function can be used.) This acts as though the function's output were created as a temporary table for the duration of this single SELECT command. When the optional WITH ORDINALITY clause is added to the function call, a new column is appended after all the function's output columns with numbering for each row.

An alias can be provided in the same way as for a table. If an alias is written, a column alias list can also be written to provide substitute names for one or more attributes of the function's composite return type, including the column added by ORDINALITY if present.

Multiple function calls can be combined into a single FROM-clause item by surrounding them with ROWS FROM(...). The output of such an item is the concatenation of the first row from each function, then the second row from each function, etc. If some of the functions produce fewer rows than others, NULLs are substituted for the missing data, so that the total number of rows returned is always the same as for the function that produced the most rows.

If the function has been defined as returning the record data type, then an alias or the key word AS must be present, followed by a column definition list in the form (*column_name data_type* [, ...]). The column definition list must match the actual number and types of columns returned by the function.

When using the ROWS FROM(...) syntax, if one of the functions requires a column definition list, it's preferred to put the column definition list after the function call inside ROWS FROM(...). A column definition list can be placed after the ROWS FROM(...) construct only if there's just a single function and no WITH ORDINALITY clause.

To use ORDINALITY together with a column definition list, you must use the ROWS FROM(...) syntax and put the column definition list inside ROWS FROM(...).

join_type

One of

- [INNER] JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN
- FULL [OUTER] JOIN
- CROSS JOIN

For the INNER and OUTER join types, a join condition must be specified, namely exactly one of NATURAL, ON *join_condition*, or USING (*join_column* [, ...]). See below for the meaning. For CROSS JOIN, none of these clauses can appear.

A JOIN clause combines two FROM items, which for convenience we will refer to as "tables", though in reality they can be any type of FROM item. Use parentheses if necessary to determine the order of nesting. In the absence of parentheses, JOINs nest left-to-right. In any case JOIN binds more tightly than the commas separating FROM-list items.

CROSS JOIN and INNER JOIN produce a simple Cartesian product, the same result as you get from listing the two tables at the top level of FROM, but restricted by the join condition (if any). CROSS JOIN is equivalent to INNER JOIN ON (TRUE), that is, no rows are removed by qualification. These

join types are just a notational convenience, since they do nothing you couldn't do with plain `FROM` and `WHERE`.

`LEFT OUTER JOIN` returns all rows in the qualified Cartesian product (i.e., all combined rows that pass its join condition), plus one copy of each row in the left-hand table for which there was no right-hand row that passed the join condition. This left-hand row is extended to the full width of the joined table by inserting null values for the right-hand columns. Note that only the `JOIN` clause's own condition is considered while deciding which rows have matches. Outer conditions are applied afterwards.

Conversely, `RIGHT OUTER JOIN` returns all the joined rows, plus one row for each unmatched right-hand row (extended with nulls on the left). This is just a notational convenience, since you could convert it to a `LEFT OUTER JOIN` by switching the left and right tables.

`FULL OUTER JOIN` returns all the joined rows, plus one row for each unmatched left-hand row (extended with nulls on the right), plus one row for each unmatched right-hand row (extended with nulls on the left).

`ON` *`join_condition`*

`join_condition` is an expression resulting in a value of type `boolean` (similar to a `WHERE` clause) that specifies which rows in a join are considered to match.

`USING ( join_column [, ...] )`

A clause of the form `USING ( a, b, ... )` is shorthand for `ON left_table.a = right_table.a AND left_table.b = right_table.b ...`. Also, `USING` implies that only one of each pair of equivalent columns will be included in the join output, not both.

`NATURAL`

`NATURAL` is shorthand for a `USING` list that mentions all columns in the two tables that have matching names. If there are no common column names, `NATURAL` is equivalent to `ON TRUE`.

`LATERAL`

The `LATERAL` key word can precede a sub-`SELECT` `FROM` item. This allows the sub-`SELECT` to refer to columns of `FROM` items that appear before it in the `FROM` list. (Without `LATERAL`, each sub-`SELECT` is evaluated independently and so cannot cross-reference any other `FROM` item.)

`LATERAL` can also precede a function-call `FROM` item, but in this case it is a noise word, because the function expression can refer to earlier `FROM` items in any case.

A `LATERAL` item can appear at top level in the `FROM` list, or within a `JOIN` tree. In the latter case it can also refer to any items that are on the left-hand side of a `JOIN` that it is on the right-hand side of.

When a `FROM` item contains `LATERAL` cross-references, evaluation proceeds as follows: for each row of the `FROM` item providing the cross-referenced column(s), or set of rows of multiple `FROM` items providing the columns, the `LATERAL` item is evaluated using that row or row set's values of the columns. The resulting row(s) are joined as usual with the rows they were computed from. This is repeated for each row or set of rows from the column source table(s).

The column source table(s) must be `INNER` or `LEFT` joined to the `LATERAL` item, else there would not be a well-defined set of rows from which to compute each set of rows for the `LATERAL` item. Thus, although a construct such as *`X`* `RIGHT JOIN LATERAL` *`Y`* is syntactically valid, it is not actually allowed for *`Y`* to reference *`X`*.

WHERE Clause

The optional WHERE clause has the general form

```
WHERE condition
```

where *condition* is any expression that evaluates to a result of type boolean. Any row that does not satisfy this condition will be eliminated from the output. A row satisfies the condition if it returns true when the actual row values are substituted for any variable references.

GROUP BY Clause

The optional GROUP BY clause has the general form

```
GROUP BY grouping_element [, ...]
```

GROUP BY will condense into a single row all selected rows that share the same values for the grouped expressions. An *expression* used inside a *grouping_element* can be an input column name, or the name or ordinal number of an output column (SELECT list item), or an arbitrary expression formed from input-column values. In case of ambiguity, a GROUP BY name will be interpreted as an input-column name rather than an output column name.

If any of GROUPING SETS, ROLLUP or CUBE are present as grouping elements, then the GROUP BY clause as a whole defines some number of independent *grouping sets*. The effect of this is equivalent to constructing a UNION ALL between subqueries with the individual grouping sets as their GROUP BY clauses. For further details on the handling of grouping sets see Section 7.2.4.

Aggregate functions, if any are used, are computed across all rows making up each group, producing a separate value for each group. (If there are aggregate functions but no GROUP BY clause, the query is treated as having a single group comprising all the selected rows.) The set of rows fed to each aggregate function can be further filtered by attaching a FILTER clause to the aggregate function call; see Section 4.2.7 for more information. When a FILTER clause is present, only those rows matching it are included in the input to that aggregate function.

When GROUP BY is present, or any aggregate functions are present, it is not valid for the SELECT list expressions to refer to ungrouped columns except within aggregate functions or when the ungrouped column is functionally dependent on the grouped columns, since there would otherwise be more than one possible value to return for an ungrouped column. A functional dependency exists if the grouped columns (or a subset thereof) are the primary key of the table containing the ungrouped column.

Keep in mind that all aggregate functions are evaluated before evaluating any “scalar” expressions in the HAVING clause or SELECT list. This means that, for example, a CASE expression cannot be used to skip evaluation of an aggregate function; see Section 4.2.14.

Currently, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE and FOR KEY SHARE cannot be specified with GROUP BY.

HAVING Clause

The optional HAVING clause has the general form

```
HAVING condition
```

where *condition* is the same as specified for the WHERE clause.

HAVING eliminates group rows that do not satisfy the condition. HAVING is different from WHERE: WHERE filters individual rows before the application of GROUP BY, while HAVING filters group rows created by GROUP BY. Each column referenced in *condition* must unambiguously reference a grouping column, unless the reference appears within an aggregate function or the ungrouped column is functionally dependent on the grouping columns.

The presence of HAVING turns a query into a grouped query even if there is no GROUP BY clause. This is the same as what happens when the query contains aggregate functions but no GROUP BY clause. All the selected rows are considered to form a single group, and the SELECT list and HAVING clause can only reference table columns from within aggregate functions. Such a query will emit a single row if the HAVING condition is true, zero rows if it is not true.

Currently, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE and FOR KEY SHARE cannot be specified with HAVING.

WINDOW Clause

The optional WINDOW clause has the general form

```
WINDOW window_name AS ( window_definition ) [, ...]
```

where *window_name* is a name that can be referenced from OVER clauses or subsequent window definitions, and *window_definition* is

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

If an *existing_window_name* is specified it must refer to an earlier entry in the WINDOW list; the new window copies its partitioning clause from that entry, as well as its ordering clause if any. In this case the new window cannot specify its own PARTITION BY clause, and it can specify ORDER BY only if the copied window does not have one. The new window always uses its own frame clause; the copied window must not specify a frame clause.

The elements of the PARTITION BY list are interpreted in much the same fashion as elements of a *GROUP BY Clause*, except that they are always simple expressions and never the name or number of an output column. Another difference is that these expressions can contain aggregate function calls, which are not allowed in a regular GROUP BY clause. They are allowed here because windowing occurs after grouping and aggregation.

Similarly, the elements of the `ORDER BY` list are interpreted in much the same fashion as elements of an *ORDER BY Clause*, except that the expressions are always taken as simple expressions and never the name or number of an output column.

The optional *`frame_clause`* defines the *window frame* for window functions that depend on the frame (not all do). The window frame is a set of related rows for each row of the query (called the *current row*). The *`frame_clause`* can be one of

```
{ RANGE | ROWS } frame_start
{ RANGE | ROWS } BETWEEN frame_start AND frame_end
```

where *`frame_start`* and *`frame_end`* can be one of

```
UNBOUNDED PRECEDING
value PRECEDING
CURRENT ROW
value FOLLOWING
UNBOUNDED FOLLOWING
```

If *`frame_end`* is omitted it defaults to `CURRENT ROW`. Restrictions are that *`frame_start`* cannot be `UNBOUNDED FOLLOWING`, *`frame_end`* cannot be `UNBOUNDED PRECEDING`, and the *`frame_end`* choice cannot appear earlier in the above list than the *`frame_start`* choice — for example `RANGE BETWEEN CURRENT ROW AND` *`value`* `PRECEDING` is not allowed.

The default framing option is `RANGE UNBOUNDED PRECEDING`, which is the same as `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`; it sets the frame to be all rows from the partition start up through the current row's last peer (a row that `ORDER BY` considers equivalent to the current row, or all rows if there is no `ORDER BY`). In general, `UNBOUNDED PRECEDING` means that the frame starts with the first row of the partition, and similarly `UNBOUNDED FOLLOWING` means that the frame ends with the last row of the partition (regardless of `RANGE` or `ROWS` mode). In `ROWS` mode, `CURRENT ROW` means that the frame starts or ends with the current row; but in `RANGE` mode it means that the frame starts or ends with the current row's first or last peer in the `ORDER BY` ordering. The *`value`* `PRECEDING` and *`value`* `FOLLOWING` cases are currently only allowed in `ROWS` mode. They indicate that the frame starts or ends with the row that many rows before or after the current row. *`value`* must be an integer expression not containing any variables, aggregate functions, or window functions. The value must not be null or negative; but it can be zero, which selects the current row itself.

Beware that the `ROWS` options can produce unpredictable results if the `ORDER BY` ordering does not order the rows uniquely. The `RANGE` options are designed to ensure that rows that are peers in the `ORDER BY` ordering are treated alike; all peer rows will be in the same frame.

The purpose of a `WINDOW` clause is to specify the behavior of *window functions* appearing in the query's *SELECT List* or *ORDER BY Clause*. These functions can reference the `WINDOW` clause entries by name in their `OVER` clauses. A `WINDOW` clause entry does not have to be referenced anywhere, however; if it is not used in the query it is simply ignored. It is possible to use window functions without any `WINDOW` clause at all, since a window function call can specify its window definition directly in its `OVER` clause. However, the `WINDOW` clause saves typing when the same window definition is needed for more than one window function.

Currently, `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` and `FOR KEY SHARE` cannot be specified with `WINDOW`.

Window functions are described in detail in Section 3.5, Section 4.2.8, and Section 7.2.5.

`SELECT` List

The `SELECT` list (between the key words `SELECT` and `FROM`) specifies expressions that form the output rows of the `SELECT` statement. The expressions can (and usually do) refer to columns computed in the `FROM` clause.

Just as in a table, every output column of a `SELECT` has a name. In a simple `SELECT` this name is just used to label the column for display, but when the `SELECT` is a sub-query of a larger query, the name is seen by the larger query as the column name of the virtual table produced by the sub-query. To specify the name to use for an output column, write `AS` *`output_name`* after the column's expression. (You can omit `AS`, but only if the desired output name does not match any PostgreSQL keyword (see Appendix C). For protection against possible future keyword additions, it is recommended that you always either write `AS` or double-quote the output name.) If you do not specify a column name, a name is chosen automatically by PostgreSQL. If the column's expression is a simple column reference then the chosen name is the same as that column's name. In more complex cases a function or type name may be used, or the system may fall back on a generated name such as `?column?`.

An output column's name can be used to refer to the column's value in `ORDER BY` and `GROUP BY` clauses, but not in the `WHERE` or `HAVING` clauses; there you must write out the expression instead.

Instead of an expression, `*` can be written in the output list as a shorthand for all the columns of the selected rows. Also, you can write *`table_name`*`.*` as a shorthand for the columns coming from just that table. In these cases it is not possible to specify new names with `AS`; the output column names will be the same as the table columns' names.

`DISTINCT` Clause

If `SELECT DISTINCT` is specified, all duplicate rows are removed from the result set (one row is kept from each group of duplicates). `SELECT ALL` specifies the opposite: all rows are kept; that is the default.

`SELECT DISTINCT ON (` *`expression`* `[, ...] )` keeps only the first row of each set of rows where the given expressions evaluate to equal. The `DISTINCT ON` expressions are interpreted using the same rules as for `ORDER BY` (see above). Note that the "first row" of each set is unpredictable unless `ORDER BY` is used to ensure that the desired row appears first. For example:

```
SELECT DISTINCT ON (location) location, time, report
    FROM weather_reports
    ORDER BY location, time DESC;
```

retrieves the most recent weather report for each location. But if we had not used `ORDER BY` to force descending order of time values for each location, we'd have gotten a report from an unpredictable time for each location.

The `DISTINCT ON` expression(s) must match the leftmost `ORDER BY` expression(s). The `ORDER BY` clause will normally contain additional expression(s) that determine the desired precedence of rows within each `DISTINCT ON` group.

Currently, `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` and `FOR KEY SHARE` cannot be specified with `DISTINCT`.

`UNION` Clause

The `UNION` clause has this general form:

```
select_statement UNION [ ALL | DISTINCT ] select_statement
```

`select_statement` is any `SELECT` statement without an `ORDER BY`, `LIMIT`, `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE`, or `FOR KEY SHARE` clause. (`ORDER BY` and `LIMIT` can be attached to a subexpression if it is enclosed in parentheses. Without parentheses, these clauses will be taken to apply to the result of the `UNION`, not to its right-hand input expression.)

The `UNION` operator computes the set union of the rows returned by the involved `SELECT` statements. A row is in the set union of two result sets if it appears in at least one of the result sets. The two `SELECT` statements that represent the direct operands of the `UNION` must produce the same number of columns, and corresponding columns must be of compatible data types.

The result of `UNION` does not contain any duplicate rows unless the `ALL` option is specified. `ALL` prevents elimination of duplicates. (Therefore, `UNION ALL` is usually significantly quicker than `UNION`; use `ALL` when you can.) `DISTINCT` can be written to explicitly specify the default behavior of eliminating duplicate rows.

Multiple `UNION` operators in the same `SELECT` statement are evaluated left to right, unless otherwise indicated by parentheses.

Currently, `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` and `FOR KEY SHARE` cannot be specified either for a `UNION` result or for any input of a `UNION`.

`INTERSECT` Clause

The `INTERSECT` clause has this general form:

```
select_statement INTERSECT [ ALL | DISTINCT ] select_statement
```

`select_statement` is any `SELECT` statement without an `ORDER BY`, `LIMIT`, `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE`, or `FOR KEY SHARE` clause.

The `INTERSECT` operator computes the set intersection of the rows returned by the involved `SELECT` statements. A row is in the intersection of two result sets if it appears in both result sets.

The result of `INTERSECT` does not contain any duplicate rows unless the `ALL` option is specified. With `ALL`, a row that has *`m`* duplicates in the left table and *`n`* duplicates in the right table will appear min(*`m`*,*`n`*) times in the result set. `DISTINCT` can be written to explicitly specify the default behavior of eliminating duplicate rows.

Multiple `INTERSECT` operators in the same `SELECT` statement are evaluated left to right, unless parentheses dictate otherwise. `INTERSECT` binds more tightly than `UNION`. That is, `A UNION B INTERSECT C` will be read as `A UNION (B INTERSECT C)`.

Currently, `FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE` and `FOR KEY SHARE` cannot be specified either for an `INTERSECT` result or for any input of an `INTERSECT`.

EXCEPT Clause

The EXCEPT clause has this general form:

```
select_statement EXCEPT [ ALL | DISTINCT ] select_statement
```

select_statement is any SELECT statement without an ORDER BY, LIMIT, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE, or FOR KEY SHARE clause.

The EXCEPT operator computes the set of rows that are in the result of the left SELECT statement but not in the result of the right one.

The result of EXCEPT does not contain any duplicate rows unless the ALL option is specified. With ALL, a row that has *m* duplicates in the left table and *n* duplicates in the right table will appear max(*m*-*n*,0) times in the result set. DISTINCT can be written to explicitly specify the default behavior of eliminating duplicate rows.

Multiple EXCEPT operators in the same SELECT statement are evaluated left to right, unless parentheses dictate otherwise. EXCEPT binds at the same level as UNION.

Currently, FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE and FOR KEY SHARE cannot be specified either for an EXCEPT result or for any input of an EXCEPT.

ORDER BY Clause

The optional ORDER BY clause has this general form:

```
ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...]
```

The ORDER BY clause causes the result rows to be sorted according to the specified expression(s). If two rows are equal according to the leftmost expression, they are compared according to the next expression and so on. If they are equal according to all specified expressions, they are returned in an implementation-dependent order.

Each *expression* can be the name or ordinal number of an output column (SELECT list item), or it can be an arbitrary expression formed from input-column values.

The ordinal number refers to the ordinal (left-to-right) position of the output column. This feature makes it possible to define an ordering on the basis of a column that does not have a unique name. This is never absolutely necessary because it is always possible to assign a name to an output column using the AS clause.

It is also possible to use arbitrary expressions in the ORDER BY clause, including columns that do not appear in the SELECT output list. Thus the following statement is valid:

```
SELECT name FROM distributors ORDER BY code;
```

A limitation of this feature is that an ORDER BY clause applying to the result of a UNION, INTERSECT, or EXCEPT clause can only specify an output column name or number, not an expression.

If an ORDER BY expression is a simple name that matches both an output column name and an input column name, ORDER BY will interpret it as the output column name. This is the opposite of the choice that GROUP BY will make in the same situation. This inconsistency is made to be compatible with the SQL standard.

Optionally one can add the key word `ASC` (ascending) or `DESC` (descending) after any expression in the `ORDER BY` clause. If not specified, `ASC` is assumed by default. Alternatively, a specific ordering operator name can be specified in the `USING` clause. An ordering operator must be a less-than or greater-than member of some B-tree operator family. `ASC` is usually equivalent to `USING <` and `DESC` is usually equivalent to `USING >`. (But the creator of a user-defined data type can define exactly what the default sort ordering is, and it might correspond to operators with other names.)

If `NULLS LAST` is specified, null values sort after all non-null values; if `NULLS FIRST` is specified, null values sort before all non-null values. If neither is specified, the default behavior is `NULLS LAST` when `ASC` is specified or implied, and `NULLS FIRST` when `DESC` is specified (thus, the default is to act as though nulls are larger than non-nulls). When `USING` is specified, the default nulls ordering depends on whether the operator is a less-than or greater-than operator.

Note that ordering options apply only to the expression they follow; for example `ORDER BY x, y DESC` does not mean the same thing as `ORDER BY x DESC, y DESC`.

Character-string data is sorted according to the collation that applies to the column being sorted. That can be overridden at need by including a `COLLATE` clause in the *`expression`*, for example `ORDER BY mycolumn COLLATE "en_US"`. For more information see Section 4.2.10 and Section 22.2.

`LIMIT` Clause

The `LIMIT` clause consists of two independent sub-clauses:

```
LIMIT { count | ALL }
OFFSET start
```

`count` specifies the maximum number of rows to return, while *`start`* specifies the number of rows to skip before starting to return rows. When both are specified, *`start`* rows are skipped before starting to count the *`count`* rows to be returned.

If the *`count`* expression evaluates to NULL, it is treated as `LIMIT ALL`, i.e., no limit. If *`start`* evaluates to NULL, it is treated the same as `OFFSET 0`.

SQL:2008 introduced a different syntax to achieve the same result, which PostgreSQL also supports. It is:

```
OFFSET start { ROW | ROWS }
FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY
```

In this syntax, the *`start`* or *`count`* value is required by the standard to be a literal constant, a parameter, or a variable name; as a PostgreSQL extension, other expressions are allowed, but will generally need to be enclosed in parentheses to avoid ambiguity. If *`count`* is omitted in a `FETCH` clause, it defaults to 1. `ROW` and `ROWS` as well as `FIRST` and `NEXT` are noise words that don't influence the effects of these clauses. According to the standard, the `OFFSET` clause must come before the `FETCH` clause if both are present; but PostgreSQL is laxer and allows either order.

When using `LIMIT`, it is a good idea to use an `ORDER BY` clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query's rows — you might be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? You don't know what ordering unless you specify `ORDER BY`.

The query planner takes `LIMIT` into account when generating a query plan, so you are very likely to get different plans (yielding different row orders) depending on what you use for `LIMIT` and `OFFSET`. Thus,

using different LIMIT/OFFSET values to select different subsets of a query result *will give inconsistent results* unless you enforce a predictable result ordering with ORDER BY. This is not a bug; it is an inherent consequence of the fact that SQL does not promise to deliver the results of a query in any particular order unless ORDER BY is used to constrain the order.

It is even possible for repeated executions of the same LIMIT query to return different subsets of the rows of a table, if there is not an ORDER BY to enforce selection of a deterministic subset. Again, this is not a bug; determinism of the results is simply not guaranteed in such a case.

The Locking Clause

FOR UPDATE, FOR NO KEY UPDATE, FOR SHARE and FOR KEY SHARE are *locking clauses*; they affect how SELECT locks rows as they are obtained from the table.

The locking clause has the general form

```
FOR lock_strength [ OF table_name [, ...] ] [ NOWAIT | SKIP LOCKED ]
```

where *lock_strength* can be one of

```
UPDATE
NO KEY UPDATE
SHARE
KEY SHARE
```

For more information on each row-level lock mode, refer to Section 13.3.2.

To prevent the operation from waiting for other transactions to commit, use either the NOWAIT or SKIP LOCKED option. With NOWAIT, the statement reports an error, rather than waiting, if a selected row cannot be locked immediately. With SKIP LOCKED, any selected rows that cannot be immediately locked are skipped. Skipping locked rows provides an inconsistent view of the data, so this is not suitable for general purpose work, but can be used to avoid lock contention with multiple consumers accessing a queue-like table. Note that NOWAIT and SKIP LOCKED apply only to the row-level lock(s) — the required ROW SHARE table-level lock is still taken in the ordinary way (see Chapter 13). You can use LOCK with the NOWAIT option first, if you need to acquire the table-level lock without waiting.

If specific tables are named in a locking clause, then only rows coming from those tables are locked; any other tables used in the SELECT are simply read as usual. A locking clause without a table list affects all tables used in the statement. If a locking clause is applied to a view or sub-query, it affects all tables used in the view or sub-query. However, these clauses do not apply to WITH queries referenced by the primary query. If you want row locking to occur within a WITH query, specify a locking clause within the WITH query.

Multiple locking clauses can be written if it is necessary to specify different locking behavior for different tables. If the same table is mentioned (or implicitly affected) by more than one locking clause, then it is processed as if it was only specified by the strongest one. Similarly, a table is processed as NOWAIT if that is specified in any of the clauses affecting it. Otherwise, it is processed as SKIP LOCKED if that is specified in any of the clauses affecting it.

The locking clauses cannot be used in contexts where returned rows cannot be clearly identified with individual table rows; for example they cannot be used with aggregation.

When a locking clause appears at the top level of a SELECT query, the rows that are locked are exactly those that are returned by the query; in the case of a join query, the rows locked are those that contribute to returned join rows. In addition, rows that satisfied the query conditions as of the query snapshot will be locked, although they will not be returned if they were updated after the snapshot and no longer satisfy the query conditions. If a LIMIT is used, locking stops once enough rows have been returned to satisfy the limit (but note that rows skipped over by OFFSET will get locked). Similarly, if a locking clause is used in a cursor's query, only rows actually fetched or stepped past by the cursor will be locked.

When a locking clause appears in a sub-SELECT, the rows locked are those returned to the outer query by the sub-query. This might involve fewer rows than inspection of the sub-query alone would suggest, since conditions from the outer query might be used to optimize execution of the sub-query. For example,

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss WHERE col1 = 5;
```

will lock only rows having `col1 = 5`, even though that condition is not textually within the sub-query.

Previous releases failed to preserve a lock which is upgraded by a later savepoint. For example, this code:

```
BEGIN;
SELECT * FROM mytable WHERE key = 1 FOR UPDATE;
SAVEPOINT s;
UPDATE mytable SET ... WHERE key = 1;
ROLLBACK TO s;
```

would fail to preserve the FOR UPDATE lock after the ROLLBACK TO. This has been fixed in release 9.3.

Caution

It is possible for a SELECT command running at the READ COMMITTED transaction isolation level and using ORDER BY and a locking clause to return rows out of order. This is because ORDER BY is applied first. The command sorts the result, but might then block trying to obtain a lock on one or more of the rows. Once the SELECT unblocks, some of the ordering column values might have been modified, leading to those rows appearing to be out of order (though they are in order in terms of the original column values). This can be worked around at need by placing the FOR UPDATE/SHARE clause in a sub-query, for example

```
SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss ORDER BY column1;
```

Note that this will result in locking all rows of `mytable`, whereas FOR UPDATE at the top level would lock only the actually returned rows. This can make for a significant performance difference, particularly if the ORDER BY is combined with LIMIT or other restrictions. So this technique is recommended only if concurrent updates of the ordering columns are expected and a strictly sorted result is required.

At the REPEATABLE READ or SERIALIZABLE transaction isolation level this would cause a serialization failure (with a SQLSTATE of '40001'), so there is no possibility of receiving rows out of order under these isolation levels.

`TABLE` Command

The command

```
TABLE name
```

is equivalent to

```
SELECT * FROM name
```

It can be used as a top-level command or as a space-saving syntax variant in parts of complex queries. Only the `WITH`, `UNION`, `INTERSECT`, `EXCEPT`, `ORDER BY`, `LIMIT`, `OFFSET`, `FETCH` and `FOR` locking clauses can be used with `TABLE`; the `WHERE` clause and any form of aggregation cannot be used.

Examples

To join the table `films` with the table `distributors`:

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
    FROM distributors d, films f
    WHERE f.did = d.did
```

```
       title       | did |     name     | date_prod  |   kind
-------------------+-----+--------------+------------+----------
 The Third Man     | 101 | British Lion | 1949-12-23 | Drama
 The African Queen | 101 | British Lion | 1951-08-11 | Romantic
 ...
```

To sum the column `len` of all films and group the results by `kind`:

```
SELECT kind, sum(len) AS total FROM films GROUP BY kind;
```

```
   kind   | total
----------+-------
 Action   | 07:34
 Comedy   | 02:58
 Drama    | 14:28
 Musical  | 06:42
 Romantic | 04:38
```

To sum the column `len` of all films, group the results by `kind` and show those group totals that are less than 5 hours:

```
SELECT kind, sum(len) AS total
    FROM films
    GROUP BY kind
    HAVING sum(len) < interval '5 hours';
```

```
   kind   | total
----------+-------
 Comedy   | 02:58
 Romantic | 04:38
```

The following two examples are identical ways of sorting the individual results according to the contents of the second column (`name`):

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;
```

```
 did |       name
-----+------------------
 109 | 20th Century Fox
 110 | Bavaria Atelier
 101 | British Lion
 107 | Columbia
 102 | Jean Luc Godard
 113 | Luso films
 104 | Mosfilm
 103 | Paramount
 106 | Toho
 105 | United Artists
 111 | Walt Disney
 112 | Warner Bros.
 108 | Westward
```

The next example shows how to obtain the union of the tables `distributors` and `actors`, restricting the results to those that begin with the letter W in each table. Only distinct rows are wanted, so the key word `ALL` is omitted.

```
distributors:               actors:
 did |     name              id |     name
-----+--------------        ----+----------------
 108 | Westward               1 | Woody Allen
 111 | Walt Disney            2 | Warren Beatty
 112 | Warner Bros.           3 | Walter Matthau
 ...                         ...
```

```
SELECT distributors.name
    FROM distributors
    WHERE distributors.name LIKE 'W%'
UNION
SELECT actors.name
    FROM actors
    WHERE actors.name LIKE 'W%';
```

```
      name
----------------
 Walt Disney
```

```
 Walter Matthau
 Warner Bros.
 Warren Beatty
 Westward
 Woody Allen
```

This example shows how to use a function in the `FROM` clause, both with and without a column definition list:

```
CREATE FUNCTION distributors(int) RETURNS SETOF distributors AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors(111);
 did |    name
-----+-------------
 111 | Walt Disney

CREATE FUNCTION distributors_2(int) RETURNS SETOF record AS $$
    SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors_2(111) AS (f1 int, f2 text);
 f1  |     f2
-----+-------------
 111 | Walt Disney
```

Here is an example of a function with an ordinality column added:

```
SELECT * FROM unnest(ARRAY['a','b','c','d','e','f']) WITH ORDINALITY;
 unnest | ordinality
--------+----------
 a      |        1
 b      |        2
 c      |        3
 d      |        4
 e      |        5
 f      |        6
(6 rows)
```

This example shows how to use a simple `WITH` clause:

```
WITH t AS (
    SELECT random() as x FROM generate_series(1, 3)
  )
SELECT * FROM t
UNION ALL
SELECT * FROM t
```

```
        x
--------------------
  0.534150459803641
  0.520092216785997
 0.0735620250925422
  0.534150459803641
  0.520092216785997
 0.0735620250925422
```

Notice that the `WITH` query was evaluated only once, so that we got two sets of the same three random values.

This example uses `WITH RECURSIVE` to find all subordinates (direct or indirect) of the employee Mary, and their level of indirectness, from a table that shows only direct subordinates:

```
WITH RECURSIVE employee_recursive(distance, employee_name, manager_name) AS (
    SELECT 1, employee_name, manager_name
    FROM employee
    WHERE manager_name = 'Mary'
  UNION ALL
    SELECT er.distance + 1, e.employee_name, e.manager_name
    FROM employee_recursive er, employee e
    WHERE er.employee_name = e.manager_name
  )
SELECT distance, employee_name FROM employee_recursive;
```

Notice the typical form of recursive queries: an initial condition, followed by `UNION`, followed by the recursive part of the query. Be sure that the recursive part of the query will eventually return no tuples, or else the query will loop indefinitely. (See Section 7.8 for more examples.)

This example uses `LATERAL` to apply a set-returning function `get_product_names()` for each row of the `manufacturers` table:

```
SELECT m.name AS mname, pname
FROM manufacturers m, LATERAL get_product_names(m.id) pname;
```

Manufacturers not currently having any products would not appear in the result, since it is an inner join. If we wished to include the names of such manufacturers in the result, we could do:

```
SELECT m.name AS mname, pname
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true;
```

Compatibility

Of course, the `SELECT` statement is compatible with the SQL standard. But there are some extensions and some missing features.

Omitted FROM Clauses

PostgreSQL allows one to omit the FROM clause. It has a straightforward use to compute the results of simple expressions:

```
SELECT 2+2;

 ?column?
----------
        4
```

Some other SQL databases cannot do this except by introducing a dummy one-row table from which to do the SELECT.

Note that if a FROM clause is not specified, the query cannot reference any database tables. For example, the following query is invalid:

```
SELECT distributors.* WHERE distributors.name = 'Westward';
```

PostgreSQL releases prior to 8.1 would accept queries of this form, and add an implicit entry to the query's FROM clause for each table referenced by the query. This is no longer allowed.

Empty SELECT Lists

The list of output expressions after SELECT can be empty, producing a zero-column result table. This is not valid syntax according to the SQL standard. PostgreSQL allows it to be consistent with allowing zero-column tables. However, an empty list is not allowed when DISTINCT is used.

Omitting the AS Key Word

In the SQL standard, the optional key word AS can be omitted before an output column name whenever the new column name is a valid column name (that is, not the same as any reserved keyword). PostgreSQL is slightly more restrictive: AS is required if the new column name matches any keyword at all, reserved or not. Recommended practice is to use AS or double-quote output column names, to prevent any possible conflict against future keyword additions.

In FROM items, both the standard and PostgreSQL allow AS to be omitted before an alias that is an unreserved keyword. But this is impractical for output column names, because of syntactic ambiguities.

ONLY and Inheritance

The SQL standard requires parentheses around the table name when writing ONLY, for example SELECT * FROM ONLY (tab1), ONLY (tab2) WHERE .... PostgreSQL considers these parentheses to be optional.

PostgreSQL allows a trailing * to be written to explicitly specify the non-ONLY behavior of including child tables. The standard does not allow this.

(These points apply equally to all SQL commands supporting the ONLY option.)

`TABLESAMPLE` Clause Restrictions

The `TABLESAMPLE` clause is currently accepted only on regular tables and materialized views. According to the SQL standard it should be possible to apply it to any `FROM` item.

Function Calls in `FROM`

PostgreSQL allows a function call to be written directly as a member of the `FROM` list. In the SQL standard it would be necessary to wrap such a function call in a sub-`SELECT`; that is, the syntax `FROM` *`func`*`(...)` *`alias`* is approximately equivalent to `FROM LATERAL (SELECT` *`func`*`(...))` *`alias`*. Note that `LATERAL` is considered to be implicit; this is because the standard requires `LATERAL` semantics for an `UNNEST()` item in `FROM`. PostgreSQL treats `UNNEST()` the same as other set-returning functions.

Namespace Available to `GROUP BY` and `ORDER BY`

In the SQL-92 standard, an `ORDER BY` clause can only use output column names or numbers, while a `GROUP BY` clause can only use expressions based on input column names. PostgreSQL extends each of these clauses to allow the other choice as well (but it uses the standard's interpretation if there is ambiguity). PostgreSQL also allows both clauses to specify arbitrary expressions. Note that names appearing in an expression will always be taken as input-column names, not as output-column names.

SQL:1999 and later use a slightly different definition which is not entirely upward compatible with SQL-92. In most cases, however, PostgreSQL will interpret an `ORDER BY` or `GROUP BY` expression the same way SQL:1999 does.

Functional Dependencies

PostgreSQL recognizes functional dependency (allowing columns to be omitted from `GROUP BY`) only when a table's primary key is included in the `GROUP BY` list. The SQL standard specifies additional conditions that should be recognized.

`WINDOW` Clause Restrictions

The SQL standard provides additional options for the window *`frame_clause`*. PostgreSQL currently supports only the options listed above.

`LIMIT` and `OFFSET`

The clauses `LIMIT` and `OFFSET` are PostgreSQL-specific syntax, also used by MySQL. The SQL:2008 standard has introduced the clauses `OFFSET ... FETCH {FIRST|NEXT} ...` for the same functionality, as shown above in *LIMIT Clause*. This syntax is also used by IBM DB2. (Applications written for Oracle frequently use a workaround involving the automatically generated `rownum` column, which is not available in PostgreSQL, to implement the effects of these clauses.)

`FOR NO KEY UPDATE`, `FOR UPDATE`, `FOR SHARE`, `FOR KEY SHARE`

Although `FOR UPDATE` appears in the SQL standard, the standard allows it only as an option of `DECLARE CURSOR`. PostgreSQL allows it in any `SELECT` query as well as in sub-`SELECT`s, but this is an extension. The `FOR NO KEY UPDATE`, `FOR SHARE` and `FOR KEY SHARE` variants, as well as the `NOWAIT` and `SKIP LOCKED` options, do not appear in the standard.

Data-Modifying Statements in `WITH`

PostgreSQL allows `INSERT`, `UPDATE`, and `DELETE` to be used as `WITH` queries. This is not found in the SQL standard.

Nonstandard Clauses

`DISTINCT ON ( ... )` is an extension of the SQL standard.

`ROWS FROM( ... )` is an extension of the SQL standard.

SELECT INTO

Name

SELECT INTO — define a new table from the results of a query

Synopsis

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ [ AS ] output_name ] [, ...]
    INTO [ TEMPORARY | TEMP | UNLOGGED ] [ TABLE ] new_table
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [...] ]
```

Description

SELECT INTO creates a new table and fills it with data computed by a query. The data is not returned to the client, as it is with a normal SELECT. The new table's columns have the names and data types associated with the output columns of the SELECT.

Parameters

TEMPORARY or TEMP

If specified, the table is created as a temporary table. Refer to CREATE TABLE for details.

UNLOGGED

If specified, the table is created as an unlogged table. Refer to CREATE TABLE for details.

new_table

The name (optionally schema-qualified) of the table to be created.

All other parameters are described in detail under SELECT.

Notes

CREATE TABLE AS is functionally similar to `SELECT INTO`. `CREATE TABLE AS` is the recommended syntax, since this form of `SELECT INTO` is not available in ECPG or PL/pgSQL, because they interpret the `INTO` clause differently. Furthermore, `CREATE TABLE AS` offers a superset of the functionality provided by `SELECT INTO`.

To add OIDs to the table created by `SELECT INTO`, enable the default_with_oids configuration variable. Alternatively, `CREATE TABLE AS` can be used with the `WITH OIDS` clause.

Examples

Create a new table `films_recent` consisting of only recent entries from the table `films`:

```
SELECT * INTO films_recent FROM films WHERE date_prod >= '2002-01-01';
```

Compatibility

The SQL standard uses `SELECT INTO` to represent selecting values into scalar variables of a host program, rather than creating a new table. This indeed is the usage found in ECPG (see Chapter 33) and PL/pgSQL (see Chapter 40). The PostgreSQL usage of `SELECT INTO` to represent table creation is historical. It is best to use `CREATE TABLE AS` for this purpose in new code.

See Also

CREATE TABLE AS

SET

Name

SET — change a run-time parameter

Synopsis

```
SET [ SESSION | LOCAL ] configuration_parameter { TO | = } { value | 'value' | DEFAULT }
SET [ SESSION | LOCAL ] TIME ZONE { timezone | LOCAL | DEFAULT }
```

Description

The SET command changes run-time configuration parameters. Many of the run-time parameters listed in Chapter 18 can be changed on-the-fly with SET. (But some require superuser privileges to change, and others cannot be changed after server or session start.) SET only affects the value used by the current session.

If SET (or equivalently SET SESSION) is issued within a transaction that is later aborted, the effects of the SET command disappear when the transaction is rolled back. Once the surrounding transaction is committed, the effects will persist until the end of the session, unless overridden by another SET.

The effects of SET LOCAL last only till the end of the current transaction, whether committed or not. A special case is SET followed by SET LOCAL within a single transaction: the SET LOCAL value will be seen until the end of the transaction, but afterwards (if the transaction is committed) the SET value will take effect.

The effects of SET or SET LOCAL are also canceled by rolling back to a savepoint that is earlier than the command.

If SET LOCAL is used within a function that has a SET option for the same variable (see CREATE FUNCTION), the effects of the SET LOCAL command disappear at function exit; that is, the value in effect when the function was called is restored anyway. This allows SET LOCAL to be used for dynamic or repeated changes of a parameter within a function, while still having the convenience of using the SET option to save and restore the caller's value. However, a regular SET command overrides any surrounding function's SET option; its effects will persist unless rolled back.

> **Note:** In PostgreSQL versions 8.0 through 8.2, the effects of a SET LOCAL would be canceled by releasing an earlier savepoint, or by successful exit from a PL/pgSQL exception block. This behavior has been changed because it was deemed unintuitive.

Parameters

SESSION

Specifies that the command takes effect for the current session. (This is the default if neither SESSION nor LOCAL appears.)

LOCAL

Specifies that the command takes effect for only the current transaction. After COMMIT or ROLLBACK, the session-level setting takes effect again. Issuing this outside of a transaction block emits a warning and otherwise has no effect.

configuration_parameter

Name of a settable run-time parameter. Available parameters are documented in Chapter 18 and below.

value

New value of parameter. Values can be specified as string constants, identifiers, numbers, or comma-separated lists of these, as appropriate for the particular parameter. DEFAULT can be written to specify resetting the parameter to its default value (that is, whatever value it would have had if no SET had been executed in the current session).

Besides the configuration parameters documented in Chapter 18, there are a few that can only be adjusted using the SET command or that have a special syntax:

SCHEMA

SET SCHEMA '*value*' is an alias for SET search_path TO *value*. Only one schema can be specified using this syntax.

NAMES

SET NAMES *value* is an alias for SET client_encoding TO *value*.

SEED

Sets the internal seed for the random number generator (the function random). Allowed values are floating-point numbers between -1 and 1, which are then multiplied by 2^{31}-1.

The seed can also be set by invoking the function setseed:

```
SELECT setseed(value);
```

TIME ZONE

SET TIME ZONE *value* is an alias for SET timezone TO *value*. The syntax SET TIME ZONE allows special syntax for the time zone specification. Here are examples of valid values:

'PST8PDT'

The time zone for Berkeley, California.

'Europe/Rome'

The time zone for Italy.

-7

The time zone 7 hours west from UTC (equivalent to PDT). Positive values are east from UTC.

`INTERVAL '-08:00' HOUR TO MINUTE`

The time zone 8 hours west from UTC (equivalent to PST).

`LOCAL`
`DEFAULT`

Set the time zone to your local time zone (that is, the server's default value of `timezone`).

Timezone settings given as numbers or intervals are internally translated to POSIX timezone syntax. For example, after `SET TIME ZONE -7`, `SHOW TIME ZONE` would report `<-07>+07`.

See Section 8.5.3 for more information about time zones.

Notes

The function `set_config` provides equivalent functionality; see Section 9.26. Also, it is possible to UPDATE the `pg_settings` system view to perform the equivalent of `SET`.

Examples

Set the schema search path:

```
SET search_path TO my_schema, public;
```

Set the style of date to traditional POSTGRES with "day before month" input convention:

```
SET datestyle TO postgres, dmy;
```

Set the time zone for Berkeley, California:

```
SET TIME ZONE 'PST8PDT';
```

Set the time zone for Italy:

```
SET TIME ZONE 'Europe/Rome';
```

Compatibility

`SET TIME ZONE` extends syntax defined in the SQL standard. The standard allows only numeric time zone offsets while PostgreSQL allows more flexible time-zone specifications. All other `SET` features are PostgreSQL extensions.

See Also

RESET, SHOW

SET CONSTRAINTS

Name

SET CONSTRAINTS — set constraint check timing for the current transaction

Synopsis

```
SET CONSTRAINTS { ALL | name [, ...] } { DEFERRED | IMMEDIATE }
```

Description

SET CONSTRAINTS sets the behavior of constraint checking within the current transaction. IMMEDIATE constraints are checked at the end of each statement. DEFERRED constraints are not checked until transaction commit. Each constraint has its own IMMEDIATE or DEFERRED mode.

Upon creation, a constraint is given one of three characteristics: DEFERRABLE INITIALLY DEFERRED, DEFERRABLE INITIALLY IMMEDIATE, or NOT DEFERRABLE. The third class is always IMMEDIATE and is not affected by the SET CONSTRAINTS command. The first two classes start every transaction in the indicated mode, but their behavior can be changed within a transaction by SET CONSTRAINTS.

SET CONSTRAINTS with a list of constraint names changes the mode of just those constraints (which must all be deferrable). Each constraint name can be schema-qualified. The current schema search path is used to find the first matching name if no schema name is specified. SET CONSTRAINTS ALL changes the mode of all deferrable constraints.

When SET CONSTRAINTS changes the mode of a constraint from DEFERRED to IMMEDIATE, the new mode takes effect retroactively: any outstanding data modifications that would have been checked at the end of the transaction are instead checked during the execution of the SET CONSTRAINTS command. If any such constraint is violated, the SET CONSTRAINTS fails (and does not change the constraint mode). Thus, SET CONSTRAINTS can be used to force checking of constraints to occur at a specific point in a transaction.

Currently, only UNIQUE, PRIMARY KEY, REFERENCES (foreign key), and EXCLUDE constraints are affected by this setting. NOT NULL and CHECK constraints are always checked immediately when a row is inserted or modified (*not* at the end of the statement). Uniqueness and exclusion constraints that have not been declared DEFERRABLE are also checked immediately.

The firing of triggers that are declared as “constraint triggers” is also controlled by this setting — they fire at the same time that the associated constraint should be checked.

Notes

Because PostgreSQL does not require constraint names to be unique within a schema (but only pertable), it is possible that there is more than one match for a specified constraint name. In this case SET CONSTRAINTS will act on all matches. For a non-schema-qualified name, once a match or matches have been found in some schema in the search path, schemas appearing later in the path are not searched.

This command only alters the behavior of constraints within the current transaction. Issuing this outside of a transaction block emits a warning and otherwise has no effect.

Compatibility

This command complies with the behavior defined in the SQL standard, except for the limitation that, in PostgreSQL, it does not apply to `NOT NULL` and `CHECK` constraints. Also, PostgreSQL checks non-deferrable uniqueness constraints immediately, not at end of statement as the standard would suggest.

SET ROLE

Name

SET ROLE — set the current user identifier of the current session

Synopsis

```
SET [ SESSION | LOCAL ] ROLE role_name
SET [ SESSION | LOCAL ] ROLE NONE
RESET ROLE
```

Description

This command sets the current user identifier of the current SQL session to be *role_name*. The role name can be written as either an identifier or a string literal. After SET ROLE, permissions checking for SQL commands is carried out as though the named role were the one that had logged in originally.

The specified *role_name* must be a role that the current session user is a member of. (If the session user is a superuser, any role can be selected.)

The SESSION and LOCAL modifiers act the same as for the regular SET command.

The NONE and RESET forms reset the current user identifier to be the current session user identifier. These forms can be executed by any user.

Notes

Using this command, it is possible to either add privileges or restrict one's privileges. If the session user role has the INHERIT attribute, then it automatically has all the privileges of every role that it could SET ROLE to; in this case SET ROLE effectively drops all the privileges assigned directly to the session user and to the other roles it is a member of, leaving only the privileges available to the named role. On the other hand, if the session user role has the NOINHERIT attribute, SET ROLE drops the privileges assigned directly to the session user and instead acquires the privileges available to the named role.

In particular, when a superuser chooses to SET ROLE to a non-superuser role, she loses her superuser privileges.

SET ROLE has effects comparable to SET SESSION AUTHORIZATION, but the privilege checks involved are quite different. Also, SET SESSION AUTHORIZATION determines which roles are allowable for later SET ROLE commands, whereas changing roles with SET ROLE does not change the set of roles allowed to a later SET ROLE.

SET ROLE does not process session variables as specified by the role's ALTER ROLE settings; this only happens during login.

SET ROLE cannot be used within a SECURITY DEFINER function.

Examples

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET ROLE 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | paul
```

Compatibility

PostgreSQL allows identifier syntax (`"rolename"`), while the SQL standard requires the role name to be written as a string literal. SQL does not allow this command during a transaction; PostgreSQL does not make this restriction because there is no reason to. The `SESSION` and `LOCAL` modifiers are a PostgreSQL extension, as is the `RESET` syntax.

See Also

SET SESSION AUTHORIZATION

SET SESSION AUTHORIZATION

Name

SET SESSION AUTHORIZATION — set the session user identifier and the current user identifier of the current session

Synopsis

```
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION user_name
SET [ SESSION | LOCAL ] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION
```

Description

This command sets the session user identifier and the current user identifier of the current SQL session to be *user_name*. The user name can be written as either an identifier or a string literal. Using this command, it is possible, for example, to temporarily become an unprivileged user and later switch back to being a superuser.

The session user identifier is initially set to be the (possibly authenticated) user name provided by the client. The current user identifier is normally equal to the session user identifier, but might change temporarily in the context of SECURITY DEFINER functions and similar mechanisms; it can also be changed by SET ROLE. The current user identifier is relevant for permission checking.

The session user identifier can be changed only if the initial session user (the *authenticated user*) had the superuser privilege. Otherwise, the command is accepted only if it specifies the authenticated user name.

The SESSION and LOCAL modifiers act the same as for the regular SET command.

The DEFAULT and RESET forms reset the session and current user identifiers to be the originally authenticated user name. These forms can be executed by any user.

Notes

SET SESSION AUTHORIZATION cannot be used within a SECURITY DEFINER function.

Examples

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';
```

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 paul         | paul
```

Compatibility

The SQL standard allows some other expressions to appear in place of the literal *user_name*, but these options are not important in practice. PostgreSQL allows identifier syntax ("username"), which SQL does not. SQL does not allow this command during a transaction; PostgreSQL does not make this restriction because there is no reason to. The SESSION and LOCAL modifiers are a PostgreSQL extension, as is the RESET syntax.

The privileges necessary to execute this command are left implementation-defined by the standard.

See Also

SET ROLE

SET TRANSACTION

Name

SET TRANSACTION — set the characteristics of the current transaction

Synopsis

```
SET TRANSACTION transaction_mode [, ...]
SET TRANSACTION SNAPSHOT snapshot_id
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]

where transaction_mode is one of:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

Description

The SET TRANSACTION command sets the characteristics of the current transaction. It has no effect on any subsequent transactions. SET SESSION CHARACTERISTICS sets the default transaction characteristics for subsequent transactions of a session. These defaults can be overridden by SET TRANSACTION for an individual transaction.

The available transaction characteristics are the transaction isolation level, the transaction access mode (read/write or read-only), and the deferrable mode. In addition, a snapshot can be selected, though only for the current transaction, not as a session default.

The isolation level of a transaction determines what data the transaction can see when other transactions are running concurrently:

READ COMMITTED

A statement can only see rows committed before it began. This is the default.

REPEATABLE READ

All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction.

SERIALIZABLE

All statements of the current transaction can only see rows committed before the first query or data-modification statement was executed in this transaction. If a pattern of reads and writes among concurrent serializable transactions would create a situation which could not have occurred for any serial (one-at-a-time) execution of those transactions, one of them will be rolled back with a serialization_failure error.

The SQL standard defines one additional level, READ UNCOMMITTED. In PostgreSQL READ UNCOMMITTED is treated as READ COMMITTED.

The transaction isolation level cannot be changed after the first query or data-modification statement (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH`, or `COPY`) of a transaction has been executed. See Chapter 13 for more information about transaction isolation and concurrency control.

The transaction access mode determines whether the transaction is read/write or read-only. Read/write is the default. When a transaction is read-only, the following SQL commands are disallowed: `INSERT`, `UPDATE`, `DELETE`, and `COPY FROM` if the table they would write to is not a temporary table; all `CREATE`, `ALTER`, and `DROP` commands; `COMMENT`, `GRANT`, `REVOKE`, `TRUNCATE`; and `EXPLAIN ANALYZE` and `EXECUTE` if the command they would execute is among those listed. This is a high-level notion of read-only that does not prevent all writes to disk.

The `DEFERRABLE` transaction property has no effect unless the transaction is also `SERIALIZABLE` and `READ ONLY`. When all three of these properties are selected for a transaction, the transaction may block when first acquiring its snapshot, after which it is able to run without the normal overhead of a `SERIALIZABLE` transaction and without any risk of contributing to or being canceled by a serialization failure. This mode is well suited for long-running reports or backups.

The `SET TRANSACTION SNAPSHOT` command allows a new transaction to run with the same *snapshot* as an existing transaction. The pre-existing transaction must have exported its snapshot with the `pg_export_snapshot` function (see Section 9.26.5). That function returns a snapshot identifier, which must be given to `SET TRANSACTION SNAPSHOT` to specify which snapshot is to be imported. The identifier must be written as a string literal in this command, for example `'000003A1-1'`. `SET TRANSACTION SNAPSHOT` can only be executed at the start of a transaction, before the first query or data-modification statement (`SELECT`, `INSERT`, `DELETE`, `UPDATE`, `FETCH`, or `COPY`) of the transaction. Furthermore, the transaction must already be set to `SERIALIZABLE` or `REPEATABLE READ` isolation level (otherwise, the snapshot would be discarded immediately, since `READ COMMITTED` mode takes a new snapshot for each command). If the importing transaction uses `SERIALIZABLE` isolation level, then the transaction that exported the snapshot must also use that isolation level. Also, a non-read-only serializable transaction cannot import a snapshot from a read-only transaction.

Notes

If `SET TRANSACTION` is executed without a prior `START TRANSACTION` or `BEGIN`, it emits a warning and otherwise has no effect.

It is possible to dispense with `SET TRANSACTION` by instead specifying the desired *`transaction_modes`* in `BEGIN` or `START TRANSACTION`. But that option is not available for `SET TRANSACTION SNAPSHOT`.

The session default transaction modes can also be set by setting the configuration parameters default_transaction_isolation, default_transaction_read_only, and default_transaction_deferrable. (In fact `SET SESSION CHARACTERISTICS` is just a verbose equivalent for setting these variables with `SET`.) This means the defaults can be set in the configuration file, via `ALTER DATABASE`, etc. Consult Chapter 18 for more information.

Examples

To begin a new transaction with the same snapshot as an already existing transaction, first export the

snapshot from the existing transaction. That will return the snapshot identifier, for example:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT pg_export_snapshot();
 pg_export_snapshot
--------------------
 000003A1-1
(1 row)
```

Then give the snapshot identifier in a `SET TRANSACTION SNAPSHOT` command at the beginning of the newly opened transaction:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION SNAPSHOT '000003A1-1';
```

Compatibility

These commands are defined in the SQL standard, except for the `DEFERRABLE` transaction mode and the `SET TRANSACTION SNAPSHOT` form, which are PostgreSQL extensions.

`SERIALIZABLE` is the default transaction isolation level in the standard. In PostgreSQL the default is ordinarily `READ COMMITTED`, but you can change it as mentioned above.

In the SQL standard, there is one other transaction characteristic that can be set with these commands: the size of the diagnostics area. This concept is specific to embedded SQL, and therefore is not implemented in the PostgreSQL server.

The SQL standard requires commas between successive *`transaction_modes`*, but for historical reasons PostgreSQL allows the commas to be omitted.

SHOW

Name

SHOW — show the value of a run-time parameter

Synopsis

```
SHOW name
SHOW ALL
```

Description

SHOW will display the current setting of run-time parameters. These variables can be set using the SET statement, by editing the postgresql.conf configuration file, through the PGOPTIONS environmental variable (when using libpq or a libpq-based application), or through command-line flags when starting the postgres server. See Chapter 18 for details.

Parameters

name

The name of a run-time parameter. Available parameters are documented in Chapter 18 and on the SET reference page. In addition, there are a few parameters that can be shown but not set:

SERVER_VERSION

Shows the server's version number.

SERVER_ENCODING

Shows the server-side character set encoding. At present, this parameter can be shown but not set, because the encoding is determined at database creation time.

LC_COLLATE

Shows the database's locale setting for collation (text ordering). At present, this parameter can be shown but not set, because the setting is determined at database creation time.

LC_CTYPE

Shows the database's locale setting for character classification. At present, this parameter can be shown but not set, because the setting is determined at database creation time.

IS_SUPERUSER

True if the current role has superuser privileges.

ALL

Show the values of all configuration parameters, with descriptions.

Notes

The function current_setting produces equivalent output; see Section 9.26. Also, the pg_settings system view produces the same information.

Examples

Show the current setting of the parameter DateStyle:

```
SHOW DateStyle;
 DateStyle
-----------
 ISO, MDY
(1 row)
```

Show the current setting of the parameter geqo:

```
SHOW geqo;
 geqo
------
 on
(1 row)
```

Show all settings:

```
SHOW ALL;
             name          | setting |                 description
---------------------------+---------+---------------------------------------------
 allow_system_table_mods | off     | Allows modifications of the structure of ...
    .
    .
    .
 xmloption               | content | Sets whether XML data in implicit parsing ...
 zero_damaged_pages      | off     | Continues processing past damaged page headers.
(196 rows)
```

Compatibility

The SHOW command is a PostgreSQL extension.

See Also

SET, RESET

START TRANSACTION

Name

START TRANSACTION — start a transaction block

Synopsis

```
START TRANSACTION [ transaction_mode [, ...] ]

where transaction_mode is one of:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

Description

This command begins a new transaction block. If the isolation level, read/write mode, or deferrable mode is specified, the new transaction has those characteristics, as if SET TRANSACTION was executed. This is the same as the BEGIN command.

Parameters

Refer to SET TRANSACTION for information on the meaning of the parameters to this statement.

Compatibility

In the standard, it is not necessary to issue START TRANSACTION to start a transaction block: any SQL command implicitly begins a block. PostgreSQL's behavior can be seen as implicitly issuing a COMMIT after each command that does not follow START TRANSACTION (or BEGIN), and it is therefore often called "autocommit". Other relational database systems might offer an autocommit feature as a convenience.

The DEFERRABLE *transaction_mode* is a PostgreSQL language extension.

The SQL standard requires commas between successive *transaction_modes*, but for historical reasons PostgreSQL allows the commas to be omitted.

See also the compatibility section of SET TRANSACTION.

See Also

BEGIN, COMMIT, ROLLBACK, SAVEPOINT, SET TRANSACTION

TRUNCATE

Name

TRUNCATE — empty a table or set of tables

Synopsis

```
TRUNCATE [ TABLE ] [ ONLY ] name [ * ] [, ... ]
    [ RESTART IDENTITY | CONTINUE IDENTITY ] [ CASCADE | RESTRICT ]
```

Description

TRUNCATE quickly removes all rows from a set of tables. It has the same effect as an unqualified DELETE on each table, but since it does not actually scan the tables it is faster. Furthermore, it reclaims disk space immediately, rather than requiring a subsequent VACUUM operation. This is most useful on large tables.

Parameters

name

The name (optionally schema-qualified) of a table to truncate. If ONLY is specified before the table name, only that table is truncated. If ONLY is not specified, the table and all its descendant tables (if any) are truncated. Optionally, * can be specified after the table name to explicitly indicate that descendant tables are included.

RESTART IDENTITY

Automatically restart sequences owned by columns of the truncated table(s).

CONTINUE IDENTITY

Do not change the values of sequences. This is the default.

CASCADE

Automatically truncate all tables that have foreign-key references to any of the named tables, or to any tables added to the group due to CASCADE.

RESTRICT

Refuse to truncate if any of the tables have foreign-key references from tables that are not listed in the command. This is the default.

Notes

You must have the TRUNCATE privilege on a table to truncate it.

`TRUNCATE` acquires an `ACCESS EXCLUSIVE` lock on each table it operates on, which blocks all other concurrent operations on the table. When `RESTART IDENTITY` is specified, any sequences that are to be restarted are likewise locked exclusively. If concurrent access to a table is required, then the `DELETE` command should be used instead.

`TRUNCATE` cannot be used on a table that has foreign-key references from other tables, unless all such tables are also truncated in the same command. Checking validity in such cases would require table scans, and the whole point is not to do one. The `CASCADE` option can be used to automatically include all dependent tables — but be very careful when using this option, or else you might lose data you did not intend to!

`TRUNCATE` will not fire any `ON DELETE` triggers that might exist for the tables. But it will fire `ON TRUNCATE` triggers. If `ON TRUNCATE` triggers are defined for any of the tables, then all `BEFORE TRUNCATE` triggers are fired before any truncation happens, and all `AFTER TRUNCATE` triggers are fired after the last truncation is performed and any sequences are reset. The triggers will fire in the order that the tables are to be processed (first those listed in the command, and then any that were added due to cascading).

`TRUNCATE` is not MVCC-safe. After truncation, the table will appear empty to concurrent transactions, if they are using a snapshot taken before the truncation occurred. See Section 13.5 for more details.

`TRUNCATE` is transaction-safe with respect to the data in the tables: the truncation will be safely rolled back if the surrounding transaction does not commit.

When `RESTART IDENTITY` is specified, the implied `ALTER SEQUENCE RESTART` operations are also done transactionally; that is, they will be rolled back if the surrounding transaction does not commit. This is unlike the normal behavior of `ALTER SEQUENCE RESTART`. Be aware that if any additional sequence operations are done on the restarted sequences before the transaction rolls back, the effects of these operations on the sequences will be rolled back, but not their effects on `currval()`; that is, after the transaction `currval()` will continue to reflect the last sequence value obtained inside the failed transaction, even though the sequence itself may no longer be consistent with that. This is similar to the usual behavior of `currval()` after a failed transaction.

`TRUNCATE` is not currently supported for foreign tables. This implies that if a specified table has any descendant tables that are foreign, the command will fail.

Examples

Truncate the tables `bigtable` and `fattable`:

```
TRUNCATE bigtable, fattable;
```

The same, and also reset any associated sequence generators:

```
TRUNCATE bigtable, fattable RESTART IDENTITY;
```

Truncate the table `othertable`, and cascade to any tables that reference `othertable` via foreign-key constraints:

```
TRUNCATE othertable CASCADE;
```

Compatibility

The SQL:2008 standard includes a `TRUNCATE` command with the syntax `TRUNCATE TABLE` *`tablename`*. The clauses `CONTINUE IDENTITY`/`RESTART IDENTITY` also appear in that standard, but have slightly different though related meanings. Some of the concurrency behavior of this command is left implementation-defined by the standard, so the above notes should be considered and compared with other implementations if necessary.

UNLISTEN

Name

UNLISTEN — stop listening for a notification

Synopsis

```
UNLISTEN { channel | * }
```

Description

UNLISTEN is used to remove an existing registration for NOTIFY events. UNLISTEN cancels any existing registration of the current PostgreSQL session as a listener on the notification channel named *channel*. The special wildcard * cancels all listener registrations for the current session.

NOTIFY contains a more extensive discussion of the use of LISTEN and NOTIFY.

Parameters

channel

Name of a notification channel (any identifier).

*

All current listen registrations for this session are cleared.

Notes

You can unlisten something you were not listening for; no warning or error will appear.

At the end of each session, UNLISTEN * is automatically executed.

A transaction that has executed UNLISTEN cannot be prepared for two-phase commit.

Examples

To make a registration:

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

Once UNLISTEN has been executed, further NOTIFY messages will be ignored:

```
UNLISTEN virtual;
NOTIFY virtual;
-- no NOTIFY event is received
```

Compatibility

There is no `UNLISTEN` command in the SQL standard.

See Also

LISTEN, NOTIFY

UPDATE

Name

UPDATE — update rows of a table

Synopsis

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    SET { column_name = { expression | DEFAULT } |
          ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] ) |
          ( column_name [, ...] ) = ( sub-SELECT )
        } [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

Description

UPDATE changes the values of the specified columns in all rows that satisfy the condition. Only the columns to be modified need be mentioned in the SET clause; columns not explicitly modified retain their previous values.

There are two ways to modify a table using information contained in other tables in the database: using sub-selects, or specifying additional tables in the FROM clause. Which technique is more appropriate depends on the specific circumstances.

The optional RETURNING clause causes UPDATE to compute and return value(s) based on each row actually updated. Any expression using the table's columns, and/or columns of other tables mentioned in FROM, can be computed. The new (post-update) values of the table's columns are used. The syntax of the RETURNING list is identical to that of the output list of SELECT.

You must have the UPDATE privilege on the table, or at least on the column(s) that are listed to be updated. You must also have the SELECT privilege on any column whose values are read in the *expressions* or *condition*.

Parameters

with_query

The WITH clause allows you to specify one or more subqueries that can be referenced by name in the UPDATE query. See Section 7.8 and SELECT for details.

table_name

The name (optionally schema-qualified) of the table to update. If ONLY is specified before the table name, matching rows are updated in the named table only. If ONLY is not specified, matching rows

are also updated in any tables inheriting from the named table. Optionally, * can be specified after the table name to explicitly indicate that descendant tables are included.

alias

A substitute name for the target table. When an alias is provided, it completely hides the actual name of the table. For example, given `UPDATE foo AS f`, the remainder of the `UPDATE` statement must refer to this table as `f` not `foo`.

column_name

The name of a column in the table named by *table_name*. The column name can be qualified with a subfield name or array subscript, if needed. Do not include the table's name in the specification of a target column — for example, `UPDATE table_name SET table_name.col = 1` is invalid.

expression

An expression to assign to the column. The expression can use the old values of this and other columns in the table.

`DEFAULT`

Set the column to its default value (which will be NULL if no specific default expression has been assigned to it).

sub-SELECT

A `SELECT` sub-query that produces as many output columns as are listed in the parenthesized column list preceding it. The sub-query must yield no more than one row when executed. If it yields one row, its column values are assigned to the target columns; if it yields no rows, NULL values are assigned to the target columns. The sub-query can refer to old values of the current row of the table being updated.

from_item

A table expression allowing columns from other tables to appear in the `WHERE` condition and update expressions. This uses the same syntax as the *FROM Clause* of a `SELECT` statement; for example, an alias for the table name can be specified. Do not repeat the target table as a *from_item* unless you intend a self-join (in which case it must appear with an alias in the *from_item*).

condition

An expression that returns a value of type `boolean`. Only rows for which this expression returns `true` will be updated.

cursor_name

The name of the cursor to use in a `WHERE CURRENT OF` condition. The row to be updated is the one most recently fetched from this cursor. The cursor must be a non-grouping query on the `UPDATE`'s target table. Note that `WHERE CURRENT OF` cannot be specified together with a Boolean condition. See DECLARE for more information about using cursors with `WHERE CURRENT OF`.

output_expression

An expression to be computed and returned by the `UPDATE` command after each row is updated. The expression can use any column names of the table named by *table_name* or table(s) listed in `FROM`. Write * to return all columns.

output_name

A name to use for a returned column.

Outputs

On successful completion, an UPDATE command returns a command tag of the form

```
UPDATE count
```

The *count* is the number of rows updated, including matched rows whose values did not change. Note that the number may be less than the number of rows that matched the *condition* when updates were suppressed by a BEFORE UPDATE trigger. If *count* is 0, no rows were updated by the query (this is not considered an error).

If the UPDATE command contains a RETURNING clause, the result will be similar to that of a SELECT statement containing the columns and values defined in the RETURNING list, computed over the row(s) updated by the command.

Notes

When a FROM clause is present, what essentially happens is that the target table is joined to the tables mentioned in the *from_item* list, and each output row of the join represents an update operation for the target table. When using FROM you should ensure that the join produces at most one output row for each row to be modified. In other words, a target row shouldn't join to more than one row from the other table(s). If it does, then only one of the join rows will be used to update the target row, but which one will be used is not readily predictable.

Because of this indeterminacy, referencing other tables only within sub-selects is safer, though often harder to read and slower than using a join.

Examples

Change the word Drama to Dramatic in the column kind of the table films:

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

Adjust temperature entries and reset precipitation to its default value in one row of the table weather:

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

Perform the same operation and return the updated entries:

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03'
```

```
  RETURNING temp_lo, temp_hi, prcp;
```

Use the alternative column-list syntax to do the same update:

```
UPDATE weather SET (temp_lo, temp_hi, prcp) = (temp_lo+1, temp_lo+15, DEFAULT)
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

Increment the sales count of the salesperson who manages the account for Acme Corporation, using the `FROM` clause syntax:

```
UPDATE employees SET sales_count = sales_count + 1 FROM accounts
  WHERE accounts.name = 'Acme Corporation'
  AND employees.id = accounts.sales_person;
```

Perform the same operation, using a sub-select in the `WHERE` clause:

```
UPDATE employees SET sales_count = sales_count + 1 WHERE id =
  (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation');
```

Update contact names in an accounts table to match the currently assigned salesmen:

```
UPDATE accounts SET (contact_first_name, contact_last_name) =
    (SELECT first_name, last_name FROM salesmen
     WHERE salesmen.id = accounts.sales_id);
```

A similar result could be accomplished with a join:

```
UPDATE accounts SET contact_first_name = first_name,
                    contact_last_name = last_name
  FROM salesmen WHERE salesmen.id = accounts.sales_id;
```

However, the second query may give unexpected results if `salesmen.id` is not a unique key, whereas the first query is guaranteed to raise an error if there are multiple `id` matches. Also, if there is no match for a particular `accounts.sales_id` entry, the first query will set the corresponding name fields to NULL, whereas the second query will not update that row at all.

Update statistics in a summary table to match the current data:

```
UPDATE summary s SET (sum_x, sum_y, avg_x, avg_y) =
    (SELECT sum(x), sum(y), avg(x), avg(y) FROM data d
     WHERE d.group_id = s.group_id);
```

Attempt to insert a new stock item along with the quantity of stock. If the item already exists, instead update the stock count of the existing item. To do this without failing the entire transaction, use savepoints:

```
BEGIN;
```

```
-- other operations
SAVEPOINT sp1;
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');
-- Assume the above fails because of a unique key violation,
-- so now we issue these commands:
ROLLBACK TO sp1;
UPDATE wines SET stock = stock + 24 WHERE winename = 'Chateau Lafite 2003';
-- continue with other operations, and eventually
COMMIT;
```

Change the `kind` column of the table `films` in the row on which the cursor `c_films` is currently positioned:

```
UPDATE films SET kind = 'Dramatic' WHERE CURRENT OF c_films;
```

Compatibility

This command conforms to the SQL standard, except that the `FROM` and `RETURNING` clauses are PostgreSQL extensions, as is the ability to use `WITH` with `UPDATE`.

Some other database systems offer a `FROM` option in which the target table is supposed to be listed again within `FROM`. That is not how PostgreSQL interprets `FROM`. Be careful when porting applications that use this extension.

According to the standard, the source value for a parenthesized sub-list of column names can be any row-valued expression yielding the correct number of columns. PostgreSQL only allows the source value to be a parenthesized list of expressions or a sub-`SELECT`. An individual column's updated value can be specified as `DEFAULT` in the list-of-expressions case, but not inside a sub-`SELECT`.

VACUUM

Name

VACUUM — garbage-collect and optionally analyze a database

Synopsis

```
VACUUM [ ( { FULL | FREEZE | VERBOSE | ANALYZE } [, ...] ) ] [ table_name [ (column_name [, .
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] [ table_name ]
VACUUM [ FULL ] [ FREEZE ] [ VERBOSE ] ANALYZE [ table_name [ (column_name [, ...] ) ] ]
```

Description

VACUUM reclaims storage occupied by dead tuples. In normal PostgreSQL operation, tuples that are deleted or obsoleted by an update are not physically removed from their table; they remain present until a VACUUM is done. Therefore it's necessary to do VACUUM periodically, especially on frequently-updated tables.

With no parameter, VACUUM processes every table in the current database that the current user has permission to vacuum. With a parameter, VACUUM processes only that table.

VACUUM ANALYZE performs a VACUUM and then an ANALYZE for each selected table. This is a handy combination form for routine maintenance scripts. See ANALYZE for more details about its processing.

Plain VACUUM (without FULL) simply reclaims space and makes it available for re-use. This form of the command can operate in parallel with normal reading and writing of the table, as an exclusive lock is not obtained. However, extra space is not returned to the operating system (in most cases); it's just kept available for re-use within the same table. VACUUM FULL rewrites the entire contents of the table into a new disk file with no extra space, allowing unused space to be returned to the operating system. This form is much slower and requires an exclusive lock on each table while it is being processed.

When the option list is surrounded by parentheses, the options can be written in any order. Without parentheses, options must be specified in exactly the order shown above. The parenthesized syntax was added in PostgreSQL 9.0; the unparenthesized syntax is deprecated.

Parameters

FULL

Selects "full" vacuum, which can reclaim more space, but takes much longer and exclusively locks the table. This method also requires extra disk space, since it writes a new copy of the table and doesn't release the old copy until the operation is complete. Usually this should only be used when a significant amount of space needs to be reclaimed from within the table.

FREEZE

Selects aggressive "freezing" of tuples. Specifying FREEZE is equivalent to performing VACUUM with the vacuum_freeze_min_age and vacuum_freeze_table_age parameters set to zero. Aggressive freez-

ing is always performed when the table is rewritten, so this option is redundant when FULL is specified.

VERBOSE

Prints a detailed vacuum activity report for each table.

ANALYZE

Updates statistics used by the planner to determine the most efficient way to execute a query.

table_name

The name (optionally schema-qualified) of a specific table to vacuum. Defaults to all tables in the current database.

column_name

The name of a specific column to analyze. Defaults to all columns. If a column list is specified, ANALYZE is implied.

Outputs

When VERBOSE is specified, VACUUM emits progress messages to indicate which table is currently being processed. Various statistics about the tables are printed as well.

Notes

To vacuum a table, one must ordinarily be the table's owner or a superuser. However, database owners are allowed to vacuum all tables in their databases, except shared catalogs. (The restriction for shared catalogs means that a true database-wide VACUUM can only be performed by a superuser.) VACUUM will skip over any tables that the calling user does not have permission to vacuum.

VACUUM cannot be executed inside a transaction block.

For tables with GIN indexes, VACUUM (in any form) also completes any pending index insertions, by moving pending index entries to the appropriate places in the main GIN index structure. See Section 61.4.1 for details.

We recommend that active production databases be vacuumed frequently (at least nightly), in order to remove dead rows. After adding or deleting a large number of rows, it might be a good idea to issue a VACUUM ANALYZE command for the affected table. This will update the system catalogs with the results of all recent changes, and allow the PostgreSQL query planner to make better choices in planning queries.

The FULL option is not recommended for routine use, but might be useful in special cases. An example is when you have deleted or updated most of the rows in a table and would like the table to physically shrink to occupy less disk space and allow faster table scans. VACUUM FULL will usually shrink the table more than a plain VACUUM would.

VACUUM causes a substantial increase in I/O traffic, which might cause poor performance for other active sessions. Therefore, it is sometimes advisable to use the cost-based vacuum delay feature. See Section 18.4.4 for details.

PostgreSQL includes an "autovacuum" facility which can automate routine vacuum maintenance. For more information about automatic and manual vacuuming, see Section 23.1.

Examples

To clean a single table `onek`, analyze it for the optimizer and print a detailed vacuum activity report:

```
VACUUM (VERBOSE, ANALYZE) onek;
```

Compatibility

There is no `VACUUM` statement in the SQL standard.

See Also

vacuumdb, Section 18.4.4, Section 23.1.6

VALUES

Name

VALUES — compute a set of rows

Synopsis

```
VALUES ( expression [, ...] ) [, ...]
    [ ORDER BY sort_expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
```

Description

VALUES computes a row value or set of row values specified by value expressions. It is most commonly used to generate a "constant table" within a larger command, but it can be used on its own.

When more than one row is specified, all the rows must have the same number of elements. The data types of the resulting table's columns are determined by combining the explicit or inferred types of the expressions appearing in that column, using the same rules as for UNION (see Section 10.5).

Within larger commands, VALUES is syntactically allowed anywhere that SELECT is. Because it is treated like a SELECT by the grammar, it is possible to use the ORDER BY, LIMIT (or equivalently FETCH FIRST), and OFFSET clauses with a VALUES command.

Parameters

expression

A constant or expression to compute and insert at the indicated place in the resulting table (set of rows). In a VALUES list appearing at the top level of an INSERT, an *expression* can be replaced by DEFAULT to indicate that the destination column's default value should be inserted. DEFAULT cannot be used when VALUES appears in other contexts.

sort_expression

An expression or integer constant indicating how to sort the result rows. This expression can refer to the columns of the VALUES result as column1, column2, etc. For more details see *ORDER BY Clause*.

operator

A sorting operator. For details see *ORDER BY Clause*.

count

The maximum number of rows to return. For details see *LIMIT Clause*.

start

The number of rows to skip before starting to return rows. For details see *LIMIT Clause*.

Notes

VALUES lists with very large numbers of rows should be avoided, as you might encounter out-of-memory failures or poor performance. VALUES appearing within INSERT is a special case (because the desired column types are known from the INSERT's target table, and need not be inferred by scanning the VALUES list), so it can handle larger lists than are practical in other contexts.

Examples

A bare VALUES command:

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

This will return a table of two columns and three rows. It's effectively equivalent to:

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

More usually, VALUES is used within a larger SQL command. The most common use is in INSERT:

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

In the context of INSERT, entries of a VALUES list can be DEFAULT to indicate that the column default should be used here instead of specifying a value:

```
INSERT INTO films VALUES
    ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes'),
    ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama', DEFAULT);
```

VALUES can also be used where a sub-SELECT might be written, for example in a FROM clause:

```
SELECT f.*
  FROM films f, (VALUES('MGM', 'Horror'), ('UA', 'Sci-Fi')) AS t (studio, kind)
  WHERE f.studio = t.studio AND f.kind = t.kind;

UPDATE employees SET salary = salary * v.increase
  FROM (VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (depno, target, increase)
  WHERE employees.depno = v.depno AND employees.sales >= v.target;
```

Note that an AS clause is required when VALUES is used in a FROM clause, just as is true for SELECT. It is not required that the AS clause specify names for all the columns, but it's good practice to do so. (The default column names for VALUES are column1, column2, etc in PostgreSQL, but these names might be different in other database systems.)

When VALUES is used in INSERT, the values are all automatically coerced to the data type of the corresponding destination column. When it's used in other contexts, it might be necessary to specify the correct data type. If the entries are all quoted literal constants, coercing the first is sufficient to determine the assumed type for all:

```
SELECT * FROM machines
WHERE ip_address IN (VALUES('192.168.0.1'::inet), ('192.168.0.10'), ('192.168.1.43'));
```

> **Tip:** For simple IN tests, it's better to rely on the list-of-scalars form of IN than to write a VALUES query as shown above. The list of scalars method requires less writing and is often more efficient.

Compatibility

VALUES conforms to the SQL standard. LIMIT and OFFSET are PostgreSQL extensions; see also under SELECT.

See Also

INSERT, SELECT

II. PostgreSQL Client Applications

This part contains reference information for PostgreSQL client applications and utilities. Not all of these commands are of general utility; some might require special privileges. The common feature of these applications is that they can be run on any host, independent of where the database server resides.

When specified on the command line, user and database names have their case preserved — the presence of spaces or special characters might require quoting. Table names and other identifiers do not have their case preserved, except where documented, and might require quoting.

clusterdb

Name

`clusterdb` — cluster a PostgreSQL database

Synopsis

`clusterdb` [*connection-option*...] [`--verbose` | `-v`] [`--table` | `-t` *table*] ... [*dbname*]

`clusterdb` [*connection-option*...] [`--verbose` | `-v`] `--all` | `-a`

Description

clusterdb is a utility for reclustering tables in a PostgreSQL database. It finds tables that have previously been clustered, and clusters them again on the same index that was last used. Tables that have never been clustered are not affected.

clusterdb is a wrapper around the SQL command CLUSTER. There is no effective difference between clustering databases via this utility and via other methods for accessing the server.

Options

clusterdb accepts the following command-line arguments:

`-a`
`--all`

Cluster all databases.

`[-d]` *dbname*
`[--dbname=]`*dbname*

Specifies the name of the database to be clustered, when `-a`/`--all` is not used. If this is not specified, the database name is read from the environment variable `PGDATABASE`. If that is not set, the user name specified for the connection is used. The *dbname* can be a connection string. If so, connection string parameters will override any conflicting command line options.

`-e`
`--echo`

Echo the commands that clusterdb generates and sends to the server.

`-q`
`--quiet`

Do not display progress messages.

`-t` *`table`*
`--table=`*`table`*

Cluster *`table`* only. Multiple tables can be clustered by writing multiple `-t` switches.

`-v`
`--verbose`

Print detailed information during processing.

`-V`
`--version`

Print the clusterdb version and exit.

`-?`
`--help`

Show help about clusterdb command line arguments, and exit.

clusterdb also accepts the following command-line arguments for connection parameters:

`-h` *`host`*
`--host=`*`host`*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *`port`*
`--port=`*`port`*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *`username`*
`--username=`*`username`*

User name to connect as.

`-w`
`--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W`
`--password`

Force clusterdb to prompt for a password before connecting to a database.

This option is never essential, since clusterdb will automatically prompt for a password if the server demands password authentication. However, clusterdb will waste a connection attempt finding out

that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

`--maintenance-db=`*dbname*

Specifies the name of the database to connect to to discover which databases should be clustered, when `-a`/`--all` is used. If not specified, the `postgres` database will be used, or if that does not exist, `template1` will be used. This can be a connection string. If so, connection string parameters will override any conflicting command line options. Also, connection string parameters other than the database name itself will be re-used when connecting to other databases.

Environment

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14).

Diagnostics

In case of difficulty, see CLUSTER and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Examples

To cluster the database `test`:

```
$ clusterdb test
```

To cluster a single table `foo` in a database named `xyzzy`:

```
$ clusterdb --table foo xyzzy
```

See Also

CLUSTER

createdb

Name

createdb — create a new PostgreSQL database

Synopsis

createdb [*connection-option*...] [*option*...] [*dbname* [*description*]]

Description

createdb creates a new PostgreSQL database.

Normally, the database user who executes this command becomes the owner of the new database. However, a different owner can be specified via the -O option, if the executing user has appropriate privileges.

createdb is a wrapper around the SQL command CREATE DATABASE. There is no effective difference between creating databases via this utility and via other methods for accessing the server.

Options

createdb accepts the following command-line arguments:

dbname

Specifies the name of the database to be created. The name must be unique among all PostgreSQL databases in this cluster. The default is to create a database with the same name as the current system user.

description

Specifies a comment to be associated with the newly created database.

-D *tablespace*
--tablespace=*tablespace*

Specifies the default tablespace for the database. (This name is processed as a double-quoted identifier.)

-e
--echo

Echo the commands that createdb generates and sends to the server.

-E *encoding*
--encoding=*encoding*

Specifies the character encoding scheme to be used in this database. The character sets supported by the PostgreSQL server are described in Section 22.3.1.

`-l` *locale*
`--locale=`*locale*

Specifies the locale to be used in this database. This is equivalent to specifying both `--lc-collate` and `--lc-ctype`.

`--lc-collate=`*locale*

Specifies the LC_COLLATE setting to be used in this database.

`--lc-ctype=`*locale*

Specifies the LC_CTYPE setting to be used in this database.

`-O` *owner*
`--owner=`*owner*

Specifies the database user who will own the new database. (This name is processed as a double-quoted identifier.)

`-T` *template*
`--template=`*template*

Specifies the template database from which to build this database. (This name is processed as a double-quoted identifier.)

`-V`
`--version`

Print the createdb version and exit.

`-?`
`--help`

Show help about createdb command line arguments, and exit.

The options `-D`, `-l`, `-E`, `-O`, and `-T` correspond to options of the underlying SQL command CREATE DATABASE; see there for more information about them.

createdb also accepts the following command-line arguments for connection parameters:

`-h` *host*
`--host=`*host*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *port*
`--port=`*port*

Specifies the TCP port or the local Unix domain socket file extension on which the server is listening for connections.

`-U` *username*
`--username=`*username*

User name to connect as.

`-w`
`--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W`
`--password`

Force createdb to prompt for a password before connecting to a database.

This option is never essential, since createdb will automatically prompt for a password if the server demands password authentication. However, createdb will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

`--maintenance-db=`*`dbname`*

Specifies the name of the database to connect to when creating the new database. If not specified, the `postgres` database will be used; if that does not exist (or if it is the name of the new database being created), `template1` will be used. This can be a connection string. If so, connection string parameters will override any conflicting command line options.

Environment

`PGDATABASE`

If set, the name of the database to create, unless overridden on the command line.

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters. `PGUSER` also determines the name of the database to create, if it is not specified on the command line or by `PGDATABASE`.

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14).

Diagnostics

In case of difficulty, see CREATE DATABASE and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Examples

To create the database `demo` using the default database server:

```
$ createdb demo
```

To create the database `demo` using the server on host `eden`, port 5000, using the `LATIN1` encoding scheme with a look at the underlying command:

```
$ createdb -p 5000 -h eden -E LATIN1 -e demo
CREATE DATABASE demo ENCODING 'LATIN1';
```

See Also

dropdb, CREATE DATABASE

createlang

Name

`createlang` — install a PostgreSQL procedural language

Synopsis

`createlang` [*connection-option*...] *langname* [*dbname*]

`createlang` [*connection-option*...] `--list` | `-l` [*dbname*]

Description

createlang is a utility for adding a procedural language to a PostgreSQL database.

createlang is just a wrapper around the CREATE EXTENSION SQL command.

> **Caution**
>
> createlang is deprecated and may be removed in a future PostgreSQL release. Direct use of the `CREATE EXTENSION` command is recommended instead.

Options

createlang accepts the following command-line arguments:

langname

Specifies the name of the procedural language to be installed. (This name is lower-cased.)

`[-d]` *dbname*
`[--dbname=]`*dbname*

Specifies the database to which the language should be added. The default is to use the database with the same name as the current system user.

`-e`
`--echo`

Display SQL commands as they are executed.

`-l`
`--list`

Show a list of already installed languages in the target database.

`-V`
`--version`

Print the createlang version and exit.

`-?`
`--help`

Show help about createlang command line arguments, and exit.

createlang also accepts the following command-line arguments for connection parameters:

`-h` *host*
`--host=`*host*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *port*
`--port=`*port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *username*
`--username=`*username*

User name to connect as.

`-w`
`--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W`
`--password`

Force createlang to prompt for a password before connecting to a database.

This option is never essential, since createlang will automatically prompt for a password if the server demands password authentication. However, createlang will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

Environment

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14).

Diagnostics

Most error messages are self-explanatory. If not, run createlang with the `--echo` option and see the respective SQL command for details. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Notes

Use droplang to remove a language.

Examples

To install the language `pltcl` into the database `template1`:

```
$ createlang pltcl template1
```

Note that installing the language into `template1` will cause it to be automatically installed into subsequently-created databases as well.

See Also

droplang, CREATE EXTENSION, CREATE LANGUAGE

createuser

Name

`createuser` — define a new PostgreSQL user account

Synopsis

`createuser` [*connection-option*...] [*option*...] [*username*]

Description

createuser creates a new PostgreSQL user (or more precisely, a role). Only superusers and users with `CREATEROLE` privilege can create new users, so createuser must be invoked by someone who can connect as a superuser or a user with `CREATEROLE` privilege.

If you wish to create a new superuser, you must connect as a superuser, not merely with `CREATEROLE` privilege. Being a superuser implies the ability to bypass all access permission checks within the database, so superuserdom should not be granted lightly.

createuser is a wrapper around the SQL command CREATE ROLE. There is no effective difference between creating users via this utility and via other methods for accessing the server.

Options

createuser accepts the following command-line arguments:

username

Specifies the name of the PostgreSQL user to be created. This name must be different from all existing roles in this PostgreSQL installation.

`-c` *number*
`--connection-limit=`*number*

Set a maximum number of connections for the new user. The default is to set no limit.

`-d`
`--createdb`

The new user will be allowed to create databases.

`-D`
`--no-createdb`

The new user will not be allowed to create databases. This is the default.

`-e`
`--echo`

Echo the commands that createuser generates and sends to the server.

`-E`
`--encrypted`

Encrypts the user's password stored in the database. If not specified, the default password behavior is used.

`-g` *role*
`--role=`*role*

Indicates role to which this role will be added immediately as a new member. Multiple roles to which this role will be added as a member can be specified by writing multiple `-g` switches.

`-i`
`--inherit`

The new role will automatically inherit privileges of roles it is a member of. This is the default.

`-I`
`--no-inherit`

The new role will not automatically inherit privileges of roles it is a member of.

`--interactive`

Prompt for the user name if none is specified on the command line, and also prompt for whichever of the options `-d`/`-D`, `-r`/`-R`, `-s`/`-S` is not specified on the command line. (This was the default behavior up to PostgreSQL 9.1.)

`-l`
`--login`

The new user will be allowed to log in (that is, the user name can be used as the initial session user identifier). This is the default.

`-L`
`--no-login`

The new user will not be allowed to log in. (A role without login privilege is still useful as a means of managing database permissions.)

`-N`
`--unencrypted`

Does not encrypt the user's password stored in the database. If not specified, the default password behavior is used.

`-P`
`--pwprompt`

If given, createuser will issue a prompt for the password of the new user. This is not necessary if you do not plan on using password authentication.

`-r`
`--createrole`

The new user will be allowed to create new roles (that is, this user will have `CREATEROLE` privilege).

`-R`
`--no-createrole`

The new user will not be allowed to create new roles. This is the default.

`-s`
`--superuser`

The new user will be a superuser.

`-S`
`--no-superuser`

The new user will not be a superuser. This is the default.

`-V`
`--version`

Print the createuser version and exit.

`--replication`

The new user will have the `REPLICATION` privilege, which is described more fully in the documentation for CREATE ROLE.

`--no-replication`

The new user will not have the `REPLICATION` privilege, which is described more fully in the documentation for CREATE ROLE.

`-?`
`--help`

Show help about createuser command line arguments, and exit.

createuser also accepts the following command-line arguments for connection parameters:

`-h` *host*
`--host=`*host*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *port*
`--port=`*port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *username*
`--username=`*username*

User name to connect as (not the user name to create).

`-w`
`--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W`
`--password`

Force createuser to prompt for a password (for connecting to the server, not for the password of the new user).

This option is never essential, since createuser will automatically prompt for a password if the server demands password authentication. However, createuser will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

Environment

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14).

Diagnostics

In case of difficulty, see CREATE ROLE and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Examples

To create a user `joe` on the default database server:

```
$ createuser joe
```

To create a user `joe` on the default database server with prompting for some additional attributes:

```
$ createuser --interactive joe
Shall the new role be a superuser? (y/n) n
Shall the new role be allowed to create databases? (y/n) n
Shall the new role be allowed to create more new roles? (y/n) n
```

To create the same user `joe` using the server on host `eden`, port 5000, with attributes explicitly specified, taking a look at the underlying command:

```
$ createuser -h eden -p 5000 -S -D -R -e joe
CREATE ROLE joe NOSUPERUSER NOCREATEDB NOCREATEROLE INHERIT LOGIN;
```

To create the user `joe` as a superuser, and assign a password immediately:

```
$ createuser -P -s -e joe
Enter password for new role: xyzzy
Enter it again: xyzzy
CREATE ROLE joe PASSWORD 'md5b5f5ba1a423792b526f799ae4eb3d59e' SUPERUSER CREATEDB CREATEROLE INHERIT
```

In the above example, the new password isn't actually echoed when typed, but we show what was typed for clarity. As you see, the password is encrypted before it is sent to the client. If the option `--unencrypted` is used, the password *will* appear in the echoed command (and possibly also in the server log and elsewhere), so you don't want to use `-e` in that case, if anyone else can see your screen.

See Also

dropuser, CREATE ROLE

dropdb

Name

`dropdb` — remove a PostgreSQL database

Synopsis

`dropdb` [*`connection-option`*...] [*`option`*...] *`dbname`*

Description

dropdb destroys an existing PostgreSQL database. The user who executes this command must be a database superuser or the owner of the database.

dropdb is a wrapper around the SQL command DROP DATABASE. There is no effective difference between dropping databases via this utility and via other methods for accessing the server.

Options

dropdb accepts the following command-line arguments:

`dbname`

Specifies the name of the database to be removed.

`-e`
`--echo`

Echo the commands that dropdb generates and sends to the server.

`-i`
`--interactive`

Issues a verification prompt before doing anything destructive.

`-V`
`--version`

Print the dropdb version and exit.

`--if-exists`

Do not throw an error if the database does not exist. A notice is issued in this case.

`-?`
`--help`

Show help about dropdb command line arguments, and exit.

dropdb also accepts the following command-line arguments for connection parameters:

`-h` *host*
`--host=`*host*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *port*
`--port=`*port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *username*
`--username=`*username*

User name to connect as.

`-w`
`--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W`
`--password`

Force dropdb to prompt for a password before connecting to a database.

This option is never essential, since dropdb will automatically prompt for a password if the server demands password authentication. However, dropdb will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

`--maintenance-db=`*dbname*

Specifies the name of the database to connect to in order to drop the target database. If not specified, the `postgres` database will be used; if that does not exist (or is the database being dropped), `template1` will be used. This can be a connection string. If so, connection string parameters will override any conflicting command line options.

Environment

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14).

Diagnostics

In case of difficulty, see DROP DATABASE and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Examples

To destroy the database `demo` on the default database server:

```
$ dropdb demo
```

To destroy the database `demo` using the server on host `eden`, port 5000, with verification and a peek at the underlying command:

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE demo;
```

See Also

createdb, DROP DATABASE

droplang

Name

`droplang` — remove a PostgreSQL procedural language

Synopsis

`droplang` [*connection-option*...] *langname* [*dbname*]

`droplang` [*connection-option*...] `--list` | `-l` [*dbname*]

Description

droplang is a utility for removing an existing procedural language from a PostgreSQL database.

droplang is just a wrapper around the DROP EXTENSION SQL command.

> **Caution**
>
> droplang is deprecated and may be removed in a future PostgreSQL release. Direct use of the `DROP EXTENSION` command is recommended instead.

Options

droplang accepts the following command line arguments:

langname

Specifies the name of the procedural language to be removed. (This name is lower-cased.)

`[-d]` *dbname*
`[--dbname=]`*dbname*

Specifies from which database the language should be removed. The default is to use the database with the same name as the current system user.

`-e`
`--echo`

Display SQL commands as they are executed.

`-l`
`--list`

Show a list of already installed languages in the target database.

`-V`
`--version`

Print the droplang version and exit.

`-?`
`--help`

Show help about droplang command line arguments, and exit.

droplang also accepts the following command line arguments for connection parameters:

`-h` *host*
`--host=`*host*

Specifies the host name of the machine on which the server is running. If host begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *port*
`--port=`*port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *username*
`--username=`*username*

User name to connect as.

`-w`
`--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W`
`--password`

Force droplang to prompt for a password before connecting to a database.

This option is never essential, since droplang will automatically prompt for a password if the server demands password authentication. However, droplang will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

Environment

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14).

Diagnostics

Most error messages are self-explanatory. If not, run droplang with the `--echo` option and see under the respective SQL command for details. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Notes

Use createlang to add a language.

Examples

To remove the language `pltcl`:

```
$ droplang pltcl dbname
```

See Also

createlang, DROP EXTENSION, DROP LANGUAGE

dropuser

Name

`dropuser` — remove a PostgreSQL user account

Synopsis

`dropuser` [*`connection-option`*...] [*`option`*...] [*`username`*]

Description

dropuser removes an existing PostgreSQL user. Only superusers and users with the `CREATEROLE` privilege can remove PostgreSQL users. (To remove a superuser, you must yourself be a superuser.)

dropuser is a wrapper around the SQL command DROP ROLE. There is no effective difference between dropping users via this utility and via other methods for accessing the server.

Options

dropuser accepts the following command-line arguments:

`username`

Specifies the name of the PostgreSQL user to be removed. You will be prompted for a name if none is specified on the command line and the `-i`/`--interactive` option is used.

`-e`
`--echo`

Echo the commands that dropuser generates and sends to the server.

`-i`
`--interactive`

Prompt for confirmation before actually removing the user, and prompt for the user name if none is specified on the command line.

`-V`
`--version`

Print the dropuser version and exit.

`--if-exists`

Do not throw an error if the user does not exist. A notice is issued in this case.

`-?`
`--help`

Show help about dropuser command line arguments, and exit.

dropuser also accepts the following command-line arguments for connection parameters:

`-h` *host*
`--host=`*host*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *port*
`--port=`*port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *username*
`--username=`*username*

User name to connect as (not the user name to drop).

`-w`
`--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W`
`--password`

Force dropuser to prompt for a password before connecting to a database.

This option is never essential, since dropuser will automatically prompt for a password if the server demands password authentication. However, dropuser will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

Environment

`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14).

Diagnostics

In case of difficulty, see DROP ROLE and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Examples

To remove user `joe` from the default database server:

```
$ dropuser joe
```

To remove user `joe` using the server on host `eden`, port 5000, with verification and a peek at the underlying command:

```
$ dropuser -p 5000 -h eden -i -e joe
Role "joe" will be permanently removed.
Are you sure? (y/n) y
DROP ROLE joe;
```

See Also

createuser, DROP ROLE

ecpg

Name

ecpg — embedded SQL C preprocessor

Synopsis

ecpg [*option*...] *file*...

Description

ecpg is the embedded SQL preprocessor for C programs. It converts C programs with embedded SQL statements to normal C code by replacing the SQL invocations with special function calls. The output files can then be processed with any C compiler tool chain.

ecpg will convert each input file given on the command line to the corresponding C output file. If an input file name does not have any extension, .pgc is assumed. The file's extension will be replaced by .c to construct the output file name. But the output file name can be overridden using the -o option.

If an input file name is just -, ecpg reads the program from standard input (and writes to standard output, unless that is overridden with -o).

This reference page does not describe the embedded SQL language. See Chapter 33 for more information on that topic.

Options

ecpg accepts the following command-line arguments:

-c

Automatically generate certain C code from SQL code. Currently, this works for EXEC SQL TYPE.

-C *mode*

Set a compatibility mode. *mode* can be INFORMIX or INFORMIX_SE.

-D *symbol*

Define a C preprocessor symbol.

-h

Process header files. When this option is specified, the output file extension becomes .h not .c, and the default input file extension is .pgh not .pgc. Also, the -c option is forced on.

-i

Parse system include files as well.

`-I` *directory*

Specify an additional include path, used to find files included via `EXEC SQL INCLUDE`. Defaults are `.` (current directory), `/usr/local/include`, the PostgreSQL include directory which is defined at compile time (default: `/usr/local/pgsql/include`), and `/usr/include`, in that order.

`-o` *filename*

Specifies that `ecpg` should write all its output to the given *filename*. Write `-o -` to send all output to standard output.

`-r` *option*

Selects run-time behavior. *Option* can be one of the following:

`no_indicator`

Do not use indicators but instead use special values to represent null values. Historically there have been databases using this approach.

`prepare`

Prepare all statements before using them. Libecpg will keep a cache of prepared statements and reuse a statement if it gets executed again. If the cache runs full, libecpg will free the least used statement.

`questionmarks`

Allow question mark as placeholder for compatibility reasons. This used to be the default long ago.

`-t`

Turn on autocommit of transactions. In this mode, each SQL command is automatically committed unless it is inside an explicit transaction block. In the default mode, commands are committed only when `EXEC SQL COMMIT` is issued.

`-v`

Print additional information including the version and the "include" path.

`--version`

Print the ecpg version and exit.

`-?`
`--help`

Show help about ecpg command line arguments, and exit.

Notes

When compiling the preprocessed C code files, the compiler needs to be able to find the ECPG header files in the PostgreSQL include directory. Therefore, you might have to use the `-I` option when invoking the compiler (e.g., `-I/usr/local/pgsql/include`).

Programs using C code with embedded SQL have to be linked against the `libecpg` library, for example using the linker options `-L/usr/local/pgsql/lib -lecpg`.

The value of either of these directories that is appropriate for the installation can be found out using pg_config.

Examples

If you have an embedded SQL C source file named `prog1.pgc`, you can create an executable program using the following sequence of commands:

```
ecpg prog1.pgc
cc -I/usr/local/pgsql/include -c prog1.c
cc -o prog1 prog1.o -L/usr/local/pgsql/lib -lecpg
```

pg_basebackup

Name

pg_basebackup — take a base backup of a PostgreSQL cluster

Synopsis

pg_basebackup [*option*...]

Description

pg_basebackup is used to take base backups of a running PostgreSQL database cluster. These are taken without affecting other clients to the database, and can be used both for point-in-time recovery (see Section 24.3) and as the starting point for a log shipping or streaming replication standby servers (see Section 25.2).

pg_basebackup makes a binary copy of the database cluster files, while making sure the system is put in and out of backup mode automatically. Backups are always taken of the entire database cluster; it is not possible to back up individual databases or database objects. For individual database backups, a tool such as pg_dump must be used.

The backup is made over a regular PostgreSQL connection, and uses the replication protocol. The connection must be made with a superuser or a user having REPLICATION permissions (see Section 20.2), and pg_hba.conf must explicitly permit the replication connection. The server must also be configured with max_wal_senders set high enough to leave at least one session available for the backup.

There can be multiple pg_basebackups running at the same time, but it is better from a performance point of view to take only one backup, and copy the result.

pg_basebackup can make a base backup from not only the master but also the standby. To take a backup from the standby, set up the standby so that it can accept replication connections (that is, set max_wal_senders and hot_standby, and configure host-based authentication). You will also need to enable full_page_writes on the master.

Note that there are some limitations in an online backup from the standby:

- The backup history file is not created in the database cluster backed up.
- There is no guarantee that all WAL files required for the backup are archived at the end of backup. If you are planning to use the backup for an archive recovery and want to ensure that all required files are available at that moment, you need to include them into the backup by using the -x option.
- If the standby is promoted to the master during online backup, the backup fails.
- All WAL records required for the backup must contain sufficient full-page writes, which requires you to enable full_page_writes on the master and not to use a tool like pg_compresslog as archive_command to remove full-page writes from WAL files.

Options

The following command-line options control the location and format of the output.

`-D` *directory*
`--pgdata=`*directory*

Directory to write the output to. pg_basebackup will create the directory and any parent directories if necessary. The directory may already exist, but it is an error if the directory already exists and is not empty.

When the backup is in tar mode, and the directory is specified as `-` (dash), the tar file will be written to `stdout`.

This option is required.

`-F` *format*
`--format=`*format*

Selects the format for the output. *format* can be one of the following:

`p`
`plain`

Write the output as plain files, with the same layout as the current data directory and tablespaces. When the cluster has no additional tablespaces, the whole database will be placed in the target directory. If the cluster contains additional tablespaces, the main data directory will be placed in the target directory, but all other tablespaces will be placed in the same absolute path as they have on the server.

This is the default format.

`t`
`tar`

Write the output as tar files in the target directory. The main data directory will be written to a file named `base.tar`, and all other tablespaces will be named after the tablespace OID.

If the value `-` (dash) is specified as target directory, the tar contents will be written to standard output, suitable for piping to for example gzip. This is only possible if the cluster has no additional tablespaces.

`-r` *rate*
`--max-rate=`*rate*

The maximum transfer rate of data transferred from the server. Values are in kilobytes per second. Use a suffix of `M` to indicate megabytes per second. A suffix of `k` is also accepted, and has no effect. Valid values are between 32 kilobytes per second and 1024 megabytes per second.

The purpose is to limit the impact of pg_basebackup on the running server.

This option always affects transfer of the data directory. Transfer of WAL files is only affected if the collection method is `fetch`.

`-R`
`--write-recovery-conf`

Write a minimal `recovery.conf` in the output directory (or into the base archive file when using tar format) to ease setting up a standby server.

`-T` *`olddir=newdir`*
`--tablespace-mapping=`*`olddir=newdir`*

Relocate the tablespace in directory *`olddir`* to *`newdir`* during the backup. To be effective, *`olddir`* must exactly match the path specification of the tablespace as it is currently defined. (But it is not an error if there is no tablespace in *`olddir`* contained in the backup.) Both *`olddir`* and *`newdir`* must be absolute paths. If a path happens to contain a = sign, escape it with a backslash. This option can be specified multiple times for multiple tablespaces. See examples below.

If a tablespace is relocated in this way, the symbolic links inside the main data directory are updated to point to the new location. So the new data directory is ready to be used for a new server instance with all tablespaces in the updated locations.

`--xlogdir=`*`xlogdir`*

Specifies the location for the transaction log directory. *`xlogdir`* must be an absolute path. The transaction log directory can only be specified when the backup is in plain mode.

`-x`
`--xlog`

Using this option is equivalent of using `-X` with method `fetch`.

`-X` *`method`*
`--xlog-method=`*`method`*

Includes the required transaction log files (WAL files) in the backup. This will include all transaction logs generated during the backup. If this option is specified, it is possible to start a postmaster directly in the extracted directory without the need to consult the log archive, thus making this a completely standalone backup.

The following methods for collecting the transaction logs are supported:

`f`
`fetch`

The transaction log files are collected at the end of the backup. Therefore, it is necessary for the wal_keep_segments parameter to be set high enough that the log is not removed before the end of the backup. If the log has been rotated when it's time to transfer it, the backup will fail and be unusable.

`s`
`stream`

Stream the transaction log while the backup is created. This will open a second connection to the server and start streaming the transaction log in parallel while running the backup. Therefore, it will use up two connections configured by the max_wal_senders parameter. As long as the client can keep up with transaction log received, using this mode requires no extra transaction logs to be saved on the master.

`-z`
`--gzip`

Enables gzip compression of tar file output, with the default compression level. Compression is only available when using the tar format.

`-Z` *`level`*
`--compress=`*`level`*

Enables gzip compression of tar file output, and specifies the compression level (0 through 9, 0 being no compression and 9 being best compression). Compression is only available when using the tar format.

The following command-line options control the generation of the backup and the running of the program.

`-c` *`fast|spread`*
`--checkpoint=`*`fast|spread`*

Sets checkpoint mode to fast (immediate) or spread (default) (see Section 24.3.3).

`-l` *`label`*
`--label=`*`label`*

Sets the label for the backup. If none is specified, a default value of "`pg_basebackup base backup`" will be used.

`-P`
`--progress`

Enables progress reporting. Turning this on will deliver an approximate progress report during the backup. Since the database may change during the backup, this is only an approximation and may not end at exactly `100%`. In particular, when WAL log is included in the backup, the total amount of data cannot be estimated in advance, and in this case the estimated target size will increase once it passes the total estimate without WAL.

When this is enabled, the backup will start by enumerating the size of the entire database, and then go back and send the actual contents. This may make the backup take slightly longer, and in particular it will take longer before the first data is sent.

`-v`
`--verbose`

Enables verbose mode. Will output some extra steps during startup and shutdown, as well as show the exact file name that is currently being processed if progress reporting is also enabled.

The following command-line options control the database connection parameters.

`-d` *`connstr`*
`--dbname=`*`connstr`*

Specifies parameters used to connect to the server, as a connction string; these will override any conflicting command line options.

The option is called `--dbname` for consistency with other client applications, but because pg_basebackup doesn't connect to any particular database in the cluster, database name in the connection string will be ignored.

`-h` *host*
`--host=`*host*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the `PGHOST` environment variable, if set, else a Unix domain socket connection is attempted.

`-p` *port*
`--port=`*port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the `PGPORT` environment variable, if set, or a compiled-in default.

`-s` *interval*
`--status-interval=`*interval*

Specifies the number of seconds between status packets sent back to the server. This allows for easier monitoring of the progress from server. A value of zero disables the periodic status updates completely, although an update will still be sent when requested by the server, to avoid timeout disconnect. The default value is 10 seconds.

`-U` *username*
`--username=`*username*

User name to connect as.

`-w`
`--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W`
`--password`

Force pg_basebackup to prompt for a password before connecting to a database.

This option is never essential, since pg_basebackup will automatically prompt for a password if the server demands password authentication. However, pg_basebackup will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

Other options are also available:

`-V`
`--version`

Print the pg_basebackup version and exit.

`-?`
`--help`

Show help about pg_basebackup command line arguments, and exit.

Environment

This utility, like most other PostgreSQL utilities, uses the environment variables supported by libpq (see Section 31.14).

Notes

At the beginning of the backup, a checkpoint needs to be written on the server the backup is taken from. Especially if the option `--checkpoint=fast` is not used, this can take some time during which pg_basebackup will be appear to be idle.

The backup will include all files in the data directory and tablespaces, including the configuration files and any additional files placed in the directory by third parties. But only regular files and directories are copied. Symbolic links (other than those used for tablespaces) and special device files are skipped. (See Section 50.3 for the precise details.)

Tablespaces will in plain format by default be backed up to the same path they have on the server, unless the option `--tablespace-mapping` is used. Without this option, running a plain format base backup on the same host as the server will not work if tablespaces are in use, because the backup would have to be written to the same directory locations as the original tablespaces.

When tar format mode is used, it is the user's responsibility to unpack each tar file before starting the PostgreSQL server. If there are additional tablespaces, the tar files for them need to be unpacked in the correct locations. In this case the symbolic links for those tablespaces will be created by the server according to the contents of the `tablespace_map` file that is included in the `base.tar` file.

pg_basebackup works with servers of the same or an older major version, down to 9.1. However, WAL streaming mode (`-X stream`) only works with server version 9.3 and later, and tar format mode (`--format=tar`) of the current version only works with server version 9.5 or later.

Examples

To create a base backup of the server at `mydbserver` and store it in the local directory `/usr/local/pgsql/data`:

```
$ pg_basebackup -h mydbserver -D /usr/local/pgsql/data
```

To create a backup of the local server with one compressed tar file for each tablespace, and store it in the directory `backup`, showing a progress report while running:

```
$ pg_basebackup -D backup -Ft -z -P
```

To create a backup of a single-tablespace local database and compress this with bzip2:

```
$ pg_basebackup -D - -Ft | bzip2 > backup.tar.bz2
```

(This command will fail if there are multiple tablespaces in the database.)

To create a backup of a local database where the tablespace in `/opt/ts` is relocated to `./backup/ts`:

```
$ pg_basebackup -D backup/data -T /opt/ts=$(pwd)/backup/ts
```

See Also

pg_dump

pgbench

Name

pgbench — run a benchmark test on PostgreSQL

Synopsis

pgbench -i [*option*...] [*dbname*]

pgbench [*option*...] [*dbname*]

Description

pgbench is a simple program for running benchmark tests on PostgreSQL. It runs the same sequence of SQL commands over and over, possibly in multiple concurrent database sessions, and then calculates the average transaction rate (transactions per second). By default, pgbench tests a scenario that is loosely based on TPC-B, involving five SELECT, UPDATE, and INSERT commands per transaction. However, it is easy to test other cases by writing your own transaction script files.

Typical output from pgbench looks like:

```
transaction type: TPC-B (sort of)
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
tps = 85.184871 (including connections establishing)
tps = 85.296346 (excluding connections establishing)
```

The first six lines report some of the most important parameter settings. The next line reports the number of transactions completed and intended (the latter being just the product of number of clients and number of transactions per client); these will be equal unless the run failed before completion. (In -T mode, only the actual number of transactions is printed.) The last two lines report the number of transactions per second, figured with and without counting the time to start database sessions.

The default TPC-B-like transaction test requires specific tables to be set up beforehand. pgbench should be invoked with the -i (initialize) option to create and populate these tables. (When you are testing a custom script, you don't need this step, but will instead need to do whatever setup your test needs.) Initialization looks like:

```
pgbench -i [ other-options ] dbname
```

where *dbname* is the name of the already-created database to test in. (You may also need -h, -p, and/or -U options to specify how to connect to the database server.)

> **Caution**
>
> `pgbench -i` creates four tables `pgbench_accounts`, `pgbench_branches`, `pgbench_history`, and `pgbench_tellers`, destroying any existing tables of these names. Be very careful to use another database if you have tables having these names!

At the default "scale factor" of 1, the tables initially contain this many rows:

```
table                   # of rows
---------------------------------
pgbench_branches        1
pgbench_tellers         10
pgbench_accounts        100000
pgbench_history         0
```

You can (and, for most purposes, probably should) increase the number of rows by using the `-s` (scale factor) option. The `-F` (fillfactor) option might also be used at this point.

Once you have done the necessary setup, you can run your benchmark with a command that doesn't include `-i`, that is

```
pgbench [ options ] dbname
```

In nearly all cases, you'll need some options to make a useful test. The most important options are `-c` (number of clients), `-t` (number of transactions), `-T` (time limit), and `-f` (specify a custom script file). See below for a full list.

Options

The following is divided into three subsections: Different options are used during database initialization and while running benchmarks, some options are useful in both cases.

Initialization Options

pgbench accepts the following command-line initialization arguments:

`-i`
`--initialize`

Required to invoke initialization mode.

`-F` *fillfactor*
`--fillfactor=`*fillfactor*

Create the `pgbench_accounts`, `pgbench_tellers` and `pgbench_branches` tables with the given fillfactor. Default is 100.

`-n`
`--no-vacuum`

Perform no vacuuming after initialization.

`-q`
`--quiet`

Switch logging to quiet mode, producing only one progress message per 5 seconds. The default logging prints one message each 100000 rows, which often outputs many lines per second (especially on good hardware).

`-s` *`scale_factor`*
`--scale=`*`scale_factor`*

Multiply the number of rows generated by the scale factor. For example, `-s 100` will create 10,000,000 rows in the `pgbench_accounts` table. Default is 1. When the scale is 20,000 or larger, the columns used to hold account identifiers (`aid` columns) will switch to using larger integers (`bigint`), in order to be big enough to hold the range of account identifiers.

`--foreign-keys`

Create foreign key constraints between the standard tables.

`--index-tablespace=`*`index_tablespace`*

Create indexes in the specified tablespace, rather than the default tablespace.

`--tablespace=`*`tablespace`*

Create tables in the specified tablespace, rather than the default tablespace.

`--unlogged-tables`

Create all tables as unlogged tables, rather than permanent tables.

Benchmarking Options

pgbench accepts the following command-line benchmarking arguments:

`-c` *`clients`*
`--client=`*`clients`*

Number of clients simulated, that is, number of concurrent database sessions. Default is 1.

`-C`
`--connect`

Establish a new connection for each transaction, rather than doing it just once per client session. This is useful to measure the connection overhead.

`-d`
`--debug`

Print debugging output.

`-D` *`varname=value`*
`--define=`*`varname=value`*

Define a variable for use by a custom script (see below). Multiple `-D` options are allowed.

`-f` *filename*
`--file=`*filename*

Read transaction script from *filename*. See below for details. `-N`, `-S`, and `-f` are mutually exclusive.

`-j` *threads*
`--jobs=`*threads*

Number of worker threads within pgbench. Using more than one thread can be helpful on multi-CPU machines. The number of clients must be a multiple of the number of threads, since each thread is given the same number of client sessions to manage. Default is 1.

`-l`
`--log`

Write the time taken by each transaction to a log file. See below for details.

`-L` *limit*
`--latency-limit=`*limit*

Transactions that last more than *limit* milliseconds are counted and reported separately, as *late*.

When throttling is used (`--rate=...`), transactions that lag behind schedule by more than *limit* ms, and thus have no hope of meeting the latency limit, are not sent to the server at all. They are counted and reported separately as *skipped*.

`-M` *querymode*
`--protocol=`*querymode*

Protocol to use for submitting queries to the server:

- `simple`: use simple query protocol.
- `extended`: use extended query protocol.
- `prepared`: use extended query protocol with prepared statements.

The default is simple query protocol. (See Chapter 50 for more information.)

`-n`
`--no-vacuum`

Perform no vacuuming before running the test. This option is *necessary* if you are running a custom test scenario that does not include the standard tables `pgbench_accounts`, `pgbench_branches`, `pgbench_history`, and `pgbench_tellers`.

`-N`
`--skip-some-updates`

Do not update `pgbench_tellers` and `pgbench_branches`. This will avoid update contention on these tables, but it makes the test case even less like TPC-B.

`-P` *sec*
`--progress=`*sec*

Show progress report every `sec` seconds. The report includes the time since the beginning of the run, the tps since the last report, and the transaction latency average and standard deviation since the last report. Under throttling (`-R`), the latency is computed with respect to the transaction scheduled start time, not the actual transaction beginning time, thus it also includes the average schedule lag time.

`-r`
`--report-latencies`

Report the average per-statement latency (execution time from the perspective of the client) of each command after the benchmark finishes. See below for details.

`-R` *rate*
`--rate=`*rate*

Execute transactions targeting the specified rate instead of running as fast as possible (the default). The rate is given in transactions per second. If the targeted rate is above the maximum possible rate, the rate limit won't impact the results.

The rate is targeted by starting transactions along a Poisson-distributed schedule time line. The expected start time schedule moves forward based on when the client first started, not when the previous transaction ended. That approach means that when transactions go past their original scheduled end time, it is possible for later ones to catch up again.

When throttling is active, the transaction latency reported at the end of the run is calculated from the scheduled start times, so it includes the time each transaction had to wait for the previous transaction to finish. The wait time is called the schedule lag time, and its average and maximum are also reported separately. The transaction latency with respect to the actual transaction start time, i.e., the time spent executing the transaction in the database, can be computed by subtracting the schedule lag time from the reported latency.

If `--latency-limit` is used together with `--rate`, a transaction can lag behind so much that it is already over the latency limit when the previous transaction ends, because the latency is calculated from the scheduled start time. Such transactions are not sent to the server, but are skipped altogether and counted separately.

A high schedule lag time is an indication that the system cannot process transactions at the specified rate, with the chosen number of clients and threads. When the average transaction execution time is longer than the scheduled interval between each transaction, each successive transaction will fall further behind, and the schedule lag time will keep increasing the longer the test run is. When that happens, you will have to reduce the specified transaction rate.

`-s` *scale_factor*
`--scale=`*scale_factor*

Report the specified scale factor in pgbench's output. With the built-in tests, this is not necessary; the correct scale factor will be detected by counting the number of rows in the `pgbench_branches` table. However, when testing custom benchmarks (`-f` option), the scale factor will be reported as 1 unless this option is used.

`-S`
`--select-only`

Perform select-only transactions instead of TPC-B-like test.

`-t` *transactions*
`--transactions=`*transactions*

Number of transactions each client runs. Default is 10.

`-T` *seconds*
`--time=`*seconds*

Run the test for this many seconds, rather than a fixed number of transactions per client. `-t` and `-T` are mutually exclusive.

`-v`
`--vacuum-all`

Vacuum all four standard tables before running the test. With neither `-n` nor `-v`, pgbench will vacuum the `pgbench_tellers` and `pgbench_branches` tables, and will truncate `pgbench_history`.

`--aggregate-interval=`*seconds*

Length of aggregation interval (in seconds). May be used only together with -l - with this option, the log contains per-interval summary (number of transactions, min/max latency and two additional fields useful for variance estimation).

This option is not currently supported on Windows.

`--sampling-rate=`*rate*

Sampling rate, used when writing data into the log, to reduce the amount of log generated. If this option is given, only the specified fraction of transactions are logged. 1.0 means all transactions will be logged, 0.05 means only 5% of the transactions will be logged.

Remember to take the sampling rate into account when processing the log file. For example, when computing tps values, you need to multiply the numbers accordingly (e.g., with 0.01 sample rate, you'll only get 1/100 of the actual tps).

Common Options

pgbench accepts the following command-line common arguments:

`-h` *hostname*
`--host=`*hostname*

The database server's host name

`-p` *port*
`--port=`*port*

The database server's port number

`-U` *login*
`--username=`*login*

The user name to connect as

`-V`
`--version`

Print the pgbench version and exit.

`-?`
`--help`

Show help about pgbench command line arguments, and exit.

Notes

What is the "Transaction" Actually Performed in pgbench?

The default transaction script issues seven commands per transaction:

1. `BEGIN;`
2. `UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;`
3. `SELECT abalance FROM pgbench_accounts WHERE aid = :aid;`
4. `UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;`
5. `UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;`
6. `INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);`
7. `END;`

If you specify `-N`, steps 4 and 5 aren't included in the transaction. If you specify `-S`, only the `SELECT` is issued.

Custom Scripts

pgbench has support for running custom benchmark scenarios by replacing the default transaction script (described above) with a transaction script read from a file (`-f` option). In this case a "transaction" counts as one execution of a script file. You can even specify multiple scripts (multiple `-f` options), in which case a random one of the scripts is chosen each time a client session starts a new transaction.

The format of a script file is one SQL command per line; multiline SQL commands are not supported. Empty lines and lines beginning with `--` are ignored. Script file lines can also be "meta commands", which are interpreted by pgbench itself, as described below.

There is a simple variable-substitution facility for script files. Variables can be set by the command-line `-D` option, explained above, or by the meta commands explained below. In addition to any variables preset by `-D` command-line options, there are a few variables that are preset automatically, listed in Table 1. A value specified for these variables using `-D` takes precedence over the automatic presets. Once set, a variable's value can be inserted into a SQL command by writing `:`*`variablename`*. When running more than one client session, each session has its own set of variables.

Table 1. Automatic variables

Variable	Description
`scale`	current scale factor
`client_id`	unique number identifying the client session (starts from zero)

Script file meta commands begin with a backslash (`\`). Arguments to a meta command are separated by white space. These meta commands are supported:

`\set` *`varname expression`*

Sets variable *`varname`* to an integer value calculated from *`expression`*. The expression may contain integer constants such as `5432`, references to variables `:`*`variablename`*, and expressions composed of unary (`-`) or binary operators (`+`, `-`, `*`, `/`, `%`) with their usual associativity, and parentheses.

Examples:

```
\set ntellers 10 * :scale
\set aid (1021 * :aid) % (100000 * :scale) + 1
```

`\setrandom` *`varname min max`* `[ uniform | { gaussian | exponential } `*`parameter`*` ]`

Sets variable *`varname`* to a random integer value between the limits *`min`* and *`max`* inclusive. Each limit can be either an integer constant or a `:`*`variablename`* reference to a variable having an integer value.

By default, or when `uniform` is specified, all values in the range are drawn with equal probability. Specifying `gaussian` or `exponential` options modifies this behavior; each requires a mandatory parameter which determines the precise shape of the distribution.

For a Gaussian distribution, the interval is mapped onto a standard normal distribution (the classical bell-shaped Gaussian curve) truncated at `-`*`parameter`* on the left and `+`*`parameter`* on the right. Values in the middle of the interval are more likely to be drawn. To be precise, if `PHI(x)` is the cumulative distribution function of the standard normal distribution, with mean `mu` defined as `(max + min) / 2.0`, with

```
f(x) = PHI(2.0 * parameter * (x - mu) / (max - min + 1)) /
    (2.0 * PHI(parameter) - 1.0)
```

then value *`i`* between *`min`* and *`max`* inclusive is drawn with probability: `f(i + 0.5) - f(i - 0.5)`. Intuitively, the larger *`parameter`*, the more frequently values close to the middle of the interval are drawn, and the less frequently values close to the *`min`* and *`max`* bounds. About 67% of values are drawn from the middle `1.0 / `*`parameter`*, that is a relative `0.5 / `*`parameter`* around the mean, and 95% in the middle `2.0 / `*`parameter`*, that is a relative `1.0 / `*`parameter`* around the mean; for instance, if *`parameter`* is 4.0, 67% of values are drawn from the middle quarter (1.0 / 4.0) of the interval (i.e., from `3.0 / 8.0` to `5.0 / 8.0`) and 95% from the middle half (`2.0 / 4.0`) of the interval (second and third quartiles). The minimum *`parameter`* is 2.0 for performance of the Box-Muller transform.

For an exponential distribution, *`parameter`* controls the distribution by truncating a quickly-decreasing exponential distribution at *`parameter`*, and then projecting onto integers between the bounds. To be precise, with

```
f(x) = exp(-parameter * (x - min) / (max - min + 1)) / (1.0 - exp(-parameter))
```

Then value *i* between *min* and *max* inclusive is drawn with probability: `f(x) - f(x + 1)`. Intuitively, the larger *parameter*, the more frequently values close to *min* are accessed, and the less frequently values close to *max* are accessed. The closer to 0 *parameter*, the flatter (more uniform) the access distribution. A crude approximation of the distribution is that the most frequent 1% values in the range, close to *min*, are drawn *parameter*% of the time. *parameter* value must be strictly positive.

Example:

```
\setrandom aid 1 :naccounts gaussian 5.0
```

`\sleep` *number* `[ us | ms | s ]`

Causes script execution to sleep for the specified duration in microseconds (`us`), milliseconds (`ms`) or seconds (`s`). If the unit is omitted then seconds are the default. *number* can be either an integer constant or a `:`*variablename* reference to a variable having an integer value.

Example:

```
\sleep 10 ms
```

`\setshell` *varname command* `[` *argument* `... ]`

Sets variable *varname* to the result of the shell command *command*. The command must return an integer value through its standard output.

argument can be either a text constant or a `:`*variablename* reference to a variable of any types. If you want to use *argument* starting with colons, you need to add an additional colon at the beginning of *argument*.

Example:

```
\setshell variable_to_be_assigned command literal_argument :variable ::literal_starting_
```

`\shell` *command* `[` *argument* `... ]`

Same as `\setshell`, but the result is ignored.

Example:

```
\shell command literal_argument :variable ::literal_starting_with_colon
```

As an example, the full definition of the built-in TPC-B-like transaction is:

```
\set nbranches :scale
\set ntellers 10 * :scale
\set naccounts 100000 * :scale
\setrandom aid 1 :naccounts
\setrandom bid 1 :nbranches
\setrandom tid 1 :ntellers
\setrandom delta -5000 5000
BEGIN;
UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta,
END;
```

This script allows each iteration of the transaction to reference different, randomly-chosen rows. (This example also shows why it's important for each client session to have its own variables — otherwise they'd not be independently touching different rows.)

Per-Transaction Logging

With the `-l` option but without the `--aggregate-interval`, pgbench writes the time taken by each transaction to a log file. The log file will be named `pgbench_log.`*`nnn`*, where *`nnn`* is the PID of the pgbench process. If the `-j` option is 2 or higher, creating multiple worker threads, each will have its own log file. The first worker will use the same name for its log file as in the standard single worker case. The additional log files for the other workers will be named `pgbench_log.`*`nnn`*`.`*`mmm`*, where *`mmm`* is a sequential number for each worker starting with 1.

The format of the log is:

```
client_id transaction_no time file_no time_epoch time_us [schedule_lag]
```

where *`time`* is the total elapsed transaction time in microseconds, *`file_no`* identifies which script file was used (useful when multiple scripts were specified with `-f`), and *`time_epoch`*/*`time_us`* are a Unix epoch format time stamp and an offset in microseconds (suitable for creating an ISO 8601 time stamp with fractional seconds) showing when the transaction completed. Field *`schedule_lag`* is the difference between the transaction's scheduled start time, and the time it actually started, in microseconds. It is only present when the `--rate` option is used. When both `--rate` and `--latency-limit` are used, the *`time`* for a skipped transaction will be reported as `skipped`.

Here is a snippet of the log file generated:

```
0 199 2241 0 1175850568 995598
0 200 2465 0 1175850568 998079
0 201 2513 0 1175850569 608
0 202 2038 0 1175850569 2663
```

Another example with --rate=100 and --latency-limit=5 (note the additional *`schedule_lag`* column):

```
0 81 4621 0 1412881037 912698 3005
0 82 6173 0 1412881037 914578 4304
0 83 skipped 0 1412881037 914578 5217
0 83 skipped 0 1412881037 914578 5099
0 83 4722 0 1412881037 916203 3108
0 84 4142 0 1412881037 918023 2333
0 85 2465 0 1412881037 919759 740
```

In this example, transaction 82 was late, because its latency (6.173 ms) was over the 5 ms limit. The next two transactions were skipped, because they were already late before they were even started.

When running a long test on hardware that can handle a lot of transactions, the log files can become very large. The `--sampling-rate` option can be used to log only a random sample of transactions.

Aggregated Logging

With the `--aggregate-interval` option, the logs use a bit different format:

```
interval_start num_of_transactions latency_sum latency_2_sum min_latency max_latency [lag_sum lag_2_
```

where *`interval_start`* is the start of the interval (Unix epoch format time stamp), *`num_of_transactions`* is the number of transactions within the interval, *`latency_sum`* is a sum of latencies (so you can compute average latency easily). The following two fields are useful for variance estimation - *`latency_sum`* is a sum of latencies and *`latency_2_sum`* is a sum of 2nd powers of latencies. The next two fields are *`min_latency`* - a minimum latency within the interval, and *`max_latency`* - maximum latency within the interval. A transaction is counted into the interval when it was committed. The fields in the end, *`lag_sum`*, *`lag_2_sum`*, *`min_lag`*, and *`max_lag`*, are only present if the `--rate` option is used. The very last one, *`skipped_transactions`*, is only present if the option `--latency-limit` is present, too. They are calculated from the time each transaction had to wait for the previous one to finish, i.e., the difference between each transaction's scheduled start time and the time it actually started.

Here is example output:

```
1345828501 5601 1542744 483552416 61 2573
1345828503 7884 1979812 565806736 60 1479
1345828505 7208 1979422 567277552 59 1391
1345828507 7685 1980268 569784714 60 1398
1345828509 7073 1979779 573489941 236 1411
```

Notice that while the plain (unaggregated) log file contains a reference to the custom script files, the aggregated log does not. Therefore if you need per script data, you need to aggregate the data on your own.

Per-Statement Latencies

With the `-r` option, pgbench collects the elapsed transaction time of each statement executed by every client. It then reports an average of those values, referred to as the latency for each statement, after the benchmark has finished.

For the default script, the output will look similar to this:

```
starting vacuum...end.
transaction type: TPC-B (sort of)
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 1
number of transactions per client: 1000
number of transactions actually processed: 10000/10000
tps = 618.764555 (including connections establishing)
tps = 622.977698 (excluding connections establishing)
statement latencies in milliseconds:
        0.004386        \set nbranches 1 * :scale
        0.001343        \set ntellers 10 * :scale
```

```
0.001212        \set naccounts 100000 * :scale
0.001310        \setrandom aid 1 :naccounts
0.001073        \setrandom bid 1 :nbranches
0.001005        \setrandom tid 1 :ntellers
0.001078        \setrandom delta -5000 5000
0.326152        BEGIN;
0.603376        UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid =
0.454643        SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
5.528491        UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid =
7.335435        UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid =
0.371851        INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:
1.212976        END;
```

If multiple script files are specified, the averages are reported separately for each script file.

Note that collecting the additional timing information needed for per-statement latency computation adds some overhead. This will slow average execution speed and lower the computed TPS. The amount of slowdown varies significantly depending on platform and hardware. Comparing average TPS values with and without latency reporting enabled is a good way to measure if the timing overhead is significant.

Good Practices

It is very easy to use pgbench to produce completely meaningless numbers. Here are some guidelines to help you get useful results.

In the first place, *never* believe any test that runs for only a few seconds. Use the `-t` or `-T` option to make the run last at least a few minutes, so as to average out noise. In some cases you could need hours to get numbers that are reproducible. It's a good idea to try the test run a few times, to find out if your numbers are reproducible or not.

For the default TPC-B-like test scenario, the initialization scale factor (`-s`) should be at least as large as the largest number of clients you intend to test (`-c`); else you'll mostly be measuring update contention. There are only `-s` rows in the `pgbench_branches` table, and every transaction wants to update one of them, so `-c` values in excess of `-s` will undoubtedly result in lots of transactions blocked waiting for other transactions.

The default test scenario is also quite sensitive to how long it's been since the tables were initialized: accumulation of dead rows and dead space in the tables changes the results. To understand the results you must keep track of the total number of updates and when vacuuming happens. If autovacuum is enabled it can result in unpredictable changes in measured performance.

A limitation of pgbench is that it can itself become the bottleneck when trying to test a large number of client sessions. This can be alleviated by running pgbench on a different machine from the database server, although low network latency will be essential. It might even be useful to run several pgbench instances concurrently, on several client machines, against the same database server.

Security

If untrusted users have access to a database that has not adopted a secure schema usage pattern, do not run pgbench in that database. pgbench uses unqualified names and does not manipulate the search path.

pg_config

Name

`pg_config` — retrieve information about the installed version of PostgreSQL

Synopsis

`pg_config` [*option*...]

Description

The pg_config utility prints configuration parameters of the currently installed version of PostgreSQL. It is intended, for example, to be used by software packages that want to interface to PostgreSQL to facilitate finding the required header files and libraries.

Options

To use pg_config, supply one or more of the following options:

`--bindir`

Print the location of user executables. Use this, for example, to find the `psql` program. This is normally also the location where the `pg_config` program resides.

`--docdir`

Print the location of documentation files.

`--htmldir`

Print the location of HTML documentation files.

`--includedir`

Print the location of C header files of the client interfaces.

`--pkgincludedir`

Print the location of other C header files.

`--includedir-server`

Print the location of C header files for server programming.

`--libdir`

Print the location of object code libraries.

`--pkglibdir`

Print the location of dynamically loadable modules, or where the server would search for them. (Other architecture-dependent data files might also be installed in this directory.)

`--localedir`

Print the location of locale support files. (This will be an empty string if locale support was not configured when PostgreSQL was built.)

`--mandir`

Print the location of manual pages.

`--sharedir`

Print the location of architecture-independent support files.

`--sysconfdir`

Print the location of system-wide configuration files.

`--pgxs`

Print the location of extension makefiles.

`--configure`

Print the options that were given to the `configure` script when PostgreSQL was configured for building. This can be used to reproduce the identical configuration, or to find out with what options a binary package was built. (Note however that binary packages often contain vendor-specific custom patches.) See also the examples below.

`--cc`

Print the value of the `CC` variable that was used for building PostgreSQL. This shows the C compiler used.

`--cppflags`

Print the value of the `CPPFLAGS` variable that was used for building PostgreSQL. This shows C compiler switches needed at preprocessing time (typically, `-I` switches).

`--cflags`

Print the value of the `CFLAGS` variable that was used for building PostgreSQL. This shows C compiler switches.

`--cflags_sl`

Print the value of the `CFLAGS_SL` variable that was used for building PostgreSQL. This shows extra C compiler switches used for building shared libraries.

`--ldflags`

Print the value of the `LDFLAGS` variable that was used for building PostgreSQL. This shows linker switches.

`--ldflags_ex`

Print the value of the `LDFLAGS_EX` variable that was used for building PostgreSQL. This shows linker switches used for building executables only.

`--ldflags_sl`

Print the value of the `LDFLAGS_SL` variable that was used for building PostgreSQL. This shows linker switches used for building shared libraries only.

`--libs`

Print the value of the `LIBS` variable that was used for building PostgreSQL. This normally contains `-l` switches for external libraries linked into PostgreSQL.

`--version`

Print the version of PostgreSQL.

`-?`
`--help`

Show help about pg_config command line arguments, and exit.

If more than one option is given, the information is printed in that order, one item per line. If no options are given, all available information is printed, with labels.

Notes

The options `--docdir`, `--pkgincludedir`, `--localedir`, `--mandir`, `--sharedir`, `--sysconfdir`, `--cc`, `--cppflags`, `--cflags`, `--cflags_sl`, `--ldflags`, `--ldflags_sl`, and `--libs` were added in PostgreSQL 8.1. The option `--htmldir` was added in PostgreSQL 8.4. The option `--ldflags_ex` was added in PostgreSQL 9.0.

Example

To reproduce the build configuration of the current PostgreSQL installation, run the following command:

```
eval ./configure `pg_config --configure`
```

The output of `pg_config --configure` contains shell quotation marks so arguments with spaces are represented correctly. Therefore, using `eval` is required for proper results.

pg_dump

Name

pg_dump — extract a PostgreSQL database into a script file or other archive file

Synopsis

pg_dump [*connection-option*...] [*option*...] [*dbname*]

Description

pg_dump is a utility for backing up a PostgreSQL database. It makes consistent backups even if the database is being used concurrently. pg_dump does not block other users accessing the database (readers or writers).

pg_dump only dumps a single database. To backup global objects that are common to all databases in a cluster, such as roles and tablespaces, use pg_dumpall.

Dumps can be output in script or archive file formats. Script dumps are plain-text files containing the SQL commands required to reconstruct the database to the state it was in at the time it was saved. To restore from such a script, feed it to psql. Script files can be used to reconstruct the database even on other machines and other architectures; with some modifications, even on other SQL database products.

The alternative archive file formats must be used with pg_restore to rebuild the database. They allow pg_restore to be selective about what is restored, or even to reorder the items prior to being restored. The archive file formats are designed to be portable across architectures.

When used with one of the archive file formats and combined with pg_restore, pg_dump provides a flexible archival and transfer mechanism. pg_dump can be used to backup an entire database, then pg_restore can be used to examine the archive and/or select which parts of the database are to be restored. The most flexible output file formats are the "custom" format (`-Fc`) and the "directory" format (`-Fd`). They allow for selection and reordering of all archived items, support parallel restoration, and are compressed by default. The "directory" format is the only format that supports parallel dumps.

While running pg_dump, one should examine the output for any warnings (printed on standard error), especially in light of the limitations listed below.

Options

The following command-line options control the content and format of the output.

dbname

Specifies the name of the database to be dumped. If this is not specified, the environment variable PGDATABASE is used. If that is not set, the user name specified for the connection is used.

`-a`
`--data-only`

Dump only the data, not the schema (data definitions). Table data, large objects, and sequence values are dumped.

This option is similar to, but for historical reasons not identical to, specifying `--section=data`.

`-b`
`--blobs`

Include large objects in the dump. This is the default behavior except when `--schema`, `--table`, or `--schema-only` is specified. The `-b` switch is therefore only useful to add large objects to dumps where a specific schema or table has been requested. Note that blobs are considered data and therefore will be included when --data-only is used, but not when --schema-only is.

`-c`
`--clean`

Output commands to clean (drop) database objects prior to outputting the commands for creating them. (Unless `--if-exists` is also specified, restore might generate some harmless error messages, if any objects were not present in the destination database.)

This option is only meaningful for the plain-text format. For the archive formats, you can specify the option when you call `pg_restore`.

`-C`
`--create`

Begin the output with a command to create the database itself and reconnect to the created database. (With a script of this form, it doesn't matter which database in the destination installation you connect to before running the script.) If `--clean` is also specified, the script drops and recreates the target database before reconnecting to it.

This option is only meaningful for the plain-text format. For the archive formats, you can specify the option when you call `pg_restore`.

`-E` *`encoding`*
`--encoding=`*`encoding`*

Create the dump in the specified character set encoding. By default, the dump is created in the database encoding. (Another way to get the same result is to set the `PGCLIENTENCODING` environment variable to the desired dump encoding.)

`-f` *`file`*
`--file=`*`file`*

Send output to the specified file. This parameter can be omitted for file based output formats, in which case the standard output is used. It must be given for the directory output format however, where it specifies the target directory instead of a file. In this case the directory is created by `pg_dump` and must not exist before.

`-F` *format*
`--format=`*format*

Selects the format of the output. *format* can be one of the following:

`p`
`plain`

Output a plain-text SQL script file (the default).

`c`
`custom`

Output a custom-format archive suitable for input into pg_restore. Together with the directory output format, this is the most flexible output format in that it allows manual selection and reordering of archived items during restore. This format is also compressed by default.

`d`
`directory`

Output a directory-format archive suitable for input into pg_restore. This will create a directory with one file for each table and blob being dumped, plus a so-called Table of Contents file describing the dumped objects in a machine-readable format that pg_restore can read. A directory format archive can be manipulated with standard Unix tools; for example, files in an uncompressed archive can be compressed with the gzip tool. This format is compressed by default and also supports parallel dumps.

`t`
`tar`

Output a `tar`-format archive suitable for input into pg_restore. The tar format is compatible with the directory format: extracting a tar-format archive produces a valid directory-format archive. However, the tar format does not support compression. Also, when using tar format the relative order of table data items cannot be changed during restore.

`-j` *njobs*
`--jobs=`*njobs*

Run the dump in parallel by dumping *njobs* tables simultaneously. This option reduces the time of the dump but it also increases the load on the database server. You can only use this option with the directory output format because this is the only output format where multiple processes can write their data at the same time.

pg_dump will open *njobs* + 1 connections to the database, so make sure your max_connections setting is high enough to accommodate all connections.

Requesting exclusive locks on database objects while running a parallel dump could cause the dump to fail. The reason is that the pg_dump master process requests shared locks on the objects that the worker processes are going to dump later in order to make sure that nobody deletes them and makes them go away while the dump is running. If another client then requests an exclusive lock on a table, that lock will not be granted but will be queued waiting for the shared lock of the master process to be released. Consequently any other access to the table will not be granted either and will queue after the exclusive lock request. This includes the worker process trying to dump the table. Without any precautions this would be a classic deadlock situation. To detect this conflict, the pg_dump worker

process requests another shared lock using the `NOWAIT` option. If the worker process is not granted this shared lock, somebody else must have requested an exclusive lock in the meantime and there is no way to continue with the dump, so pg_dump has no choice but to abort the dump.

For a consistent backup, the database server needs to support synchronized snapshots, a feature that was introduced in PostgreSQL 9.2. With this feature, database clients can ensure they see the same data set even though they use different connections. `pg_dump -j` uses multiple database connections; it connects to the database once with the master process and once again for each worker job. Without the synchronized snapshot feature, the different worker jobs wouldn't be guaranteed to see the same data in each connection, which could lead to an inconsistent backup.

If you want to run a parallel dump of a pre-9.2 server, you need to make sure that the database content doesn't change from between the time the master connects to the database until the last worker job has connected to the database. The easiest way to do this is to halt any data modifying processes (DDL and DML) accessing the database before starting the backup. You also need to specify the `--no-synchronized-snapshots` parameter when running `pg_dump -j` against a pre-9.2 PostgreSQL server.

`-n` *`schema`*
`--schema=`*`schema`*

Dump only schemas matching *`schema`*; this selects both the schema itself, and all its contained objects. When this option is not specified, all non-system schemas in the target database will be dumped. Multiple schemas can be selected by writing multiple `-n` switches. Also, the *`schema`* parameter is interpreted as a pattern according to the same rules used by psql's `\d` commands (see *Patterns*), so multiple schemas can also be selected by writing wildcard characters in the pattern. When using wildcards, be careful to quote the pattern if needed to prevent the shell from expanding the wildcards; see *Examples*.

> **Note:** When `-n` is specified, pg_dump makes no attempt to dump any other database objects that the selected schema(s) might depend upon. Therefore, there is no guarantee that the results of a specific-schema dump can be successfully restored by themselves into a clean database.

> **Note:** Non-schema objects such as blobs are not dumped when `-n` is specified. You can add blobs back to the dump with the `--blobs` switch.

`-N` *`schema`*
`--exclude-schema=`*`schema`*

Do not dump any schemas matching the *`schema`* pattern. The pattern is interpreted according to the same rules as for `-n`. `-N` can be given more than once to exclude schemas matching any of several patterns.

When both `-n` and `-N` are given, the behavior is to dump just the schemas that match at least one `-n` switch but no `-N` switches. If `-N` appears without `-n`, then schemas matching `-N` are excluded from what is otherwise a normal dump.

`-o`
`--oids`

Dump object identifiers (OIDs) as part of the data for every table. Use this option if your application references the OID columns in some way (e.g., in a foreign key constraint). Otherwise, this option should not be used.

`-O`
`--no-owner`

Do not output commands to set ownership of objects to match the original database. By default, pg_dump issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created database objects. These statements will fail when the script is run unless it is started by a superuser (or the same user that owns all of the objects in the script). To make a script that can be restored by any user, but will give that user ownership of all the objects, specify `-O`.

This option is only meaningful for the plain-text format. For the archive formats, you can specify the option when you call `pg_restore`.

`-R`
`--no-reconnect`

This option is obsolete but still accepted for backwards compatibility.

`-s`
`--schema-only`

Dump only the object definitions (schema), not data.

This option is the inverse of `--data-only`. It is similar to, but for historical reasons not identical to, specifying `--section=pre-data --section=post-data`.

(Do not confuse this with the `--schema` option, which uses the word "schema" in a different meaning.)

To exclude table data for only a subset of tables in the database, see `--exclude-table-data`.

`-S` *username*
`--superuser=`*username*

Specify the superuser user name to use when disabling triggers. This is relevant only if `--disable-triggers` is used. (Usually, it's better to leave this out, and instead start the resulting script as superuser.)

`-t` *table*
`--table=`*table*

Dump only tables (or views or sequences or foreign tables) matching *table*. Multiple tables can be selected by writing multiple `-t` switches. Also, the *table* parameter is interpreted as a pattern according to the same rules used by psql's `\d` commands (see *Patterns*), so multiple tables can also be selected by writing wildcard characters in the pattern. When using wildcards, be careful to quote the pattern if needed to prevent the shell from expanding the wildcards; see *Examples*.

The `-n` and `-N` switches have no effect when `-t` is used, because tables selected by `-t` will be dumped regardless of those switches, and non-table objects will not be dumped.

Note: When `-t` is specified, pg_dump makes no attempt to dump any other database objects that the selected table(s) might depend upon. Therefore, there is no guarantee that the results of a specific-table dump can be successfully restored by themselves into a clean database.

Note: The behavior of the `-t` switch is not entirely upward compatible with pre-8.2 PostgreSQL versions. Formerly, writing `-t tab` would dump all tables named `tab`, but now it just dumps whichever one is visible in your default search path. To get the old behavior you can write `-t 'tab'`. Also, you must write something like `-t sch.tab` to select a table in a particular schema, rather than the old locution of `-n sch -t tab`.

`-T` *table*
`--exclude-table=`*table*

Do not dump any tables matching the *table* pattern. The pattern is interpreted according to the same rules as for `-t`. `-T` can be given more than once to exclude tables matching any of several patterns.

When both `-t` and `-T` are given, the behavior is to dump just the tables that match at least one `-t` switch but no `-T` switches. If `-T` appears without `-t`, then tables matching `-T` are excluded from what is otherwise a normal dump.

`-v`
`--verbose`

Specifies verbose mode. This will cause pg_dump to output detailed object comments and start/stop times to the dump file, and progress messages to standard error.

`-V`
`--version`

Print the pg_dump version and exit.

`-x`
`--no-privileges`
`--no-acl`

Prevent dumping of access privileges (grant/revoke commands).

`-Z` *0..9*
`--compress=`*0..9*

Specify the compression level to use. Zero means no compression. For the custom archive format, this specifies compression of individual table-data segments, and the default is to compress at a moderate level. For plain text output, setting a nonzero compression level causes the entire output file to be compressed, as though it had been fed through gzip; but the default is not to compress. The tar archive format currently does not support compression at all.

`--binary-upgrade`

This option is for use by in-place upgrade utilities. Its use for other purposes is not recommended or supported. The behavior of the option may change in future releases without notice.

`--column-inserts`
`--attribute-inserts`

Dump data as `INSERT` commands with explicit column names (`INSERT INTO` *`table`* `(`*`column`*`, ...) VALUES ...`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL databases. However, since this option generates a separate command for each row, an error in reloading a row causes only that row to be lost rather than the entire table contents.

`--disable-dollar-quoting`

This option disables the use of dollar quoting for function bodies, and forces them to be quoted using SQL standard string syntax.

`--disable-triggers`

This option is relevant only when creating a data-only dump. It instructs pg_dump to include commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have referential integrity checks or other triggers on the tables that you do not want to invoke during data reload.

Presently, the commands emitted for `--disable-triggers` must be done as superuser. So, you should also specify a superuser name with `-S`, or preferably be careful to start the resulting script as a superuser.

This option is only meaningful for the plain-text format. For the archive formats, you can specify the option when you call `pg_restore`.

`--enable-row-security`

This option is relevant only when dumping the contents of a table which has row security. By default, pg_dump will set row_security to off, to ensure that all data is dumped from the table. If the user does not have sufficient privileges to bypass row security, then an error is thrown. This parameter instructs pg_dump to set row_security to on instead, allowing the user to dump the parts of the contents of the table that they have access to.

Note that if you use this option currently, you probably also want the dump be in `INSERT` format, as the `COPY FROM` during restore does not support row security.

`--exclude-table-data=`*`table`*

Do not dump data for any tables matching the *`table`* pattern. The pattern is interpreted according to the same rules as for `-t`. `--exclude-table-data` can be given more than once to exclude tables matching any of several patterns. This option is useful when you need the definition of a particular table even though you do not need the data in it.

To exclude data for all tables in the database, see `--schema-only`.

`--if-exists`

Use conditional commands (i.e., add an `IF EXISTS` clause) when cleaning database objects. This option is not valid unless `--clean` is also specified.

`--inserts`

Dump data as `INSERT` commands (rather than `COPY`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL databases. However, since this option generates a separate command for each row, an error in reloading a row causes only

that row to be lost rather than the entire table contents. Note that the restore might fail altogether if you have rearranged column order. The `--column-inserts` option is safe against column order changes, though even slower.

`--lock-wait-timeout=`*`timeout`*

Do not wait forever to acquire shared table locks at the beginning of the dump. Instead fail if unable to lock a table within the specified *`timeout`*. The timeout may be specified in any of the formats accepted by `SET statement_timeout`. (Allowed values vary depending on the server version you are dumping from, but an integer number of milliseconds is accepted by all versions since 7.3. This option is ignored when dumping from a pre-7.3 server.)

`--no-security-labels`

Do not dump security labels.

`--no-synchronized-snapshots`

This option allows running `pg_dump -j` against a pre-9.2 server, see the documentation of the `-j` parameter for more details.

`--no-tablespaces`

Do not output commands to select tablespaces. With this option, all objects will be created in whichever tablespace is the default during restore.

This option is only meaningful for the plain-text format. For the archive formats, you can specify the option when you call `pg_restore`.

`--no-unlogged-table-data`

Do not dump the contents of unlogged tables. This option has no effect on whether or not the table definitions (schema) are dumped; it only suppresses dumping the table data. Data in unlogged tables is always excluded when dumping from a standby server.

`--quote-all-identifiers`

Force quoting of all identifiers. This option is recommended when dumping a database from a server whose PostgreSQL major version is different from pg_dump's, or when the output is intended to be loaded into a server of a different major version. By default, pg_dump quotes only identifiers that are reserved words in its own major version. This sometimes results in compatibility issues when dealing with servers of other versions that may have slightly different sets of reserved words. Using `--quote-all-identifiers` prevents such issues, at the price of a harder-to-read dump script.

`--section=`*`sectionname`*

Only dump the named section. The section name can be `pre-data`, `data`, or `post-data`. This option can be specified more than once to select multiple sections. The default is to dump all sections.

The data section contains actual table data, large-object contents, and sequence values. Post-data items include definitions of indexes, triggers, rules, and constraints other than validated check constraints. Pre-data items include all other data definition items.

`--serializable-deferrable`

Use a `serializable` transaction for the dump, to ensure that the snapshot used is consistent with later database states; but do this by waiting for a point in the transaction stream at which no anomalies can be present, so that there isn't a risk of the dump failing or causing other transactions to roll back

with a `serialization_failure`. See Chapter 13 for more information about transaction isolation and concurrency control.

This option is not beneficial for a dump which is intended only for disaster recovery. It could be useful for a dump used to load a copy of the database for reporting or other read-only load sharing while the original database continues to be updated. Without it the dump may reflect a state which is not consistent with any serial execution of the transactions eventually committed. For example, if batch processing techniques are used, a batch may show as closed in the dump without all of the items which are in the batch appearing.

This option will make no difference if there are no read-write transactions active when pg_dump is started. If read-write transactions are active, the start of the dump may be delayed for an indeterminate length of time. Once running, performance with or without the switch is the same.

`--snapshot=`*`snapshotname`*

Use the specified synchronized snapshot when making a dump of the database (see Table 9-71 for more details).

This option is useful when needing to synchronize the dump with a logical replication slot (see Chapter 46) or with a concurrent session.

In the case of a parallel dump, the snapshot name defined by this option is used rather than taking a new snapshot.

`--use-set-session-authorization`

Output SQL-standard `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to determine object ownership. This makes the dump more standards-compatible, but depending on the history of the objects in the dump, might not restore properly. Also, a dump using `SET SESSION AUTHORIZATION` will certainly require superuser privileges to restore correctly, whereas `ALTER OWNER` requires lesser privileges.

`-?`
`--help`

Show help about pg_dump command line arguments, and exit.

The following command-line options control the database connection parameters.

`-d` *`dbname`*
`--dbname=`*`dbname`*

Specifies the name of the database to connect to. This is equivalent to specifying *`dbname`* as the first non-option argument on the command line. The *`dbname`* can be a connection string. If so, connection string parameters will override any conflicting command line options.

`-h` *`host`*
`--host=`*`host`*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the `PGHOST` environment variable, if set, else a Unix domain socket connection is attempted.

`-p` *port*
`--port=`*port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the `PGPORT` environment variable, if set, or a compiled-in default.

`-U` *username*
`--username=`*username*

User name to connect as.

`-w`
`--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W`
`--password`

Force pg_dump to prompt for a password before connecting to a database.

This option is never essential, since pg_dump will automatically prompt for a password if the server demands password authentication. However, pg_dump will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

`--role=`*rolename*

Specifies a role name to be used to create the dump. This option causes pg_dump to issue a `SET ROLE` *rolename* command after connecting to the database. It is useful when the authenticated user (specified by `-U`) lacks privileges needed by pg_dump, but can switch to a role with the required rights. Some installations have a policy against logging in directly as a superuser, and use of this option allows dumps to be made without violating the policy.

Environment

`PGDATABASE`
`PGHOST`
`PGOPTIONS`
`PGPORT`
`PGUSER`

Default connection parameters.

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14).

Diagnostics

pg_dump internally executes `SELECT` statements. If you have problems running pg_dump, make sure you are able to select information from the database using, for example, psql. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

The database activity of pg_dump is normally collected by the statistics collector. If this is undesirable, you can set parameter `track_counts` to false via `PGOPTIONS` or the `ALTER USER` command.

Notes

If your database cluster has any local additions to the `template1` database, be careful to restore the output of pg_dump into a truly empty database; otherwise you are likely to get errors due to duplicate definitions of the added objects. To make an empty database without any local additions, copy from `template0` not `template1`, for example:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

When a data-only dump is chosen and the option `--disable-triggers` is used, pg_dump emits commands to disable triggers on user tables before inserting the data, and then commands to re-enable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs might be left in the wrong state.

The dump file produced by pg_dump does not contain the statistics used by the optimizer to make query planning decisions. Therefore, it is wise to run `ANALYZE` after restoring from a dump file to ensure optimal performance; see Section 23.1.3 and Section 23.1.6 for more information. The dump file also does not contain any `ALTER DATABASE ... SET` commands; these settings are dumped by pg_dumpall, along with database users and other installation-wide settings.

Because pg_dump is used to transfer data to newer versions of PostgreSQL, the output of pg_dump can be expected to load into PostgreSQL server versions newer than pg_dump's version. pg_dump can also dump from PostgreSQL servers older than its own version. (Currently, servers back to version 7.0 are supported.) However, pg_dump cannot dump from PostgreSQL servers newer than its own major version; it will refuse to even try, rather than risk making an invalid dump. Also, it is not guaranteed that pg_dump's output can be loaded into a server of an older major version — not even if the dump was taken from a server of that version. Loading a dump file into an older server may require manual editing of the dump file to remove syntax not understood by the older server. Use of the `--quote-all-identifiers` option is recommended in cross-version cases, as it can prevent problems arising from varying reserved-word lists in different PostgreSQL versions.

Examples

To dump a database called `mydb` into a SQL-script file:

```
$ pg_dump mydb > db.sql
```

To reload such a script into a (freshly created) database named `newdb`:

```
$ psql -d newdb -f db.sql
```

To dump a database into a custom-format archive file:

```
$ pg_dump -Fc mydb > db.dump
```

To dump a database into a directory-format archive:

```
$ pg_dump -Fd mydb -f dumpdir
```

To dump a database into a directory-format archive in parallel with 5 worker jobs:

```
$ pg_dump -Fd mydb -j 5 -f dumpdir
```

To reload an archive file into a (freshly created) database named `newdb`:

```
$ pg_restore -d newdb db.dump
```

To dump a single table named `mytab`:

```
$ pg_dump -t mytab mydb > db.sql
```

To dump all tables whose names start with `emp` in the `detroit` schema, except for the table named `employee_log`:

```
$ pg_dump -t 'detroit.emp*' -T detroit.employee_log mydb > db.sql
```

To dump all schemas whose names start with `east` or `west` and end in `gsm`, excluding any schemas whose names contain the word `test`:

```
$ pg_dump -n 'east*gsm' -n 'west*gsm' -N '*test*' mydb > db.sql
```

The same, using regular expression notation to consolidate the switches:

```
$ pg_dump -n '(east|west)*gsm' -N '*test*' mydb > db.sql
```

To dump all database objects except for tables whose names begin with `ts_`:

```
$ pg_dump -T 'ts_*' mydb > db.sql
```

To specify an upper-case or mixed-case name in `-t` and related switches, you need to double-quote the name; else it will be folded to lower case (see *Patterns*). But double quotes are special to the shell, so in turn they must be quoted. Thus, to dump a single table with a mixed-case name, you need something like

```
$ pg_dump -t "\"MixedCaseName\"" mydb > mytab.sql
```

See Also

pg_dumpall, pg_restore, psql

pg_dumpall

Name

pg_dumpall — extract a PostgreSQL database cluster into a script file

Synopsis

pg_dumpall [*connection-option*...] [*option*...]

Description

pg_dumpall is a utility for writing out ("dumping") all PostgreSQL databases of a cluster into one script file. The script file contains SQL commands that can be used as input to psql to restore the databases. It does this by calling pg_dump for each database in a cluster. pg_dumpall also dumps global objects that are common to all databases. (pg_dump does not save these objects.) This currently includes information about database users and groups, tablespaces, and properties such as access permissions that apply to databases as a whole.

Since pg_dumpall reads tables from all databases you will most likely have to connect as a database superuser in order to produce a complete dump. Also you will need superuser privileges to execute the saved script in order to be allowed to add users and groups, and to create databases.

The SQL script will be written to the standard output. Use the -f/--file option or shell operators to redirect it into a file.

pg_dumpall needs to connect several times to the PostgreSQL server (once per database). If you use password authentication it will ask for a password each time. It is convenient to have a ~/.pgpass file in such cases. See Section 31.15 for more information.

Options

The following command-line options control the content and format of the output.

-a
--data-only

Dump only the data, not the schema (data definitions).

-c
--clean

Include SQL commands to clean (drop) databases before recreating them. DROP commands for roles and tablespaces are added as well.

`-f` *filename*
`--file=`*filename*

Send output to the specified file. If this is omitted, the standard output is used.

`-g`
`--globals-only`

Dump only global objects (roles and tablespaces), no databases.

`-o`
`--oids`

Dump object identifiers (OIDs) as part of the data for every table. Use this option if your application references the OID columns in some way (e.g., in a foreign key constraint). Otherwise, this option should not be used.

`-O`
`--no-owner`

Do not output commands to set ownership of objects to match the original database. By default, pg_dumpall issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created schema elements. These statements will fail when the script is run unless it is started by a superuser (or the same user that owns all of the objects in the script). To make a script that can be restored by any user, but will give that user ownership of all the objects, specify `-O`.

`-r`
`--roles-only`

Dump only roles, no databases or tablespaces.

`-s`
`--schema-only`

Dump only the object definitions (schema), not data.

`-S` *username*
`--superuser=`*username*

Specify the superuser user name to use when disabling triggers. This is relevant only if `--disable-triggers` is used. (Usually, it's better to leave this out, and instead start the resulting script as superuser.)

`-t`
`--tablespaces-only`

Dump only tablespaces, no databases or roles.

`-v`
`--verbose`

Specifies verbose mode. This will cause pg_dumpall to output start/stop times to the dump file, and progress messages to standard error. It will also enable verbose output in pg_dump.

`-V`
`--version`

Print the pg_dumpall version and exit.

`-x`
`--no-privileges`
`--no-acl`

Prevent dumping of access privileges (grant/revoke commands).

`--binary-upgrade`

This option is for use by in-place upgrade utilities. Its use for other purposes is not recommended or supported. The behavior of the option may change in future releases without notice.

`--column-inserts`
`--attribute-inserts`

Dump data as `INSERT` commands with explicit column names (`INSERT INTO` *`table`* `(`*`column`*`, ...) VALUES ...`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL databases.

`--disable-dollar-quoting`

This option disables the use of dollar quoting for function bodies, and forces them to be quoted using SQL standard string syntax.

`--disable-triggers`

This option is relevant only when creating a data-only dump. It instructs pg_dumpall to include commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have referential integrity checks or other triggers on the tables that you do not want to invoke during data reload.

Presently, the commands emitted for `--disable-triggers` must be done as superuser. So, you should also specify a superuser name with `-S`, or preferably be careful to start the resulting script as a superuser.

`--if-exists`

Use conditional commands (i.e., add an `IF EXISTS` clause) to clean databases and other objects. This option is not valid unless `--clean` is also specified.

`--inserts`

Dump data as `INSERT` commands (rather than `COPY`). This will make restoration very slow; it is mainly useful for making dumps that can be loaded into non-PostgreSQL databases. Note that the restore might fail altogether if you have rearranged column order. The `--column-inserts` option is safer, though even slower.

`--lock-wait-timeout=`*`timeout`*

Do not wait forever to acquire shared table locks at the beginning of the dump. Instead, fail if unable to lock a table within the specified *`timeout`*. The timeout may be specified in any of the formats accepted by `SET statement_timeout`. Allowed values vary depending on the server version you are dumping from, but an integer number of milliseconds is accepted by all versions since 7.3. This option is ignored when dumping from a pre-7.3 server.

`--no-security-labels`

Do not dump security labels.

`--no-tablespaces`

Do not output commands to create tablespaces nor select tablespaces for objects. With this option, all objects will be created in whichever tablespace is the default during restore.

`--no-unlogged-table-data`

Do not dump the contents of unlogged tables. This option has no effect on whether or not the table definitions (schema) are dumped; it only suppresses dumping the table data.

`--quote-all-identifiers`

Force quoting of all identifiers. This option is recommended when dumping a database from a server whose PostgreSQL major version is different from pg_dumpall's, or when the output is intended to be loaded into a server of a different major version. By default, pg_dumpall quotes only identifiers that are reserved words in its own major version. This sometimes results in compatibility issues when dealing with servers of other versions that may have slightly different sets of reserved words. Using `--quote-all-identifiers` prevents such issues, at the price of a harder-to-read dump script.

`--use-set-session-authorization`

Output SQL-standard `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to determine object ownership. This makes the dump more standards compatible, but depending on the history of the objects in the dump, might not restore properly.

`-?`
`--help`

Show help about pg_dumpall command line arguments, and exit.

The following command-line options control the database connection parameters.

`-d` *`connstr`*
`--dbname=`*`connstr`*

Specifies parameters used to connect to the server, as a connction string; these will override any conflicting command line options.

The option is called `--dbname` for consistency with other client applications, but because pg_dumpall needs to connect to many databases, database name in the connection string will be ignored. Use `-l` option to specify the name of the database used to dump global objects and to discover what other databases should be dumped.

`-h` *`host`*
`--host=`*`host`*

Specifies the host name of the machine on which the database server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the `PGHOST` environment variable, if set, else a Unix domain socket connection is attempted.

`-l` *`dbname`*
`--database=`*`dbname`*

Specifies the name of the database to connect to for dumping global objects and discovering what other databases should be dumped. If not specified, the `postgres` database will be used, and if that does not exist, `template1` will be used.

`-p` *port*
`--port=`*port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the `PGPORT` environment variable, if set, or a compiled-in default.

`-U` *username*
`--username=`*username*

User name to connect as.

`-w`
`--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W`
`--password`

Force pg_dumpall to prompt for a password before connecting to a database.

This option is never essential, since pg_dumpall will automatically prompt for a password if the server demands password authentication. However, pg_dumpall will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

Note that the password prompt will occur again for each database to be dumped. Usually, it's better to set up a `~/.pgpass` file than to rely on manual password entry.

`--role=`*rolename*

Specifies a role name to be used to create the dump. This option causes pg_dumpall to issue a `SET ROLE` *rolename* command after connecting to the database. It is useful when the authenticated user (specified by `-U`) lacks privileges needed by pg_dumpall, but can switch to a role with the required rights. Some installations have a policy against logging in directly as a superuser, and use of this option allows dumps to be made without violating the policy.

Environment

`PGHOST`
`PGOPTIONS`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14).

Notes

Since pg_dumpall calls pg_dump internally, some diagnostic messages will refer to pg_dump.

Once restored, it is wise to run `ANALYZE` on each database so the optimizer has useful statistics. You can also run `vacuumdb -a -z` to analyze all databases.

pg_dumpall requires all needed tablespace directories to exist before the restore; otherwise, database creation will fail for databases in non-default locations.

Examples

To dump all databases:

```
$ pg_dumpall > db.out
```

To reload database(s) from this file, you can use:

```
$ psql -f db.out postgres
```

(It is not important to which database you connect here since the script file created by pg_dumpall will contain the appropriate commands to create and connect to the saved databases.)

See Also

Check pg_dump for details on possible error conditions.

pg_isready

Name

pg_isready — check the connection status of a PostgreSQL server

Synopsis

pg_isready [*connection-option*...] [*option*...]

Description

pg_isready is a utility for checking the connection status of a PostgreSQL database server. The exit status specifies the result of the connection check.

Options

-d *dbname*
--dbname=*dbname*

Specifies the name of the database to connect to. The *dbname* can be a connection string. If so, connection string parameters will override any conflicting command line options.

-h *hostname*
--host=*hostname*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix-domain socket.

-p *port*
--port=*port*

Specifies the TCP port or the local Unix-domain socket file extension on which the server is listening for connections. Defaults to the value of the PGPORT environment variable or, if not set, to the port specified at compile time, usually 5432.

-q
--quiet

Do not display status message. This is useful when scripting.

-t *seconds*
--timeout=*seconds*

The maximum number of seconds to wait when attempting connection before returning that the server is not responding. Setting to 0 disables. The default is 3 seconds.

`-U` *username*
`--username=`*username*

Connect to the database as the user *username* instead of the default.

`-V`
`--version`

Print the pg_isready version and exit.

`-?`
`--help`

Show help about pg_isready command line arguments, and exit.

Exit Status

pg_isready returns `0` to the shell if the server is accepting connections normally, `1` if the server is rejecting connections (for example during startup), `2` if there was no response to the connection attempt, and `3` if no attempt was made (for example due to invalid parameters).

Environment

`pg_isready`, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14).

Notes

It is not necessary to supply correct user name, password, or database name values to obtain the server status; however, if incorrect values are provided, the server will log a failed connection attempt.

Examples

Standard Usage:

```
$ pg_isready
/tmp:5432 - accepting connections
$ echo $?
0
```

Running with connection parameters to a PostgreSQL cluster in startup:

```
$ pg_isready -h localhost -p 5433
localhost:5433 - rejecting connections
$ echo $?
1
```

Running with connection parameters to a non-responsive PostgreSQL cluster:

```
$ pg_isready -h someremotehost
someremotehost:5432 - no response
$ echo $?
2
```

pg_receivexlog

Name

pg_receivexlog — stream transaction logs from a PostgreSQL server

Synopsis

pg_receivexlog [*option*...]

Description

pg_receivexlog is used to stream the transaction log from a running PostgreSQL cluster. The transaction log is streamed using the streaming replication protocol, and is written to a local directory of files. This directory can be used as the archive location for doing a restore using point-in-time recovery (see Section 24.3).

pg_receivexlog streams the transaction log in real time as it's being generated on the server, and does not wait for segments to complete like archive_command does. For this reason, it is not necessary to set archive_timeout when using pg_receivexlog.

Unlike the WAL receiver of a PostgreSQL standby server, pg_receivexlog by default flushes WAL data only when a WAL file is closed. The option --synchronous must be specified to flush WAL data in real time.

The transaction log is streamed over a regular PostgreSQL connection and uses the replication protocol. The connection must be made with a superuser or a user having REPLICATION permissions (see Section 20.2), and pg_hba.conf must permit the replication connection. The server must also be configured with max_wal_senders set high enough to leave at least one session available for the stream.

If the connection is lost, or if it cannot be initially established, with a non-fatal error, pg_receivexlog will retry the connection indefinitely, and reestablish streaming as soon as possible. To avoid this behavior, use the -n parameter.

Options

-D *directory*
--directory=*directory*

Directory to write the output to.

This parameter is required.

--if-not-exists

Do not error out when --create-slot is specified and a slot with the specified name already exists.

`-n`
`--no-loop`

Don't loop on connection errors. Instead, exit right away with an error.

`-s` *interval*
`--status-interval=`*interval*

Specifies the number of seconds between status packets sent back to the server. This allows for easier monitoring of the progress from server. A value of zero disables the periodic status updates completely, although an update will still be sent when requested by the server, to avoid timeout disconnect. The default value is 10 seconds.

`-S` *slotname*
`--slot=`*slotname*

Require pg_receivexlog to use an existing replication slot (see Section 25.2.6). When this option is used, pg_receivexlog will report a flush position to the server, indicating when each segment has been synchronized to disk so that the server can remove that segment if it is not otherwise needed.

When the replication client of pg_receivexlog is configured on the server as a synchronous standby, then using a replication slot will report the flush position to the server, but only when a WAL file is closed. Therefore, that configuration will cause transactions on the primary to wait for a long time and effectively not work satisfactorily. The option `--synchronous` (see below) must be specified in addition to make this work correctly.

`--synchronous`

Flush the WAL data to disk immediately after it has been received. Also send a status packet back to the server immediately after flushing, regardless of `--status-interval`.

This option should be specified if the replication client of pg_receivexlog is configured on the server as a synchronous standby, to ensure that timely feedback is sent to the server.

`-v`
`--verbose`

Enables verbose mode.

The following command-line options control the database connection parameters.

`-d` *connstr*
`--dbname=`*connstr*

Specifies parameters used to connect to the server, as a connection string. See Section 31.1.1 for more information.

The option is called `--dbname` for consistency with other client applications, but because pg_receivexlog doesn't connect to any particular database in the cluster, database name in the connection string will be ignored.

`-h` *host*
`--host=`*host*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the `PGHOST` environment variable, if set, else a Unix domain socket connection is attempted.

`-p` *port*
`--port=`*port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the `PGPORT` environment variable, if set, or a compiled-in default.

`-U` *username*
`--username=`*username*

User name to connect as.

`-w`
`--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W`
`--password`

Force pg_receivexlog to prompt for a password before connecting to a database.

This option is never essential, since pg_receivexlog will automatically prompt for a password if the server demands password authentication. However, pg_receivexlog will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

pg_receivexlog can perform one of the two following actions in order to control physical replication slots:

`--create-slot`

Create a new physical replication slot with the name specified in `--slot`, then exit.

`--drop-slot`

Drop the replication slot with the name specified in `--slot`, then exit.

Other options are also available:

`-V`
`--version`

Print the pg_receivexlog version and exit.

`-?`
`--help`

Show help about pg_receivexlog command line arguments, and exit.

Environment

This utility, like most other PostgreSQL utilities, uses the environment variables supported by libpq (see Section 31.14).

Notes

When using pg_receivexlog instead of archive_command as the main WAL backup method, it is strongly recommended to use replication slots. Otherwise, the server is free to recycle or remove transaction log files before they are backed up, because it does not have any information, either from archive_command or the replication slots, about how far the WAL stream has been archived. Note, however, that a replication slot will fill up the server's disk space if the receiver does not keep up with fetching the WAL data.

Examples

To stream the transaction log from the server at `mydbserver` and store it in the local directory `/usr/local/pgsql/archive`:

```
$ pg_receivexlog -h mydbserver -D /usr/local/pgsql/archive
```

See Also

pg_basebackup

pg_recvlogical

Name

pg_recvlogical — control PostgreSQL logical decoding streams

Synopsis

pg_recvlogical [*option*...]

Description

pg_recvlogical controls logical decoding replication slots and streams data from such replication slots.

It creates a replication-mode connection, so it is subject to the same constraints as pg_receivexlog, plus those for logical replication (see Chapter 46).

Options

At least one of the following options must be specified to select an action:

--create-slot

Create a new logical replication slot with the name specified by --slot, using the output plugin specified by --plugin, for the database specified by --dbname.

--drop-slot

Drop the replication slot with the name specified by --slot, then exit.

--start

Begin streaming changes from the logical replication slot specified by --slot, continuing until terminated by a signal. If the server side change stream ends with a server shutdown or disconnect, retry in a loop unless --no-loop is specified.

The stream format is determined by the output plugin specified when the slot was created.

The connection must be to the same database used to create the slot.

--create-slot and --start can be specified together. --drop-slot cannot be combined with another action.

The following command-line options control the location and format of the output and other replication behavior:

`-f` *filename*
`--file=`*filename*

Write received and decoded transaction data into this file. Use `-` for stdout.

`-F` *interval_seconds*
`--fsync-interval=`*interval_seconds*

Specifies how often pg_recvlogical should issue `fsync()` calls to ensure the output file is safely flushed to disk.

The server will occasionally request the client to perform a flush and report the flush position to the server. This setting is in addition to that, to perform flushes more frequently.

Specifying an interval of `0` disables issuing `fsync()` calls altogether, while still reporting progress to the server. In this case, data could be lost in the event of a crash.

`-I` *lsn*
`--startpos=`*lsn*

In `--start` mode, start replication from the given LSN. For details on the effect of this, see the documentation in Chapter 46 and Section 50.3. Ignored in other modes.

`--if-not-exists`

Do not error out when `--create-slot` is specified and a slot with the specified name already exists.

`-n`
`--no-loop`

When the connection to the server is lost, do not retry in a loop, just exit.

`-o` *name*`[=`*value*`]`
`--option=`*name*`[=`*value*`]`

Pass the option *name* to the output plugin with, if specified, the option value *value*. Which options exist and their effects depends on the used output plugin.

`-P` *plugin*
`--plugin=`*plugin*

When creating a slot, use the specified logical decoding output plugin. See Chapter 46. This option has no effect if the slot already exists.

`-s` *interval_seconds*
`--status-interval=`*interval_seconds*

This option has the same effect as the option of the same name in pg_receivexlog. See the description there.

`-S` *slot_name*
`--slot=`*slot_name*

In `--start` mode, use the existing logical replication slot named *slot_name*. In `--create-slot` mode, create the slot with this name. In `--drop-slot` mode, delete the slot with this name.

`-v`
`--verbose`

Enables verbose mode.

The following command-line options control the database connection parameters.

`-d` *`dbname`*
`--dbname=`*`dbname`*

The database to connect to. See the description of the actions for what this means in detail. The *`dbname`* can be a connection string. If so, connection string parameters will override any conflicting command line options. Defaults to the user name.

`-h` *`hostname-or-ip`*
`--host=`*`hostname-or-ip`*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the `PGHOST` environment variable, if set, else a Unix domain socket connection is attempted.

`-p` *`port`*
`--port=`*`port`*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the `PGPORT` environment variable, if set, or a compiled-in default.

`-U` *`user`*
`--username=`*`user`*

User name to connect as. Defaults to current operating system user name.

`-w`
`--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W`
`--password`

Force pg_recvlogical to prompt for a password before connecting to a database.

This option is never essential, since pg_recvlogical will automatically prompt for a password if the server demands password authentication. However, pg_recvlogical will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

The following additional options are available:

`-V`
`--version`

Print the pg_recvlogical version and exit.

`-?`
`--help`

Show help about pg_recvlogical command line arguments, and exit.

Environment

This utility, like most other PostgreSQL utilities, uses the environment variables supported by libpq (see Section 31.14).

Examples

See Section 46.1 for an example.

See Also

pg_receivexlog

pg_restore

Name

pg_restore — restore a PostgreSQL database from an archive file created by pg_dump

Synopsis

pg_restore [*connection-option*...] [*option*...] [*filename*]

Description

pg_restore is a utility for restoring a PostgreSQL database from an archive created by pg_dump in one of the non-plain-text formats. It will issue the commands necessary to reconstruct the database to the state it was in at the time it was saved. The archive files also allow pg_restore to be selective about what is restored, or even to reorder the items prior to being restored. The archive files are designed to be portable across architectures.

pg_restore can operate in two modes. If a database name is specified, pg_restore connects to that database and restores archive contents directly into the database. Otherwise, a script containing the SQL commands necessary to rebuild the database is created and written to a file or standard output. This script output is equivalent to the plain text output format of pg_dump. Some of the options controlling the output are therefore analogous to pg_dump options.

Obviously, pg_restore cannot restore information that is not present in the archive file. For instance, if the archive was made using the "dump data as INSERT commands" option, pg_restore will not be able to load the data using COPY statements.

Options

pg_restore accepts the following command line arguments.

filename

Specifies the location of the archive file (or directory, for a directory-format archive) to be restored. If not specified, the standard input is used.

-a
--data-only

Restore only the data, not the schema (data definitions). Table data, large objects, and sequence values are restored, if present in the archive.

This option is similar to, but for historical reasons not identical to, specifying --section=data.

`-c`
`--clean`

Clean (drop) database objects before recreating them. (Unless `--if-exists` is used, this might generate some harmless error messages, if any objects were not present in the destination database.)

`-C`
`--create`

Create the database before restoring into it. If `--clean` is also specified, drop and recreate the target database before connecting to it.

When this option is used, the database named with `-d` is used only to issue the initial `DROP DATABASE` and `CREATE DATABASE` commands. All data is restored into the database name that appears in the archive.

`-d` *`dbname`*
`--dbname=`*`dbname`*

Connect to database *`dbname`* and restore directly into the database. The *`dbname`* can be a connection string. If so, connection string parameters will override any conflicting command line options.

`-e`
`--exit-on-error`

Exit if an error is encountered while sending SQL commands to the database. The default is to continue and to display a count of errors at the end of the restoration.

`-f` *`filename`*
`--file=`*`filename`*

Specify output file for generated script, or for the listing when used with `-l`. Use `-` for the standard output, which is also the default.

`-F` *`format`*
`--format=`*`format`*

Specify format of the archive. It is not necessary to specify the format, since pg_restore will determine the format automatically. If specified, it can be one of the following:

`c`
`custom`

The archive is in the custom format of pg_dump.

`d`
`directory`

The archive is a directory archive.

`t`
`tar`

The archive is a `tar` archive.

`-I` *index*
`--index=`*index*

Restore definition of named index only. Multiple indexes may be specified with multiple `-I` switches.

`-j` *number-of-jobs*
`--jobs=`*number-of-jobs*

Run the most time-consuming parts of pg_restore — those which load data, create indexes, or create constraints — using multiple concurrent jobs. This option can dramatically reduce the time to restore a large database to a server running on a multiprocessor machine.

Each job is one process or one thread, depending on the operating system, and uses a separate connection to the server.

The optimal value for this option depends on the hardware setup of the server, of the client, and of the network. Factors include the number of CPU cores and the disk setup. A good place to start is the number of CPU cores on the server, but values larger than that can also lead to faster restore times in many cases. Of course, values that are too high will lead to decreased performance because of thrashing.

Only the custom and directory archive formats are supported with this option. The input must be a regular file or directory (not, for example, a pipe). This option is ignored when emitting a script rather than connecting directly to a database server. Also, multiple jobs cannot be used together with the option `--single-transaction`.

`-l`
`--list`

List the contents of the archive. The output of this operation can be used as input to the `-L` option. Note that if filtering switches such as `-n` or `-t` are used with `-l`, they will restrict the items listed.

`-L` *list-file*
`--use-list=`*list-file*

Restore only those archive elements that are listed in *list-file*, and restore them in the order they appear in the file. Note that if filtering switches such as `-n` or `-t` are used with `-L`, they will further restrict the items restored.

list-file is normally created by editing the output of a previous `-l` operation. Lines can be moved or removed, and can also be commented out by placing a semicolon (`;`) at the start of the line. See below for examples.

`-n` *namespace*
`--schema=`*schema*

Restore only objects that are in the named schema. Multiple schemas may be specified with multiple `-n` switches. This can be combined with the `-t` option to restore just a specific table.

`-O`
`--no-owner`

Do not output commands to set ownership of objects to match the original database. By default, pg_restore issues `ALTER OWNER` or `SET SESSION AUTHORIZATION` statements to set ownership of created schema elements. These statements will fail unless the initial connection to the database is made by a superuser (or the same user that owns all of the objects in the script). With `-O`, any user name can be used for the initial connection, and this user will own all the created objects.

`-P` *function-name(argtype [, ...])*
`--function=`*function-name(argtype [, ...])*

Restore the named function only. Be careful to spell the function name and arguments exactly as they appear in the dump file's table of contents. Multiple functions may be specified with multiple `-P` switches.

`-R`
`--no-reconnect`

This option is obsolete but still accepted for backwards compatibility.

`-s`
`--schema-only`

Restore only the schema (data definitions), not data, to the extent that schema entries are present in the archive.

This option is the inverse of `--data-only`. It is similar to, but for historical reasons not identical to, specifying `--section=pre-data --section=post-data`.

(Do not confuse this with the `--schema` option, which uses the word "schema" in a different meaning.)

`-S` *username*
`--superuser=`*username*

Specify the superuser user name to use when disabling triggers. This is relevant only if `--disable-triggers` is used.

`-t` *table*
`--table=`*table*

Restore definition and/or data of named table only. Multiple tables may be specified with multiple `-t` switches. This can be combined with the `-n` option to specify a schema.

`-T` *trigger*
`--trigger=`*trigger*

Restore named trigger only. Multiple triggers may be specified with multiple `-T` switches.

`-v`
`--verbose`

Specifies verbose mode.

`-V`
`--version`

Print the pg_restore version and exit.

`-x`
`--no-privileges`
`--no-acl`

Prevent restoration of access privileges (grant/revoke commands).

`-1`
`--single-transaction`

Execute the restore as a single transaction (that is, wrap the emitted commands in `BEGIN`/`COMMIT`). This ensures that either all the commands complete successfully, or no changes are applied. This option implies `--exit-on-error`.

`--disable-triggers`

This option is relevant only when performing a data-only restore. It instructs pg_restore to execute commands to temporarily disable triggers on the target tables while the data is reloaded. Use this if you have referential integrity checks or other triggers on the tables that you do not want to invoke during data reload.

Presently, the commands emitted for `--disable-triggers` must be done as superuser. So you should also specify a superuser name with `-S` or, preferably, run pg_restore as a PostgreSQL superuser.

`--enable-row-security`

This option is relevant only when restoring the contents of a table which has row security. By default, pg_restore will set row_security to off, to ensure that all data is restored in to the table. If the user does not have sufficient privileges to bypass row security, then an error is thrown. This parameter instructs pg_restore to set row_security to on instead, allowing the user to attempt to restore the contents of the table with row security enabled. This might still fail if the user does not have the right to insert the rows from the dump into the table.

Note that this option currently also requires the dump be in `INSERT` format, as `COPY FROM` does not support row security.

`--if-exists`

Use conditional commands (i.e., add an `IF EXISTS` clause) when cleaning database objects. This option is not valid unless `--clean` is also specified.

`--no-data-for-failed-tables`

By default, table data is restored even if the creation command for the table failed (e.g., because it already exists). With this option, data for such a table is skipped. This behavior is useful if the target database already contains the desired table contents. For example, auxiliary tables for PostgreSQL extensions such as PostGIS might already be loaded in the target database; specifying this option prevents duplicate or obsolete data from being loaded into them.

This option is effective only when restoring directly into a database, not when producing SQL script output.

`--no-security-labels`

Do not output commands to restore security labels, even if the archive contains them.

`--no-tablespaces`

Do not output commands to select tablespaces. With this option, all objects will be created in whichever tablespace is the default during restore.

`--section=`*`sectionname`*

Only restore the named section. The section name can be `pre-data`, `data`, or `post-data`. This option can be specified more than once to select multiple sections. The default is to restore all sections.

The data section contains actual table data as well as large-object definitions. Post-data items consist of definitions of indexes, triggers, rules and constraints other than validated check constraints. Pre-data items consist of all other data definition items.

`--use-set-session-authorization`

Output SQL-standard `SET SESSION AUTHORIZATION` commands instead of `ALTER OWNER` commands to determine object ownership. This makes the dump more standards-compatible, but depending on the history of the objects in the dump, might not restore properly.

`-?`
`--help`

Show help about pg_restore command line arguments, and exit.

pg_restore also accepts the following command line arguments for connection parameters:

`-h` *`host`*
`--host=`*`host`*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket. The default is taken from the `PGHOST` environment variable, if set, else a Unix domain socket connection is attempted.

`-p` *`port`*
`--port=`*`port`*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections. Defaults to the `PGPORT` environment variable, if set, or a compiled-in default.

`-U` *`username`*
`--username=`*`username`*

User name to connect as.

`-w`
`--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W`
`--password`

Force pg_restore to prompt for a password before connecting to a database.

This option is never essential, since pg_restore will automatically prompt for a password if the server demands password authentication. However, pg_restore will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

`--role=`*`rolename`*

Specifies a role name to be used to perform the restore. This option causes pg_restore to issue a `SET ROLE` *`rolename`* command after connecting to the database. It is useful when the authenticated user (specified by `-U`) lacks privileges needed by pg_restore, but can switch to a role with the required

rights. Some installations have a policy against logging in directly as a superuser, and use of this option allows restores to be performed without violating the policy.

Environment

`PGHOST`
`PGOPTIONS`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14). However, it does not read `PGDATABASE` when a database name is not supplied.

Diagnostics

When a direct database connection is specified using the `-d` option, pg_restore internally executes SQL statements. If you have problems running pg_restore, make sure you are able to select information from the database using, for example, psql. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Notes

If your installation has any local additions to the `template1` database, be careful to load the output of pg_restore into a truly empty database; otherwise you are likely to get errors due to duplicate definitions of the added objects. To make an empty database without any local additions, copy from `template0` not `template1`, for example:

```
CREATE DATABASE foo WITH TEMPLATE template0;
```

The limitations of pg_restore are detailed below.

- When restoring data to a pre-existing table and the option `--disable-triggers` is used, pg_restore emits commands to disable triggers on user tables before inserting the data, then emits commands to re-enable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs might be left in the wrong state.
- pg_restore cannot restore large objects selectively; for instance, only those for a specific table. If an archive contains large objects, then all large objects will be restored, or none of them if they are excluded via `-L`, `-t`, or other options.

See also the pg_dump documentation for details on limitations of pg_dump.

Once restored, it is wise to run `ANALYZE` on each restored table so the optimizer has useful statistics; see Section 23.1.3 and Section 23.1.6 for more information.

Examples

Assume we have dumped a database called `mydb` into a custom-format dump file:

```
$ pg_dump -Fc mydb > db.dump
```

To drop the database and recreate it from the dump:

```
$ dropdb mydb
$ pg_restore -C -d postgres db.dump
```

The database named in the `-d` switch can be any database existing in the cluster; pg_restore only uses it to issue the `CREATE DATABASE` command for `mydb`. With `-C`, data is always restored into the database name that appears in the dump file.

To reload the dump into a new database called `newdb`:

```
$ createdb -T template0 newdb
$ pg_restore -d newdb db.dump
```

Notice we don't use `-C`, and instead connect directly to the database to be restored into. Also note that we clone the new database from `template0` not `template1`, to ensure it is initially empty.

To reorder database items, it is first necessary to dump the table of contents of the archive:

```
$ pg_restore -l db.dump > db.list
```

The listing file consists of a header and one line for each item, e.g.:

```
;
; Archive created at Mon Sep 14 13:55:39 2009
;     dbname: DBDEMOS
;     TOC Entries: 81
;     Compression: 9
;     Dump Version: 1.10-0
;     Format: CUSTOM
;     Integer: 4 bytes
;     Offset: 8 bytes
;     Dumped from database version: 8.3.5
;     Dumped by pg_dump version: 8.3.8
;
;
; Selected TOC Entries:
;
3; 2615 2200 SCHEMA - public pasha
1861; 0 0 COMMENT - SCHEMA public pasha
1862; 0 0 ACL - public pasha
317; 1247 17715 TYPE public composite pasha
```

```
319; 1247 25899 DOMAIN public domain0 pasha
```

Semicolons start a comment, and the numbers at the start of lines refer to the internal archive ID assigned to each item.

Lines in the file can be commented out, deleted, and reordered. For example:

```
10; 145433 TABLE map_resolutions postgres
;2; 145344 TABLE species postgres
;4; 145359 TABLE nt_header postgres
6; 145402 TABLE species_records postgres
;8; 145416 TABLE ss_old postgres
```

could be used as input to pg_restore and would only restore items 10 and 6, in that order:

```
$ pg_restore -L db.list db.dump
```

See Also

pg_dump, pg_dumpall, psql

psql

Name

psql — PostgreSQL interactive terminal

Synopsis

psql [*option*...] [*dbname* [*username*]]

Description

psql is a terminal-based front-end to PostgreSQL. It enables you to type in queries interactively, issue them to PostgreSQL, and see the query results. Alternatively, input can be from a file. In addition, it provides a number of meta-commands and various shell-like features to facilitate writing scripts and automating a wide variety of tasks.

Options

-a
--echo-all

Print all nonempty input lines to standard output as they are read. (This does not apply to lines read interactively.) This is equivalent to setting the variable ECHO to all.

-A
--no-align

Switches to unaligned output mode. (The default output mode is otherwise aligned.)

-b
--echo-errors

Print failed SQL commands to standard error output. This is equivalent to setting the variable ECHO to errors.

-c *command*
--command=*command*

Specifies that psql is to execute one command string, *command*, and then exit. This is useful in shell scripts. Start-up files (psqlrc and ~/.psqlrc) are ignored with this option.

command must be either a command string that is completely parsable by the server (i.e., it contains no psql-specific features), or a single backslash command. Thus you cannot mix SQL and psql meta-commands with this option. To achieve that, you could pipe the string into psql, for example: echo '\x \\ SELECT * FROM foo;' | psql. (\\ is the separator meta-command.)

If the command string contains multiple SQL commands, they are processed in a single transaction, unless there are explicit `BEGIN`/`COMMIT` commands included in the string to divide it into multiple transactions. This is different from the behavior when the same string is fed to psql's standard input. Also, only the result of the last SQL command is returned.

Because of these legacy behaviors, putting more than one command in the `-c` string often has unexpected results. It's better to feed multiple commands to psql's standard input, either using echo as illustrated above, or via a shell here-document, for example:

```
psql <<EOF
\x
SELECT * FROM foo;
EOF
```

`-d` *`dbname`*
`--dbname=`*`dbname`*

Specifies the name of the database to connect to. This is equivalent to specifying *`dbname`* as the first non-option argument on the command line. The *`dbname`* can be a connection string. If so, connection string parameters will override any conflicting command line options.

`-e`
`--echo-queries`

Copy all SQL commands sent to the server to standard output as well. This is equivalent to setting the variable `ECHO` to `queries`.

`-E`
`--echo-hidden`

Echo the actual queries generated by `\d` and other backslash commands. You can use this to study psql's internal operations. This is equivalent to setting the variable `ECHO_HIDDEN` to `on`.

`-f` *`filename`*
`--file=`*`filename`*

Use the file *`filename`* as the source of commands instead of reading commands interactively. After the file is processed, psql terminates. This is in many ways equivalent to the meta-command `\i`.

If *`filename`* is `-` (hyphen), then standard input is read until an EOF indication or `\q` meta-command. Note however that Readline is not used in this case (much as if `-n` had been specified).

Using this option is subtly different from writing `psql < `*`filename`*. In general, both will do what you expect, but using `-f` enables some nice features such as error messages with line numbers. There is also a slight chance that using this option will reduce the start-up overhead. On the other hand, the variant using the shell's input redirection is (in theory) guaranteed to yield exactly the same output you would have received had you entered everything by hand.

`-F` *`separator`*
`--field-separator=`*`separator`*

Use *`separator`* as the field separator for unaligned output. This is equivalent to `\pset fieldsep` or `\f`.

`-h` *hostname*
`--host=`*hostname*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix-domain socket.

`-H`
`--html`

Turn on HTML tabular output. This is equivalent to `\pset format html` or the `\H` command.

`-l`
`--list`

List all available databases, then exit. Other non-connection options are ignored. This is similar to the meta-command `\list`.

`-L` *filename*
`--log-file=`*filename*

Write all query output into file *filename*, in addition to the normal output destination.

`-n`
`--no-readline`

Do not use Readline for line editing and do not use the command history. This can be useful to turn off tab expansion when cutting and pasting.

`-o` *filename*
`--output=`*filename*

Put all query output into file *filename*. This is equivalent to the command `\o`.

`-p` *port*
`--port=`*port*

Specifies the TCP port or the local Unix-domain socket file extension on which the server is listening for connections. Defaults to the value of the `PGPORT` environment variable or, if not set, to the port specified at compile time, usually 5432.

`-P` *assignment*
`--pset=`*assignment*

Specifies printing options, in the style of `\pset`. Note that here you have to separate name and value with an equal sign instead of a space. For example, to set the output format to LaTeX, you could write `-P format=latex`.

`-q`
`--quiet`

Specifies that psql should do its work quietly. By default, it prints welcome messages and various informational output. If this option is used, none of this happens. This is useful with the `-c` option. This is equivalent to setting the variable `QUIET` to `on`.

`-R` *separator*
`--record-separator=`*separator*

Use *separator* as the record separator for unaligned output. This is equivalent to the `\pset recordsep` command.

`-s`
`--single-step`

Run in single-step mode. That means the user is prompted before each command is sent to the server, with the option to cancel execution as well. Use this to debug scripts.

`-S`
`--single-line`

Runs in single-line mode where a newline terminates an SQL command, as a semicolon does.

> **Note:** This mode is provided for those who insist on it, but you are not necessarily encouraged to use it. In particular, if you mix SQL and meta-commands on a line the order of execution might not always be clear to the inexperienced user.

`-t`
`--tuples-only`

Turn off printing of column names and result row count footers, etc. This is equivalent to the `\t` command.

`-T` *`table_options`*
`--table-attr=`*`table_options`*

Specifies options to be placed within the HTML `table` tag. See `\pset` for details.

`-U` *`username`*
`--username=`*`username`*

Connect to the database as the user *`username`* instead of the default. (You must have permission to do so, of course.)

`-v` *`assignment`*
`--set=`*`assignment`*
`--variable=`*`assignment`*

Perform a variable assignment, like the `\set` meta-command. Note that you must separate name and value, if any, by an equal sign on the command line. To unset a variable, leave off the equal sign. To set a variable with an empty value, use the equal sign but leave off the value. These assignments are done during a very early stage of start-up, so variables reserved for internal purposes might get overwritten later.

`-V`
`--version`

Print the psql version and exit.

`-w`
`--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available from other sources such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

Note that this option will remain set for the entire session, and so it affects uses of the meta-command `\connect` as well as the initial connection attempt.

`-W`
`--password`

Force psql to prompt for a password before connecting to a database, even if the password will not be used.

If the server requires password authentication and a password is not available from other sources such as a `.pgpass` file, psql will prompt for a password in any case. However, psql will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

Note that this option will remain set for the entire session, and so it affects uses of the meta-command `\connect` as well as the initial connection attempt.

`-x`
`--expanded`

Turn on the expanded table formatting mode. This is equivalent to the `\x` command.

`-X,`
`--no-psqlrc`

Do not read the start-up file (neither the system-wide `psqlrc` file nor the user's `~/.psqlrc` file).

`-z`
`--field-separator-zero`

Set the field separator for unaligned output to a zero byte.

`-0`
`--record-separator-zero`

Set the record separator for unaligned output to a zero byte. This is useful for interfacing, for example, with `xargs -0`.

`-1`
`--single-transaction`

When psql executes a script, adding this option wraps `BEGIN`/`COMMIT` around the script to execute it as a single transaction. This ensures that either all the commands complete successfully, or no changes are applied.

If the script itself uses `BEGIN`, `COMMIT`, or `ROLLBACK`, this option will not have the desired effects. Also, if the script contains any command that cannot be executed inside a transaction block, specifying this option will cause that command (and hence the whole transaction) to fail.

`-?`
`--help[=`*`topic`*`]`

Show help about psql and exit. The optional *`topic`* parameter (defaulting to `options`) selects which part of psql is explained: `commands` describes psql's backslash commands; `options` describes the command-line options that can be passed to psql; and `variables` shows help about psql configuration variables.

Exit Status

psql returns 0 to the shell if it finished normally, 1 if a fatal error of its own occurs (e.g., out of memory, file not found), 2 if the connection to the server went bad and the session was not interactive, and 3 if an error occurred in a script and the variable `ON_ERROR_STOP` was set.

Usage

Connecting to a Database

psql is a regular PostgreSQL client application. In order to connect to a database you need to know the name of your target database, the host name and port number of the server, and what user name you want to connect as. psql can be told about those parameters via command line options, namely `-d`, `-h`, `-p`, and `-U` respectively. If an argument is found that does not belong to any option it will be interpreted as the database name (or the user name, if the database name is already given). Not all of these options are required; there are useful defaults. If you omit the host name, psql will connect via a Unix-domain socket to a server on the local host, or via TCP/IP to `localhost` on machines that don't have Unix-domain sockets. The default port number is determined at compile time. Since the database server uses the same default, you will not have to specify the port in most cases. The default user name is your operating-system user name, as is the default database name. Note that you cannot just connect to any database under any user name. Your database administrator should have informed you about your access rights.

When the defaults aren't quite right, you can save yourself some typing by setting the environment variables `PGDATABASE`, `PGHOST`, `PGPORT` and/or `PGUSER` to appropriate values. (For additional environment variables, see Section 31.14.) It is also convenient to have a `~/.pgpass` file to avoid regularly having to type in passwords. See Section 31.15 for more information.

An alternative way to specify connection parameters is in a `conninfo` string or a URI, which is used instead of a database name. This mechanism give you very wide control over the connection. For example:

```
$ psql "service=myservice sslmode=require"
$ psql postgresql://dbmaster:5433/mydb?sslmode=require
```

This way you can also use LDAP for connection parameter lookup as described in Section 31.17. See Section 31.1.2 for more information on all the available connection options.

If the connection could not be made for any reason (e.g., insufficient privileges, server is not running on the targeted host, etc.), psql will return an error and terminate.

If both standard input and standard output are a terminal, then psql sets the client encoding to "auto", which will detect the appropriate client encoding from the locale settings (`LC_CTYPE` environment variable on Unix systems). If this doesn't work out as expected, the client encoding can be overridden using the environment variable `PGCLIENTENCODING`.

Entering SQL Commands

In normal operation, psql provides a prompt with the name of the database to which psql is currently

connected, followed by the string =>. For example:

```
$ psql testdb
psql (9.5.24)
Type "help" for help.

testdb=>
```

At the prompt, the user can type in SQL commands. Ordinarily, input lines are sent to the server when a command-terminating semicolon is reached. An end of line does not terminate a command. Thus commands can be spread over several lines for clarity. If the command was sent and executed without error, the results of the command are displayed on the screen.

If untrusted users have access to a database that has not adopted a secure schema usage pattern, begin your session by removing publicly-writable schemas from `search_path`. One can add `options=-csearch_path=` to the connection string or issue `SELECT pg_catalog.set_config('search_path', '', false)` before other SQL commands. This consideration is not specific to psql; it applies to every interface for executing arbitrary SQL commands.

Whenever a command is executed, psql also polls for asynchronous notification events generated by LISTEN and NOTIFY.

While C-style block comments are passed to the server for processing and removal, SQL-standard comments are removed by psql.

Meta-Commands

Anything you enter in psql that begins with an unquoted backslash is a psql meta-command that is processed by psql itself. These commands make psql more useful for administration or scripting. Meta-commands are often called slash or backslash commands.

The format of a psql command is the backslash, followed immediately by a command verb, then any arguments. The arguments are separated from the command verb and each other by any number of whitespace characters.

To include whitespace in an argument you can quote it with single quotes. To include a single quote in an argument, write two single quotes within single-quoted text. Anything contained in single quotes is furthermore subject to C-like substitutions for `\n` (new line), `\t` (tab), `\b` (backspace), `\r` (carriage return), `\f` (form feed), `\`*`digits`* (octal), and `\x`*`digits`* (hexadecimal). A backslash preceding any other character within single-quoted text quotes that single character, whatever it is.

Within an argument, text that is enclosed in backquotes (`) is taken as a command line that is passed to the shell. The output of the command (with any trailing newline removed) replaces the backquoted text.

If an unquoted colon (:) followed by a psql variable name appears within an argument, it is replaced by the variable's value, as described in *SQL Interpolation*.

Some commands take an SQL identifier (such as a table name) as argument. These arguments follow the syntax rules of SQL: Unquoted letters are forced to lowercase, while double quotes (") protect letters from case conversion and allow incorporation of whitespace into the identifier. Within double quotes, paired double quotes reduce to a single double quote in the resulting name. For example, `FOO"BAR"BAZ` is interpreted as `fooBARbaz`, and `"A weird"" name"` becomes `A weird" name`.

Parsing for arguments stops at the end of the line, or when another unquoted backslash is found. An unquoted backslash is taken as the beginning of a new meta-command. The special sequence `\\` (two backslashes) marks the end of arguments and continues parsing SQL commands, if any. That way SQL and psql commands can be freely mixed on a line. But in any case, the arguments of a meta-command cannot continue beyond the end of the line.

The following meta-commands are defined:

`\a`

If the current table output format is unaligned, it is switched to aligned. If it is not unaligned, it is set to unaligned. This command is kept for backwards compatibility. See `\pset` for a more general solution.

`\c` or `\connect [ -reuse-previous=`*`on|off`*` ] [ `*`dbname`*` [ `*`username`*` ] [ `*`host`*` ] [ `*`port`*` ] | `*`conninfo`*` ]`

Establishes a new connection to a PostgreSQL server. The connection parameters to use can be specified either using a positional syntax (one or more of database name, user, host, and port), or using a *`conninfo`* connection string as detailed in Section 31.1.1. If no arguments are given, a new connection is made using the same parameters as before.

Specifying any of *`dbname`*, *`username`*, *`host`* or *`port`* as `-` is equivalent to omitting that parameter.

The new connection can re-use connection parameters from the previous connection; not only database name, user, host, and port, but other settings such as *`sslmode`*. By default, parameters are re-used in the positional syntax, but not when a *`conninfo`* string is given. Passing a first argument of `-reuse-previous=on` or `-reuse-previous=off` overrides that default. If parameters are re-used, then any parameter not explicitly specified as a positional parameter or in the *`conninfo`* string is taken from the existing connection's parameters. An exception is that if the *`host`* setting is changed from its previous value using the positional syntax, any *`hostaddr`* setting present in the existing connection's parameters is dropped. When the command neither specifies nor reuses a particular parameter, the libpq default is used.

If the new connection is successfully made, the previous connection is closed. If the connection attempt fails (wrong user name, access denied, etc.), the previous connection will be kept if psql is in interactive mode. But when executing a non-interactive script, processing will immediately stop with an error. This distinction was chosen as a user convenience against typos on the one hand, and a safety mechanism that scripts are not accidentally acting on the wrong database on the other hand.

Examples:

```
=> \c mydb myuser host.dom 6432
=> \c service=foo
=> \c "host=localhost port=5432 dbname=mydb connect_timeout=10 sslmode=disable"
=> \c -reuse-previous=on sslmode=require    -- changes only sslmode
=> \c postgresql://tom@localhost/mydb?application_name=myapp
```

`\C [ `*`title`*` ]`

Sets the title of any tables being printed as the result of a query or unset any such title. This command is equivalent to `\pset title `*`title`*. (The name of this command derives from "caption", as it was previously only used to set the caption in an HTML table.)

`\cd [ directory ]`

Changes the current working directory to *directory*. Without argument, changes to the current user's home directory.

Tip: To print your current working directory, use `\! pwd`.

`\conninfo`

Outputs information about the current database connection.

`\copy { table [ ( column_list ) ] | ( query ) } { from | to } { 'filename' | program 'command' | stdin | stdout | pstdin | pstdout } [ [ with ] ( option [, ...] ) ]`

Performs a frontend (client) copy. This is an operation that runs an SQL COPY command, but instead of the server reading or writing the specified file, psql reads or writes the file and routes the data between the server and the local file system. This means that file accessibility and privileges are those of the local user, not the server, and no SQL superuser privileges are required.

When `program` is specified, *command* is executed by psql and the data passed from or to *command* is routed between the server and the client. Again, the execution privileges are those of the local user, not the server, and no SQL superuser privileges are required.

For `\copy ... from stdin`, data rows are read from the same source that issued the command, continuing until `\.` is read or the stream reaches EOF. This option is useful for populating tables inline within a SQL script file. For `\copy ... to stdout`, output is sent to the same place as psql command output, and the `COPY` *count* command status is not printed (since it might be confused with a data row). To read/write psql's standard input or output regardless of the current command source or `\o` option, write `from pstdin` or `to pstdout`.

The syntax of this command is similar to that of the SQL COPY command. All options other than the data source/destination are as specified for COPY. Because of this, special parsing rules apply to the `\copy` command. In particular, psql's variable substitution rules and backslash escapes do not apply.

Tip: Another way to obtain the same result as `\copy ... to` is to use the SQL `COPY ... TO STDOUT` command and terminate it with `\g` *filename* or `\g |`*program*. Unlike `\copy`, this method allows the command to span multiple lines; also, variable interpolation and backquote expansion can be used.

Tip: These operations are not as efficient as the SQL `COPY` command with a file or program data source or destination, because all data must pass through the client/server connection. For large amounts of data the SQL command might be preferable.

`\copyright`

Shows the copyright and distribution terms of PostgreSQL.

`\d[S+] [ pattern ]`

For each relation (table, view, materialized view, index, sequence, or foreign table) or composite type matching the *pattern*, show all columns, their types, the tablespace (if not the default) and any special attributes such as `NOT NULL` or defaults. Associated indexes, constraints, rules, and triggers are also shown. For foreign tables, the associated foreign server is shown as well. ("Matching the pattern" is defined in *Patterns* below.)

For some types of relation, `\d` shows additional information for each column: column values for sequences, indexed expressions for indexes, and foreign data wrapper options for foreign tables.

The command form `\d+` is identical, except that more information is displayed: any comments associated with the columns of the table are shown, as is the presence of OIDs in the table, the view definition if the relation is a view, a non-default replica identity setting.

By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

> **Note:** If `\d` is used without a *pattern* argument, it is equivalent to `\dtvmsE` which will show a list of all visible tables, views, materialized views, sequences and foreign tables. This is purely a convenience measure.

`\da[S] [ pattern ]`

Lists aggregate functions, together with their return type and the data types they operate on. If *pattern* is specified, only aggregates whose names match the pattern are shown. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

`\db[+] [ pattern ]`

Lists tablespaces. If *pattern* is specified, only tablespaces whose names match the pattern are shown. If + is appended to the command name, each tablespace is listed with its associated options, on-disk size, permissions and description.

`\dc[S+] [ pattern ]`

Lists conversions between character-set encodings. If *pattern* is specified, only conversions whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects. If + is appended to the command name, each object is listed with its associated description.

`\dC[+] [ pattern ]`

Lists type casts. If *pattern* is specified, only casts whose source or target types match the pattern are listed. If + is appended to the command name, each object is listed with its associated description.

`\dd[S] [ pattern ]`

Shows the descriptions of objects of type `constraint`, `operator class`, `operator family`, `rule`, and `trigger`. All other comments may be viewed by the respective backslash commands for those object types.

`\dd` displays descriptions for objects matching the *pattern*, or of visible objects of the appropriate type if no argument is given. But in either case, only objects that have a description are listed. By

default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

Descriptions for objects can be created with the COMMENT SQL command.

`\ddp [ pattern ]`

Lists default access privilege settings. An entry is shown for each role (and schema, if applicable) for which the default privilege settings have been changed from the built-in defaults. If *pattern* is specified, only entries whose role name or schema name matches the pattern are listed.

The ALTER DEFAULT PRIVILEGES command is used to set default access privileges. The meaning of the privilege display is explained under GRANT.

`\dD[S+] [ pattern ]`

Lists domains. If *pattern* is specified, only domains whose names match the pattern are shown. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects. If `+` is appended to the command name, each object is listed with its associated permissions and description.

`\dE[S+] [ pattern ]`
`\di[S+] [ pattern ]`
`\dm[S+] [ pattern ]`
`\ds[S+] [ pattern ]`
`\dt[S+] [ pattern ]`
`\dv[S+] [ pattern ]`

In this group of commands, the letters `E`, `i`, `m`, `s`, `t`, and `v` stand for foreign table, index, materialized view, sequence, table, and view, respectively. You can specify any or all of these letters, in any order, to obtain a listing of objects of these types. For example, `\dit` lists indexes and tables. If `+` is appended to the command name, each object is listed with its physical size on disk and its associated description, if any. If *pattern* is specified, only objects whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

`\des[+] [ pattern ]`

Lists foreign servers (mnemonic: “external servers”). If *pattern* is specified, only those servers whose name matches the pattern are listed. If the form `\des+` is used, a full description of each server is shown, including the server’s ACL, type, version, options, and description.

`\det[+] [ pattern ]`

Lists foreign tables (mnemonic: “external tables”). If *pattern* is specified, only entries whose table name or schema name matches the pattern are listed. If the form `\det+` is used, generic options and the foreign table description are also displayed.

`\deu[+] [ pattern ]`

Lists user mappings (mnemonic: “external users”). If *pattern* is specified, only those mappings whose user names match the pattern are listed. If the form `\deu+` is used, additional information about each mapping is shown.

> **Caution**
>
> `\deu+` might also display the user name and password of the remote user, so care should be taken not to disclose them.

`\dew[+] [ pattern ]`

Lists foreign-data wrappers (mnemonic: "external wrappers"). If *pattern* is specified, only those foreign-data wrappers whose name matches the pattern are listed. If the form `\dew+` is used, the ACL, options, and description of the foreign-data wrapper are also shown.

`\df[antwS+] [ pattern ]`

Lists functions, together with their arguments, return types, and function types, which are classified as "agg" (aggregate), "normal", "trigger", or "window". To display only functions of specific type(s), add the corresponding letters `a`, `n`, `t`, or `w` to the command. If *pattern* is specified, only functions whose names match the pattern are shown. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects. If the form `\df+` is used, additional information about each function is shown, including security classification, volatility, owner, language, source code and description.

> **Tip:** To look up functions taking arguments or returning values of a specific type, use your pager's search capability to scroll through the `\df` output.

`\dF[+] [ pattern ]`

Lists text search configurations. If *pattern* is specified, only configurations whose names match the pattern are shown. If the form `\dF+` is used, a full description of each configuration is shown, including the underlying text search parser and the dictionary list for each parser token type.

`\dFd[+] [ pattern ]`

Lists text search dictionaries. If *pattern* is specified, only dictionaries whose names match the pattern are shown. If the form `\dFd+` is used, additional information is shown about each selected dictionary, including the underlying text search template and the option values.

`\dFp[+] [ pattern ]`

Lists text search parsers. If *pattern* is specified, only parsers whose names match the pattern are shown. If the form `\dFp+` is used, a full description of each parser is shown, including the underlying functions and the list of recognized token types.

`\dFt[+] [ pattern ]`

Lists text search templates. If *pattern* is specified, only templates whose names match the pattern are shown. If the form `\dFt+` is used, additional information is shown about each template, including the underlying function names.

`\dg[+] [ pattern ]`

Lists database roles. (Since the concepts of "users" and "groups" have been unified into "roles", this command is now equivalent to `\du`.) If *pattern* is specified, only those roles whose names match the pattern are listed. If the form `\dg+` is used, additional information is shown about each role; currently this adds the comment for each role.

`\dl`

This is an alias for `\lo_list`, which shows a list of large objects.

`\dL[S+] [ pattern ]`

Lists procedural languages. If *pattern* is specified, only languages whose names match the pattern are listed. By default, only user-created languages are shown; supply the `S` modifier to include system objects. If + is appended to the command name, each language is listed with its call handler, validator, access privileges, and whether it is a system object.

`\dn[S+] [ pattern ]`

Lists schemas (namespaces). If *pattern* is specified, only schemas whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects. If + is appended to the command name, each object is listed with its associated permissions and description, if any.

`\do[S+] [ pattern ]`

Lists operators with their operand and result types. If *pattern* is specified, only operators whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects. If + is appended to the command name, additional information about each operator is shown, currently just the name of the underlying function.

`\dO[S+] [ pattern ]`

Lists collations. If *pattern* is specified, only collations whose names match the pattern are listed. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects. If + is appended to the command name, each collation is listed with its associated description, if any. Note that only collations usable with the current database's encoding are shown, so the results may vary in different databases of the same installation.

`\dp [ pattern ]`

Lists tables, views and sequences with their associated access privileges. If *pattern* is specified, only tables, views and sequences whose names match the pattern are listed.

The GRANT and REVOKE commands are used to set access privileges. The meaning of the privilege display is explained under GRANT.

`\drds [ role-pattern [ database-pattern ] ]`

Lists defined configuration settings. These settings can be role-specific, database-specific, or both. *role-pattern* and *database-pattern* are used to select specific roles and databases to list, respectively. If omitted, or if * is specified, all settings are listed, including those not role-specific or database-specific, respectively.

The ALTER ROLE and ALTER DATABASE commands are used to define per-role and per-database configuration settings.

`\dT[S+] [ pattern ]`

Lists data types. If *pattern* is specified, only types whose names match the pattern are listed. If + is appended to the command name, each type is listed with its internal name and size, its allowed values if it is an `enum` type, and its associated permissions. By default, only user-created objects are shown; supply a pattern or the `S` modifier to include system objects.

`\du[+] [ pattern ]`

Lists database roles. (Since the concepts of "users" and "groups" have been unified into "roles", this command is now equivalent to `\dg`.) If *pattern* is specified, only those roles whose names match the pattern are listed. If the form `\du+` is used, additional information is shown about each role; currently this adds the comment for each role.

`\dx[+] [ pattern ]`

Lists installed extensions. If *pattern* is specified, only those extensions whose names match the pattern are listed. If the form `\dx+` is used, all the objects belonging to each matching extension are listed.

`\dy[+] [ pattern ]`

Lists event triggers. If *pattern* is specified, only those event triggers whose names match the pattern are listed. If + is appended to the command name, each object is listed with its associated description.

`\e` or `\edit [ filename ] [ line_number ]`

If *filename* is specified, the file is edited; after the editor exits, its content is copied back to the query buffer. If no *filename* is given, the current query buffer is copied to a temporary file which is then edited in the same fashion.

The new query buffer is then re-parsed according to the normal rules of psql, where the whole buffer is treated as a single line. (Thus you cannot make scripts this way. Use `\i` for that.) This means that if the query ends with (or contains) a semicolon, it is immediately executed. Otherwise it will merely wait in the query buffer; type semicolon or `\g` to send it, or `\r` to cancel.

If a line number is specified, psql will position the cursor on the specified line of the file or query buffer. Note that if a single all-digits argument is given, psql assumes it is a line number, not a file name.

> **Tip:** See under *Environment* for how to configure and customize your editor.

`\echo text [ ... ]`

Prints the arguments to the standard output, separated by one space and followed by a newline. This can be useful to intersperse information in the output of scripts. For example:

```
=> \echo `date`
Tue Oct 26 21:40:57 CEST 1999
```

If the first argument is an unquoted `-n` the trailing newline is not written.

> **Tip:** If you use the `\o` command to redirect your query output you might wish to use `\qecho` instead of this command.

`\ef [ function_description [ line_number ] ]`

This command fetches and edits the definition of the named function, in the form of a `CREATE OR REPLACE FUNCTION` command. Editing is done in the same way as for `\edit`. After the editor exits, the updated command waits in the query buffer; type semicolon or `\g` to send it, or `\r` to cancel.

The target function can be specified by name alone, or by name and arguments, for example `foo(integer, text)`. The argument types must be given if there is more than one function of the same name.

If no function is specified, a blank `CREATE FUNCTION` template is presented for editing.

If a line number is specified, psql will position the cursor on the specified line of the function body. (Note that the function body typically does not begin on the first line of the file.)

> **Tip:** See under *Environment* for how to configure and customize your editor.

`\encoding [ *encoding* ]`

Sets the client character set encoding. Without an argument, this command shows the current encoding.

`\f [ *string* ]`

Sets the field separator for unaligned query output. The default is the vertical bar (`|`). See also `\pset` for a generic way of setting output options.

`\g [ *filename* ]`
`\g [ |*command* ]`

Sends the current query input buffer to the server, and optionally stores the query's output in *`filename`* or pipes the output to the shell command *`command`*. The file or command is written to only if the query successfully returns zero or more tuples, not if the query fails or is a non-data-returning SQL command.

A bare `\g` is essentially equivalent to a semicolon. A `\g` with argument is a "one-shot" alternative to the `\o` command.

`\gset [ *prefix* ]`

Sends the current query input buffer to the server and stores the query's output into psql variables (see *Variables*). The query to be executed must return exactly one row. Each column of the row is stored into a separate variable, named the same as the column. For example:

```
=> SELECT 'hello' AS var1, 10 AS var2
-> \gset
=> \echo :var1 :var2
hello 10
```

If you specify a *`prefix`*, that string is prepended to the query's column names to create the variable names to use:

```
=> SELECT 'hello' AS var1, 10 AS var2
-> \gset result_
=> \echo :result_var1 :result_var2
hello 10
```

If a column result is NULL, the corresponding variable is unset rather than being set.

If the query fails or does not return one row, no variables are changed.

`\h` or `\help [ command ]`

Gives syntax help on the specified SQL command. If *command* is not specified, then psql will list all the commands for which syntax help is available. If *command* is an asterisk (`*`), then syntax help on all SQL commands is shown.

> **Note:** To simplify typing, commands that consists of several words do not have to be quoted. Thus it is fine to type **`\help alter table`**.

`\H` or `\html`

Turns on HTML query output format. If the HTML format is already on, it is switched back to the default aligned text format. This command is for compatibility and convenience, but see `\pset` about setting other output options.

`\i` or `\include` *filename*

Reads input from the file *filename* and executes it as though it had been typed on the keyboard.

If *filename* is – (hyphen), then standard input is read until an EOF indication or `\q` meta-command. This can be used to intersperse interactive input with input from files. Note that Readline behavior will be used only if it is active at the outermost level.

> **Note:** If you want to see the lines on the screen as they are read you must set the variable `ECHO` to `all`.

`\ir` or `\include_relative` *filename*

The `\ir` command is similar to `\i`, but resolves relative file names differently. When executing in interactive mode, the two commands behave identically. However, when invoked from a script, `\ir` interprets file names relative to the directory in which the script is located, rather than the current working directory.

`\l[+]` or `\list[+] [ pattern ]`

List the databases in the server and show their names, owners, character set encodings, and access privileges. If *pattern* is specified, only databases whose names match the pattern are listed. If + is appended to the command name, database sizes, default tablespaces, and descriptions are also displayed. (Size information is only available for databases that the current user can connect to.)

`\lo_export` *loid filename*

Reads the large object with OID *loid* from the database and writes it to *filename*. Note that this is subtly different from the server function `lo_export`, which acts with the permissions of the user that the database server runs as and on the server's file system.

> **Tip:** Use `\lo_list` to find out the large object's OID.

`\lo_import` *filename* [*comment*]

Stores the file into a PostgreSQL large object. Optionally, it associates the given comment with the object. Example:

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of me'
lo_import 152801
```

The response indicates that the large object received object ID 152801, which can be used to access the newly-created large object in the future. For the sake of readability, it is recommended to always associate a human-readable comment with every object. Both OIDs and comments can be viewed with the `\lo_list` command.

Note that this command is subtly different from the server-side `lo_import` because it acts as the local user on the local file system, rather than the server's user and file system.

`\lo_list`

Shows a list of all PostgreSQL large objects currently stored in the database, along with any comments provided for them.

`\lo_unlink` *loid*

Deletes the large object with OID *loid* from the database.

> **Tip:** Use `\lo_list` to find out the large object's OID.

`\o` or `\out` [*filename*]
`\o` or `\out` [|*command*]

Arranges to save future query results to the file *filename* or pipe future results to the shell command *command*. If no argument is specified, the query output is reset to the standard output.

"Query results" includes all tables, command responses, and notices obtained from the database server, as well as output of various backslash commands that query the database (such as `\d`), but not error messages.

> **Tip:** To intersperse text output in between query results, use `\qecho`.

`\p` or `\print`

Print the current query buffer to the standard output.

`\password` [*username*]

Changes the password of the specified user (by default, the current user). This command prompts for the new password, encrypts it, and sends it to the server as an `ALTER ROLE` command. This makes sure that the new password does not appear in cleartext in the command history, the server log, or elsewhere.

`\prompt` [*text*] *name*

Prompts the user to supply text, which is assigned to the variable *name*. An optional prompt string, *text*, can be specified. (For multiword prompts, surround the text with single quotes.)

By default, `\prompt` uses the terminal for input and output. However, if the `-f` command line switch was used, `\prompt` uses standard input and standard output.

`\pset [ option [ value ] ]`

This command sets options affecting the output of query result tables. *option* indicates which option is to be set. The semantics of *value* vary depending on the selected option. For some options, omitting *value* causes the option to be toggled or unset, as described under the particular option. If no such behavior is mentioned, then omitting *value* just results in the current setting being displayed.

`\pset` without any arguments displays the current status of all printing options.

Adjustable printing options are:

`border`

The *value* must be a number. In general, the higher the number the more borders and lines the tables will have, but details depend on the particular format. In HTML format, this will translate directly into the `border=...` attribute. In most other formats only values 0 (no border), 1 (internal dividing lines), and 2 (table frame) make sense, and values above 2 will be treated the same as `border = 2`. The `latex` and `latex-longtable` formats additionally allow a value of 3 to add dividing lines between data rows.

`columns`

Sets the target width for the `wrapped` format, and also the width limit for determining whether output is wide enough to require the pager or switch to the vertical display in expanded auto mode. Zero (the default) causes the target width to be controlled by the environment variable `COLUMNS`, or the detected screen width if `COLUMNS` is not set. In addition, if `columns` is zero then the `wrapped` format only affects screen output. If `columns` is nonzero then file and pipe output is wrapped to that width as well.

`expanded` (or `x`)

If *value* is specified it must be either `on` or `off`, which will enable or disable expanded mode, or `auto`. If *value* is omitted the command toggles between the on and off settings. When expanded mode is enabled, query results are displayed in two columns, with the column name on the left and the data on the right. This mode is useful if the data wouldn't fit on the screen in the normal "horizontal" mode. In the auto setting, the expanded mode is used whenever the query output is wider than the screen, otherwise the regular mode is used. The auto setting is only effective in the aligned and wrapped formats. In other formats, it always behaves as if the expanded mode is off.

`fieldsep`

Specifies the field separator to be used in unaligned output format. That way one can create, for example, tab- or comma-separated output, which other programs might prefer. To set a tab as field separator, type `\pset fieldsep '\t'`. The default field separator is `'|'` (a vertical bar).

`fieldsep_zero`

Sets the field separator to use in unaligned output format to a zero byte.

`footer`

If *`value`* is specified it must be either `on` or `off` which will enable or disable display of the table footer (the `(`*`n`*` rows)` count). If *`value`* is omitted the command toggles footer display on or off.

`format`

Sets the output format to one of `unaligned`, `aligned`, `wrapped`, `html`, `asciidoc`, `latex` (uses `tabular`), `latex-longtable`, or `troff-ms`. Unique abbreviations are allowed.

`unaligned` format writes all columns of a row on one line, separated by the currently active field separator. This is useful for creating output that might be intended to be read in by other programs (for example, tab-separated or comma-separated format).

`aligned` format is the standard, human-readable, nicely formatted text output; this is the default.

`wrapped` format is like `aligned` but wraps wide data values across lines to make the output fit in the target column width. The target width is determined as described under the `columns` option. Note that psql will not attempt to wrap column header titles; therefore, `wrapped` format behaves the same as `aligned` if the total width needed for column headers exceeds the target.

The `html`, `asciidoc`, `latex`, `latex-longtable`, and `troff-ms` formats put out tables that are intended to be included in documents using the respective mark-up language. They are not complete documents! This might not be necessary in HTML, but in LaTeX you must have a complete document wrapper. `latex-longtable` also requires the LaTeX `longtable` and `booktabs` packages.

`linestyle`

Sets the border line drawing style to one of `ascii`, `old-ascii` or `unicode`. Unique abbreviations are allowed. (That would mean one letter is enough.) The default setting is `ascii`. This option only affects the `aligned` and `wrapped` output formats.

`ascii` style uses plain ASCII characters. Newlines in data are shown using a `+` symbol in the right-hand margin. When the `wrapped` format wraps data from one line to the next without a newline character, a dot (`.`) is shown in the right-hand margin of the first line, and again in the left-hand margin of the following line.

`old-ascii` style uses plain ASCII characters, using the formatting style used in PostgreSQL 8.4 and earlier. Newlines in data are shown using a `:` symbol in place of the left-hand column separator. When the data is wrapped from one line to the next without a newline character, a `;` symbol is used in place of the left-hand column separator.

`unicode` style uses Unicode box-drawing characters. Newlines in data are shown using a carriage return symbol in the right-hand margin. When the data is wrapped from one line to the next without a newline character, an ellipsis symbol is shown in the right-hand margin of the first line, and again in the left-hand margin of the following line.

When the `border` setting is greater than zero, the `linestyle` option also determines the characters with which the border lines are drawn. Plain ASCII characters work everywhere, but Unicode characters look nicer on displays that recognize them.

`null`

Sets the string to be printed in place of a null value. The default is to print nothing, which can easily be mistaken for an empty string. For example, one might prefer `\pset null '(null)'`.

`numericlocale`

If *`value`* is specified it must be either `on` or `off` which will enable or disable display of a locale-specific character to separate groups of digits to the left of the decimal marker. If *`value`* is omitted the command toggles between regular and locale-specific numeric output.

`pager`

Controls use of a pager program for query and psql help output. If the environment variable `PAGER` is set, the output is piped to the specified program. Otherwise a platform-dependent default (such as `more`) is used.

When the `pager` option is `off`, the pager program is not used. When the `pager` option is `on`, the pager is used when appropriate, i.e., when the output is to a terminal and will not fit on the screen. The `pager` option can also be set to `always`, which causes the pager to be used for all terminal output regardless of whether it fits on the screen. `\pset pager` without a *`value`* toggles pager use on and off.

`pager_min_lines`

If `pager_min_lines` is set to a number greater than the page height, the pager program will not be called unless there are at least this many lines of output to show. The default setting is 0.

`recordsep`

Specifies the record (line) separator to use in unaligned output format. The default is a newline character.

`recordsep_zero`

Sets the record separator to use in unaligned output format to a zero byte.

`tableattr` (or `T`)

In HTML format, this specifies attributes to be placed inside the `table` tag. This could for example be `cellpadding` or `bgcolor`. Note that you probably don't want to specify `border` here, as that is already taken care of by `\pset border`. If no *`value`* is given, the table attributes are unset.

In `latex-longtable` format, this controls the proportional width of each column containing a left-aligned data type. It is specified as a whitespace-separated list of values, e.g., `'0.2 0.2 0.6'`. Unspecified output columns use the last specified value.

`title`

Sets the table title for any subsequently printed tables. This can be used to give your output descriptive tags. If no *`value`* is given, the title is unset.

`tuples_only` (or `t`)

If *`value`* is specified it must be either `on` or `off` which will enable or disable tuples-only mode. If *`value`* is omitted the command toggles between regular and tuples-only output. Regular output includes extra information such as column headers, titles, and various footers. In tuples-only mode, only actual table data is shown.

`unicode_border_linestyle`

Sets the border drawing style for the `unicode` line style to one of `single` or `double`.

`unicode_column_linestyle`

Sets the column drawing style for the `unicode` line style to one of `single` or `double`.

`unicode_header_linestyle`

Sets the header drawing style for the `unicode` line style to one of `single` or `double`.

Illustrations of how these different formats look can be seen in the *Examples* section.

Tip: There are various shortcut commands for `\pset`. See `\a`, `\C`, `\f`, `\H`, `\t`, `\T`, and `\x`.

`\q` or `\quit`

Quits the psql program. In a script file, only execution of that script is terminated.

`\qecho` *`text`* `[ ... ]`

This command is identical to `\echo` except that the output will be written to the query output channel, as set by `\o`.

`\r` or `\reset`

Resets (clears) the query buffer.

`\s [` *`filename`* `]`

Print psql's command line history to *`filename`*. If *`filename`* is omitted, the history is written to the standard output (using the pager if appropriate). This command is not available if psql was built without Readline support.

`\set [` *`name`* `[` *`value`* `[ ... ] ] ]`

Sets the psql variable *`name`* to *`value`*, or if more than one value is given, to the concatenation of all of them. If only one argument is given, the variable is set with an empty value. To unset a variable, use the `\unset` command.

`\set` without any arguments displays the names and values of all currently-set psql variables.

Valid variable names can contain letters, digits, and underscores. See the section *Variables* below for details. Variable names are case-sensitive.

Although you are welcome to set any variable to anything you want, psql treats several variables as special. They are documented in the section about variables.

Note: This command is unrelated to the SQL command SET.

`\setenv` *name* [*value*]

Sets the environment variable *name* to *value*, or if the *value* is not supplied, unsets the environment variable. Example:

```
testdb=> \setenv PAGER less
testdb=> \setenv LESS -imx4F
```

`\sf[+]` *function_description*

This command fetches and shows the definition of the named function, in the form of a `CREATE OR REPLACE FUNCTION` command. The definition is printed to the current query output channel, as set by `\o`.

The target function can be specified by name alone, or by name and arguments, for example `foo(integer, text)`. The argument types must be given if there is more than one function of the same name.

If + is appended to the command name, then the output lines are numbered, with the first line of the function body being line 1.

`\t`

Toggles the display of output column name headings and row count footer. This command is equivalent to `\pset tuples_only` and is provided for convenience.

`\T` *table_options*

Specifies attributes to be placed within the `table` tag in HTML output format. This command is equivalent to `\pset tableattr` *table_options*.

`\timing` [*on* | *off*]

Without parameter, toggles a display of how long each SQL statement takes, in milliseconds. With parameter, sets same.

`\unset` *name*

Unsets (deletes) the psql variable *name*.

`\w` or `\write` *filename*
`\w` or `\write` `|`*command*

Outputs the current query buffer to the file *filename* or pipes it to the shell command *command*.

`\watch` [*seconds*]

Repeatedly execute the current query buffer (like `\g`) until interrupted or the query fails. Wait the specified number of seconds (default 2) between executions.

`\x` [*on* | *off* | *auto*]

Sets or toggles expanded table formatting mode. As such it is equivalent to `\pset expanded`.

`\z` [*pattern*]

Lists tables, views and sequences with their associated access privileges. If a *pattern* is specified, only tables, views and sequences whose names match the pattern are listed.

This is an alias for `\dp` ("display privileges").

`\! [ command ]`

Escapes to a separate shell or executes the shell command *command*. The arguments are not further interpreted; the shell will see them as-is. In particular, the variable substitution rules and backslash escapes do not apply.

`\? [ topic ]`

Shows help information. The optional *topic* parameter (defaulting to `commands`) selects which part of psql is explained: `commands` describes psql's backslash commands; `options` describes the command-line options that can be passed to psql; and `variables` shows help about psql configuration variables.

Patterns

The various `\d` commands accept a *pattern* parameter to specify the object name(s) to be displayed. In the simplest case, a pattern is just the exact name of the object. The characters within a pattern are normally folded to lower case, just as in SQL names; for example, `\dt FOO` will display the table named `foo`. As in SQL names, placing double quotes around a pattern stops folding to lower case. Should you need to include an actual double quote character in a pattern, write it as a pair of double quotes within a double-quote sequence; again this is in accord with the rules for SQL quoted identifiers. For example, `\dt "FOO""BAR"` will display the table named `FOO"BAR` (not `foo"bar`). Unlike the normal rules for SQL names, you can put double quotes around just part of a pattern, for instance `\dt FOO"FOO"BAR` will display the table named `fooFOObar`.

Whenever the *pattern* parameter is omitted completely, the `\d` commands display all objects that are visible in the current schema search path — this is equivalent to using `*` as the pattern. (An object is said to be *visible* if its containing schema is in the search path and no object of the same kind and name appears earlier in the search path. This is equivalent to the statement that the object can be referenced by name without explicit schema qualification.) To see all objects in the database regardless of visibility, use `*.*` as the pattern.

Within a pattern, `*` matches any sequence of characters (including no characters) and `?` matches any single character. (This notation is comparable to Unix shell file name patterns.) For example, `\dt int*` displays tables whose names begin with `int`. But within double quotes, `*` and `?` lose these special meanings and are just matched literally.

A pattern that contains a dot (`.`) is interpreted as a schema name pattern followed by an object name pattern. For example, `\dt foo*.*bar*` displays all tables whose table name includes `bar` that are in schemas whose schema name starts with `foo`. When no dot appears, then the pattern matches only objects that are visible in the current schema search path. Again, a dot within double quotes loses its special meaning and is matched literally.

Advanced users can use regular-expression notations such as character classes, for example `[0-9]` to match any digit. All regular expression special characters work as specified in Section 9.7.3, except for `.` which is taken as a separator as mentioned above, `*` which is translated to the regular-expression notation `.*`, `?` which is translated to `.`, and `$` which is matched literally. You can emulate these pattern characters at need by writing `?` for `.`, `(R+|)` for `R*`, or `(R|)` for `R?`. `$` is not needed as a regular-expression character since the pattern must match the whole name, unlike the usual interpretation of regular expressions (in other words, `$` is automatically appended to your pattern). Write `*` at the beginning and/or end if you don't wish the pattern to be anchored. Note that within double quotes, all regular expression special characters

lose their special meanings and are matched literally. Also, the regular expression special characters are matched literally in operator name patterns (i.e., the argument of `\do`).

Advanced Features

Variables

psql provides variable substitution features similar to common Unix command shells. Variables are simply name/value pairs, where the value can be any string of any length. The name must consist of letters (including non-Latin letters), digits, and underscores.

To set a variable, use the psql meta-command `\set`. For example,

```
testdb=> \set foo bar
```

sets the variable `foo` to the value `bar`. To retrieve the content of the variable, precede the name with a colon, for example:

```
testdb=> \echo :foo
bar
```

This works in both regular SQL commands and meta-commands; there is more detail in *SQL Interpolation*, below.

If you call `\set` without a second argument, the variable is set, with an empty string as value. To unset (i.e., delete) a variable, use the command `\unset`. To show the values of all variables, call `\set` without any argument.

> **Note:** The arguments of `\set` are subject to the same substitution rules as with other commands. Thus you can construct interesting references such as `\set :foo 'something'` and get "soft links" or "variable variables" of Perl or PHP fame, respectively. Unfortunately (or fortunately?), there is no way to do anything useful with these constructs. On the other hand, `\set bar :foo` is a perfectly valid way to copy a variable.

A number of these variables are treated specially by psql. They represent certain option settings that can be changed at run time by altering the value of the variable, or in some cases represent changeable state of psql. Although you can use these variables for other purposes, this is not recommended, as the program behavior might grow really strange really quickly. By convention, all specially treated variables' names consist of all upper-case ASCII letters (and possibly digits and underscores). To ensure maximum compatibility in the future, avoid using such variable names for your own purposes. A list of all specially treated variables follows.

`AUTOCOMMIT`

> When `on` (the default), each SQL command is automatically committed upon successful completion. To postpone commit in this mode, you must enter a `BEGIN` or `START TRANSACTION` SQL command. When `off` or unset, SQL commands are not committed until you explicitly issue `COMMIT` or `END`. The autocommit-off mode works by issuing an implicit `BEGIN` for you, just before any command that

is not already in a transaction block and is not itself a `BEGIN` or other transaction-control command, nor a command that cannot be executed inside a transaction block (such as `VACUUM`).

> **Note:** In autocommit-off mode, you must explicitly abandon any failed transaction by entering `ABORT` or `ROLLBACK`. Also keep in mind that if you exit the session without committing, your work will be lost.

> **Note:** The autocommit-on mode is PostgreSQL's traditional behavior, but autocommit-off is closer to the SQL spec. If you prefer autocommit-off, you might wish to set it in the system-wide `psqlrc` file or your `~/.psqlrc` file.

`COMP_KEYWORD_CASE`

Determines which letter case to use when completing an SQL key word. If set to `lower` or `upper`, the completed word will be in lower or upper case, respectively. If set to `preserve-lower` or `preserve-upper` (the default), the completed word will be in the case of the word already entered, but words being completed without anything entered will be in lower or upper case, respectively.

`DBNAME`

The name of the database you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

`ECHO`

If set to `all`, all nonempty input lines are printed to standard output as they are read. (This does not apply to lines read interactively.) To select this behavior on program start-up, use the switch `-a`. If set to `queries`, psql prints each query to standard output as it is sent to the server. The switch for this is `-e`. If set to `errors`, then only failed queries are displayed on standard error output. The switch for this is `-b`. If unset, or if set to `none` (or any other value than those above) then no queries are displayed.

`ECHO_HIDDEN`

When this variable is set to `on` and a backslash command queries the database, the query is first shown. This feature helps you to study PostgreSQL internals and provide similar functionality in your own programs. (To select this behavior on program start-up, use the switch `-E`.) If you set the variable to the value `noexec`, the queries are just shown but are not actually sent to the server and executed.

`ENCODING`

The current client character set encoding.

`FETCH_COUNT`

If this variable is set to an integer value > 0, the results of `SELECT` queries are fetched and displayed in groups of that many rows, rather than the default behavior of collecting the entire result set before display. Therefore only a limited amount of memory is used, regardless of the size of the result set. Settings of 100 to 1000 are commonly used when enabling this feature. Keep in mind that when using this feature, a query might fail after having already displayed some rows.

Tip: Although you can use any output format with this feature, the default `aligned` format tends to look bad because each group of `FETCH_COUNT` rows will be formatted separately, leading to varying column widths across the row groups. The other output formats work better.

`HISTCONTROL`

If this variable is set to `ignorespace`, lines which begin with a space are not entered into the history list. If set to a value of `ignoredups`, lines matching the previous history line are not entered. A value of `ignoreboth` combines the two options. If unset, or if set to `none` (or any other value than those above), all lines read in interactive mode are saved on the history list.

Note: This feature was shamelessly plagiarized from Bash.

`HISTFILE`

The file name that will be used to store the history list. The default value is `~/.psql_history`. For example, putting:

```
\set HISTFILE ~/.psql_history- :DBNAME
```

in `~/.psqlrc` will cause psql to maintain a separate history for each database.

Note: This feature was shamelessly plagiarized from Bash.

`HISTSIZE`

The number of commands to store in the command history. The default value is 500.

Note: This feature was shamelessly plagiarized from Bash.

`HOST`

The database server host you are currently connected to. This is set every time you connect to a database (including program start-up), but can be unset.

`IGNOREEOF`

If unset, sending an EOF character (usually **Control+D**) to an interactive session of psql will terminate the application. If set to a numeric value, that many EOF characters are ignored before the application terminates. If the variable is set but has no numeric value, the default is 10.

Note: This feature was shamelessly plagiarized from Bash.

`LASTOID`

The value of the last affected OID, as returned from an `INSERT` or `\lo_import` command. This variable is only guaranteed to be valid until after the result of the next SQL command has been displayed.

`ON_ERROR_ROLLBACK`

When set to `on`, if a statement in a transaction block generates an error, the error is ignored and the transaction continues. When set to `interactive`, such errors are only ignored in interactive sessions, and not when reading script files. When unset or set to `off`, a statement in a transaction block that generates an error aborts the entire transaction. The error rollback mode works by issuing an implicit `SAVEPOINT` for you, just before each command that is in a transaction block, and then rolling back to the savepoint if the command fails.

`ON_ERROR_STOP`

By default, command processing continues after an error. When this variable is set to `on`, processing will instead stop immediately. In interactive mode, psql will return to the command prompt; otherwise, psql will exit, returning error code 3 to distinguish this case from fatal error conditions, which are reported using error code 1. In either case, any currently running scripts (the top-level script, if any, and any other scripts which it may have in invoked) will be terminated immediately. If the top-level command string contained multiple SQL commands, processing will stop with the current command.

`PORT`

The database server port to which you are currently connected. This is set every time you connect to a database (including program start-up), but can be unset.

`PROMPT1`
`PROMPT2`
`PROMPT3`

These specify what the prompts psql issues should look like. See *Prompting* below.

`QUIET`

Setting this variable to `on` is equivalent to the command line option `-q`. It is probably not too useful in interactive mode.

`SINGLELINE`

Setting this variable to `on` is equivalent to the command line option `-S`.

`SINGLESTEP`

Setting this variable to `on` is equivalent to the command line option `-s`.

`USER`

The database user you are currently connected as. This is set every time you connect to a database (including program start-up), but can be unset.

`VERBOSITY`

This variable can be set to the values `default`, `verbose`, or `terse` to control the verbosity of error reports.

SQL Interpolation

A key feature of psql variables is that you can substitute ("interpolate") them into regular SQL statements, as well as the arguments of meta-commands. Furthermore, psql provides facilities for ensuring that vari-

able values used as SQL literals and identifiers are properly quoted. The syntax for interpolating a value without any quoting is to prepend the variable name with a colon (`:`). For example,

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

would query the table `my_table`. Note that this may be unsafe: the value of the variable is copied literally, so it can contain unbalanced quotes, or even backslash commands. You must make sure that it makes sense where you put it.

When a value is to be used as an SQL literal or identifier, it is safest to arrange for it to be quoted. To quote the value of a variable as an SQL literal, write a colon followed by the variable name in single quotes. To quote the value as an SQL identifier, write a colon followed by the variable name in double quotes. These constructs deal correctly with quotes and other special characters embedded within the variable value. The previous example would be more safely written this way:

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :"foo";
```

Variable interpolation will not be performed within quoted SQL literals and identifiers. Therefore, a construction such as `':foo'` doesn't work to produce a quoted literal from a variable's value (and it would be unsafe if it did work, since it wouldn't correctly handle quotes embedded in the value).

One example use of this mechanism is to copy the contents of a file into a table column. First load the file into a variable and then interpolate the variable's value as a quoted string:

```
testdb=> \set content `cat my_file.txt`
testdb=> INSERT INTO my_table VALUES (:'content');
```

(Note that this still won't work if `my_file.txt` contains NUL bytes. psql does not support embedded NUL bytes in variable values.)

Since colons can legally appear in SQL commands, an apparent attempt at interpolation (that is, `:name`, `:'name'`, or `:"name"`) is not replaced unless the named variable is currently set. In any case, you can escape a colon with a backslash to protect it from substitution.

The colon syntax for variables is standard SQL for embedded query languages, such as ECPG. The colon syntaxes for array slices and type casts are PostgreSQL extensions, which can sometimes conflict with the standard usage. The colon-quote syntax for escaping a variable's value as an SQL literal or identifier is a psql extension.

Prompting

The prompts psql issues can be customized to your preference. The three variables `PROMPT1`, `PROMPT2`, and `PROMPT3` contain strings and special escape sequences that describe the appearance of the prompt. Prompt 1 is the normal prompt that is issued when psql requests a new command. Prompt 2 is issued when more input is expected during command entry, for example because the command was not terminated with a semicolon or a quote was not closed. Prompt 3 is issued when you are running an SQL `COPY FROM STDIN` command and you need to type in a row value on the terminal.

The value of the selected prompt variable is printed literally, except where a percent sign (`%`) is encountered. Depending on the next character, certain other text is substituted instead. Defined substitutions are:

`%M`

The full host name (with domain name) of the database server, or `[local]` if the connection is over a Unix domain socket, or `[local:`*`/dir/name`*`]`, if the Unix domain socket is not at the compiled in default location.

`%m`

The host name of the database server, truncated at the first dot, or `[local]` if the connection is over a Unix domain socket.

`%>`

The port number at which the database server is listening.

`%n`

The database session user name. (The expansion of this value might change during a database session as the result of the command `SET SESSION AUTHORIZATION`.)

`%/`

The name of the current database.

`%~`

Like `%/`, but the output is `~` (tilde) if the database is your default database.

`%#`

If the session user is a database superuser, then a `#`, otherwise a `>`. (The expansion of this value might change during a database session as the result of the command `SET SESSION AUTHORIZATION`.)

`%R`

In prompt 1 normally `=`, but `^` if in single-line mode, or `!` if the session is disconnected from the database (which can happen if `\connect` fails). In prompt 2 `%R` is replaced by a character that depends on why psql expects more input: `-` if the command simply wasn't terminated yet, but `*` if there is an unfinished `/* ... */` comment, a single quote if there is an unfinished quoted string, a double quote if there is an unfinished quoted identifier, a dollar sign if there is an unfinished dollar-quoted string, or `(` if there is an unmatched left parenthesis. In prompt 3 `%R` doesn't produce anything.

`%x`

Transaction status: an empty string when not in a transaction block, or `*` when in a transaction block, or `!` when in a failed transaction block, or `?` when the transaction state is indeterminate (for example, because there is no connection).

`%l`

The line number inside the current statement, starting from `1`.

`%`*`digits`*

The character with the indicated octal code is substituted.

`%:`*`name`*`:`

The value of the psql variable *`name`*. See the section *Variables* for details.

`%`*command*`

The output of *command*, similar to ordinary "back-tick" substitution.

`%[ ... %]`

Prompts can contain terminal control characters which, for example, change the color, background, or style of the prompt text, or change the title of the terminal window. In order for the line editing features of Readline to work properly, these non-printing control characters must be designated as invisible by surrounding them with `%[` and `%]`. Multiple pairs of these can occur within the prompt. For example:

```
testdb=> \set PROMPT1 '%[%033[1;33;40m%]%n@%/%R%[%033[0m%]%# '
```

results in a boldfaced (`1;`) yellow-on-black (`33;40`) prompt on VT100-compatible, color-capable terminals.

To insert a percent sign into your prompt, write `%%`. The default prompts are `'%/%R%# '` for prompts 1 and 2, and `'>> '` for prompt 3.

Note: This feature was shamelessly plagiarized from tcsh.

Command-Line Editing

psql supports the Readline library for convenient line editing and retrieval. The command history is automatically saved when psql exits and is reloaded when psql starts up. Tab-completion is also supported, although the completion logic makes no claim to be an SQL parser. The queries generated by tab-completion can also interfere with other SQL commands, e.g., `SET TRANSACTION ISOLATION LEVEL`. If for some reason you do not like the tab completion, you can turn it off by putting this in a file named `.inputrc` in your home directory:

```
$if psql
set disable-completion on
$endif
```

(This is not a psql but a Readline feature. Read its documentation for further details.)

Environment

`COLUMNS`

If `\pset columns` is zero, controls the width for the `wrapped` format and width for determining if wide output requires the pager or should be switched to the vertical format in expanded auto mode.

`PAGER`

If the query results do not fit on the screen, they are piped through this command. Typical values are `more` or `less`. The default is platform-dependent. Use of the pager can be disabled by setting `PAGER` to empty, or by using pager-related options of the `\pset` command.

PGDATABASE
PGHOST
PGPORT
PGUSER

Default connection parameters (see Section 31.14).

PSQL_EDITOR
EDITOR
VISUAL

Editor used by the \e and \ef commands. The variables are examined in the order listed; the first that is set is used.

The built-in default editors are vi on Unix systems and notepad.exe on Windows systems.

PSQL_EDITOR_LINENUMBER_ARG

When \e or \ef is used with a line number argument, this variable specifies the command-line argument used to pass the starting line number to the user's editor. For editors such as Emacs or vi, this is a plus sign. Include a trailing space in the value of the variable if there needs to be space between the option name and the line number. Examples:

```
PSQL_EDITOR_LINENUMBER_ARG='+'
PSQL_EDITOR_LINENUMBER_ARG='--line '
```

The default is + on Unix systems (corresponding to the default editor vi, and useful for many other common editors); but there is no default on Windows systems.

PSQL_HISTORY

Alternative location for the command history file. Tilde (~) expansion is performed.

PSQLRC

Alternative location of the user's .psqlrc file. Tilde (~) expansion is performed.

SHELL

Command executed by the \! command.

TMPDIR

Directory for storing temporary files. The default is /tmp.

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14).

Files

psqlrc and ~/.psqlrc

Unless it is passed an -X or -c option, psql attempts to read and execute commands from the system-wide startup file (psqlrc) and then the user's personal startup file (~/.psqlrc), after connecting to the database but before accepting normal commands. These files can be used to set up the client and/or the server to taste, typically with \set and SET commands.

The system-wide startup file is named `psqlrc` and is sought in the installation's "system configuration" directory, which is most reliably identified by running `pg_config --sysconfdir`. By default this directory will be `../etc/` relative to the directory containing the PostgreSQL executables. The name of this directory can be set explicitly via the `PGSYSCONFDIR` environment variable.

The user's personal startup file is named `.psqlrc` and is sought in the invoking user's home directory. On Windows, which lacks such a concept, the personal startup file is named `%APPDATA%\postgresql\psqlrc.conf`. The location of the user's startup file can be set explicitly via the `PSQLRC` environment variable.

Both the system-wide startup file and the user's personal startup file can be made psql-version-specific by appending a dash and the PostgreSQL major or minor release number to the file name, for example `~/.psqlrc-9.2` or `~/.psqlrc-9.2.5`. The most specific version-matching file will be read in preference to a non-version-specific file.

`.psql_history`

The command-line history is stored in the file `~/.psql_history`, or `%APPDATA%\postgresql\psql_history` on Windows.

The location of the history file can be set explicitly via the `PSQL_HISTORY` environment variable.

Notes

- In an earlier life psql allowed the first argument of a single-letter backslash command to start directly after the command, without intervening whitespace. As of PostgreSQL 8.4 this is no longer allowed.
- psql works best with servers of the same or an older major version. Backslash commands are particularly likely to fail if the server is of a newer version than psql itself. However, backslash commands of the `\d` family should work with servers of versions back to 7.4, though not necessarily with servers newer than psql itself. The general functionality of running SQL commands and displaying query results should also work with servers of a newer major version, but this cannot be guaranteed in all cases.

  If you want to use psql to connect to several servers of different major versions, it is recommended that you use the newest version of psql. Alternatively, you can keep a copy of psql from each major version around and be sure to use the version that matches the respective server. But in practice, this additional complication should not be necessary.

Notes for Windows Users

psql is built as a "console application". Since the Windows console windows use a different encoding than the rest of the system, you must take special care when using 8-bit characters within psql. If psql detects a problematic console code page, it will warn you at startup. To change the console code page, two things are necessary:

- Set the code page by entering **`cmd.exe /c chcp 1252`**. (1252 is a code page that is appropriate for German; replace it with your value.) If you are using Cygwin, you can put this command in `/etc/profile`.

- Set the console font to `Lucida Console`, because the raster font does not work with the ANSI code page.

Examples

The first example shows how to spread a command over several lines of input. Notice the changing prompt:

```
testdb=> CREATE TABLE my_table (
testdb(>  first integer not null default 0,
testdb(>  second text)
testdb-> ;
CREATE TABLE
```

Now look at the table definition again:

```
testdb=> \d my_table
             Table "my_table"
 Attribute |  Type   |      Modifier
-----------+---------+--------------------
 first     | integer | not null default 0
 second    | text    |
```

Now we change the prompt to something more interesting:

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

Let's assume you have filled the table with data and want to take a look at it:

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-------+--------
     1 | one
     2 | two
     3 | three
     4 | four
(4 rows)
```

You can display tables in different ways by using the `\pset` command:

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM my_table;
+-------+--------+
| first | second |
+-------+--------+
|     1 | one    |
|     2 | two    |
|     3 | three  |
|     4 | four   |
+-------+--------+
```

```
(4 rows)

peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
----- ------
    1 one
    2 two
    3 three
    4 four
(4 rows)

peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep ","
Field separator is ",".
peter@localhost testdb=> \pset tuples_only
Showing only tuples.
peter@localhost testdb=> SELECT second, first FROM my_table;
one,1
two,2
three,3
four,4
```

Alternatively, use the short commands:

```
peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

reindexdb

Name

`reindexdb` — reindex a PostgreSQL database

Synopsis

`reindexdb` [*connection-option*...] [*option*...] [`--schema` | `-S` *schema*] ... [`--table` | `-t` *table*] ... [`--index` | `-i` *index*] ... [*dbname*]

`reindexdb` [*connection-option*...] [*option*...] `--all` | `-a`

`reindexdb` [*connection-option*...] [*option*...] `--system` | `-s` [*dbname*]

Description

reindexdb is a utility for rebuilding indexes in a PostgreSQL database.

reindexdb is a wrapper around the SQL command REINDEX. There is no effective difference between reindexing databases via this utility and via other methods for accessing the server.

Options

reindexdb accepts the following command-line arguments:

`-a`
`--all`

Reindex all databases.

`[-d]` *dbname*
`[--dbname=]`*dbname*

Specifies the name of the database to be reindexed, when `-a`/`--all` is not used. If this is not specified, the database name is read from the environment variable `PGDATABASE`. If that is not set, the user name specified for the connection is used. The *dbname* can be a connection string. If so, connection string parameters will override any conflicting command line options.

`-e`
`--echo`

Echo the commands that reindexdb generates and sends to the server.

`-i` *index*
`--index=`*index*

Recreate *index* only. Multiple indexes can be recreated by writing multiple `-i` switches.

`-q`
`--quiet`

Do not display progress messages.

`-s`
`--system`

Reindex database's system catalogs.

`-S` *schema*
`--schema=`*schema*

Reindex *schema* only. Multiple schemas can be reindexed by writing multiple `-S` switches.

`-t` *table*
`--table=`*table*

Reindex *table* only. Multiple tables can be reindexed by writing multiple `-t` switches.

`-v`
`--verbose`

Print detailed information during processing.

`-V`
`--version`

Print the reindexdb version and exit.

`-?`
`--help`

Show help about reindexdb command line arguments, and exit.

reindexdb also accepts the following command-line arguments for connection parameters:

`-h` *host*
`--host=`*host*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *port*
`--port=`*port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *username*
`--username=`*username*

User name to connect as.

`-w`
`--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W`
`--password`

Force reindexdb to prompt for a password before connecting to a database.

This option is never essential, since reindexdb will automatically prompt for a password if the server demands password authentication. However, reindexdb will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

`--maintenance-db=`*dbname*

Specifies the name of the database to connect to to discover which databases should be reindexed, when `-a`/`--all` is used. If not specified, the `postgres` database will be used, or if that does not exist, `template1` will be used. This can be a connection string. If so, connection string parameters will override any conflicting command line options. Also, connection string parameters other than the database name itself will be re-used when connecting to other databases.

Environment

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14).

Diagnostics

In case of difficulty, see REINDEX and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Notes

reindexdb might need to connect several times to the PostgreSQL server, asking for a password each time. It is convenient to have a `~/.pgpass` file in such cases. See Section 31.15 for more information.

Examples

To reindex the database `test`:

```
$ reindexdb test
```

To reindex the table `foo` and the index `bar` in a database named `abcd`:

```
$ reindexdb --table foo --index bar abcd
```

See Also

REINDEX

vacuumdb

Name

vacuumdb — garbage-collect and analyze a PostgreSQL database

Synopsis

vacuumdb [*connection-option*...] [*option*...] [--table | -t *table* [(*column* [,...])]] ... [*dbname*]

vacuumdb [*connection-option*...] [*option*...] --all | -a

Description

vacuumdb is a utility for cleaning a PostgreSQL database. vacuumdb will also generate internal statistics used by the PostgreSQL query optimizer.

vacuumdb is a wrapper around the SQL command VACUUM. There is no effective difference between vacuuming and analyzing databases via this utility and via other methods for accessing the server.

Options

vacuumdb accepts the following command-line arguments:

-a
--all

Vacuum all databases.

[-d] *dbname*
[--dbname=]*dbname*

Specifies the name of the database to be cleaned or analyzed, when -a/--all is not used. If this is not specified, the database name is read from the environment variable PGDATABASE. If that is not set, the user name specified for the connection is used. The *dbname* can be a connection string. If so, connection string parameters will override any conflicting command line options.

-e
--echo

Echo the commands that vacuumdb generates and sends to the server.

-f
--full

Perform “full” vacuuming.

`-F`
`--freeze`

Aggressively “freeze” tuples.

`-j` *`njobs`*
`--jobs=`*`njobs`*

Execute the vacuum or analyze commands in parallel by running *`njobs`* commands simultaneously. This option reduces the time of the processing but it also increases the load on the database server.

vacuumdb will open *`njobs`* connections to the database, so make sure your max_connections setting is high enough to accommodate all connections.

Note that using this mode together with the `-f` (`FULL`) option might cause deadlock failures if certain system catalogs are processed in parallel.

`-q`
`--quiet`

Do not display progress messages.

`-t` *`table`* `[ (`*`column`* `[,...]) ]`
`--table=`*`table`* `[ (`*`column`* `[,...]) ]`

Clean or analyze *`table`* only. Column names can be specified only in conjunction with the `--analyze` or `--analyze-only` options. Multiple tables can be vacuumed by writing multiple `-t` switches.

> **Tip:** If you specify columns, you probably have to escape the parentheses from the shell. (See examples below.)

`-v`
`--verbose`

Print detailed information during processing.

`-V`
`--version`

Print the vacuumdb version and exit.

`-z`
`--analyze`

Also calculate statistics for use by the optimizer.

`-Z`
`--analyze-only`

Only calculate statistics for use by the optimizer (no vacuum).

`--analyze-in-stages`

Only calculate statistics for use by the optimizer (no vacuum), like `--analyze-only`. Run several (currently three) stages of analyze with different configuration settings, to produce usable statistics faster.

This option is useful to analyze a database that was newly populated from a restored dump or by `pg_upgrade`. This option will try to create some statistics as fast as possible, to make the database usable, and then produce full statistics in the subsequent stages.

`-?`
`--help`

Show help about vacuumdb command line arguments, and exit.

vacuumdb also accepts the following command-line arguments for connection parameters:

`-h` *host*
`--host=`*host*

Specifies the host name of the machine on which the server is running. If the value begins with a slash, it is used as the directory for the Unix domain socket.

`-p` *port*
`--port=`*port*

Specifies the TCP port or local Unix domain socket file extension on which the server is listening for connections.

`-U` *username*
`--username=`*username*

User name to connect as.

`-w`
`--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W`
`--password`

Force vacuumdb to prompt for a password before connecting to a database.

This option is never essential, since vacuumdb will automatically prompt for a password if the server demands password authentication. However, vacuumdb will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

`--maintenance-db=`*dbname*

Specifies the name of the database to connect to to discover which databases should be vacuumed, when `-a`/`--all` is used. If not specified, the `postgres` database will be used, or if that does not exist, `template1` will be used. This can be a connection string. If so, connection string parameters will override any conflicting command line options. Also, connection string parameters other than the database name itself will be re-used when connecting to other databases.

Environment

`PGDATABASE`
`PGHOST`
`PGPORT`
`PGUSER`

Default connection parameters

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14).

Diagnostics

In case of difficulty, see VACUUM and psql for discussions of potential problems and error messages. The database server must be running at the targeted host. Also, any default connection settings and environment variables used by the libpq front-end library will apply.

Notes

vacuumdb might need to connect several times to the PostgreSQL server, asking for a password each time. It is convenient to have a `~/.pgpass` file in such cases. See Section 31.15 for more information.

Examples

To clean the database `test`:

```
$ vacuumdb test
```

To clean and analyze for the optimizer a database named `bigdb`:

```
$ vacuumdb --analyze bigdb
```

To clean a single table `foo` in a database named `xyzzy`, and analyze a single column `bar` of the table for the optimizer:

```
$ vacuumdb --analyze --verbose --table 'foo(bar)' xyzzy
```

See Also

VACUUM

III. PostgreSQL Server Applications

This part contains reference information for PostgreSQL server applications and support utilities. These commands can only be run usefully on the host where the database server resides. Other utility programs are listed in Reference II, *PostgreSQL Client Applications*.

initdb

Name

initdb — create a new PostgreSQL database cluster

Synopsis

initdb [*option*...] [--pgdata | -D] *directory*

Description

initdb creates a new PostgreSQL database cluster. A database cluster is a collection of databases that are managed by a single server instance.

Creating a database cluster consists of creating the directories in which the database data will live, generating the shared catalog tables (tables that belong to the whole cluster rather than to any particular database), and creating the template1 and postgres databases. When you later create a new database, everything in the template1 database is copied. (Therefore, anything installed in template1 is automatically copied into each database created later.) The postgres database is a default database meant for use by users, utilities and third party applications.

Although initdb will attempt to create the specified data directory, it might not have permission if the parent directory of the desired data directory is root-owned. To initialize in such a setup, create an empty data directory as root, then use chown to assign ownership of that directory to the database user account, then su to become the database user to run initdb.

initdb must be run as the user that will own the server process, because the server needs to have access to the files and directories that initdb creates. Since the server cannot be run as root, you must not run initdb as root either. (It will in fact refuse to do so.)

initdb initializes the database cluster's default locale and character set encoding. The character set encoding, collation order (LC_COLLATE) and character set classes (LC_CTYPE, e.g., upper, lower, digit) can be set separately for a database when it is created. initdb determines those settings for the template1 database, which will serve as the default for all other databases.

To alter the default collation order or character set classes, use the --lc-collate and --lc-ctype options. Collation orders other than C or POSIX also have a performance penalty. For these reasons it is important to choose the right locale when running initdb.

The remaining locale categories can be changed later when the server is started. You can also use --locale to set the default for all locale categories, including collation order and character set classes. All server locale values (lc_*) can be displayed via SHOW ALL. More details can be found in Section 22.1.

To alter the default encoding, use the --encoding. More details can be found in Section 22.3.

Options

`-A` *authmethod*
`--auth=`*authmethod*

This option specifies the authentication method for local users used in `pg_hba.conf` (`host` and `local` lines). Do not use `trust` unless you trust all local users on your system. `trust` is the default for ease of installation.

`--auth-host=`*authmethod*

This option specifies the authentication method for local users via TCP/IP connections used in `pg_hba.conf` (`host` lines).

`--auth-local=`*authmethod*

This option specifies the authentication method for local users via Unix-domain socket connections used in `pg_hba.conf` (`local` lines).

`-D` *directory*
`--pgdata=`*directory*

This option specifies the directory where the database cluster should be stored. This is the only information required by `initdb`, but you can avoid writing it by setting the `PGDATA` environment variable, which can be convenient since the database server (`postgres`) can find the database directory later by the same variable.

`-E` *encoding*
`--encoding=`*encoding*

Selects the encoding of the template database. This will also be the default encoding of any database you create later, unless you override it there. The default is derived from the locale, or `SQL_ASCII` if that does not work. The character sets supported by the PostgreSQL server are described in Section 22.3.1.

`-k`
`--data-checksums`

Use checksums on data pages to help detect corruption by the I/O system that would otherwise be silent. Enabling checksums may incur a noticeable performance penalty. This option can only be set during initialization, and cannot be changed later. If set, checksums are calculated for all objects, in all databases.

`--locale=`*locale*

Sets the default locale for the database cluster. If this option is not specified, the locale is inherited from the environment that `initdb` runs in. Locale support is described in Section 22.1.

`--lc-collate=`*locale*
`--lc-ctype=`*locale*
`--lc-messages=`*locale*
`--lc-monetary=`*locale*
`--lc-numeric=`*locale*
`--lc-time=`*locale*

Like `--locale`, but only sets the locale in the specified category.

`--no-locale`

Equivalent to `--locale=C`.

`-N`
`--nosync`

By default, `initdb` will wait for all files to be written safely to disk. This option causes `initdb` to return without waiting, which is faster, but means that a subsequent operating system crash can leave the data directory corrupt. Generally, this option is useful for testing, but should not be used when creating a production installation.

`--pwfile=`*`filename`*

Makes `initdb` read the database superuser's password from a file. The first line of the file is taken as the password.

`-S`
`--sync-only`

Safely write all database files to disk and exit. This does not perform any of the normal initdb operations.

`-T` *`CFG`*
`--text-search-config=`*`CFG`*

Sets the default text search configuration. See default_text_search_config for further information.

`-U` *`username`*
`--username=`*`username`*

Selects the user name of the database superuser. This defaults to the name of the effective user running `initdb`. It is really not important what the superuser's name is, but one might choose to keep the customary name postgres, even if the operating system user's name is different.

`-W`
`--pwprompt`

Makes `initdb` prompt for a password to give the database superuser. If you don't plan on using password authentication, this is not important. Otherwise you won't be able to use password authentication until you have a password set up.

`-X` *`directory`*
`--xlogdir=`*`directory`*

This option specifies the directory where the transaction log should be stored.

Other, less commonly used, options are also available:

`-d`
`--debug`

Print debugging output from the bootstrap backend and a few other messages of lesser interest for the general public. The bootstrap backend is the program `initdb` uses to create the catalog tables. This option generates a tremendous amount of extremely boring output.

`-L` *directory*

Specifies where `initdb` should find its input files to initialize the database cluster. This is normally not necessary. You will be told if you need to specify their location explicitly.

`-n`
`--noclean`

By default, when `initdb` determines that an error prevented it from completely creating the database cluster, it removes any files it might have created before discovering that it cannot finish the job. This option inhibits tidying-up and is thus useful for debugging.

Other options:

`-V`
`--version`

Print the initdb version and exit.

`-?`
`--help`

Show help about initdb command line arguments, and exit.

Environment

`PGDATA`

Specifies the directory where the database cluster is to be stored; can be overridden using the `-D` option.

`TZ`

Specifies the default time zone of the created database cluster. The value should be a full time zone name (see Section 8.5.3).

This utility, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14).

Notes

`initdb` can also be invoked via `pg_ctl initdb`.

See Also

pg_ctl, postgres

pg_archivecleanup

Name

pg_archivecleanup — clean up PostgreSQL WAL archive files

Synopsis

pg_archivecleanup [*option*...] *archivelocation oldestkeptwalfile*

Description

pg_archivecleanup is designed to be used as an archive_cleanup_command to clean up WAL file archives when running as a standby server (see Section 25.2). pg_archivecleanup can also be used as a standalone program to clean WAL file archives.

To configure a standby server to use pg_archivecleanup, put this into its recovery.conf configuration file:

```
archive_cleanup_command = 'pg_archivecleanup archivelocation %r'
```

where *archivelocation* is the directory from which WAL segment files should be removed.

When used within archive_cleanup_command, all WAL files logically preceding the value of the %r argument will be removed from *archivelocation*. This minimizes the number of files that need to be retained, while preserving crash-restart capability. Use of this parameter is appropriate if the *archivelocation* is a transient staging area for this particular standby server, but *not* when the *archivelocation* is intended as a long-term WAL archive area, or when multiple standby servers are recovering from the same archive location.

When used as a standalone program all WAL files logically preceding the *oldestkeptwalfile* will be removed from *archivelocation*. In this mode, if you specify a .partial or .backup file name, then only the file prefix will be used as the *oldestkeptwalfile*. This treatment of .backup file name allows you to remove all WAL files archived prior to a specific base backup without error. For example, the following example will remove all files older than WAL file name 000000010000003700000010:

```
pg_archivecleanup -d archive 000000010000003700000010.00000020.backup

pg_archivecleanup:  keep WAL file "archive/000000010000003700000010" and later
pg_archivecleanup:  removing file "archive/00000001000000370000000F"
pg_archivecleanup:  removing file "archive/00000001000000370000000E"
```

pg_archivecleanup assumes that *archivelocation* is a directory readable and writable by the server-owning user.

Options

pg_archivecleanup accepts the following command-line arguments:

`-d`

Print lots of debug logging output on `stderr`.

`-n`

Print the names of the files that would have been removed on `stdout` (performs a dry run).

`-V`
`--version`

Print the pg_archivecleanup version and exit.

`-x` *extension*

When using the program as a standalone utility, provide an extension that will be stripped from all file names before deciding if they should be deleted. This is typically useful for cleaning up archives that have been compressed during storage, and therefore have had an extension added by the compression program. For example: `-x .gz`.

`-?`
`--help`

Show help about pg_archivecleanup command line arguments, and exit.

Notes

pg_archivecleanup is designed to work with PostgreSQL 8.0 and later when used as a standalone utility, or with PostgreSQL 9.0 and later when used as an archive cleanup command.

pg_archivecleanup is written in C and has an easy-to-modify source code, with specifically designated sections to modify for your own needs

Examples

On Linux or Unix systems, you might use:

```
archive_cleanup_command = 'pg_archivecleanup -d /mnt/standby/archive %r 2>>cleanup.log'
```

where the archive directory is physically located on the standby server, so that the `archive_command` is accessing it across NFS, but the files are local to the standby. This will:

- produce debugging output in `cleanup.log`
- remove no-longer-needed files from the archive directory

See Also

pg_standby

pg_controldata

Name

pg_controldata — display control information of a PostgreSQL database cluster

Synopsis

pg_controldata [*option*] [[-D] *datadir*]

Description

pg_controldata prints information initialized during initdb, such as the catalog version. It also shows information about write-ahead logging and checkpoint processing. This information is cluster-wide, and not specific to any one database.

This utility can only be run by the user who initialized the cluster because it requires read access to the data directory. You can specify the data directory on the command line, or use the environment variable PGDATA. This utility supports the options -V and --version, which print the pg_controldata version and exit. It also supports options -? and --help, which output the supported arguments.

Environment

PGDATA

Default data directory location

pg_ctl

Name

pg_ctl — initialize, start, stop, or control a PostgreSQL server

Synopsis

pg_ctl init[db] [-s] [-D *datadir*] [-o *initdb-options*]

pg_ctl start [-w] [-t *seconds*] [-s] [-D *datadir*] [-l *filename*] [-o *options*] [-p *path*] [-c]

pg_ctl stop [-W] [-t *seconds*] [-s] [-D *datadir*] [-m s[mart]|f[ast]|i[mmediate]]

pg_ctl restart [-w] [-t *seconds*] [-s] [-D *datadir*] [-c] [-m s[mart]|f[ast]|i[mmediate]] [-o *options*]

pg_ctl reload [-s] [-D *datadir*]

pg_ctl status [-D *datadir*]

pg_ctl promote [-s] [-D *datadir*]

pg_ctl kill *signal_name process_id*

pg_ctl register [-N *servicename*] [-U *username*] [-P *password*] [-D *datadir*] [-S a[uto]|d[emand]] [-w] [-t *seconds*] [-s] [-o *options*]

pg_ctl unregister [-N *servicename*]

Description

pg_ctl is a utility for initializing a PostgreSQL database cluster, starting, stopping, or restarting the PostgreSQL database server (postgres), or displaying the status of a running server. Although the server can be started manually, pg_ctl encapsulates tasks such as redirecting log output and properly detaching from the terminal and process group. It also provides convenient options for controlled shutdown.

The init or initdb mode creates a new PostgreSQL database cluster. A database cluster is a collection of databases that are managed by a single server instance. This mode invokes the initdb command. See initdb for details.

In `start` mode, a new server is launched. The server is started in the background, and its standard input is attached to `/dev/null` (or `nul` on Windows). On Unix-like systems, by default, the server's standard output and standard error are sent to pg_ctl's standard output (not standard error). The standard output of pg_ctl should then be redirected to a file or piped to another process such as a log rotating program like rotatelogs; otherwise `postgres` will write its output to the controlling terminal (from the background) and will not leave the shell's process group. On Windows, by default the server's standard output and standard error are sent to the terminal. These default behaviors can be changed by using `-l` to append the server's output to a log file. Use of either `-l` or output redirection is recommended.

In `stop` mode, the server that is running in the specified data directory is shut down. Three different shutdown methods can be selected with the `-m` option. "Smart" mode waits for all active clients to disconnect and any online backup to finish. If the server is in hot standby, recovery and streaming replication will be terminated once all clients have disconnected. "Fast" mode (the default) does not wait for clients to disconnect and will terminate an online backup in progress. All active transactions are rolled back and clients are forcibly disconnected, then the server is shut down. "Immediate" mode will abort all server processes immediately, without a clean shutdown. This will lead to a crash-recovery run on the next restart.

`restart` mode effectively executes a stop followed by a start. This allows changing the `postgres` command-line options. `restart` might fail if relative paths specified were specified on the command-line during server start.

`reload` mode simply sends the `postgres` process a SIGHUP signal, causing it to reread its configuration files (`postgresql.conf`, `pg_hba.conf`, etc.). This allows changing of configuration-file options that do not require a complete restart to take effect.

`status` mode checks whether a server is running in the specified data directory. If it is, the PID and the command line options that were used to invoke it are displayed. If the server is not running, the process returns an exit status of 3. If an accessible data directory is not specified, the process returns an exit status of 4.

In `promote` mode, the standby server that is running in the specified data directory is commanded to exit recovery and begin read-write operations.

`kill` mode allows you to send a signal to a specified process. This is particularly valuable for Microsoft Windows which does not have a kill command. Use `--help` to see a list of supported signal names.

`register` mode allows you to register a system service on Microsoft Windows. The `-S` option allows selection of service start type, either "auto" (start service automatically on system startup) or "demand" (start service on demand).

`unregister` mode allows you to unregister a system service on Microsoft Windows. This undoes the effects of the `register` command.

Options

`-c`
`--core-file`

Attempt to allow server crashes to produce core files, on platforms where this is possible, by lifting any soft resource limit placed on core files. This is useful in debugging or diagnosing problems by allowing a stack trace to be obtained from a failed server process.

`-D` *`datadir`*
`--pgdata` *`datadir`*

Specifies the file system location of the database configuration files. If this is omitted, the environment variable `PGDATA` is used.

`-l` *`filename`*
`--log` *`filename`*

Append the server log output to *`filename`*. If the file does not exist, it is created. The umask is set to 077, so access to the log file is disallowed to other users by default.

`-m` *`mode`*
`--mode` *`mode`*

Specifies the shutdown mode. *`mode`* can be `smart`, `fast`, or `immediate`, or the first letter of one of these three. If this is omitted, `fast` is used.

`-o` *`options`*

Specifies options to be passed directly to the `postgres` command; multiple option invocations are appended.

The options should usually be surrounded by single or double quotes to ensure that they are passed through as a group.

`-o` *`initdb-options`*

Specifies options to be passed directly to the `initdb` command.

The options should usually be surrounded by single or double quotes to ensure that they are passed through as a group.

`-p` *`path`*

Specifies the location of the `postgres` executable. By default the `postgres` executable is taken from the same directory as `pg_ctl`, or failing that, the hard-wired installation directory. It is not necessary to use this option unless you are doing something unusual and get errors that the `postgres` executable was not found.

In `init` mode, this option analogously specifies the location of the `initdb` executable.

`-s`
`--silent`

Print only errors, no informational messages.

`-t`
`--timeout`

The maximum number of seconds to wait when waiting for startup or shutdown to complete. Defaults to the value of the `PGCTLTIMEOUT` environment variable or, if not set, to 60 seconds.

`-V`
`--version`

Print the pg_ctl version and exit.

`-w`

Wait for the startup or shutdown to complete. Waiting is the default option for shutdowns, but not startups. When waiting for startup, `pg_ctl` repeatedly attempts to connect to the server. When waiting for shutdown, `pg_ctl` waits for the server to remove its PID file. This option allows the entry of an SSL passphrase on startup. `pg_ctl` returns an exit code based on the success of the startup or shutdown.

`-W`

Do not wait for startup or shutdown to complete. This is the default for start and restart modes.

`-?`
`--help`

Show help about pg_ctl command line arguments, and exit.

Options for Windows

`-e` *source*

Name of the event source for pg_ctl to use for logging to the event log when running as a Windows service. The default is `PostgreSQL`. Note that this only controls the logging from pg_ctl itself; once started, the server will use the event source specified by event_source. Should the server fail during early startup, it might also log using the default event source `PostgreSQL`.

`-N` *servicename*

Name of the system service to register. The name will be used as both the service name and the display name.

`-P` *password*

Password for the user to start the service.

`-S` *start-type*

Start type of the system service to register. start-type can be `auto`, or `demand`, or the first letter of one of these two. If this is omitted, `auto` is used.

`-U` *username*

User name for the user to start the service. For domain users, use the format `DOMAIN\username`.

Environment

`PGCTLTIMEOUT`

Default limit on the number of seconds to wait when waiting for startup or shutdown to complete. If not set, the default is 60 seconds.

`PGDATA`

Default data directory location.

`pg_ctl`, like most other PostgreSQL utilities, also uses the environment variables supported by libpq (see Section 31.14). For additional server variables, see postgres.

Files

`postmaster.pid`

The existence of this file in the data directory is used to help pg_ctl determine if the server is currently running.

`postmaster.opts`

If this file exists in the data directory, pg_ctl (in `restart` mode) will pass the contents of the file as options to postgres, unless overridden by the `-o` option. The contents of this file are also displayed in `status` mode.

Examples

Starting the Server

To start the server:

```
$ pg_ctl start
```

To start the server, waiting until the server is accepting connections:

```
$ pg_ctl -w start
```

To start the server using port 5433, and running without `fsync`, use:

```
$ pg_ctl -o "-F -p 5433" start
```

Stopping the Server

To stop the server, use:

```
$ pg_ctl stop
```

The `-m` option allows control over *how* the server shuts down:

```
$ pg_ctl stop -m fast
```

Restarting the Server

Restarting the server is almost equivalent to stopping the server and starting it again, except that `pg_ctl` saves and reuses the command line options that were passed to the previously running instance. To restart the server in the simplest form, use:

```
$ pg_ctl restart
```

To restart the server, waiting for it to shut down and restart:

```
$ pg_ctl -w restart
```

To restart using port 5433, disabling `fsync` upon restart:

```
$ pg_ctl -o "-F -p 5433" restart
```

Showing the Server Status

Here is sample status output from pg_ctl:

```
$ pg_ctl status
pg_ctl: server is running (PID: 13718)
/usr/local/pgsql/bin/postgres "-D" "/usr/local/pgsql/data" "-p" "5433" "-B" "128"
```

This is the command line that would be invoked in restart mode.

See Also

initdb, postgres

pg_resetxlog

Name

pg_resetxlog — reset the write-ahead log and other control information of a PostgreSQL database cluster

Synopsis

pg_resetxlog [-c *xid*,*xid*] [-f] [-n] [-o *oid*] [-x *xid*] [-e *xid_epoch*] [-m *mxid*,*mxid*] [-O *mxoff*] [-l *xlogfile*] {[-D] *datadir*}

Description

pg_resetxlog clears the write-ahead log (WAL) and optionally resets some other control information stored in the pg_control file. This function is sometimes needed if these files have become corrupted. It should be used only as a last resort, when the server will not start due to such corruption.

After running this command, it should be possible to start the server, but bear in mind that the database might contain inconsistent data due to partially-committed transactions. You should immediately dump your data, run initdb, and reload. After reload, check for inconsistencies and repair as needed.

This utility can only be run by the user who installed the server, because it requires read/write access to the data directory. For safety reasons, you must specify the data directory on the command line. pg_resetxlog does not use the environment variable PGDATA.

If pg_resetxlog complains that it cannot determine valid data for pg_control, you can force it to proceed anyway by specifying the -f (force) option. In this case plausible values will be substituted for the missing data. Most of the fields can be expected to match, but manual assistance might be needed for the next OID, next transaction ID and epoch, next multitransaction ID and offset, and WAL starting address fields. These fields can be set using the options discussed below. If you are not able to determine correct values for all these fields, -f can still be used, but the recovered database must be treated with even more suspicion than usual: an immediate dump and reload is imperative. *Do not* execute any data-modifying operations in the database before you dump, as any such action is likely to make the corruption worse.

The -o, -x, -e, -m, -O, -c and -l options allow the next OID, next transaction ID, next transaction ID's epoch, next and oldest multitransaction ID, next multitransaction offset, oldest and newest transaction IDs for which the commit time can be retrieved, and WAL starting address values to be set manually. These are only needed when pg_resetxlog is unable to determine appropriate values by reading pg_control. Safe values can be determined as follows:

- A safe value for the next transaction ID (-x) can be determined by looking for the numerically largest file name in the directory pg_clog under the data directory, adding one, and then multiplying by 1048576. Note that the file names are in hexadecimal. It is usually easiest to specify the option value in hexadecimal too. For example, if 0011 is the largest entry in pg_clog, -x 0x1200000 will work (five trailing zeroes provide the proper multiplier).

- A safe value for the next multitransaction ID (first part of `-m`) can be determined by looking for the numerically largest file name in the directory `pg_multixact/offsets` under the data directory, adding one, and then multiplying by 65536. Conversely, a safe value for the oldest multitransaction ID (second part of `-m`) can be determined by looking for the numerically smallest file name in the same directory and multiplying by 65536. As above, the file names are in hexadecimal, so the easiest way to do this is to specify the option value in hexadecimal and append four zeroes.
- A safe value for the next multitransaction offset (`-O`) can be determined by looking for the numerically largest file name in the directory `pg_multixact/members` under the data directory, adding one, and then multiplying by 52352. As above, the file names are in hexadecimal. There is no simple recipe such as the ones above of appending zeroes.
- A safe value for the oldest transaction ID for which the commit time can be retrieved (first part of `-c`) can be determined by looking for the numerically smallest file name in the directory `pg_commit_ts` under the data directory. Conversely, a safe value for the newest transaction ID for which the commit time can be retrieved (second part of `-c`) can be determined by looking for the numerically greatest file name in the same directory. As above, the file names are in hexadecimal.
- The WAL starting address (`-l`) should be larger than any WAL segment file name currently existing in the directory `pg_xlog` under the data directory. These names are also in hexadecimal and have three parts. The first part is the "timeline ID" and should usually be kept the same. For example, if `000000010000000320000004A` is the largest entry in `pg_xlog`, use `-l 000000010000000320000004B` or higher.

  > **Note:** `pg_resetxlog` itself looks at the files in `pg_xlog` and chooses a default `-l` setting beyond the last existing file name. Therefore, manual adjustment of `-l` should only be needed if you are aware of WAL segment files that are not currently present in `pg_xlog`, such as entries in an offline archive; or if the contents of `pg_xlog` have been lost entirely.

- There is no comparably easy way to determine a next OID that's beyond the largest one in the database, but fortunately it is not critical to get the next-OID setting right.
- The transaction ID epoch is not actually stored anywhere in the database except in the field that is set by `pg_resetxlog`, so any value will work so far as the database itself is concerned. You might need to adjust this value to ensure that replication systems such as Slony-I and Skytools work correctly — if so, an appropriate value should be obtainable from the state of the downstream replicated database.

The `-n` (no operation) option instructs `pg_resetxlog` to print the values reconstructed from `pg_control` and values about to be changed, and then exit without modifying anything. This is mainly a debugging tool, but can be useful as a sanity check before allowing `pg_resetxlog` to proceed for real.

The `-V` and `--version` options print the pg_resetxlog version and exit. The options `-?` and `--help` show supported arguments, and exit.

Notes

This command must not be used when the server is running. `pg_resetxlog` will refuse to start up if it finds a server lock file in the data directory. If the server crashed then a lock file might have been left behind; in that case you can remove the lock file to allow `pg_resetxlog` to run. But before you do so, make doubly certain that there is no server process still alive.

`pg_resetxlog` works only with servers of the same major version.

pg_rewind

Name

pg_rewind — synchronize a PostgreSQL data directory with another data directory that was forked from the first one

Synopsis

pg_rewind [*option*...] {-D | --target-pgdata} *directory* {--source-pgdata=*directory* | --source-server=*connstr*}

Description

pg_rewind is a tool for synchronizing a PostgreSQL cluster with another copy of the same cluster, after the clusters' timelines have diverged. A typical scenario is to bring an old master server back online after failover, as a standby that follows the new master.

The result is equivalent to replacing the target data directory with the source one. All files are copied, including configuration files. The advantage of pg_rewind over taking a new base backup, or tools like rsync, is that pg_rewind does not require reading through all unchanged files in the cluster. That makes it a lot faster when the database is large and only a small portion of it differs between the clusters.

pg_rewind examines the timeline histories of the source and target clusters to determine the point where they diverged, and expects to find WAL in the target cluster's pg_xlog directory reaching all the way back to the point of divergence. In the typical failover scenario where the target cluster was shut down soon after the divergence, that is not a problem, but if the target cluster had run for a long time after the divergence, the old WAL files might not be present anymore. In that case, they can be manually copied from the WAL archive to the pg_xlog directory. Fetching missing files from a WAL archive automatically is currently not supported.

When the target server is started up for the first time after running pg_rewind, it will go into recovery mode and replay all WAL generated in the source server after the point of divergence. If some of the WAL was no longer available in the source server when pg_rewind was run, and therefore could not be copied by pg_rewind session, it needs to be made available when the target server is started up. That can be done by creating a recovery.conf file in the target data directory with a suitable restore_command.

pg_rewind requires that the target server either has the wal_log_hints option is enabled in postgresql.conf or that data checksums were enabled when the cluster was initialized with initdb. Neither of these are currently on by default. full_page_writes must also be enabled. That is the default.

> **Warning**
>
> If pg_rewind fails while processing, then the data folder of the target is likely not in a state that can be recovered. In such a case, taking a new fresh backup is recommended.
>
> pg_rewind will fail immediately if it finds files it cannot write directly to. This can happen for example when the source and the target server use the same file mapping for read-only SSL keys and certificates. If such files are present on the target server it is recommended to remove them before running pg_rewind. After doing the rewind, some of those files may have been copied from the source, in which case it may be necessary to remove the data copied and restore back the set of links used before the rewind.

Options

pg_rewind accepts the following command-line arguments:

`-D` *`directory`*
`--target-pgdata=`*`directory`*

This option specifies the target data directory that is synchronized with the source. The target server must shut down cleanly before running pg_rewind

`--source-pgdata=`*`directory`*

Specifies path to the data directory of the source server, to synchronize the target with. When `--source-pgdata` is used, the source server must be cleanly shut down.

`--source-server=`*`connstr`*

Specifies a libpq connection string to connect to the source PostgreSQL server to synchronize the target with. The connection must be a normal (non-replication) connection with superuser access. The server must be up and running, and must not be in recovery mode.

`-n`
`--dry-run`

Do everything except actually modifying the target directory.

`-P`
`--progress`

Enables progress reporting. Turning this on will deliver an approximate progress report while copying data over from the source cluster.

`--debug`

Print verbose debugging output that is mostly useful for developers debugging pg_rewind.

`-V`
`--version`

Display version information, then exit.

`-?`
`--help`

Show help, then exit.

Environment

When `--source-server` option is used, pg_rewind also uses the environment variables supported by libpq (see Section 31.14).

Notes

When executing pg_rewind using an online cluster as source which has been recently promoted, it is necessary to execute a `CHECKPOINT` after promotion so as its control file reflects up-to-date timeline information, which is used by pg_rewind to check if the target cluster can be rewound using the designated source cluster.

How it works

The basic idea is to copy everything from the new cluster to the old cluster, except for the blocks that we know to be the same.

1. Scan the WAL log of the old cluster, starting from the last checkpoint before the point where the new cluster's timeline history forked off from the old cluster. For each WAL record, make a note of the data blocks that were touched. This yields a list of all the data blocks that were changed in the old cluster, after the new cluster forked off.
2. Copy all those changed blocks from the new cluster to the old cluster.
3. Copy all other files such as `clog` and configuration files from the new cluster to the old cluster, everything except the relation files.
4. Apply the WAL from the new cluster, starting from the checkpoint created at failover. (Strictly speaking, pg_rewind doesn't apply the WAL, it just creates a backup label file indicating that when PostgreSQL is started, it will start replay from that checkpoint and apply all the required WAL.)

pg_test_fsync

Name

pg_test_fsync — determine fastest wal_sync_method for PostgreSQL

Synopsis

pg_test_fsync [*option*...]

Description

pg_test_fsync is intended to give you a reasonable idea of what the fastest wal_sync_method is on your specific system, as well as supplying diagnostic information in the event of an identified I/O problem. However, differences shown by pg_test_fsync might not make any significant difference in real database throughput, especially since many database servers are not speed-limited by their transaction logs. pg_test_fsync reports average file sync operation time in microseconds for each wal_sync_method, which can also be used to inform efforts to optimize the value of commit_delay.

Options

pg_test_fsync accepts the following command-line options:

-f
--filename

Specifies the file name to write test data in. This file should be in the same file system that the pg_xlog directory is or will be placed in. (pg_xlog contains the WAL files.) The default is pg_test_fsync.out in the current directory.

-s
--secs-per-test

Specifies the number of seconds for each test. The more time per test, the greater the test's accuracy, but the longer it takes to run. The default is 5 seconds, which allows the program to complete in under 2 minutes.

-V
--version

Print the pg_test_fsync version and exit.

-?
--help

Show help about pg_test_fsync command line arguments, and exit.

See Also

postgres

pg_test_timing

Name

pg_test_timing — measure timing overhead

Synopsis

pg_test_timing [*option*...]

Description

pg_test_timing is a tool to measure the timing overhead on your system and confirm that the system time never moves backwards. Systems that are slow to collect timing data can give less accurate EXPLAIN ANALYZE results.

Options

pg_test_timing accepts the following command-line options:

-d *duration*
--duration=*duration*

Specifies the test duration, in seconds. Longer durations give slightly better accuracy, and are more likely to discover problems with the system clock moving backwards. The default test duration is 3 seconds.

-V
--version

Print the pg_test_timing version and exit.

-?
--help

Show help about pg_test_timing command line arguments, and exit.

Usage

Interpreting results

Good results will show most (>90%) individual timing calls take less than one microsecond. Average per loop overhead will be even lower, below 100 nanoseconds. This example from an Intel i7-860 system

using a TSC clock source shows excellent performance:

```
Testing timing overhead for 3 seconds.
Per loop time including overhead: 35.96 nsec
Histogram of timing durations:
< usec   % of total      count
     1     96.40465   80435604
     2      3.59518    2999652
     4      0.00015        126
     8      0.00002         13
    16      0.00000          2
```

Note that different units are used for the per loop time than the histogram. The loop can have resolution within a few nanoseconds (nsec), while the individual timing calls can only resolve down to one microsecond (usec).

Measuring executor timing overhead

When the query executor is running a statement using `EXPLAIN ANALYZE`, individual operations are timed as well as showing a summary. The overhead of your system can be checked by counting rows with the psql program:

```
CREATE TABLE t AS SELECT * FROM generate_series(1,100000);
\timing
SELECT COUNT(*) FROM t;
EXPLAIN ANALYZE SELECT COUNT(*) FROM t;
```

The i7-860 system measured runs the count query in 9.8 ms while the `EXPLAIN ANALYZE` version takes 16.6 ms, each processing just over 100,000 rows. That 6.8 ms difference means the timing overhead per row is 68 ns, about twice what pg_test_timing estimated it would be. Even that relatively small amount of overhead is making the fully timed count statement take almost 70% longer. On more substantial queries, the timing overhead would be less problematic.

Changing time sources

On some newer Linux systems, it's possible to change the clock source used to collect timing data at any time. A second example shows the slowdown possible from switching to the slower acpi_pm time source, on the same system used for the fast results above:

```
# cat /sys/devices/system/clocksource/clocksource0/available_clocksource
tsc hpet acpi_pm
# echo acpi_pm > /sys/devices/system/clocksource/clocksource0/current_clocksource
# pg_test_timing
Per loop time including overhead: 722.92 nsec
Histogram of timing durations:
< usec   % of total      count
     1     27.84870    1155682
```

```
     2     72.05956    2990371
     4      0.07810       3241
     8      0.01357        563
    16      0.00007          3
```

In this configuration, the sample `EXPLAIN ANALYZE` above takes 115.9 ms. That's 1061 nsec of timing overhead, again a small multiple of what's measured directly by this utility. That much timing overhead means the actual query itself is only taking a tiny fraction of the accounted for time, most of it is being consumed in overhead instead. In this configuration, any `EXPLAIN ANALYZE` totals involving many timed operations would be inflated significantly by timing overhead.

FreeBSD also allows changing the time source on the fly, and it logs information about the timer selected during boot:

```
# dmesg | grep "Timecounter"
Timecounter "ACPI-fast" frequency 3579545 Hz quality 900
Timecounter "i8254" frequency 1193182 Hz quality 0
Timecounters tick every 10.000 msec
Timecounter "TSC" frequency 2531787134 Hz quality 800
# sysctl kern.timecounter.hardware=TSC
kern.timecounter.hardware: ACPI-fast -> TSC
```

Other systems may only allow setting the time source on boot. On older Linux systems the "clock" kernel setting is the only way to make this sort of change. And even on some more recent ones, the only option you'll see for a clock source is "jiffies". Jiffies are the older Linux software clock implementation, which can have good resolution when it's backed by fast enough timing hardware, as in this example:

```
$ cat /sys/devices/system/clocksource/clocksource0/available_clocksource
jiffies
$ dmesg | grep time.c
time.c: Using 3.579545 MHz WALL PM GTOD PIT/TSC timer.
time.c: Detected 2400.153 MHz processor.
$ pg_test_timing
Testing timing overhead for 3 seconds.
Per timing duration including loop overhead: 97.75 ns
Histogram of timing durations:
< usec   % of total      count
     1     90.23734   27694571
     2      9.75277    2993204
     4      0.00981       3010
     8      0.00007         22
    16      0.00000          1
    32      0.00000          1
```

Clock hardware and timing accuracy

Collecting accurate timing information is normally done on computers using hardware clocks with various levels of accuracy. With some hardware the operating systems can pass the system clock time almost directly to programs. A system clock can also be derived from a chip that simply provides timing interrupts, periodic ticks at some known time interval. In either case, operating system kernels provide a clock source that hides these details. But the accuracy of that clock source and how quickly it can return results varies based on the underlying hardware.

Inaccurate time keeping can result in system instability. Test any change to the clock source very carefully. Operating system defaults are sometimes made to favor reliability over best accuracy. And if you are using a virtual machine, look into the recommended time sources compatible with it. Virtual hardware faces additional difficulties when emulating timers, and there are often per operating system settings suggested by vendors.

The Time Stamp Counter (TSC) clock source is the most accurate one available on current generation CPUs. It's the preferred way to track the system time when it's supported by the operating system and the TSC clock is reliable. There are several ways that TSC can fail to provide an accurate timing source, making it unreliable. Older systems can have a TSC clock that varies based on the CPU temperature, making it unusable for timing. Trying to use TSC on some older multicore CPUs can give a reported time that's inconsistent among multiple cores. This can result in the time going backwards, a problem this program checks for. And even the newest systems can fail to provide accurate TSC timing with very aggressive power saving configurations.

Newer operating systems may check for the known TSC problems and switch to a slower, more stable clock source when they are seen. If your system supports TSC time but doesn't default to that, it may be disabled for a good reason. And some operating systems may not detect all the possible problems correctly, or will allow using TSC even in situations where it's known to be inaccurate.

The High Precision Event Timer (HPET) is the preferred timer on systems where it's available and TSC is not accurate. The timer chip itself is programmable to allow up to 100 nanosecond resolution, but you may not see that much accuracy in your system clock.

Advanced Configuration and Power Interface (ACPI) provides a Power Management (PM) Timer, which Linux refers to as the acpi_pm. The clock derived from acpi_pm will at best provide 300 nanosecond resolution.

Timers used on older PC hardware include the 8254 Programmable Interval Timer (PIT), the real-time clock (RTC), the Advanced Programmable Interrupt Controller (APIC) timer, and the Cyclone timer. These timers aim for millisecond resolution.

See Also

EXPLAIN

pg_upgrade

Name

pg_upgrade — upgrade a PostgreSQL server instance

Synopsis

```
pg_upgrade -b oldbindir -B newbindir -d olddatadir -D newdatadir [option...]
```

Description

pg_upgrade (formerly called pg_migrator) allows data stored in PostgreSQL data files to be upgraded to a later PostgreSQL major version without the data dump/reload typically required for major version upgrades, e.g., from 8.4.7 to the current major release of PostgreSQL. It is not required for minor version upgrades, e.g., from 9.0.1 to 9.0.4.

Major PostgreSQL releases regularly add new features that often change the layout of the system tables, but the internal data storage format rarely changes. pg_upgrade uses this fact to perform rapid upgrades by creating new system tables and simply reusing the old user data files. If a future major release ever changes the data storage format in a way that makes the old data format unreadable, pg_upgrade will not be usable for such upgrades. (The community will attempt to avoid such situations.)

pg_upgrade does its best to make sure the old and new clusters are binary-compatible, e.g., by checking for compatible compile-time settings, including 32/64-bit binaries. It is important that any external modules are also binary compatible, though this cannot be checked by pg_upgrade.

pg_upgrade supports upgrades from 8.4.X and later to the current major release of PostgreSQL, including snapshot and alpha releases.

Options

pg_upgrade accepts the following command-line arguments:

`-b` *bindir*
`--old-bindir=`*bindir*

the old PostgreSQL executable directory; environment variable `PGBINOLD`

`-B` *bindir*
`--new-bindir=`*bindir*

the new PostgreSQL executable directory; environment variable `PGBINNEW`

`-c`
`--check`

check clusters only, don't change any data

`-d` *datadir*
`--old-datadir=`*datadir*

the old cluster data directory; environment variable `PGDATAOLD`

`-D` *datadir*
`--new-datadir=`*datadir*

the new cluster data directory; environment variable `PGDATANEW`

`-j` *njobs*
`--jobs=`*njobs*

number of simultaneous processes or threads to use

`-k`
`--link`

use hard links instead of copying files to the new cluster

`-o` *options*
`--old-options` *options*

options to be passed directly to the old `postgres` command; multiple option invocations are appended

`-O` *options*
`--new-options` *options*

options to be passed directly to the new `postgres` command; multiple option invocations are appended

`-p` *port*
`--old-port=`*port*

the old cluster port number; environment variable `PGPORTOLD`

`-P` *port*
`--new-port=`*port*

the new cluster port number; environment variable `PGPORTNEW`

`-r`
`--retain`

retain SQL and log files even after successful completion

`-U` *username*
`--username=`*username*

cluster's install user name; environment variable `PGUSER`

`-v`
`--verbose`

enable verbose internal logging

`-V`
`--version`

display version information, then exit

`-?`
`--help`

show help, then exit

Usage

These are the steps to perform an upgrade with pg_upgrade:

1. Optionally move the old cluster

   If you are using a version-specific installation directory, e.g., `/opt/PostgreSQL/9.1`, you do not need to move the old cluster. The graphical installers all use version-specific installation directories.

   If your installation directory is not version-specific, e.g., `/usr/local/pgsql`, it is necessary to move the current PostgreSQL install directory so it does not interfere with the new PostgreSQL installation. Once the current PostgreSQL server is shut down, it is safe to rename the PostgreSQL installation directory; assuming the old directory is `/usr/local/pgsql`, you can do:

   ```
   mv /usr/local/pgsql /usr/local/pgsql.old
   ```
   to rename the directory.

2. For source installs, build the new version

   Build the new PostgreSQL source with `configure` flags that are compatible with the old cluster. pg_upgrade will check `pg_controldata` to make sure all settings are compatible before starting the upgrade.

3. Install the new PostgreSQL binaries

   Install the new server's binaries and support files. pg_upgrade is included in a default installation.

   For source installs, if you wish to install the new server in a custom location, use the `prefix` variable:

   ```
   make prefix=/usr/local/pgsql.new install
   ```

4. Initialize the new PostgreSQL cluster

   Initialize the new cluster using `initdb`. Again, use compatible `initdb` flags that match the old cluster. Many prebuilt installers do this step automatically. There is no need to start the new cluster.

5. Install custom shared object files

   Install any custom shared object files (or DLLs) used by the old cluster into the new cluster, e.g., `pgcrypto.so`, whether they are from `contrib` or some other source. Do not install the schema definitions, e.g., `CREATE EXTENSION pgcrypto`, because these will be upgraded from the old cluster. Also, any custom full text search files (dictionary, synonym, thesaurus, stop words) must also be copied to the new cluster.

6. Adjust authentication

   `pg_upgrade` will connect to the old and new servers several times, so you might want to set authentication to `peer` in `pg_hba.conf` or use a `~/.pgpass` file (see Section 31.15).

7. Stop both servers

Make sure both database servers are stopped using, on Unix, e.g.:

```
pg_ctl -D /opt/PostgreSQL/8.4 stop
pg_ctl -D /opt/PostgreSQL/9.0 stop
```

or on Windows, using the proper service names:

```
NET STOP postgresql-8.4
NET STOP postgresql-9.0
```

Streaming replication and log-shipping standby servers can remain running until a later step.

8. Prepare for standby server upgrades

   If you are upgrading standby servers using methods outlined in section step 10, verify that the old standby servers are caught up by running pg_controldata against the old primary and standby clusters. Verify that the "Latest checkpoint location" values match in all clusters. (There will be a mismatch if old standby servers were shut down before the old primary or if the old standby servers are still running.) Also, make sure `wal_level` is not set to `minimal` in the `postgresql.conf` file on the new primary cluster.

9. Run pg_upgrade

   Always run the pg_upgrade binary of the new server, not the old one. pg_upgrade requires the specification of the old and new cluster's data and executable (`bin`) directories. You can also specify user and port values, and whether you want the data linked instead of copied (the default).

   If you use link mode, the upgrade will be much faster (no file copying) and use less disk space, but you will not be able to access your old cluster once you start the new cluster after the upgrade. Link mode also requires that the old and new cluster data directories be in the same file system. (Tablespaces and `pg_xlog` can be on different file systems.) See `pg_upgrade --help` for a full list of options.

   The `--jobs` option allows multiple CPU cores to be used for copying/linking of files and to dump and reload database schemas in parallel; a good place to start is the maximum of the number of CPU cores and tablespaces. This option can dramatically reduce the time to upgrade a multi-database server running on a multiprocessor machine.

   For Windows users, you must be logged into an administrative account, and then start a shell as the `postgres` user and set the proper path:

   ```
   RUNAS /USER:postgres "CMD.EXE"
   SET PATH=%PATH%;C:\Program Files\PostgreSQL\9.0\bin;
   ```

   and then run pg_upgrade with quoted directories, e.g.:

   ```
   pg_upgrade.exe
           --old-datadir "C:/Program Files/PostgreSQL/8.4/data"
           --new-datadir "C:/Program Files/PostgreSQL/9.0/data"
           --old-bindir "C:/Program Files/PostgreSQL/8.4/bin"
           --new-bindir "C:/Program Files/PostgreSQL/9.0/bin"
   ```

   Once started, `pg_upgrade` will verify the two clusters are compatible and then do the upgrade. You can use `pg_upgrade --check` to perform only the checks, even if the old server is still running. `pg_upgrade --check` will also outline any manual adjustments you will need to make after the upgrade. If you are going to be using link mode, you should use the `--link` option with `--check` to enable link-mode-specific checks. `pg_upgrade` requires write permission in the current directory.

   Obviously, no one should be accessing the clusters during the upgrade. pg_upgrade defaults to running servers on port 50432 to avoid unintended client connections. You can use the same port number for both clusters when doing an upgrade because the old and new clusters will not be running at the

same time. However, when checking an old running server, the old and new port numbers must be different.

If an error occurs while restoring the database schema, `pg_upgrade` will exit and you will have to revert to the old cluster as outlined in step 16 below. To try `pg_upgrade` again, you will need to modify the old cluster so the pg_upgrade schema restore succeeds. If the problem is a `contrib` module, you might need to uninstall the `contrib` module from the old cluster and install it in the new cluster after the upgrade, assuming the module is not being used to store user data.

10. Upgrade Streaming Replication and Log-Shipping standby servers

    If you used link mode and have Streaming Replication (see Section 25.2.5) or Log-Shipping (see Section 25.2) standby servers, you can follow these steps to quickly upgrade them. You will not be running pg_upgrade on the standby servers, but rather rsync on the primary. Do not start any servers yet.

    If you did *not* use link mode, do not have or do not want to use rsync, or want an easier solution, skip the instructions in this section and simply recreate the standby servers once pg_upgrade completes and the new primary is running.

    a. Install the new PostgreSQL binaries on standby servers

       Make sure the new binaries and support files are installed on all standby servers.

    b. Make sure the new standby data directories do *not* exist

       Make sure the new standby data directories do *not* exist or are empty. If initdb was run, delete the standby servers' new data directories.

    c. Install custom shared object files

       Install the same custom shared object files on the new standbys that you installed in the new primary cluster.

    d. Stop standby servers

       If the standby servers are still running, stop them now using the above instructions.

    e. Save configuration files

       Save any configuration files from the old standbys' configuration directories you need to keep, e.g., `postgresql.conf` (and any files included by it), `postgresql.auto.conf`, `recovery.conf`, `pg_hba.conf`, because these will be overwritten or removed in the next step.

    f. Run rsync

       When using link mode, standby servers can be quickly upgraded using rsync. To accomplish this, from a directory on the primary server that is above the old and new database cluster directories, run this on the *primary* for each standby server:

       ```
       rsync --archive --delete --hard-links --size-only --no-inc-recursive old_cluster
       ```

       where `old_cluster` and `new_cluster` are relative to the current directory on the primary, and `remote_dir` is *above* the old and new cluster directories on the standby. The directory structure under the specified directories on the primary and standbys must match. Consult the rsync manual page for details on specifying the remote directory, e.g.,

       ```
       rsync --archive --delete --hard-links --size-only --no-inc-recursive /opt/Postgre
             /opt/PostgreSQL/9.6 standby.example.com:/opt/PostgreSQL
       ```

You can verify what the command will do using rsync's `--dry-run` option. While rsync must be run on the primary for at least one standby, it is possible to run rsync on an upgraded standby to upgrade other standbys, as long as the upgraded standby has not been started.

What this does is to record the links created by pg_upgrade's link mode that connect files in the old and new clusters on the primary server. It then finds matching files in the standby's old cluster and creates links for them in the standby's new cluster. Files that were not linked on the primary are copied from the primary to the standby. (They are usually small.) This provides rapid standby upgrades. Unfortunately, rsync needlessly copies files associated with temporary and unlogged tables because these files don't normally exist on standby servers.

If you have tablespaces, you will need to run a similar rsync command for each tablespace directory, e.g.:

```
rsync --archive --delete --hard-links --size-only --no-inc-recursive /vol1/pg_tbl
      /vol1/pg_tblsp/PG_9.6_201608131 standby.example.com:/vol1/pg_tblsp
```

If you have relocated `pg_xlog` outside the data directories, rsync must be run on those directories too.

g. Configure streaming replication and log-shipping standby servers

Configure the servers for log shipping. (You do not need to run `pg_start_backup()` and `pg_stop_backup()` or take a file system backup as the standbys are still synchronized with the primary.)

11. Restore `pg_hba.conf`

If you modified `pg_hba.conf`, restore its original settings. It might also be necessary to adjust other configuration files in the new cluster to match the old cluster, e.g., `postgresql.conf` (and any files included by it), `postgresql.auto.conf`.

12. Start the new server

The new server can now be safely started, and then any rsync'ed standby servers.

13. Post-Upgrade processing

If any post-upgrade processing is required, pg_upgrade will issue warnings as it completes. It will also generate script files that must be run by the administrator. The script files will connect to each database that needs post-upgrade processing. Each script should be run using:

```
psql --username postgres --file script.sql postgres
```

The scripts can be run in any order and can be deleted once they have been run.

> **Caution**
>
> In general it is unsafe to access tables referenced in rebuild scripts until the rebuild scripts have run to completion; doing so could yield incorrect results or poor performance. Tables not referenced in rebuild scripts can be accessed immediately.

14. Statistics

Because optimizer statistics are not transferred by `pg_upgrade`, you will be instructed to run a command to regenerate that information at the end of the upgrade. You might need to set connection parameters to match your new cluster.

15. Delete old cluster

    Once you are satisfied with the upgrade, you can delete the old cluster's data directories by running the script mentioned when `pg_upgrade` completes. (Automatic deletion is not possible if you have user-defined tablespaces inside the old data directory.) You can also delete the old installation directories (e.g., `bin`, `share`).

16. Reverting to old cluster

    If, after running `pg_upgrade`, you wish to revert to the old cluster, there are several options:

    - If the `--check` option was used, the old cluster was unmodified; it can be restarted.
    - If the `--link` option was *not* used, the old cluster was unmodified; it can be restarted.
    - If the `--link` option was used, the data files might be shared between the old and new cluster:
      - If `pg_upgrade` aborted before linking started, the old cluster was unmodified; it can be restarted.
      - If you did *not* start the new cluster, the old cluster was unmodified except that, when linking started, a `.old` suffix was appended to `$PGDATA/global/pg_control`. To reuse the old cluster, remove the `.old` suffix from `$PGDATA/global/pg_control`; you can then restart the old cluster.
      - If you did start the new cluster, it has written to shared files and it is unsafe to use the old cluster. The old cluster will need to be restored from backup in this case.

Notes

pg_upgrade does not support upgrading of databases containing these `reg*` OID-referencing system data types: `regproc`, `regprocedure`, `regoper`, `regoperator`, `regconfig`, and `regdictionary`. (`regtype` can be upgraded.)

All failure, rebuild, and reindex cases will be reported by pg_upgrade if they affect your installation; post-upgrade scripts to rebuild tables and indexes will be generated automatically. If you are trying to automate the upgrade of many clusters, you should find that clusters with identical database schemas require the same post-upgrade steps for all cluster upgrades; this is because the post-upgrade steps are based on the database schemas, and not user data.

For deployment testing, create a schema-only copy of the old cluster, insert dummy data, and upgrade that.

If you are upgrading a pre-PostgreSQL 9.2 cluster that uses a configuration-file-only directory, you must pass the real data directory location to pg_upgrade, and pass the configuration directory location to the server, e.g., `-d /real-data-directory -o '-D /configuration-directory'`.

If using a pre-9.1 old server that is using a non-default Unix-domain socket directory or a default that differs from the default of the new cluster, set `PGHOST` to point to the old server's socket location. (This is not relevant on Windows.)

If you want to use link mode and you do not want your old cluster to be modified when the new cluster is started, make a copy of the old cluster and upgrade that in link mode. To make a valid copy of the old cluster, use `rsync` to create a dirty copy of the old cluster while the server is running, then shut down the

old server and run `rsync --checksum` again to update the copy with any changes to make it consistent. (`--checksum` is necessary because `rsync` only has file modification-time granularity of one second.) You might want to exclude some files, e.g., `postmaster.pid`, as documented in Section 24.3.3. If your file system supports file system snapshots or copy-on-write file copies, you can use that to make a backup of the old cluster and tablespaces, though the snapshot and copies must be created simultaneously or while the database server is down.

See Also

initdb, pg_ctl, pg_dump, postgres

pg_xlogdump

Name

pg_xlogdump — display a human-readable rendering of the write-ahead log of a PostgreSQL database cluster

Synopsis

pg_xlogdump [option...] [startseg [endseg]]

Description

pg_xlogdump displays the write-ahead log (WAL) and is mainly useful for debugging or educational purposes.

This utility can only be run by the user who installed the server, because it requires read-only access to the data directory.

Options

The following command-line options control the location and format of the output:

startseg

Start reading at the specified log segment file. This implicitly determines the path in which files will be searched for, and the timeline to use.

endseg

Stop after reading the specified log segment file.

-b
--bkp-details

Output detailed information about backup blocks.

-e *end*
--end=*end*

Stop reading at the specified log position, instead of reading to the end of the log stream.

-f
--follow

After reaching the end of valid WAL, keep polling once per second for new WAL to appear.

-n *limit*
--limit=*limit*

Display the specified number of records, then stop.

`-p` *path*
`--path=`*path*

Directory in which to find log segment files. The default is to search for them in the `pg_xlog` subdirectory of the current directory.

`-r` *rmgr*
`--rmgr=`*rmgr*

Only display records generated by the specified resource manager. If `list` is passed as name, print a list of valid resource manager names, and exit.

`-s` *start*
`--start=`*start*

Log position at which to start reading. The default is to start reading the first valid log record found in the earliest file found.

`-t` *timeline*
`--timeline=`*timeline*

Timeline from which to read log records. The default is to use the value in `startseg`, if that is specified; otherwise, the default is 1.

`-V`
`--version`

Print the pg_xlogdump version and exit.

`-x` *xid*
`--xid=`*xid*

Only display records marked with the given transaction ID.

`-z`
`--stats[=record]`

Display summary statistics (number and size of records and full-page images) instead of individual records. Optionally generate statistics per-record instead of per-rmgr.

`-?`
`--help`

Show help about pg_xlogdump command line arguments, and exit.

Notes

Can give wrong results when the server is running.

Only the specified timeline is displayed (or the default, if none is specified). Records in other timelines are ignored.

pg_xlogdump cannot read WAL files with suffix `.partial`. If those files need to be read, `.partial` suffix needs to be removed from the file name.

See Also

Section 29.5

postgres

Name

postgres — PostgreSQL database server

Synopsis

postgres [*option*...]

Description

postgres is the PostgreSQL database server. In order for a client application to access a database it connects (over a network or locally) to a running postgres instance. The postgres instance then starts a separate server process to handle the connection.

One postgres instance always manages the data of exactly one database cluster. A database cluster is a collection of databases that is stored at a common file system location (the "data area"). More than one postgres instance can run on a system at one time, so long as they use different data areas and different communication ports (see below). When postgres starts it needs to know the location of the data area. The location must be specified by the -D option or the PGDATA environment variable; there is no default. Typically, -D or PGDATA points directly to the data area directory created by initdb. Other possible file layouts are discussed in Section 18.2.

By default postgres starts in the foreground and prints log messages to the standard error stream. In practical applications postgres should be started as a background process, perhaps at boot time.

The postgres command can also be called in single-user mode. The primary use for this mode is during bootstrapping by initdb. Sometimes it is used for debugging or disaster recovery; note that running a single-user server is not truly suitable for debugging the server, since no realistic interprocess communication and locking will happen. When invoked in single-user mode from the shell, the user can enter queries and the results will be printed to the screen, but in a form that is more useful for developers than end users. In the single-user mode, the session user will be set to the user with ID 1, and implicit superuser powers are granted to this user. This user does not actually have to exist, so the single-user mode can be used to manually recover from certain kinds of accidental damage to the system catalogs.

Options

postgres accepts the following command-line arguments. For a detailed discussion of the options consult Chapter 18. You can save typing most of these options by setting up a configuration file. Some (safe) options can also be set from the connecting client in an application-dependent way to apply only for that session. For example, if the environment variable PGOPTIONS is set, then libpq-based clients will pass that string to the server, which will interpret it as postgres command-line options.

General Purpose

`-B` *`nbuffers`*

Sets the number of shared buffers for use by the server processes. The default value of this parameter is chosen automatically by initdb. Specifying this option is equivalent to setting the shared_buffers configuration parameter.

`-c` *`name=value`*

Sets a named run-time parameter. The configuration parameters supported by PostgreSQL are described in Chapter 18. Most of the other command line options are in fact short forms of such a parameter assignment. `-c` can appear multiple times to set multiple parameters.

`-C` *`name`*

Prints the value of the named run-time parameter, and exits. (See the `-c` option above for details.) This can be used on a running server, and returns values from `postgresql.conf`, modified by any parameters supplied in this invocation. It does not reflect parameters supplied when the cluster was started.

This option is meant for other programs that interact with a server instance, such as pg_ctl, to query configuration parameter values. User-facing applications should instead use SHOW or the `pg_settings` view.

`-d` *`debug-level`*

Sets the debug level. The higher this value is set, the more debugging output is written to the server log. Values are from 1 to 5. It is also possible to pass `-d 0` for a specific session, which will prevent the server log level of the parent `postgres` process from being propagated to this session.

`-D` *`datadir`*

Specifies the file system location of the database configuration files. See Section 18.2 for details.

`-e`

Sets the default date style to "European", that is `DMY` ordering of input date fields. This also causes the day to be printed before the month in certain date output formats. See Section 8.5 for more information.

`-F`

Disables `fsync` calls for improved performance, at the risk of data corruption in the event of a system crash. Specifying this option is equivalent to disabling the fsync configuration parameter. Read the detailed documentation before using this!

`-h` *`hostname`*

Specifies the IP host name or address on which `postgres` is to listen for TCP/IP connections from client applications. The value can also be a comma-separated list of addresses, or `*` to specify listening on all available interfaces. An empty value specifies not listening on any IP addresses, in which case only Unix-domain sockets can be used to connect to the server. Defaults to listening only on localhost. Specifying this option is equivalent to setting the listen_addresses configuration parameter.

`-i`

Allows remote clients to connect via TCP/IP (Internet domain) connections. Without this option, only local connections are accepted. This option is equivalent to setting `listen_addresses` to `*` in

postgresql.conf or via -h.

This option is deprecated since it does not allow access to the full functionality of listen_addresses. It's usually better to set listen_addresses directly.

-k *directory*

Specifies the directory of the Unix-domain socket on which postgres is to listen for connections from client applications. The value can also be a comma-separated list of directories. An empty value specifies not listening on any Unix-domain sockets, in which case only TCP/IP sockets can be used to connect to the server. The default value is normally /tmp, but that can be changed at build time. Specifying this option is equivalent to setting the unix_socket_directories configuration parameter.

-l

Enables secure connections using SSL. PostgreSQL must have been compiled with support for SSL for this option to be available. For more information on using SSL, refer to Section 17.9.

-N *max-connections*

Sets the maximum number of client connections that this server will accept. The default value of this parameter is chosen automatically by initdb. Specifying this option is equivalent to setting the max_connections configuration parameter.

-o *extra-options*

The command-line-style arguments specified in *extra-options* are passed to all server processes started by this postgres process.

Spaces within *extra-options* are considered to separate arguments, unless escaped with a backslash (\); write \\ to represent a literal backslash. Multiple arguments can also be specified via multiple uses of -o.

The use of this option is obsolete; all command-line options for server processes can be specified directly on the postgres command line.

-p *port*

Specifies the TCP/IP port or local Unix domain socket file extension on which postgres is to listen for connections from client applications. Defaults to the value of the PGPORT environment variable, or if PGPORT is not set, then defaults to the value established during compilation (normally 5432). If you specify a port other than the default port, then all client applications must specify the same port using either command-line options or PGPORT.

-s

Print time information and other statistics at the end of each command. This is useful for benchmarking or for use in tuning the number of buffers.

-S *work-mem*

Specifies the amount of memory to be used by internal sorts and hashes before resorting to temporary disk files. See the description of the work_mem configuration parameter in Section 18.4.1.

-V

--version

Print the postgres version and exit.

`--*name*=*value*`

Sets a named run-time parameter; a shorter form of `-c`.

`--describe-config`

This option dumps out the server's internal configuration variables, descriptions, and defaults in tab-delimited `COPY` format. It is designed primarily for use by administration tools.

`-?`
`--help`

Show help about postgres command line arguments, and exit.

Semi-internal Options

The options described here are used mainly for debugging purposes, and in some cases to assist with recovery of severely damaged databases. There should be no reason to use them in a production database setup. They are listed here only for use by PostgreSQL system developers. Furthermore, these options might change or be removed in a future release without notice.

`-f { s | i | o | b | t | n | m | h }`

Forbids the use of particular scan and join methods: `s` and `i` disable sequential and index scans respectively, `o`, `b` and `t` disable index-only scans, bitmap index scans, and TID scans respectively, while `n`, `m`, and `h` disable nested-loop, merge and hash joins respectively.

Neither sequential scans nor nested-loop joins can be disabled completely; the `-fs` and `-fn` options simply discourage the optimizer from using those plan types if it has any other alternative.

`-n`

This option is for debugging problems that cause a server process to die abnormally. The ordinary strategy in this situation is to notify all other server processes that they must terminate and then reinitialize the shared memory and semaphores. This is because an errant server process could have corrupted some shared state before terminating. This option specifies that `postgres` will not reinitialize shared data structures. A knowledgeable system programmer can then use a debugger to examine shared memory and semaphore state.

`-O`

Allows the structure of system tables to be modified. This is used by `initdb`.

`-P`

Ignore system indexes when reading system tables, but still update the indexes when modifying the tables. This is useful when recovering from damaged system indexes.

`-t pa[rser] | pl[anner] | e[xecutor]`

Print timing statistics for each query relating to each of the major system modules. This option cannot be used together with the `-s` option.

`-T`

This option is for debugging problems that cause a server process to die abnormally. The ordinary strategy in this situation is to notify all other server processes that they must terminate and then reinitialize the shared memory and semaphores. This is because an errant server process could have

corrupted some shared state before terminating. This option specifies that `postgres` will stop all other server processes by sending the signal `SIGSTOP`, but will not cause them to terminate. This permits system programmers to collect core dumps from all server processes by hand.

`-v` *`protocol`*

Specifies the version number of the frontend/backend protocol to be used for a particular session. This option is for internal use only.

`-W` *`seconds`*

A delay of this many seconds occurs when a new server process is started, after it conducts the authentication procedure. This is intended to give an opportunity to attach to the server process with a debugger.

Options for Single-User Mode

The following options only apply to the single-user mode.

`--single`

Selects the single-user mode. This must be the first argument on the command line.

`database`

Specifies the name of the database to be accessed. This must be the last argument on the command line. If it is omitted it defaults to the user name.

`-E`

Echo all commands.

`-j`

Disables use of newline as a statement delimiter.

`-r` *`filename`*

Send all server log output to *`filename`*. This option is only honored when supplied as a command-line option.

Environment

`PGCLIENTENCODING`

Default character encoding used by clients. (The clients can override this individually.) This value can also be set in the configuration file.

`PGDATA`

Default data directory location

`PGDATESTYLE`

Default value of the DateStyle run-time parameter. (The use of this environment variable is deprecated.)

`PGPORT`

Default port number (preferably set in the configuration file)

Diagnostics

A failure message mentioning `semget` or `shmget` probably indicates you need to configure your kernel to provide adequate shared memory and semaphores. For more discussion see Section 17.4. You might be able to postpone reconfiguring your kernel by decreasing shared_buffers to reduce the shared memory consumption of PostgreSQL, and/or by reducing max_connections to reduce the semaphore consumption.

A failure message suggesting that another server is already running should be checked carefully, for example by using the command

```
$ ps ax | grep postgres
```

or

```
$ ps -ef | grep postgres
```

depending on your system. If you are certain that no conflicting server is running, you can remove the lock file mentioned in the message and try again.

A failure message indicating inability to bind to a port might indicate that that port is already in use by some non-PostgreSQL process. You might also get this error if you terminate `postgres` and immediately restart it using the same port; in this case, you must simply wait a few seconds until the operating system closes the port before trying again. Finally, you might get this error if you specify a port number that your operating system considers to be reserved. For example, many versions of Unix consider port numbers under 1024 to be "trusted" and only permit the Unix superuser to access them.

Notes

The utility command pg_ctl can be used to start and shut down the `postgres` server safely and comfortably.

If at all possible, *do not* use `SIGKILL` to kill the main `postgres` server. Doing so will prevent `postgres` from freeing the system resources (e.g., shared memory and semaphores) that it holds before terminating. This might cause problems for starting a fresh `postgres` run.

To terminate the `postgres` server normally, the signals `SIGTERM`, `SIGINT`, or `SIGQUIT` can be used. The first will wait for all clients to terminate before quitting, the second will forcefully disconnect all clients, and the third will quit immediately without proper shutdown, resulting in a recovery run during restart.

The `SIGHUP` signal will reload the server configuration files. It is also possible to send `SIGHUP` to an individual server process, but that is usually not sensible.

To cancel a running query, send the `SIGINT` signal to the process running that command. To terminate a backend process cleanly, send `SIGTERM` to that process. See also `pg_cancel_backend` and `pg_terminate_backend` in Section 9.26.2 for the SQL-callable equivalents of these two actions.

The `postgres` server uses `SIGQUIT` to tell subordinate server processes to terminate without normal cleanup. This signal *should not* be used by users. It is also unwise to send `SIGKILL` to a server process —

the main `postgres` process will interpret this as a crash and will force all the sibling processes to quit as part of its standard crash-recovery procedure.

Bugs

The `--` options will not work on FreeBSD or OpenBSD. Use `-c` instead. This is a bug in the affected operating systems; a future release of PostgreSQL will provide a workaround if this is not fixed.

Usage

To start a single-user mode server, use a command like

```
postgres --single -D /usr/local/pgsql/data other-options my_database
```

Provide the correct path to the database directory with `-D`, or make sure that the environment variable `PGDATA` is set. Also specify the name of the particular database you want to work in.

Normally, the single-user mode server treats newline as the command entry terminator; there is no intelligence about semicolons, as there is in psql. To continue a command across multiple lines, you must type backslash just before each newline except the last one.

But if you use the `-j` command line switch, then newline does not terminate command entry. In this case, the server will read the standard input until the end-of-file (EOF) marker, then process the input as a single command string. Backslash-newline is not treated specially in this case.

To quit the session, type EOF (**Control+D**, usually). If you've used `-j`, two consecutive EOFs are needed to exit.

Note that the single-user mode server does not provide sophisticated line-editing features (no command history, for example). Single-User mode also does not do any background processing, like automatic checkpoints.

Examples

To start `postgres` in the background using default values, type:

```
$ nohup postgres >logfile 2>&1 </dev/null &
```

To start `postgres` with a specific port, e.g., 1234:

```
$ postgres -p 1234
```

To connect to this server using psql, specify this port with the -p option:

```
$ psql -p 1234
```

or set the environment variable `PGPORT`:

```
$ export PGPORT=1234
$ psql
```

Named run-time parameters can be set in either of these styles:

```
$ postgres -c work_mem=1234
$ postgres --work-mem=1234
```

Either form overrides whatever setting might exist for `work_mem` in `postgresql.conf`. Notice that underscores in parameter names can be written as either underscore or dash on the command line. Except for short-term experiments, it's probably better practice to edit the setting in `postgresql.conf` than to rely on a command-line switch to set a parameter.

See Also

initdb, pg_ctl

postmaster

Name

`postmaster` — PostgreSQL database server

Synopsis

`postmaster` [*`option`*...]

Description

`postmaster` is a deprecated alias of `postgres`.

See Also

postgres

VII. Internals

This part contains assorted information that might be of use to PostgreSQL developers.

postmaster

1996

Chapter 48. Overview of PostgreSQL Internals

Author: This chapter originated as part of *Enhancement of the ANSI SQL Implementation of PostgreSQL*, Stefan Simkovics' Master's Thesis prepared at Vienna University of Technology under the direction of O.Univ.Prof.Dr. Georg Gottlob and Univ.Ass. Mag. Katrin Seyr.

This chapter gives an overview of the internal structure of the backend of PostgreSQL. After having read the following sections you should have an idea of how a query is processed. This chapter does not aim to provide a detailed description of the internal operation of PostgreSQL, as such a document would be very extensive. Rather, this chapter is intended to help the reader understand the general sequence of operations that occur within the backend from the point at which a query is received, to the point at which the results are returned to the client.

48.1. The Path of a Query

Here we give a short overview of the stages a query has to pass in order to obtain a result.

1. A connection from an application program to the PostgreSQL server has to be established. The application program transmits a query to the server and waits to receive the results sent back by the server.
2. The *parser stage* checks the query transmitted by the application program for correct syntax and creates a *query tree*.
3. The *rewrite system* takes the query tree created by the parser stage and looks for any *rules* (stored in the *system catalogs*) to apply to the query tree. It performs the transformations given in the *rule bodies*.

   One application of the rewrite system is in the realization of *views*. Whenever a query against a view (i.e., a *virtual table*) is made, the rewrite system rewrites the user's query to a query that accesses the *base tables* given in the *view definition* instead.
4. The *planner/optimizer* takes the (rewritten) query tree and creates a *query plan* that will be the input to the *executor*.

   It does so by first creating all possible *paths* leading to the same result. For example if there is an index on a relation to be scanned, there are two paths for the scan. One possibility is a simple sequential scan and the other possibility is to use the index. Next the cost for the execution of each path is estimated and the cheapest path is chosen. The cheapest path is expanded into a complete plan that the executor can use.
5. The executor recursively steps through the *plan tree* and retrieves rows in the way represented by the plan. The executor makes use of the *storage system* while scanning relations, performs *sorts* and *joins*, evaluates *qualifications* and finally hands back the rows derived.

In the following sections we will cover each of the above listed items in more detail to give a better understanding of PostgreSQL's internal control and data structures.

48.2. How Connections are Established

PostgreSQL is implemented using a simple "process per user" client/server model. In this model there is one *client process* connected to exactly one *server process*. As we do not know ahead of time how many connections will be made, we have to use a *master process* that spawns a new server process every time a connection is requested. This master process is called `postgres` and listens at a specified TCP/IP port for incoming connections. Whenever a request for a connection is detected the `postgres` process spawns a new server process. The server tasks communicate with each other using *semaphores* and *shared memory* to ensure data integrity throughout concurrent data access.

The client process can be any program that understands the PostgreSQL protocol described in Chapter 50. Many clients are based on the C-language library libpq, but several independent implementations of the protocol exist, such as the Java JDBC driver.

Once a connection is established the client process can send a query to the *backend* (server). The query is transmitted using plain text, i.e., there is no parsing done in the *frontend* (client). The server parses the query, creates an *execution plan*, executes the plan and returns the retrieved rows to the client by transmitting them over the established connection.

48.3. The Parser Stage

The *parser stage* consists of two parts:

- The *parser* defined in `gram.y` and `scan.l` is built using the Unix tools bison and flex.
- The *transformation process* does modifications and augmentations to the data structures returned by the parser.

48.3.1. Parser

The parser has to check the query string (which arrives as plain text) for valid syntax. If the syntax is correct a *parse tree* is built up and handed back; otherwise an error is returned. The parser and lexer are implemented using the well-known Unix tools bison and flex.

The *lexer* is defined in the file `scan.l` and is responsible for recognizing *identifiers*, the *SQL key words* etc. For every key word or identifier that is found, a *token* is generated and handed to the parser.

The parser is defined in the file `gram.y` and consists of a set of *grammar rules* and *actions* that are executed whenever a rule is fired. The code of the actions (which is actually C code) is used to build up the parse tree.

The file `scan.l` is transformed to the C source file `scan.c` using the program flex and `gram.y` is transformed to `gram.c` using bison. After these transformations have taken place a normal C compiler can be used to create the parser. Never make any changes to the generated C files as they will be overwritten the next time flex or bison is called.

Note: The mentioned transformations and compilations are normally done automatically using the *makefiles* shipped with the PostgreSQL source distribution.

A detailed description of bison or the grammar rules given in `gram.y` would be beyond the scope of this paper. There are many books and documents dealing with flex and bison. You should be familiar with bison before you start to study the grammar given in `gram.y` otherwise you won't understand what happens there.

48.3.2. Transformation Process

The parser stage creates a parse tree using only fixed rules about the syntactic structure of SQL. It does not make any lookups in the system catalogs, so there is no possibility to understand the detailed semantics of the requested operations. After the parser completes, the *transformation process* takes the tree handed back by the parser as input and does the semantic interpretation needed to understand which tables, functions, and operators are referenced by the query. The data structure that is built to represent this information is called the *query tree*.

The reason for separating raw parsing from semantic analysis is that system catalog lookups can only be done within a transaction, and we do not wish to start a transaction immediately upon receiving a query string. The raw parsing stage is sufficient to identify the transaction control commands (`BEGIN`, `ROLLBACK`, etc), and these can then be correctly executed without any further analysis. Once we know that we are dealing with an actual query (such as `SELECT` or `UPDATE`), it is okay to start a transaction if we're not already in one. Only then can the transformation process be invoked.

The query tree created by the transformation process is structurally similar to the raw parse tree in most places, but it has many differences in detail. For example, a `FuncCall` node in the parse tree represents something that looks syntactically like a function call. This might be transformed to either a `FuncExpr` or `Aggref` node depending on whether the referenced name turns out to be an ordinary function or an aggregate function. Also, information about the actual data types of columns and expression results is added to the query tree.

48.4. The PostgreSQL Rule System

PostgreSQL supports a powerful *rule system* for the specification of *views* and ambiguous *view updates*. Originally the PostgreSQL rule system consisted of two implementations:

- The first one worked using *row level* processing and was implemented deep in the *executor*. The rule system was called whenever an individual row had been accessed. This implementation was removed in 1995 when the last official release of the Berkeley Postgres project was transformed into Postgres95.
- The second implementation of the rule system is a technique called *query rewriting*. The *rewrite system* is a module that exists between the *parser stage* and the *planner/optimizer*. This technique is still implemented.

The query rewriter is discussed in some detail in Chapter 38, so there is no need to cover it here. We will only point out that both the input and the output of the rewriter are query trees, that is, there is no change in the representation or level of semantic detail in the trees. Rewriting can be thought of as a form of macro expansion.

48.5. Planner/Optimizer

The task of the *planner/optimizer* is to create an optimal execution plan. A given SQL query (and hence, a query tree) can be actually executed in a wide variety of different ways, each of which will produce the same set of results. If it is computationally feasible, the query optimizer will examine each of these possible execution plans, ultimately selecting the execution plan that is expected to run the fastest.

> **Note:** In some situations, examining each possible way in which a query can be executed would take an excessive amount of time and memory space. In particular, this occurs when executing queries involving large numbers of join operations. In order to determine a reasonable (not necessarily optimal) query plan in a reasonable amount of time, PostgreSQL uses a *Genetic Query Optimizer* (see Chapter 57) when the number of joins exceeds a threshold (see geqo_threshold).

The planner's search procedure actually works with data structures called *paths*, which are simply cut-down representations of plans containing only as much information as the planner needs to make its decisions. After the cheapest path is determined, a full-fledged *plan tree* is built to pass to the executor. This represents the desired execution plan in sufficient detail for the executor to run it. In the rest of this section we'll ignore the distinction between paths and plans.

48.5.1. Generating Possible Plans

The planner/optimizer starts by generating plans for scanning each individual relation (table) used in the query. The possible plans are determined by the available indexes on each relation. There is always the possibility of performing a sequential scan on a relation, so a sequential scan plan is always created. Assume an index is defined on a relation (for example a B-tree index) and a query contains the restriction `relation.attribute OPR constant`. If `relation.attribute` happens to match the key of the B-tree index and `OPR` is one of the operators listed in the index's *operator class*, another plan is created using the B-tree index to scan the relation. If there are further indexes present and the restrictions in the query happen to match a key of an index, further plans will be considered. Index scan plans are also generated for indexes that have a sort ordering that can match the query's `ORDER BY` clause (if any), or a sort ordering that might be useful for merge joining (see below).

If the query requires joining two or more relations, plans for joining relations are considered after all feasible plans have been found for scanning single relations. The three available join strategies are:

- *nested loop join*: The right relation is scanned once for every row found in the left relation. This strategy is easy to implement but can be very time consuming. (However, if the right relation can be scanned with an index scan, this can be a good strategy. It is possible to use values from the current row of the left relation as keys for the index scan of the right.)

- *merge join*: Each relation is sorted on the join attributes before the join starts. Then the two relations are scanned in parallel, and matching rows are combined to form join rows. This kind of join is more attractive because each relation has to be scanned only once. The required sorting might be achieved either by an explicit sort step, or by scanning the relation in the proper order using an index on the join key.
- *hash join*: the right relation is first scanned and loaded into a hash table, using its join attributes as hash keys. Next the left relation is scanned and the appropriate values of every row found are used as hash keys to locate the matching rows in the table.

When the query involves more than two relations, the final result must be built up by a tree of join steps, each with two inputs. The planner examines different possible join sequences to find the cheapest one.

If the query uses fewer than geqo_threshold relations, a near-exhaustive search is conducted to find the best join sequence. The planner preferentially considers joins between any two relations for which there exist a corresponding join clause in the `WHERE` qualification (i.e., for which a restriction like `where rel1.attr1=rel2.attr2` exists). Join pairs with no join clause are considered only when there is no other choice, that is, a particular relation has no available join clauses to any other relation. All possible plans are generated for every join pair considered by the planner, and the one that is (estimated to be) the cheapest is chosen.

When `geqo_threshold` is exceeded, the join sequences considered are determined by heuristics, as described in Chapter 57. Otherwise the process is the same.

The finished plan tree consists of sequential or index scans of the base relations, plus nested-loop, merge, or hash join nodes as needed, plus any auxiliary steps needed, such as sort nodes or aggregate-function calculation nodes. Most of these plan node types have the additional ability to do *selection* (discarding rows that do not meet a specified Boolean condition) and *projection* (computation of a derived column set based on given column values, that is, evaluation of scalar expressions where needed). One of the responsibilities of the planner is to attach selection conditions from the `WHERE` clause and computation of required output expressions to the most appropriate nodes of the plan tree.

48.6. Executor

The *executor* takes the plan created by the planner/optimizer and recursively processes it to extract the required set of rows. This is essentially a demand-pull pipeline mechanism. Each time a plan node is called, it must deliver one more row, or report that it is done delivering rows.

To provide a concrete example, assume that the top node is a `MergeJoin` node. Before any merge can be done two rows have to be fetched (one from each subplan). So the executor recursively calls itself to process the subplans (it starts with the subplan attached to `lefttree`). The new top node (the top node of the left subplan) is, let's say, a `Sort` node and again recursion is needed to obtain an input row. The child node of the `Sort` might be a `SeqScan` node, representing actual reading of a table. Execution of this node causes the executor to fetch a row from the table and return it up to the calling node. The `Sort` node will repeatedly call its child to obtain all the rows to be sorted. When the input is exhausted (as indicated by the child node returning a NULL instead of a row), the `Sort` code performs the sort, and finally is able to return its first output row, namely the first one in sorted order. It keeps the remaining rows stored so that it can deliver them in sorted order in response to later demands.

The `MergeJoin` node similarly demands the first row from its right subplan. Then it compares the two rows to see if they can be joined; if so, it returns a join row to its caller. On the next call, or immediately if it cannot join the current pair of inputs, it advances to the next row of one table or the other (depending on how the comparison came out), and again checks for a match. Eventually, one subplan or the other is exhausted, and the `MergeJoin` node returns NULL to indicate that no more join rows can be formed.

Complex queries can involve many levels of plan nodes, but the general approach is the same: each node computes and returns its next output row each time it is called. Each node is also responsible for applying any selection or projection expressions that were assigned to it by the planner.

The executor mechanism is used to evaluate all four basic SQL query types: `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. For `SELECT`, the top-level executor code only needs to send each row returned by the query plan tree off to the client. For `INSERT`, each returned row is inserted into the target table specified for the `INSERT`. This is done in a special top-level plan node called `ModifyTable`. (A simple `INSERT ... VALUES` command creates a trivial plan tree consisting of a single `Result` node, which computes just one result row, and `ModifyTable` above it to perform the insertion. But `INSERT ... SELECT` can demand the full power of the executor mechanism.) For `UPDATE`, the planner arranges that each computed row includes all the updated column values, plus the *TID* (tuple ID, or row ID) of the original target row; this data is fed into a `ModifyTable` node, which uses the information to create a new updated row and mark the old row deleted. For `DELETE`, the only column that is actually returned by the plan is the TID, and the `ModifyTable` node simply uses the TID to visit each target row and mark it deleted.

Chapter 49. System Catalogs

The system catalogs are the place where a relational database management system stores schema metadata, such as information about tables and columns, and internal bookkeeping information. PostgreSQL's system catalogs are regular tables. You can drop and recreate the tables, add columns, insert and update values, and severely mess up your system that way. Normally, one should not change the system catalogs by hand, there are always SQL commands to do that. (For example, `CREATE DATABASE` inserts a row into the `pg_database` catalog — and actually creates the database on disk.) There are some exceptions for particularly esoteric operations, such as adding index access methods.

49.1. Overview

Table 49-1 lists the system catalogs. More detailed documentation of each catalog follows below.

Most system catalogs are copied from the template database during database creation and are thereafter database-specific. A few catalogs are physically shared across all databases in a cluster; these are noted in the descriptions of the individual catalogs.

Table 49-1. System Catalogs

Catalog Name	Purpose
`pg_aggregate`	aggregate functions
`pg_am`	index access methods
`pg_amop`	access method operators
`pg_amproc`	access method support procedures
`pg_attrdef`	column default values
`pg_attribute`	table columns ("attributes")
`pg_authid`	authorization identifiers (roles)
`pg_auth_members`	authorization identifier membership relationships
`pg_cast`	casts (data type conversions)
`pg_class`	tables, indexes, sequences, views ("relations")
`pg_collation`	collations (locale information)
`pg_constraint`	check constraints, unique constraints, primary key constraints, foreign key constraints
`pg_conversion`	encoding conversion information
`pg_database`	databases within this database cluster
`pg_db_role_setting`	per-role and per-database settings
`pg_default_acl`	default privileges for object types
`pg_depend`	dependencies between database objects
`pg_description`	descriptions or comments on database objects

Catalog Name	Purpose
pg_enum	enum label and value definitions
pg_event_trigger	event triggers
pg_extension	installed extensions
pg_foreign_data_wrapper	foreign-data wrapper definitions
pg_foreign_server	foreign server definitions
pg_foreign_table	additional foreign table information
pg_index	additional index information
pg_inherits	table inheritance hierarchy
pg_language	languages for writing functions
pg_largeobject	data pages for large objects
pg_largeobject_metadata	metadata for large objects
pg_namespace	schemas
pg_opclass	access method operator classes
pg_operator	operators
pg_opfamily	access method operator families
pg_pltemplate	template data for procedural languages
pg_policy	row-security policies
pg_proc	functions and procedures
pg_range	information about range types
pg_replication_origin	registered replication origins
pg_rewrite	query rewrite rules
pg_seclabel	security labels on database objects
pg_shdepend	dependencies on shared objects
pg_shdescription	comments on shared objects
pg_shseclabel	security labels on shared database objects
pg_statistic	planner statistics
pg_tablespace	tablespaces within this database cluster
pg_transform	transforms (data type to procedural language conversions)
pg_trigger	triggers
pg_ts_config	text search configurations
pg_ts_config_map	text search configurations' token mappings
pg_ts_dict	text search dictionaries
pg_ts_parser	text search parsers
pg_ts_template	text search templates
pg_type	data types
pg_user_mapping	mappings of users to foreign servers

49.2. pg_aggregate

The catalog pg_aggregate stores information about aggregate functions. An aggregate function is a function that operates on a set of values (typically one column from each row that matches a query condition) and returns a single value computed from all these values. Typical aggregate functions are sum, count, and max. Each entry in pg_aggregate is an extension of an entry in pg_proc. The pg_proc entry carries the aggregate's name, input and output data types, and other information that is similar to ordinary functions.

Table 49-2. pg_aggregate Columns

Name	Type	References	Description
aggfnoid	regproc	pg_proc.oid	pg_proc OID of the aggregate function
aggkind	char		Aggregate kind: n for "normal" aggregates, o for "ordered-set" aggregates, or h for "hypothetical-set" aggregates
aggnumdirectargs	int2		Number of direct (non-aggregated) arguments of an ordered-set or hypothetical-set aggregate, counting a variadic array as one argument. If equal to pronargs, the aggregate must be variadic and the variadic array describes the aggregated arguments as well as the final direct arguments. Always zero for normal aggregates.
aggtransfn	regproc	pg_proc.oid	Transition function
aggfinalfn	regproc	pg_proc.oid	Final function (zero if none)
aggmtransfn	regproc	pg_proc.oid	Forward transition function for moving-aggregate mode (zero if none)

Name	Type	References	Description
aggminvtransfn	regproc	pg_proc.oid	Inverse transition function for moving-aggregate mode (zero if none)
aggmfinalfn	regproc	pg_proc.oid	Final function for moving-aggregate mode (zero if none)
aggfinalextra	bool		True to pass extra dummy arguments to aggfinalfn
aggmfinalextra	bool		True to pass extra dummy arguments to aggmfinalfn
aggsortop	oid	pg_operator.oid	Associated sort operator (zero if none)
aggtranstype	oid	pg_type.oid	Data type of the aggregate function's internal transition (state) data
aggtransspace	int4		Approximate average size (in bytes) of the transition state data, or zero to use a default estimate
aggmtranstype	oid	pg_type.oid	Data type of the aggregate function's internal transition (state) data for moving-aggregate mode (zero if none)
aggmtransspace	int4		Approximate average size (in bytes) of the transition state data for moving-aggregate mode, or zero to use a default estimate

Name	Type	References	Description
agginitval	text		The initial value of the transition state. This is a text field containing the initial value in its external string representation. If this field is null, the transition state value starts out null.
aggminitval	text		The initial value of the transition state for moving-aggregate mode. This is a text field containing the initial value in its external string representation. If this field is null, the transition state value starts out null.

New aggregate functions are registered with the CREATE AGGREGATE command. See Section 35.10 for more information about writing aggregate functions and the meaning of the transition functions, etc.

49.3. pg_am

The catalog pg_am stores information about index access methods. There is one row for each index access method supported by the system. The contents of this catalog are discussed in detail in Chapter 58.

Table 49-3. pg_am Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
amname	name		Name of the access method
amstrategies	int2		Number of operator strategies for this access method, or zero if access method does not have a fixed set of operator strategies

Name	Type	References	Description
amsupport	int2		Number of support routines for this access method
amcanorder	bool		Does the access method support ordered scans sorted by the indexed column's value?
amcanorderbyop	bool		Does the access method support ordered scans sorted by the result of an operator on the indexed column?
amcanbackward	bool		Does the access method support backward scanning?
amcanunique	bool		Does the access method support unique indexes?
amcanmulticol	bool		Does the access method support multicolumn indexes?
amoptionalkey	bool		Does the access method support a scan without any constraint for the first index column?
amsearcharray	bool		Does the access method support ScalarArrayOpExpr searches?
amsearchnulls	bool		Does the access method support IS NULL/NOT NULL searches?
amstorage	bool		Can index storage data type differ from column data type?
amclusterable	bool		Can an index of this type be clustered on?
ampredlocks	bool		Does an index of this type manage fine-grained predicate locks?
amkeytype	oid	pg_type.oid	Type of data stored in index, or zero if not a fixed type

Name	Type	References	Description
aminsert	regproc	pg_proc.oid	"Insert this tuple" function
ambeginscan	regproc	pg_proc.oid	"Prepare for index scan" function
amgettuple	regproc	pg_proc.oid	"Next valid tuple" function, or zero if none
amgetbitmap	regproc	pg_proc.oid	"Fetch all valid tuples" function, or zero if none
amrescan	regproc	pg_proc.oid	"(Re)start index scan" function
amendscan	regproc	pg_proc.oid	"Clean up after index scan" function
ammarkpos	regproc	pg_proc.oid	"Mark current scan position" function
amrestrpos	regproc	pg_proc.oid	"Restore marked scan position" function
ambuild	regproc	pg_proc.oid	"Build new index" function
ambuildempty	regproc	pg_proc.oid	"Build empty index" function
ambulkdelete	regproc	pg_proc.oid	Bulk-delete function
amvacuumcleanup	regproc	pg_proc.oid	Post-VACUUM cleanup function
amcanreturn	regproc	pg_proc.oid	Function to check whether an index column supports index-only scans. Can be zero if index-only scans are never supported.
amcostestimate	regproc	pg_proc.oid	Function to estimate cost of an index scan
amoptions	regproc	pg_proc.oid	Function to parse and validate reloptions for an index

49.4. pg_amop

The catalog pg_amop stores information about operators associated with access method operator families. There is one row for each operator that is a member of an operator family. A family member can be either a *search* operator or an *ordering* operator. An operator can appear in more than one family, but cannot appear in more than one search position nor more than one ordering position within a family. (It is allowed,

though unlikely, for an operator to be used for both search and ordering purposes.)

Table 49-4. `pg_amop` Columns

Name	Type	References	Description
`oid`	`oid`		Row identifier (hidden attribute; must be explicitly selected)
`amopfamily`	`oid`	`pg_opfamily.oid`	The operator family this entry is for
`amoplefttype`	`oid`	`pg_type.oid`	Left-hand input data type of operator
`amoprighttype`	`oid`	`pg_type.oid`	Right-hand input data type of operator
`amopstrategy`	`int2`		Operator strategy number
`amoppurpose`	`char`		Operator purpose, either `s` for search or `o` for ordering
`amopopr`	`oid`	`pg_operator.oid`	OID of the operator
`amopmethod`	`oid`	`pg_am.oid`	Index access method operator family is for
`amopsortfamily`	`oid`	`pg_opfamily.oid`	The B-tree operator family this entry sorts according to, if an ordering operator; zero if a search operator

A "search" operator entry indicates that an index of this operator family can be searched to find all rows satisfying `WHERE` *`indexed_column operator constant`*. Obviously, such an operator must return `boolean`, and its left-hand input type must match the index's column data type.

An "ordering" operator entry indicates that an index of this operator family can be scanned to return rows in the order represented by `ORDER BY` *`indexed_column operator constant`*. Such an operator could return any sortable data type, though again its left-hand input type must match the index's column data type. The exact semantics of the `ORDER BY` are specified by the `amopsortfamily` column, which must reference a B-tree operator family for the operator's result type.

> **Note:** At present, it's assumed that the sort order for an ordering operator is the default for the referenced operator family, i.e., `ASC NULLS LAST`. This might someday be relaxed by adding additional columns to specify sort options explicitly.

An entry's `amopmethod` must match the `opfmethod` of its containing operator family (including `amopmethod` here is an intentional denormalization of the catalog structure for performance reasons). Also, `amoplefttype` and `amoprighttype` must match the `oprleft` and `oprright` fields of the referenced `pg_operator` entry.

49.5. pg_amproc

The catalog pg_amproc stores information about support procedures associated with access method operator families. There is one row for each support procedure belonging to an operator family.

Table 49-5. pg_amproc Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
amprocfamily	oid	pg_opfamily.oid	The operator family this entry is for
amproclefttype	oid	pg_type.oid	Left-hand input data type of associated operator
amprocrighttype	oid	pg_type.oid	Right-hand input data type of associated operator
amprocnum	int2		Support procedure number
amproc	regproc	pg_proc.oid	OID of the procedure

The usual interpretation of the amproclefttype and amprocrighttype fields is that they identify the left and right input types of the operator(s) that a particular support procedure supports. For some access methods these match the input data type(s) of the support procedure itself, for others not. There is a notion of "default" support procedures for an index, which are those with amproclefttype and amprocrighttype both equal to the index operator class's opcintype.

49.6. pg_attrdef

The catalog pg_attrdef stores column default values. The main information about columns is stored in pg_attribute (see below). Only columns that explicitly specify a default value (when the table is created or the column is added) will have an entry here.

Table 49-6. pg_attrdef Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
adrelid	oid	pg_class.oid	The table this column belongs to
adnum	int2	pg_attribute.attnum	The number of the column

Name	Type	References	Description
adbin	pg_node_tree		The internal representation of the column default value
adsrc	text		A human-readable representation of the default value

The adsrc field is historical, and is best not used, because it does not track outside changes that might affect the representation of the default value. Reverse-compiling the adbin field (with pg_get_expr for example) is a better way to display the default value.

49.7. pg_attribute

The catalog pg_attribute stores information about table columns. There will be exactly one pg_attribute row for every column in every table in the database. (There will also be attribute entries for indexes, and indeed all objects that have pg_class entries.)

The term attribute is equivalent to column and is used for historical reasons.

Table 49-7. pg_attribute Columns

Name	Type	References	Description
attrelid	oid	pg_class.oid	The table this column belongs to
attname	name		The column name
atttypid	oid	pg_type.oid	The data type of this column

Name	Type	References	Description
attstattarget	int4		attstattarget controls the level of detail of statistics accumulated for this column by ANALYZE. A zero value indicates that no statistics should be collected. A negative value says to use the system default statistics target. The exact meaning of positive values is data type-dependent. For scalar data types, attstattarget is both the target number of "most common values" to collect, and the target number of histogram bins to create.
attlen	int2		A copy of pg_type.typlen of this column's type
attnum	int2		The number of the column. Ordinary columns are numbered from 1 up. System columns, such as oid, have (arbitrary) negative numbers.
attndims	int4		Number of dimensions, if the column is an array type; otherwise 0. (Presently, the number of dimensions of an array is not enforced, so any nonzero value effectively means "it's an array".)

Name	Type	References	Description
attcacheoff	int4		Always -1 in storage, but when loaded into a row descriptor in memory this might be updated to cache the offset of the attribute within the row
atttypmod	int4		atttypmod records type-specific data supplied at table creation time (for example, the maximum length of a varchar column). It is passed to type-specific input functions and length coercion functions. The value will generally be -1 for types that do not need atttypmod.
attbyval	bool		A copy of pg_type.typbyval of this column's type
attstorage	char		Normally a copy of pg_type.typstorage of this column's type. For TOAST-able data types, this can be altered after column creation to control storage policy.
attalign	char		A copy of pg_type.typalign of this column's type
attnotnull	bool		This represents a not-null constraint.
atthasdef	bool		This column has a default value, in which case there will be a corresponding entry in the pg_attrdef catalog that actually defines the value.

Name	Type	References	Description
attisdropped	bool		This column has been dropped and is no longer valid. A dropped column is still physically present in the table, but is ignored by the parser and so cannot be accessed via SQL.
attislocal	bool		This column is defined locally in the relation. Note that a column can be locally defined and inherited simultaneously.
attinhcount	int4		The number of direct ancestors this column has. A column with a nonzero number of ancestors cannot be dropped nor renamed.
attcollation	oid	pg_collation.oid	The defined collation of the column, or zero if the column is not of a collatable data type.
attacl	aclitem[]		Column-level access privileges, if any have been granted specifically on this column
attoptions	text[]		Attribute-level options, as "keyword=value" strings
attfdwoptions	text[]		Attribute-level foreign data wrapper options, as "keyword=value" strings

In a dropped column's pg_attribute entry, atttypid is reset to zero, but attlen and the other fields copied from pg_type are still valid. This arrangement is needed to cope with the situation where the dropped column's data type was later dropped, and so there is no pg_type row anymore. attlen and the other fields can be used to interpret the contents of a row of the table.

49.8. pg_authid

The catalog pg_authid contains information about database authorization identifiers (roles). A role sub-

sumes the concepts of "users" and "groups". A user is essentially just a role with the `rolcanlogin` flag set. Any role (with or without `rolcanlogin`) can have other roles as members; see `pg_auth_members`.

Since this catalog contains passwords, it must not be publicly readable. `pg_roles` is a publicly readable view on `pg_authid` that blanks out the password field.

Chapter 20 contains detailed information about user and privilege management.

Because user identities are cluster-wide, `pg_authid` is shared across all databases of a cluster: there is only one copy of `pg_authid` per cluster, not one per database.

Table 49-8. `pg_authid` Columns

Name	Type	Description
`oid`	`oid`	Row identifier (hidden attribute; must be explicitly selected)
`rolname`	`name`	Role name
`rolsuper`	`bool`	Role has superuser privileges
`rolinherit`	`bool`	Role automatically inherits privileges of roles it is a member of
`rolcreaterole`	`bool`	Role can create more roles
`rolcreatedb`	`bool`	Role can create databases
`rolcanlogin`	`bool`	Role can log in. That is, this role can be given as the initial session authorization identifier.
`rolreplication`	`bool`	Role is a replication role. That is, this role can initiate streaming replication (see Section 25.2.5) and set/unset the system backup mode using `pg_start_backup` and `pg_stop_backup`
`rolbypassrls`	`bool`	Role bypasses every row level security policy, see Section 5.7 for more information.
`rolconnlimit`	`int4`	For roles that can log in, this sets maximum number of concurrent connections this role can make. -1 means no limit.

Name	Type	Description
rolpassword	text	Password (possibly encrypted); null if none. If the password is encrypted, this column will begin with the string md5 followed by a 32-character hexadecimal MD5 hash. The MD5 hash will be of the user's password concatenated to their user name. For example, if user joe has password xyzzy, PostgreSQL will store the md5 hash of xyzzyjoe. A password that does not follow that format is assumed to be unencrypted.
rolvaliduntil	timestamptz	Password expiry time (only used for password authentication); null if no expiration

49.9. pg_auth_members

The catalog pg_auth_members shows the membership relations between roles. Any non-circular set of relationships is allowed.

Because user identities are cluster-wide, pg_auth_members is shared across all databases of a cluster: there is only one copy of pg_auth_members per cluster, not one per database.

Table 49-9. pg_auth_members Columns

Name	Type	References	Description
roleid	oid	pg_authid.oid	ID of a role that has a member
member	oid	pg_authid.oid	ID of a role that is a member of roleid
grantor	oid	pg_authid.oid	ID of the role that granted this membership
admin_option	bool		True if member can grant membership in roleid to others

49.10. pg_cast

The catalog pg_cast stores data type conversion paths, both built-in and user-defined.

It should be noted that pg_cast does not represent every type conversion that the system knows how to perform; only those that cannot be deduced from some generic rule. For example, casting between a domain and its base type is not explicitly represented in pg_cast. Another important exception is that "automatic I/O conversion casts", those performed using a data type's own I/O functions to convert to or from text or other string types, are not explicitly represented in pg_cast.

Table 49-10. pg_cast Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
castsource	oid	pg_type.oid	OID of the source data type
casttarget	oid	pg_type.oid	OID of the target data type
castfunc	oid	pg_proc.oid	The OID of the function to use to perform this cast. Zero is stored if the cast method doesn't require a function.
castcontext	char		Indicates what contexts the cast can be invoked in. e means only as an explicit cast (using CAST or :: syntax). a means implicitly in assignment to a target column, as well as explicitly. i means implicitly in expressions, as well as the other cases.
castmethod	char		Indicates how the cast is performed. f means that the function specified in the castfunc field is used. i means that the input/output functions are used. b means that the types are binary-coercible, thus no conversion is required.

The cast functions listed in pg_cast must always take the cast source type as their first argument type, and return the cast destination type as their result type. A cast function can have up to three arguments. The second argument, if present, must be type integer; it receives the type modifier associated with the

destination type, or -1 if there is none. The third argument, if present, must be type boolean; it receives true if the cast is an explicit cast, false otherwise.

It is legitimate to create a pg_cast entry in which the source and target types are the same, if the associated function takes more than one argument. Such entries represent "length coercion functions" that coerce values of the type to be legal for a particular type modifier value.

When a pg_cast entry has different source and target types and a function that takes more than one argument, it represents converting from one type to another and applying a length coercion in a single step. When no such entry is available, coercion to a type that uses a type modifier involves two steps, one to convert between data types and a second to apply the modifier.

49.11. pg_class

The catalog pg_class catalogs tables and most everything else that has columns or is otherwise similar to a table. This includes indexes (but see also pg_index), sequences, views, materialized views, composite types, and TOAST tables; see relkind. Below, when we mean all of these kinds of objects we speak of "relations". Not all columns are meaningful for all relation types.

Table 49-11. pg_class Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
relname	name		Name of the table, index, view, etc.
relnamespace	oid	pg_namespace.oid	The OID of the namespace that contains this relation
reltype	oid	pg_type.oid	The OID of the data type that corresponds to this table's row type, if any (zero for indexes, which have no pg_type entry)
reloftype	oid	pg_type.oid	For typed tables, the OID of the underlying composite type, zero for all other relations
relowner	oid	pg_authid.oid	Owner of the relation
relam	oid	pg_am.oid	If this is an index, the access method used (B-tree, hash, etc.)

Name	Type	References	Description
relfilenode	oid		Name of the on-disk file of this relation; zero means this is a "mapped" relation whose disk file name is determined by low-level state
reltablespace	oid	pg_tablespace.oid	The tablespace in which this relation is stored. If zero, the database's default tablespace is implied. (Not meaningful if the relation has no on-disk file.)
relpages	int4		Size of the on-disk representation of this table in pages (of size BLCKSZ). This is only an estimate used by the planner. It is updated by VACUUM, ANALYZE, and a few DDL commands such as CREATE INDEX.
reltuples	float4		Number of rows in the table. This is only an estimate used by the planner. It is updated by VACUUM, ANALYZE, and a few DDL commands such as CREATE INDEX.
relallvisible	int4		Number of pages that are marked all-visible in the table's visibility map. This is only an estimate used by the planner. It is updated by VACUUM, ANALYZE, and a few DDL commands such as CREATE INDEX.

Name	Type	References	Description
reltoastrelid	oid	pg_class.oid	OID of the TOAST table associated with this table, 0 if none. The TOAST table stores large attributes “out of line” in a secondary table.
relhasindex	bool		True if this is a table and it has (or recently had) any indexes
relisshared	bool		True if this table is shared across all databases in the cluster. Only certain system catalogs (such as pg_database) are shared.
relpersistence	char		p = permanent table, u = unlogged table, t = temporary table
relkind	char		r = ordinary table, i = index, S = sequence, v = view, m = materialized view, c = composite type, t = TOAST table, f = foreign table
relnatts	int2		Number of user columns in the relation (system columns not counted). There must be this many corresponding entries in pg_attribute. See also pg_attribute.attnum.
relchecks	int2		Number of CHECK constraints on the table; see pg_constraint catalog
relhasoids	bool		True if we generate an OID for each row of the relation

Name	Type	References	Description
relhaspkey	bool		True if the table has (or once had) a primary key
relhasrules	bool		True if table has (or once had) rules; see pg_rewrite catalog
relhastriggers	bool		True if table has (or once had) triggers; see pg_trigger catalog
relhassubclass	bool		True if table has (or once had) any inheritance children
relrowsecurity	bool		True if table has row level security enabled; see pg_policy catalog
relforcerowsecurity	bool		True if row level security (when enabled) will also apply to table owner; see pg_policy catalog
relispopulated	bool		True if relation is populated (this is true for all relations other than some materialized views)
relreplident	char		Columns used to form "replica identity" for rows: d = default (primary key, if any), n = nothing, f = all columns, i = index with indisreplident set (same as nothing if the index used has been dropped)

Name	Type	References	Description
relfrozenxid	xid		All transaction IDs before this one have been replaced with a permanent ("frozen") transaction ID in this table. This is used to track whether the table needs to be vacuumed in order to prevent transaction ID wraparound or to allow pg_clog to be shrunk. Zero (InvalidTransactionId) if the relation is not a table.
relminmxid	xid		All multixact IDs before this one have been replaced by a transaction ID in this table. This is used to track whether the table needs to be vacuumed in order to prevent multixact ID wraparound or to allow pg_multixact to be shrunk. Zero (InvalidMultiXactId) if the relation is not a table.
relacl	aclitem[]		Access privileges; see GRANT and REVOKE for details
reloptions	text[]		Access-method-specific options, as "keyword=value" strings

Several of the Boolean flags in pg_class are maintained lazily: they are guaranteed to be true if that's the correct state, but may not be reset to false immediately when the condition is no longer true. For example, relhasindex is set by CREATE INDEX, but it is never cleared by DROP INDEX. Instead, VACUUM clears relhasindex if it finds the table has no indexes. This arrangement avoids race conditions and improves concurrency.

49.12. pg_collation

The catalog pg_collation describes the available collations, which are essentially mappings from an SQL name to operating system locale categories. See Section 22.2 for more information.

Table 49-12. pg_collation Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
collname	name		Collation name (unique per namespace and encoding)
collnamespace	oid	pg_namespace.oid	The OID of the namespace that contains this collation
collowner	oid	pg_authid.oid	Owner of the collation
collencoding	int4		Encoding in which the collation is applicable, or -1 if it works for any encoding
collcollate	name		LC_COLLATE for this collation object
collctype	name		LC_CTYPE for this collation object

Note that the unique key on this catalog is (collname, collencoding, collnamespace) not just (collname, collnamespace). PostgreSQL generally ignores all collations that do not have collencoding equal to either the current database's encoding or -1, and creation of new entries with the same name as an entry with collencoding = -1 is forbidden. Therefore it is sufficient to use a qualified SQL name (*schema*.*name*) to identify a collation, even though this is not unique according to the catalog definition. The reason for defining the catalog this way is that initdb fills it in at cluster initialization time with entries for all locales available on the system, so it must be able to hold entries for all encodings that might ever be used in the cluster.

In the template0 database, it could be useful to create collations whose encoding does not match the database encoding, since they could match the encodings of databases later cloned from template0. This would currently have to be done manually.

49.13. pg_constraint

The catalog pg_constraint stores check, primary key, unique, foreign key, and exclusion constraints on tables. (Column constraints are not treated specially. Every column constraint is equivalent to some table constraint.) Not-null constraints are represented in the pg_attribute catalog, not here.

User-defined constraint triggers (created with `CREATE CONSTRAINT TRIGGER`) also give rise to an entry in this table.

Check constraints on domains are stored here, too.

Table 49-13. `pg_constraint` Columns

Name	Type	References	Description
`oid`	`oid`		Row identifier (hidden attribute; must be explicitly selected)
`conname`	`name`		Constraint name (not necessarily unique!)
`connamespace`	`oid`	`pg_namespace.oid`	The OID of the namespace that contains this constraint
`contype`	`char`		`c` = check constraint, `f` = foreign key constraint, `p` = primary key constraint, `u` = unique constraint, `t` = constraint trigger, `x` = exclusion constraint
`condeferrable`	`bool`		Is the constraint deferrable?
`condeferred`	`bool`		Is the constraint deferred by default?
`convalidated`	`bool`		Has the constraint been validated? Currently, can only be false for foreign keys and CHECK constraints
`conrelid`	`oid`	`pg_class.oid`	The table this constraint is on; 0 if not a table constraint
`contypid`	`oid`	`pg_type.oid`	The domain this constraint is on; 0 if not a domain constraint
`conindid`	`oid`	`pg_class.oid`	The index supporting this constraint, if it's a unique, primary key, foreign key, or exclusion constraint; else 0
`confrelid`	`oid`	`pg_class.oid`	If a foreign key, the referenced table; else 0

Name	Type	References	Description
confupdtype	char		Foreign key update action code: a = no action, r = restrict, c = cascade, n = set null, d = set default
confdeltype	char		Foreign key deletion action code: a = no action, r = restrict, c = cascade, n = set null, d = set default
confmatchtype	char		Foreign key match type: f = full, p = partial, s = simple
conislocal	bool		This constraint is defined locally for the relation. Note that a constraint can be locally defined and inherited simultaneously.
coninhcount	int4		The number of direct inheritance ancestors this constraint has. A constraint with a nonzero number of ancestors cannot be dropped nor renamed.
connoinherit	bool		This constraint is defined locally for the relation. It is a non-inheritable constraint.
conkey	int2[]	pg_attribute.attnum	If a table constraint (including foreign keys, but not constraint triggers), list of the constrained columns
confkey	int2[]	pg_attribute.attnum	If a foreign key, list of the referenced columns
conpfeqop	oid[]	pg_operator.oid	If a foreign key, list of the equality operators for PK = FK comparisons

Name	Type	References	Description
conppeqop	oid[]	pg_operator.oid	If a foreign key, list of the equality operators for PK = PK comparisons
conffeqop	oid[]	pg_operator.oid	If a foreign key, list of the equality operators for FK = FK comparisons
conexclop	oid[]	pg_operator.oid	If an exclusion constraint, list of the per-column exclusion operators
conbin	pg_node_tree		If a check constraint, an internal representation of the expression
consrc	text		If a check constraint, a human-readable representation of the expression

In the case of an exclusion constraint, conkey is only useful for constraint elements that are simple column references. For other cases, a zero appears in conkey and the associated index must be consulted to discover the expression that is constrained. (conkey thus has the same contents as pg_index.indkey for the index.)

Note: consrc is not updated when referenced objects change; for example, it won't track renaming of columns. Rather than relying on this field, it's best to use pg_get_constraintdef() to extract the definition of a check constraint.

Note: pg_class.relchecks needs to agree with the number of check-constraint entries found in this table for each relation.

49.14. pg_conversion

The catalog pg_conversion describes encoding conversion procedures. See CREATE CONVERSION for more information.

Table 49-14. pg_conversion Columns

Name	Type	References	Description

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
conname	name		Conversion name (unique within a namespace)
connamespace	oid	pg_namespace.oid	The OID of the namespace that contains this conversion
conowner	oid	pg_authid.oid	Owner of the conversion
conforencoding	int4		Source encoding ID
contoencoding	int4		Destination encoding ID
conproc	regproc	pg_proc.oid	Conversion procedure
condefault	bool		True if this is the default conversion

49.15. pg_database

The catalog pg_database stores information about the available databases. Databases are created with the CREATE DATABASE command. Consult Chapter 21 for details about the meaning of some of the parameters.

Unlike most system catalogs, pg_database is shared across all databases of a cluster: there is only one copy of pg_database per cluster, not one per database.

Table 49-15. pg_database Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
datname	name		Database name
datdba	oid	pg_authid.oid	Owner of the database, usually the user who created it
encoding	int4		Character encoding for this database (pg_encoding_to_char() can translate this number to the encoding name)

Name	Type	References	Description
datcollate	name		LC_COLLATE for this database
datctype	name		LC_CTYPE for this database
datistemplate	bool		If true, then this database can be cloned by any user with CREATEDB privileges; if false, then only superusers or the owner of the database can clone it.
datallowconn	bool		If false then no one can connect to this database. This is used to protect the template0 database from being altered.
datconnlimit	int4		Sets maximum number of concurrent connections that can be made to this database. -1 means no limit.
datlastsysoid	oid		Last system OID in the database; useful particularly to pg_dump
datfrozenxid	xid		All transaction IDs before this one have been replaced with a permanent ("frozen") transaction ID in this database. This is used to track whether the database needs to be vacuumed in order to prevent transaction ID wraparound or to allow pg_clog to be shrunk. It is the minimum of the per-table pg_class.relfrozenxid values.

Name	Type	References	Description
datminmxid	xid		All multixact IDs before this one have been replaced with a transaction ID in this database. This is used to track whether the database needs to be vacuumed in order to prevent multixact ID wraparound or to allow pg_multixact to be shrunk. It is the minimum of the per-table pg_class.relminmxid values.
dattablespace	oid	pg_tablespace.oid	The default tablespace for the database. Within this database, all tables for which pg_class.reltablespace is zero will be stored in this tablespace; in particular, all the non-shared system catalogs will be there.
datacl	aclitem[]		Access privileges; see GRANT and REVOKE for details

49.16. pg_db_role_setting

The catalog pg_db_role_setting records the default values that have been set for run-time configuration variables, for each role and database combination.

Unlike most system catalogs, pg_db_role_setting is shared across all databases of a cluster: there is only one copy of pg_db_role_setting per cluster, not one per database.

Table 49-16. pg_db_role_setting Columns

Name	Type	References	Description

Name	Type	References	Description
setdatabase	oid	pg_database.oid	The OID of the database the setting is applicable to, or zero if not database-specific
setrole	oid	pg_authid.oid	The OID of the role the setting is applicable to, or zero if not role-specific
setconfig	text[]		Defaults for run-time configuration variables

49.17. pg_default_acl

The catalog pg_default_acl stores initial privileges to be assigned to newly created objects.

Table 49-17. pg_default_acl Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
defaclrole	oid	pg_authid.oid	The OID of the role associated with this entry
defaclnamespace	oid	pg_namespace.oid	The OID of the namespace associated with this entry, or 0 if none
defaclobjtype	char		Type of object this entry is for: r = relation (table, view), S = sequence, f = function, T = type
defaclacl	aclitem[]		Access privileges that this type of object should have on creation

A pg_default_acl entry shows the initial privileges to be assigned to an object belonging to the indicated user. There are currently two types of entry: "global" entries with defaclnamespace = 0, and "per-schema" entries that reference a particular schema. If a global entry is present then it *overrides* the normal hard-wired default privileges for the object type. A per-schema entry, if present, represents privileges to be *added to* the global or hard-wired default privileges.

Note that when an ACL entry in another catalog is null, it is taken to represent the hard-wired default

privileges for its object, *not* whatever might be in pg_default_acl at the moment. pg_default_acl is only consulted during object creation.

49.18. pg_depend

The catalog pg_depend records the dependency relationships between database objects. This information allows DROP commands to find which other objects must be dropped by DROP CASCADE or prevent dropping in the DROP RESTRICT case.

See also pg_shdepend, which performs a similar function for dependencies involving objects that are shared across a database cluster.

Table 49-18. pg_depend Columns

Name	Type	References	Description
classid	oid	pg_class.oid	The OID of the system catalog the dependent object is in
objid	oid	any OID column	The OID of the specific dependent object
objsubid	int4		For a table column, this is the column number (the objid and classid refer to the table itself). For all other object types, this column is zero.
refclassid	oid	pg_class.oid	The OID of the system catalog the referenced object is in
refobjid	oid	any OID column	The OID of the specific referenced object
refobjsubid	int4		For a table column, this is the column number (the refobjid and refclassid refer to the table itself). For all other object types, this column is zero.
deptype	char		A code defining the specific semantics of this dependency relationship; see text

In all cases, a pg_depend entry indicates that the referenced object cannot be dropped without also dropping the dependent object. However, there are several subflavors identified by deptype:

DEPENDENCY_NORMAL (n)

A normal relationship between separately-created objects. The dependent object can be dropped without affecting the referenced object. The referenced object can only be dropped by specifying CASCADE, in which case the dependent object is dropped, too. Example: a table column has a normal dependency on its data type.

DEPENDENCY_AUTO (a)

The dependent object can be dropped separately from the referenced object, and should be automatically dropped (regardless of RESTRICT or CASCADE mode) if the referenced object is dropped. Example: a named constraint on a table is made autodependent on the table, so that it will go away if the table is dropped.

DEPENDENCY_INTERNAL (i)

The dependent object was created as part of creation of the referenced object, and is really just a part of its internal implementation. A DROP of the dependent object will be disallowed outright (we'll tell the user to issue a DROP against the referenced object, instead). A DROP of the referenced object will be propagated through to drop the dependent object whether CASCADE is specified or not. Example: a trigger that's created to enforce a foreign-key constraint is made internally dependent on the constraint's pg_constraint entry.

DEPENDENCY_EXTENSION (e)

The dependent object is a member of the *extension* that is the referenced object (see pg_extension). The dependent object can be dropped only via DROP EXTENSION on the referenced object. Functionally this dependency type acts the same as an internal dependency, but it's kept separate for clarity and to simplify pg_dump.

DEPENDENCY_PIN (p)

There is no dependent object; this type of entry is a signal that the system itself depends on the referenced object, and so that object must never be deleted. Entries of this type are created only by initdb. The columns for the dependent object contain zeroes.

Other dependency flavors might be needed in future.

49.19. pg_description

The catalog pg_description stores optional descriptions (comments) for each database object. Descriptions can be manipulated with the COMMENT command and viewed with psql's \d commands. Descriptions of many built-in system objects are provided in the initial contents of pg_description.

See also pg_shdescription, which performs a similar function for descriptions involving objects that are shared across a database cluster.

Table 49-19. pg_description Columns

Name	Type	References	Description
objoid	oid	any OID column	The OID of the object this description pertains to

Name	Type	References	Description
classoid	oid	pg_class.oid	The OID of the system catalog this object appears in
objsubid	int4		For a comment on a table column, this is the column number (the objoid and classoid refer to the table itself). For all other object types, this column is zero.
description	text		Arbitrary text that serves as the description of this object

49.20. pg_enum

The pg_enum catalog contains entries showing the values and labels for each enum type. The internal representation of a given enum value is actually the OID of its associated row in pg_enum.

Table 49-20. pg_enum Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
enumtypid	oid	pg_type.oid	The OID of the pg_type entry owning this enum value
enumsortorder	float4		The sort position of this enum value within its enum type
enumlabel	name		The textual label for this enum value

The OIDs for pg_enum rows follow a special rule: even-numbered OIDs are guaranteed to be ordered in the same way as the sort ordering of their enum type. That is, if two even OIDs belong to the same enum type, the smaller OID must have the smaller enumsortorder value. Odd-numbered OID values need bear no relationship to the sort order. This rule allows the enum comparison routines to avoid catalog lookups in many common cases. The routines that create and alter enum types attempt to assign even OIDs to enum values whenever possible.

When an enum type is created, its members are assigned sort-order positions 1..*n*. But members added later might be given negative or fractional values of enumsortorder. The only requirement on these

values is that they be correctly ordered and unique within each enum type.

49.21. pg_event_trigger

The catalog pg_event_trigger stores event triggers. See Chapter 37 for more information.

Table 49-21. pg_event_trigger Columns

Name	Type	References	Description
evtname	name		Trigger name (must be unique)
evtevent	name		Identifies the event for which this trigger fires
evtowner	oid	pg_authid.oid	Owner of the event trigger
evtfoid	oid	pg_proc.oid	The function to be called
evtenabled	char		Controls in which session_replication_role modes the event trigger fires. O = trigger fires in "origin" and "local" modes, D = trigger is disabled, R = trigger fires in "replica" mode, A = trigger fires always.
evttags	text[]		Command tags for which this trigger will fire. If NULL, the firing of this trigger is not restricted on the basis of the command tag.

49.22. pg_extension

The catalog pg_extension stores information about the installed extensions. See Section 35.15 for details about extensions.

Table 49-22. pg_extension Columns

Name	Type	References	Description

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
extname	name		Name of the extension
extowner	oid	pg_authid.oid	Owner of the extension
extnamespace	oid	pg_namespace.oid	Schema containing the extension's exported objects
extrelocatable	bool		True if extension can be relocated to another schema
extversion	text		Version name for the extension
extconfig	oid[]	pg_class.oid	Array of regclass OIDs for the extension's configuration table(s), or NULL if none
extcondition	text[]		Array of WHERE-clause filter conditions for the extension's configuration table(s), or NULL if none

Note that unlike most catalogs with a "namespace" column, extnamespace is not meant to imply that the extension belongs to that schema. Extension names are never schema-qualified. Rather, extnamespace indicates the schema that contains most or all of the extension's objects. If extrelocatable is true, then this schema must in fact contain all schema-qualifiable objects belonging to the extension.

49.23. pg_foreign_data_wrapper

The catalog pg_foreign_data_wrapper stores foreign-data wrapper definitions. A foreign-data wrapper is the mechanism by which external data, residing on foreign servers, is accessed.

Table 49-23. pg_foreign_data_wrapper Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
fdwname	name		Name of the foreign-data wrapper
fdwowner	oid	pg_authid.oid	Owner of the foreign-data wrapper

Name	Type	References	Description
fdwhandler	oid	pg_proc.oid	References a handler function that is responsible for supplying execution routines for the foreign-data wrapper. Zero if no handler is provided
fdwvalidator	oid	pg_proc.oid	References a validator function that is responsible for checking the validity of the options given to the foreign-data wrapper, as well as options for foreign servers and user mappings using the foreign-data wrapper. Zero if no validator is provided
fdwacl	aclitem[]		Access privileges; see GRANT and REVOKE for details
fdwoptions	text[]		Foreign-data wrapper specific options, as "keyword=value" strings

49.24. pg_foreign_server

The catalog pg_foreign_server stores foreign server definitions. A foreign server describes a source of external data, such as a remote server. Foreign servers are accessed via foreign-data wrappers.

Table 49-24. pg_foreign_server Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
srvname	name		Name of the foreign server
srvowner	oid	pg_authid.oid	Owner of the foreign server

Name	Type	References	Description
srvfdw	oid	pg_foreign_data_wrapper.oid	OID of the foreign-data wrapper of this foreign server
srvtype	text		Type of the server (optional)
srvversion	text		Version of the server (optional)
srvacl	aclitem[]		Access privileges; see GRANT and REVOKE for details
srvoptions	text[]		Foreign server specific options, as "keyword=value" strings

49.25. pg_foreign_table

The catalog pg_foreign_table contains auxiliary information about foreign tables. A foreign table is primarily represented by a pg_class entry, just like a regular table. Its pg_foreign_table entry contains the information that is pertinent only to foreign tables and not any other kind of relation.

Table 49-25. pg_foreign_table Columns

Name	Type	References	Description
ftrelid	oid	pg_class.oid	OID of the pg_class entry for this foreign table
ftserver	oid	pg_foreign_server.oid	OID of the foreign server for this foreign table
ftoptions	text[]		Foreign table options, as "keyword=value" strings

49.26. pg_index

The catalog pg_index contains part of the information about indexes. The rest is mostly in pg_class.

Table 49-26. pg_index Columns

Name	Type	References	Description

Name	Type	References	Description
indexrelid	oid	pg_class.oid	The OID of the pg_class entry for this index
indrelid	oid	pg_class.oid	The OID of the pg_class entry for the table this index is for
indnatts	int2		The number of columns in the index (duplicates pg_class.relnatts)
indisunique	bool		If true, this is a unique index
indisprimary	bool		If true, this index represents the primary key of the table (indisunique should always be true when this is true)
indisexclusion	bool		If true, this index supports an exclusion constraint
indimmediate	bool		If true, the uniqueness check is enforced immediately on insertion (irrelevant if indisunique is not true)
indisclustered	bool		If true, the table was last clustered on this index
indisvalid	bool		If true, the index is currently valid for queries. False means the index is possibly incomplete: it must still be modified by INSERT/UPDATE operations, but it cannot safely be used for queries. If it is unique, the uniqueness property is not guaranteed true either.

Name	Type	References	Description
indcheckxmin	bool		If true, queries must not use the index until the xmin of this pg_index row is below their TransactionXmin event horizon, because the table may contain broken HOT chains with incompatible rows that they can see
indisready	bool		If true, the index is currently ready for inserts. False means the index must be ignored by INSERT/UPDATE operations.
indislive	bool		If false, the index is in process of being dropped, and should be ignored for all purposes (including HOT-safety decisions)
indisreplident	bool		If true this index has been chosen as "replica identity" using ALTER TABLE ... REPLICA IDENTITY USING INDEX ...
indkey	int2vector	pg_attribute.attnum	This is an array of indnatts values that indicate which table columns this index indexes. For example a value of 1 3 would mean that the first and the third table columns make up the index key. A zero in this array indicates that the corresponding index attribute is an expression over the table columns, rather than a simple column reference.

Name	Type	References	Description
indcollation	oidvector	pg_collation.oid	For each column in the index key, this contains the OID of the collation to use for the index.
indclass	oidvector	pg_opclass.oid	For each column in the index key, this contains the OID of the operator class to use. See pg_opclass for details.
indoption	int2vector		This is an array of indnatts values that store per-column flag bits. The meaning of the bits is defined by the index's access method.
indexprs	pg_node_tree		Expression trees (in nodeToString() representation) for index attributes that are not simple column references. This is a list with one element for each zero entry in indkey. Null if all index attributes are simple references.
indpred	pg_node_tree		Expression tree (in nodeToString() representation) for partial index predicate. Null if not a partial index.

49.27. pg_inherits

The catalog pg_inherits records information about table inheritance hierarchies. There is one entry for each direct child table in the database. (Indirect inheritance can be determined by following chains of entries.)

Table 49-27. pg_inherits Columns

Name	Type	References	Description

Name	Type	References	Description
inhrelid	oid	pg_class.oid	The OID of the child table
inhparent	oid	pg_class.oid	The OID of the parent table
inhseqno	int4		If there is more than one direct parent for a child table (multiple inheritance), this number tells the order in which the inherited columns are to be arranged. The count starts at 1.

49.28. pg_language

The catalog pg_language registers languages in which you can write functions or stored procedures. See CREATE LANGUAGE and Chapter 39 for more information about language handlers.

Table 49-28. pg_language Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
lanname	name		Name of the language
lanowner	oid	pg_authid.oid	Owner of the language
lanispl	bool		This is false for internal languages (such as SQL) and true for user-defined languages. Currently, pg_dump still uses this to determine which languages need to be dumped, but this might be replaced by a different mechanism in the future.

Name	Type	References	Description
lanpltrusted	bool		True if this is a trusted language, which means that it is believed not to grant access to anything outside the normal SQL execution environment. Only superusers can create functions in untrusted languages.
lanplcallfoid	oid	pg_proc.oid	For noninternal languages this references the language handler, which is a special function that is responsible for executing all functions that are written in the particular language
laninline	oid	pg_proc.oid	This references a function that is responsible for executing "inline" anonymous code blocks (DO blocks). Zero if inline blocks are not supported.
lanvalidator	oid	pg_proc.oid	This references a language validator function that is responsible for checking the syntax and validity of new functions when they are created. Zero if no validator is provided.
lanacl	aclitem[]		Access privileges; see GRANT and REVOKE for details

49.29. pg_largeobject

The catalog pg_largeobject holds the data making up "large objects". A large object is identified by an OID assigned when it is created. Each large object is broken into segments or "pages" small enough to be conveniently stored as rows in pg_largeobject. The amount of data per page is defined to be

LOBLKSIZE (which is currently BLCKSZ/4, or typically 2 kB).

Prior to PostgreSQL 9.0, there was no permission structure associated with large objects. As a result, pg_largeobject was publicly readable and could be used to obtain the OIDs (and contents) of all large objects in the system. This is no longer the case; use pg_largeobject_metadata to obtain a list of large object OIDs.

Table 49-29. pg_largeobject Columns

Name	Type	References	Description
loid	oid	pg_largeobject_metadata.oid	Identifier of the large object that includes this page
pageno	int4		Page number of this page within its large object (counting from zero)
data	bytea		Actual data stored in the large object. This will never be more than LOBLKSIZE bytes and might be less.

Each row of pg_largeobject holds data for one page of a large object, beginning at byte offset (pageno * LOBLKSIZE) within the object. The implementation allows sparse storage: pages might be missing, and might be shorter than LOBLKSIZE bytes even if they are not the last page of the object. Missing regions within a large object read as zeroes.

49.30. pg_largeobject_metadata

The catalog pg_largeobject_metadata holds metadata associated with large objects. The actual large object data is stored in pg_largeobject.

Table 49-30. pg_largeobject_metadata Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
lomowner	oid	pg_authid.oid	Owner of the large object
lomacl	aclitem[]		Access privileges; see GRANT and REVOKE for details

49.31. pg_namespace

The catalog pg_namespace stores namespaces. A namespace is the structure underlying SQL schemas: each namespace can have a separate collection of relations, types, etc. without name conflicts.

Table 49-31. pg_namespace Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
nspname	name		Name of the namespace
nspowner	oid	pg_authid.oid	Owner of the namespace
nspacl	aclitem[]		Access privileges; see GRANT and REVOKE for details

49.32. pg_opclass

The catalog pg_opclass defines index access method operator classes. Each operator class defines semantics for index columns of a particular data type and a particular index access method. An operator class essentially specifies that a particular operator family is applicable to a particular indexable column data type. The set of operators from the family that are actually usable with the indexed column are whichever ones accept the column's data type as their left-hand input.

Operator classes are described at length in Section 35.14.

Table 49-32. pg_opclass Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
opcmethod	oid	pg_am.oid	Index access method operator class is for
opcname	name		Name of this operator class
opcnamespace	oid	pg_namespace.oid	Namespace of this operator class
opcowner	oid	pg_authid.oid	Owner of the operator class

Name	Type	References	Description
opcfamily	oid	pg_opfamily.oid	Operator family containing the operator class
opcintype	oid	pg_type.oid	Data type that the operator class indexes
opcdefault	bool		True if this operator class is the default for opcintype
opckeytype	oid	pg_type.oid	Type of data stored in index, or zero if same as opcintype

An operator class's opcmethod must match the opfmethod of its containing operator family. Also, there must be no more than one pg_opclass row having opcdefault true for any given combination of opcmethod and opcintype.

49.33. pg_operator

The catalog pg_operator stores information about operators. See CREATE OPERATOR and Section 35.12 for more information.

Table 49-33. pg_operator Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
oprname	name		Name of the operator
oprnamespace	oid	pg_namespace.oid	The OID of the namespace that contains this operator
oprowner	oid	pg_authid.oid	Owner of the operator
oprkind	char		b = infix ("both"), l = prefix ("left"), r = postfix ("right")
oprcanmerge	bool		This operator supports merge joins
oprcanhash	bool		This operator supports hash joins
oprleft	oid	pg_type.oid	Type of the left operand
oprright	oid	pg_type.oid	Type of the right operand

Name	Type	References	Description
oprresult	oid	pg_type.oid	Type of the result
oprcom	oid	pg_operator.oid	Commutator of this operator, if any
oprnegate	oid	pg_operator.oid	Negator of this operator, if any
oprcode	regproc	pg_proc.oid	Function that implements this operator
oprrest	regproc	pg_proc.oid	Restriction selectivity estimation function for this operator
oprjoin	regproc	pg_proc.oid	Join selectivity estimation function for this operator

Unused column contain zeroes. For example, oprleft is zero for a prefix operator.

49.34. pg_opfamily

The catalog pg_opfamily defines operator families. Each operator family is a collection of operators and associated support routines that implement the semantics specified for a particular index access method. Furthermore, the operators in a family are all "compatible", in a way that is specified by the access method. The operator family concept allows cross-data-type operators to be used with indexes and to be reasoned about using knowledge of access method semantics.

Operator families are described at length in Section 35.14.

Table 49-34. pg_opfamily Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
opfmethod	oid	pg_am.oid	Index access method operator family is for
opfname	name		Name of this operator family
opfnamespace	oid	pg_namespace.oid	Namespace of this operator family
opfowner	oid	pg_authid.oid	Owner of the operator family

The majority of the information defining an operator family is not in its pg_opfamily row, but in the associated rows in pg_amop, pg_amproc, and pg_opclass.

49.35. pg_pltemplate

The catalog pg_pltemplate stores "template" information for procedural languages. A template for a language allows the language to be created in a particular database by a simple CREATE LANGUAGE command, with no need to specify implementation details.

Unlike most system catalogs, pg_pltemplate is shared across all databases of a cluster: there is only one copy of pg_pltemplate per cluster, not one per database. This allows the information to be accessible in each database as it is needed.

Table 49-35. pg_pltemplate Columns

Name	Type	Description
tmplname	name	Name of the language this template is for
tmpltrusted	boolean	True if language is considered trusted
tmpldbacreate	boolean	True if language may be created by a database owner
tmplhandler	text	Name of call handler function
tmplinline	text	Name of anonymous-block handler function, or null if none
tmplvalidator	text	Name of validator function, or null if none
tmpllibrary	text	Path of shared library that implements language
tmplacl	aclitem[]	Access privileges for template (not actually used)

There are not currently any commands that manipulate procedural language templates; to change the built-in information, a superuser must modify the table using ordinary INSERT, DELETE, or UPDATE commands.

> **Note:** It is likely that pg_pltemplate will be removed in some future release of PostgreSQL, in favor of keeping this knowledge about procedural languages in their respective extension installation scripts.

49.36. pg_policy

The catalog pg_policy stores row level security policies for tables. A policy includes the kind of command that it applies to (possibly all commands), the roles that it applies to, the expression to be added as a security-barrier qualification to queries that include the table, and the expression to be added as a WITH CHECK option for queries that attempt to add new records to the table.

Table 49-36. pg_policy Columns

Name	Type	References	Description
polname	name		The name of the policy
polrelid	oid	pg_class.oid	The table to which the policy applies
polcmd	char		The command type to which the policy is applied: r for SELECT, a for INSERT, w for UPDATE, d for DELETE, or * for all
polroles	oid[]	pg_authid.oid	The roles to which the policy is applied
polqual	pg_node_tree		The expression tree to be added to the security barrier qualifications for queries that use the table
polwithcheck	pg_node_tree		The expression tree to be added to the WITH CHECK qualifications for queries that attempt to add rows to the table

Note: Policies stored in pg_policy are applied only when pg_class.relrowsecurity is set for their table.

49.37. pg_proc

The catalog pg_proc stores information about functions (or procedures). See CREATE FUNCTION and Section 35.3 for more information.

The table contains data for aggregate functions as well as plain functions. If proisagg is true, there should be a matching row in pg_aggregate.

Table 49-37. pg_proc Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
proname	name		Name of the function

Name	Type	References	Description
pronamespace	oid	pg_namespace.oid	The OID of the namespace that contains this function
proowner	oid	pg_authid.oid	Owner of the function
prolang	oid	pg_language.oid	Implementation language or call interface of this function
procost	float4		Estimated execution cost (in units of cpu_operator_cost); if proretset, this is cost per row returned
prorows	float4		Estimated number of result rows (zero if not proretset)
provariadic	oid	pg_type.oid	Data type of the variadic array parameter's elements, or zero if the function does not have a variadic parameter
protransform	regproc	pg_proc.oid	Calls to this function can be simplified by this other function (see Section 35.9.11)
proisagg	bool		Function is an aggregate function
proiswindow	bool		Function is a window function
prosecdef	bool		Function is a security definer (i.e., a "setuid" function)
proleakproof	bool		The function has no side effects. No information about the arguments is conveyed except via the return value. Any function that might throw an error depending on the values of its arguments is not leak-proof.

Name	Type	References	Description
proisstrict	bool		Function returns null if any call argument is null. In that case the function won't actually be called at all. Functions that are not "strict" must be prepared to handle null inputs.
proretset	bool		Function returns a set (i.e., multiple values of the specified data type)
provolatile	char		provolatile tells whether the function's result depends only on its input arguments, or is affected by outside factors. It is i for "immutable" functions, which always deliver the same result for the same inputs. It is s for "stable" functions, whose results (for fixed inputs) do not change within a scan. It is v for "volatile" functions, whose results might change at any time. (Use v also for functions with side-effects, so that calls to them cannot get optimized away.)
pronargs	int2		Number of input arguments
pronargdefaults	int2		Number of arguments that have defaults
prorettype	oid	pg_type.oid	Data type of the return value

Name	Type	References	Description
proargtypes	oidvector	pg_type.oid	An array with the data types of the function arguments. This includes only input arguments (including INOUT and VARIADIC arguments), and thus represents the call signature of the function.
proallargtypes	oid[]	pg_type.oid	An array with the data types of the function arguments. This includes all arguments (including OUT and INOUT arguments); however, if all the arguments are IN arguments, this field will be null. Note that subscripting is 1-based, whereas for historical reasons proargtypes is subscripted from 0.
proargmodes	char[]		An array with the modes of the function arguments, encoded as i for IN arguments, o for OUT arguments, b for INOUT arguments, v for VARIADIC arguments, t for TABLE arguments. If all the arguments are IN arguments, this field will be null. Note that subscripts correspond to positions of proallargtypes not proargtypes.

Name	Type	References	Description
proargnames	text[]		An array with the names of the function arguments. Arguments without a name are set to empty strings in the array. If none of the arguments have a name, this field will be null. Note that subscripts correspond to positions of proallargtypes not proargtypes.
proargdefaults	pg_node_tree		Expression trees (in nodeToString() representation) for default values. This is a list with pronargdefaults elements, corresponding to the last *N* *input* arguments (i.e., the last *N* proargtypes positions). If none of the arguments have defaults, this field will be null.
protrftypes	oid[]		Data type OIDs for which to apply transforms.
prosrc	text		This tells the function handler how to invoke the function. It might be the actual source code of the function for interpreted languages, a link symbol, a file name, or just about anything else, depending on the implementation language/call convention.
probin	text		Additional information about how to invoke the function. Again, the interpretation is language-specific.

Name	Type	References	Description
proconfig	text[]		Function's local settings for run-time configuration variables
proacl	aclitem[]		Access privileges; see GRANT and REVOKE for details

For compiled functions, both built-in and dynamically loaded, prosrc contains the function's C-language name (link symbol). For all other currently-known language types, prosrc contains the function's source text. probin is unused except for dynamically-loaded C functions, for which it gives the name of the shared library file containing the function.

49.38. pg_range

The catalog pg_range stores information about range types. This is in addition to the types' entries in pg_type.

Table 49-38. pg_range Columns

Name	Type	References	Description
rngtypid	oid	pg_type.oid	OID of the range type
rngsubtype	oid	pg_type.oid	OID of the element type (subtype) of this range type
rngcollation	oid	pg_collation.oid	OID of the collation used for range comparisons, or 0 if none
rngsubopc	oid	pg_opclass.oid	OID of the subtype's operator class used for range comparisons
rngcanonical	regproc	pg_proc.oid	OID of the function to convert a range value into canonical form, or 0 if none
rngsubdiff	regproc	pg_proc.oid	OID of the function to return the difference between two element values as double precision, or 0 if none

rngsubopc (plus rngcollation, if the element type is collatable) determines the sort ordering used by the range type. rngcanonical is used when the element type is discrete. rngsubdiff is optional but

should be supplied to improve performance of GiST indexes on the range type.

49.39. pg_replication_origin

The pg_replication_origin catalog contains all replication origins created. For more on replication origins see Chapter 47.

Table 49-39. pg_replication_origin Columns

Name	Type	References	Description
roident	Oid		A unique, cluster-wide identifier for the replication origin. Should never leave the system.
roname	text		The external, user defined, name of a replication origin.

49.40. pg_rewrite

The catalog pg_rewrite stores rewrite rules for tables and views.

Table 49-40. pg_rewrite Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
rulename	name		Rule name
ev_class	oid	pg_class.oid	The table this rule is for
ev_type	char		Event type that the rule is for: 1 = SELECT, 2 = UPDATE, 3 = INSERT, 4 = DELETE
ev_enabled	char		Controls in which session_replication_role modes the rule fires. O = rule fires in "origin" and "local" modes, D = rule is disabled, R = rule fires in "replica" mode, A = rule fires always.

Name	Type	References	Description
is_instead	bool		True if the rule is an INSTEAD rule
ev_qual	pg_node_tree		Expression tree (in the form of a nodeToString() representation) for the rule's qualifying condition
ev_action	pg_node_tree		Query tree (in the form of a nodeToString() representation) for the rule's action

Note: pg_class.relhasrules must be true if a table has any rules in this catalog.

49.41. pg_seclabel

The catalog pg_seclabel stores security labels on database objects. Security labels can be manipulated with the SECURITY LABEL command. For an easier way to view security labels, see Section 49.72.

See also pg_shseclabel, which performs a similar function for security labels of database objects that are shared across a database cluster.

Table 49-41. pg_seclabel Columns

Name	Type	References	Description
objoid	oid	any OID column	The OID of the object this security label pertains to
classoid	oid	pg_class.oid	The OID of the system catalog this object appears in
objsubid	int4		For a security label on a table column, this is the column number (the objoid and classoid refer to the table itself). For all other object types, this column is zero.

Name	Type	References	Description
provider	text		The label provider associated with this label.
label	text		The security label applied to this object.

49.42. pg_shdepend

The catalog pg_shdepend records the dependency relationships between database objects and shared objects, such as roles. This information allows PostgreSQL to ensure that those objects are unreferenced before attempting to delete them.

See also pg_depend, which performs a similar function for dependencies involving objects within a single database.

Unlike most system catalogs, pg_shdepend is shared across all databases of a cluster: there is only one copy of pg_shdepend per cluster, not one per database.

Table 49-42. pg_shdepend Columns

Name	Type	References	Description
dbid	oid	pg_database.oid	The OID of the database the dependent object is in, or zero for a shared object
classid	oid	pg_class.oid	The OID of the system catalog the dependent object is in
objid	oid	any OID column	The OID of the specific dependent object
objsubid	int4		For a table column, this is the column number (the objid and classid refer to the table itself). For all other object types, this column is zero.
refclassid	oid	pg_class.oid	The OID of the system catalog the referenced object is in (must be a shared catalog)
refobjid	oid	any OID column	The OID of the specific referenced object

Name	Type	References	Description
deptype	char		A code defining the specific semantics of this dependency relationship; see text

In all cases, a pg_shdepend entry indicates that the referenced object cannot be dropped without also dropping the dependent object. However, there are several subflavors identified by deptype:

SHARED_DEPENDENCY_OWNER (o)

The referenced object (which must be a role) is the owner of the dependent object.

SHARED_DEPENDENCY_ACL (a)

The referenced object (which must be a role) is mentioned in the ACL (access control list, i.e., privileges list) of the dependent object. (A SHARED_DEPENDENCY_ACL entry is not made for the owner of the object, since the owner will have a SHARED_DEPENDENCY_OWNER entry anyway.)

SHARED_DEPENDENCY_POLICY (r)

The referenced object (which must be a role) is mentioned as the target of a dependent policy object.

SHARED_DEPENDENCY_PIN (p)

There is no dependent object; this type of entry is a signal that the system itself depends on the referenced object, and so that object must never be deleted. Entries of this type are created only by initdb. The columns for the dependent object contain zeroes.

Other dependency flavors might be needed in future. Note in particular that the current definition only supports roles as referenced objects.

49.43. pg_shdescription

The catalog pg_shdescription stores optional descriptions (comments) for shared database objects. Descriptions can be manipulated with the COMMENT command and viewed with psql's \d commands.

See also pg_description, which performs a similar function for descriptions involving objects within a single database.

Unlike most system catalogs, pg_shdescription is shared across all databases of a cluster: there is only one copy of pg_shdescription per cluster, not one per database.

Table 49-43. pg_shdescription Columns

Name	Type	References	Description
objoid	oid	any OID column	The OID of the object this description pertains to

Name	Type	References	Description
classoid	oid	pg_class.oid	The OID of the system catalog this object appears in
description	text		Arbitrary text that serves as the description of this object

49.44. pg_shseclabel

The catalog pg_shseclabel stores security labels on shared database objects. Security labels can be manipulated with the SECURITY LABEL command. For an easier way to view security labels, see Section 49.72.

See also pg_seclabel, which performs a similar function for security labels involving objects within a single database.

Unlike most system catalogs, pg_shseclabel is shared across all databases of a cluster: there is only one copy of pg_shseclabel per cluster, not one per database.

Table 49-44. pg_shseclabel Columns

Name	Type	References	Description
objoid	oid	any OID column	The OID of the object this security label pertains to
classoid	oid	pg_class.oid	The OID of the system catalog this object appears in
provider	text		The label provider associated with this label.
label	text		The security label applied to this object.

49.45. pg_statistic

The catalog pg_statistic stores statistical data about the contents of the database. Entries are created by ANALYZE and subsequently used by the query planner. Note that all the statistical data is inherently approximate, even assuming that it is up-to-date.

Normally there is one entry, with stainherit = false, for each table column that has been analyzed. If the table has inheritance children, a second entry with stainherit = true is also created. This row represents the column's statistics over the inheritance tree, i.e., statistics for the data you'd see with SELECT

column FROM *table**, whereas the stainherit = false row represents the results of SELECT *column* FROM ONLY *table*.

pg_statistic also stores statistical data about the values of index expressions. These are described as if they were actual data columns; in particular, starelid references the index. No entry is made for an ordinary non-expression index column, however, since it would be redundant with the entry for the underlying table column. Currently, entries for index expressions always have stainherit = false.

Since different kinds of statistics might be appropriate for different kinds of data, pg_statistic is designed not to assume very much about what sort of statistics it stores. Only extremely general statistics (such as nullness) are given dedicated columns in pg_statistic. Everything else is stored in "slots", which are groups of associated columns whose content is identified by a code number in one of the slot's columns. For more information see src/include/catalog/pg_statistic.h.

pg_statistic should not be readable by the public, since even statistical information about a table's contents might be considered sensitive. (Example: minimum and maximum values of a salary column might be quite interesting.) pg_stats is a publicly readable view on pg_statistic that only exposes information about those tables that are readable by the current user.

Table 49-45. pg_statistic Columns

Name	Type	References	Description
starelid	oid	pg_class.oid	The table or index that the described column belongs to
staattnum	int2	pg_attribute.attnum	The number of the described column
stainherit	bool		If true, the stats include inheritance child columns, not just the values in the specified relation
stanullfrac	float4		The fraction of the column's entries that are null
stawidth	int4		The average stored width, in bytes, of nonnull entries

Name	Type	References	Description
stadistinct	float4		The number of distinct nonnull data values in the column. A value greater than zero is the actual number of distinct values. A value less than zero is the negative of a multiplier for the number of rows in the table; for example, a column in which about 80% of the values are nonnull and each nonnull value appears about twice on average could be represented by stadistinct = -0.4. A zero value means the number of distinct values is unknown.
stakind*N*	int2		A code number indicating the kind of statistics stored in the *N*th "slot" of the pg_statistic row.
staop*N*	oid	pg_operator.oid	An operator used to derive the statistics stored in the *N*th "slot". For example, a histogram slot would show the < operator that defines the sort order of the data.
stanumbers*N*	float4[]		Numerical statistics of the appropriate kind for the *N*th "slot", or null if the slot kind does not involve numerical values

Name	Type	References	Description
stavalues*N*	anyarray		Column data values of the appropriate kind for the *N*th "slot", or null if the slot kind does not store any data values. Each array's element values are actually of the specific column's data type, or a related type such as an array's element type, so there is no way to define these columns' type more specifically than anyarray.

49.46. pg_tablespace

The catalog pg_tablespace stores information about the available tablespaces. Tables can be placed in particular tablespaces to aid administration of disk layout.

Unlike most system catalogs, pg_tablespace is shared across all databases of a cluster: there is only one copy of pg_tablespace per cluster, not one per database.

Table 49-46. pg_tablespace Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
spcname	name		Tablespace name
spcowner	oid	pg_authid.oid	Owner of the tablespace, usually the user who created it
spcacl	aclitem[]		Access privileges; see GRANT and REVOKE for details
spcoptions	text[]		Tablespace-level options, as "keyword=value" strings

49.47. pg_transform

The catalog pg_transform stores information about transforms, which are a mechanism to adapt data types to procedural languages. See CREATE TRANSFORM for more information.

Table 49-47. pg_transform Columns

Name	Type	References	Description
trftype	oid	pg_type.oid	OID of the data type this transform is for
trflang	oid	pg_language.oid	OID of the language this transform is for
trffromsql	regproc	pg_proc.oid	The OID of the function to use when converting the data type for input to the procedural language (e.g., function parameters). Zero is stored if this operation is not supported.
trftosql	regproc	pg_proc.oid	The OID of the function to use when converting output from the procedural language (e.g., return values) to the data type. Zero is stored if this operation is not supported.

49.48. pg_trigger

The catalog pg_trigger stores triggers on tables and views. See CREATE TRIGGER for more information.

Table 49-48. pg_trigger Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
tgrelid	oid	pg_class.oid	The table this trigger is on

Name	Type	References	Description
tgname	name		Trigger name (must be unique among triggers of same table)
tgfoid	oid	pg_proc.oid	The function to be called
tgtype	int2		Bit mask identifying trigger firing conditions
tgenabled	char		Controls in which session_replication_role modes the trigger fires. O = trigger fires in "origin" and "local" modes, D = trigger is disabled, R = trigger fires in "replica" mode, A = trigger fires always.
tgisinternal	bool		True if trigger is internally generated (usually, to enforce the constraint identified by tgconstraint)
tgconstrrelid	oid	pg_class.oid	The table referenced by a referential integrity constraint
tgconstrindid	oid	pg_class.oid	The index supporting a unique, primary key, referential integrity, or exclusion constraint
tgconstraint	oid	pg_constraint.oid	The pg_constraint entry associated with the trigger, if any
tgdeferrable	bool		True if constraint trigger is deferrable
tginitdeferred	bool		True if constraint trigger is initially deferred
tgnargs	int2		Number of argument strings passed to trigger function
tgattr	int2vector	pg_attribute.attnum	Column numbers, if trigger is column-specific; otherwise an empty array

Name	Type	References	Description
tgargs	bytea		Argument strings to pass to trigger, each NULL-terminated
tgqual	pg_node_tree		Expression tree (in nodeToString() representation) for the trigger's WHEN condition, or null if none

Currently, column-specific triggering is supported only for UPDATE events, and so tgattr is relevant only for that event type. tgtype might contain bits for other event types as well, but those are presumed to be table-wide regardless of what is in tgattr.

Note: When tgconstraint is nonzero, tgconstrrelid, tgconstrindid, tgdeferrable, and tginitdeferred are largely redundant with the referenced pg_constraint entry. However, it is possible for a non-deferrable trigger to be associated with a deferrable constraint: foreign key constraints can have some deferrable and some non-deferrable triggers.

Note: pg_class.relhastriggers must be true if a relation has any triggers in this catalog.

49.49. pg_ts_config

The pg_ts_config catalog contains entries representing text search configurations. A configuration specifies a particular text search parser and a list of dictionaries to use for each of the parser's output token types. The parser is shown in the pg_ts_config entry, but the token-to-dictionary mapping is defined by subsidiary entries in pg_ts_config_map.

PostgreSQL's text search features are described at length in Chapter 12.

Table 49-49. pg_ts_config Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
cfgname	name		Text search configuration name
cfgnamespace	oid	pg_namespace.oid	The OID of the namespace that contains this configuration

Name	Type	References	Description
cfgowner	oid	pg_authid.oid	Owner of the configuration
cfgparser	oid	pg_ts_parser.oid	The OID of the text search parser for this configuration

49.50. pg_ts_config_map

The pg_ts_config_map catalog contains entries showing which text search dictionaries should be consulted, and in what order, for each output token type of each text search configuration's parser.

PostgreSQL's text search features are described at length in Chapter 12.

Table 49-50. pg_ts_config_map Columns

Name	Type	References	Description
mapcfg	oid	pg_ts_config.oid	The OID of the pg_ts_config entry owning this map entry
maptokentype	integer		A token type emitted by the configuration's parser
mapseqno	integer		Order in which to consult this entry (lower mapseqnos first)
mapdict	oid	pg_ts_dict.oid	The OID of the text search dictionary to consult

49.51. pg_ts_dict

The pg_ts_dict catalog contains entries defining text search dictionaries. A dictionary depends on a text search template, which specifies all the implementation functions needed; the dictionary itself provides values for the user-settable parameters supported by the template. This division of labor allows dictionaries to be created by unprivileged users. The parameters are specified by a text string dictinitoption, whose format and meaning vary depending on the template.

PostgreSQL's text search features are described at length in Chapter 12.

Table 49-51. pg_ts_dict Columns

Name	Type	References	Description

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
dictname	name		Text search dictionary name
dictnamespace	oid	pg_namespace.oid	The OID of the namespace that contains this dictionary
dictowner	oid	pg_authid.oid	Owner of the dictionary
dicttemplate	oid	pg_ts_template.oid	The OID of the text search template for this dictionary
dictinitoption	text		Initialization option string for the template

49.52. pg_ts_parser

The pg_ts_parser catalog contains entries defining text search parsers. A parser is responsible for splitting input text into lexemes and assigning a token type to each lexeme. Since a parser must be implemented by C-language-level functions, creation of new parsers is restricted to database superusers.

PostgreSQL's text search features are described at length in Chapter 12.

Table 49-52. pg_ts_parser Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
prsname	name		Text search parser name
prsnamespace	oid	pg_namespace.oid	The OID of the namespace that contains this parser
prsstart	regproc	pg_proc.oid	OID of the parser's startup function
prstoken	regproc	pg_proc.oid	OID of the parser's next-token function
prsend	regproc	pg_proc.oid	OID of the parser's shutdown function
prsheadline	regproc	pg_proc.oid	OID of the parser's headline function

Name	Type	References	Description
prslextype	regproc	pg_proc.oid	OID of the parser's lextype function

49.53. pg_ts_template

The pg_ts_template catalog contains entries defining text search templates. A template is the implementation skeleton for a class of text search dictionaries. Since a template must be implemented by C-language-level functions, creation of new templates is restricted to database superusers.

PostgreSQL's text search features are described at length in Chapter 12.

Table 49-53. pg_ts_template Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
tmplname	name		Text search template name
tmplnamespace	oid	pg_namespace.oid	The OID of the namespace that contains this template
tmplinit	regproc	pg_proc.oid	OID of the template's initialization function
tmpllexize	regproc	pg_proc.oid	OID of the template's lexize function

49.54. pg_type

The catalog pg_type stores information about data types. Base types and enum types (scalar types) are created with CREATE TYPE, and domains with CREATE DOMAIN. A composite type is automatically created for each table in the database, to represent the row structure of the table. It is also possible to create composite types with CREATE TYPE AS.

Table 49-54. pg_type Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
typname	name		Data type name

Name	Type	References	Description
typnamespace	oid	pg_namespace.oid	The OID of the namespace that contains this type
typowner	oid	pg_authid.oid	Owner of the type
typlen	int2		For a fixed-size type, typlen is the number of bytes in the internal representation of the type. But for a variable-length type, typlen is negative. -1 indicates a "varlena" type (one that has a length word), -2 indicates a null-terminated C string.
typbyval	bool		typbyval determines whether internal routines pass a value of this type by value or by reference. typbyval had better be false if typlen is not 1, 2, or 4 (or 8 on machines where Datum is 8 bytes). Variable-length types are always passed by reference. Note that typbyval can be false even if the length would allow pass-by-value.
typtype	char		typtype is b for a base type, c for a composite type (e.g., a table's row type), d for a domain, e for an enum type, p for a pseudo-type, or r for a range type. See also typrelid and typbasetype.

Name	Type	References	Description
typcategory	char		typcategory is an arbitrary classification of data types that is used by the parser to determine which implicit casts should be "preferred". See Table 49-55.
typispreferred	bool		True if the type is a preferred cast target within its typcategory
typisdefined	bool		True if the type is defined, false if this is a placeholder entry for a not-yet-defined type. When typisdefined is false, nothing except the type name, namespace, and OID can be relied on.
typdelim	char		Character that separates two values of this type when parsing array input. Note that the delimiter is associated with the array element data type, not the array data type.
typrelid	oid	pg_class.oid	If this is a composite type (see typtype), then this column points to the pg_class entry that defines the corresponding table. (For a free-standing composite type, the pg_class entry doesn't really represent a table, but it is needed anyway for the type's pg_attribute entries to link to.) Zero for non-composite types.

Name	Type	References	Description
typelem	oid	pg_type.oid	If typelem is not 0 then it identifies another row in pg_type. The current type can then be subscripted like an array yielding values of type typelem. A "true" array type is variable length (typlen = -1), but some fixed-length (typlen > 0) types also have nonzero typelem, for example name and point. If a fixed-length type has a typelem then its internal representation must be some number of values of the typelem data type with no other data. Variable-length array types have a header defined by the array subroutines.
typarray	oid	pg_type.oid	If typarray is not 0 then it identifies another row in pg_type, which is the "true" array type having this type as element
typinput	regproc	pg_proc.oid	Input conversion function (text format)
typoutput	regproc	pg_proc.oid	Output conversion function (text format)
typreceive	regproc	pg_proc.oid	Input conversion function (binary format), or 0 if none
typsend	regproc	pg_proc.oid	Output conversion function (binary format), or 0 if none
typmodin	regproc	pg_proc.oid	Type modifier input function, or 0 if type does not support modifiers

Name	Type	References	Description
typmodout	regproc	pg_proc.oid	Type modifier output function, or 0 to use the standard format
typanalyze	regproc	pg_proc.oid	Custom ANALYZE function, or 0 to use the standard function

Name	Type	References	Description
`typalign`	`char`		`typalign` is the alignment required when storing a value of this type. It applies to storage on disk as well as most representations of the value inside PostgreSQL. When multiple values are stored consecutively, such as in the representation of a complete row on disk, padding is inserted before a datum of this type so that it begins on the specified boundary. The alignment reference is the beginning of the first datum in the sequence. Possible values are: • `c` = `char` alignment, i.e., no alignment needed. • `s` = `short` alignment (2 bytes on most machines). • `i` = `int` alignment (4 bytes on most machines). • `d` = `double` alignment (8 bytes on many machines, but by no means all). **Note:** For types used in system tables, it is critical that the size and alignment defined in `pg_type` agree with the way that the compiler will lay out the column in a structure representing a table row.

Name	Type	References	Description
typstorage	char		typstorage tells for varlena types (those with typlen = -1) if the type is prepared for toasting and what the default strategy for attributes of this type should be. Possible values are • p: Value must always be stored plain. • e: Value can be stored in a "secondary" relation (if relation has one, see pg_class.reltoastrelid). • m: Value can be stored compressed inline. • x: Value can be stored compressed inline or stored in "secondary" storage. Note that m columns can also be moved out to secondary storage, but only as a last resort (e and x columns are moved first).
typnotnull	bool		typnotnull represents a not-null constraint on a type. Used for domains only.
typbasetype	oid	pg_type.oid	If this is a domain (see typtype), then typbasetype identifies the type that this one is based on. Zero if this type is not a domain.

Name	Type	References	Description
typtypmod	int4		Domains use typtypmod to record the typmod to be applied to their base type (-1 if base type does not use a typmod). -1 if this type is not a domain.
typndims	int4		typndims is the number of array dimensions for a domain over an array (that is, typbasetype is an array type). Zero for types other than domains over array types.
typcollation	oid	pg_collation.oid	typcollation specifies the collation of the type. If the type does not support collations, this will be zero. A base type that supports collations will have DEFAULT_COLLATION_OID here. A domain over a collatable type can have some other collation OID, if one was specified for the domain.
typdefaultbin	pg_node_tree		If typdefaultbin is not null, it is the nodeToString() representation of a default expression for the type. This is only used for domains.

Name	Type	References	Description
typdefault	text		typdefault is null if the type has no associated default value. If typdefaultbin is not null, typdefault must contain a human-readable version of the default expression represented by typdefaultbin. If typdefaultbin is null and typdefault is not, then typdefault is the external representation of the type's default value, which can be fed to the type's input converter to produce a constant.
typacl	aclitem[]		Access privileges; see GRANT and REVOKE for details

Table 49-55 lists the system-defined values of typcategory. Any future additions to this list will also be upper-case ASCII letters. All other ASCII characters are reserved for user-defined categories.

Table 49-55. typcategory Codes

Code	Category
A	Array types
B	Boolean types
C	Composite types
D	Date/time types
E	Enum types
G	Geometric types
I	Network address types
N	Numeric types
P	Pseudo-types
R	Range types
S	String types
T	Timespan types
U	User-defined types
V	Bit-string types

Code	Category
X	unknown type

49.55. pg_user_mapping

The catalog pg_user_mapping stores the mappings from local user to remote. Access to this catalog is restricted from normal users, use the view pg_user_mappings instead.

Table 49-56. pg_user_mapping Columns

Name	Type	References	Description
oid	oid		Row identifier (hidden attribute; must be explicitly selected)
umuser	oid	pg_authid.oid	OID of the local role being mapped, 0 if the user mapping is public
umserver	oid	pg_foreign_server.oid	The OID of the foreign server that contains this mapping
umoptions	text[]		User mapping specific options, as "keyword=value" strings

49.56. System Views

In addition to the system catalogs, PostgreSQL provides a number of built-in views. Some system views provide convenient access to some commonly used queries on the system catalogs. Other views provide access to internal server state.

The information schema (Chapter 34) provides an alternative set of views which overlap the functionality of the system views. Since the information schema is SQL-standard whereas the views described here are PostgreSQL-specific, it's usually better to use the information schema if it provides all the information you need.

Table 49-57 lists the system views described here. More detailed documentation of each view follows below. There are some additional views that provide access to the results of the statistics collector; they are described in Table 27-2.

Except where noted, all the views described here are read-only.

Table 49-57. System Views

View Name	Purpose
pg_available_extensions	available extensions
pg_available_extension_versions	available versions of extensions
pg_cursors	open cursors
pg_file_settings	summary of configuration file contents
pg_group	groups of database users
pg_indexes	indexes
pg_locks	currently held locks
pg_matviews	materialized views
pg_policies	policies
pg_prepared_statements	prepared statements
pg_prepared_xacts	prepared transactions
pg_replication_origin_status	information about replication origins, including replication progress
pg_replication_slots	replication slot information
pg_roles	database roles
pg_rules	rules
pg_seclabels	security labels
pg_settings	parameter settings
pg_shadow	database users
pg_stats	planner statistics
pg_tables	tables
pg_timezone_abbrevs	time zone abbreviations
pg_timezone_names	time zone names
pg_user	database users
pg_user_mappings	user mappings
pg_views	views

49.57. pg_available_extensions

The pg_available_extensions view lists the extensions that are available for installation. See also the pg_extension catalog, which shows the extensions currently installed.

Table 49-58. pg_available_extensions Columns

Name	Type	Description
name	name	Extension name
default_version	text	Name of default version, or NULL if none is specified

Name	Type	Description
installed_version	text	Currently installed version of the extension, or NULL if not installed
comment	text	Comment string from the extension's control file

The pg_available_extensions view is read only.

49.58. pg_available_extension_versions

The pg_available_extension_versions view lists the specific extension versions that are available for installation. See also the pg_extension catalog, which shows the extensions currently installed.

Table 49-59. pg_available_extension_versions Columns

Name	Type	Description
name	name	Extension name
version	text	Version name
installed	bool	True if this version of this extension is currently installed
superuser	bool	True if only superusers are allowed to install this extension
relocatable	bool	True if extension can be relocated to another schema
schema	name	Name of the schema that the extension must be installed into, or NULL if partially or fully relocatable
requires	name[]	Names of prerequisite extensions, or NULL if none
comment	text	Comment string from the extension's control file

The pg_available_extension_versions view is read only.

49.59. pg_cursors

The pg_cursors view lists the cursors that are currently available. Cursors can be defined in several ways:

- via the DECLARE statement in SQL

- via the Bind message in the frontend/backend protocol, as described in Section 50.2.3
- via the Server Programming Interface (SPI), as described in Section 44.1

The pg_cursors view displays cursors created by any of these means. Cursors only exist for the duration of the transaction that defines them, unless they have been declared WITH HOLD. Therefore non-holdable cursors are only present in the view until the end of their creating transaction.

> **Note:** Cursors are used internally to implement some of the components of PostgreSQL, such as procedural languages. Therefore, the pg_cursors view might include cursors that have not been explicitly created by the user.

Table 49-60. pg_cursors Columns

Name	Type	Description
name	text	The name of the cursor
statement	text	The verbatim query string submitted to declare this cursor
is_holdable	boolean	true if the cursor is holdable (that is, it can be accessed after the transaction that declared the cursor has committed); false otherwise
is_binary	boolean	true if the cursor was declared BINARY; false otherwise
is_scrollable	boolean	true if the cursor is scrollable (that is, it allows rows to be retrieved in a nonsequential manner); false otherwise
creation_time	timestamptz	The time at which the cursor was declared

The pg_cursors view is read only.

49.60. pg_file_settings

The view pg_file_settings provides a summary of the contents of the server's configuration file(s). A row appears in this view for each "name = value" entry appearing in the files, with annotations indicating whether the value could be applied successfully. Additional row(s) may appear for problems not linked to a "name = value" entry, such as syntax errors in the files.

This view is helpful for checking whether planned changes in the configuration files will work, or for diagnosing a previous failure. Note that this view reports on the *current* contents of the files, not on what was last applied by the server. (The pg_settings view is usually sufficient to determine that.)

The pg_file_settings view can be read only by superusers.

Table 49-61. pg_file_settings Columns

Name	Type	Description
sourcefile	text	Full path name of the configuration file
sourceline	integer	Line number within the configuration file where the entry appears
seqno	integer	Order in which the entries are processed (1..*n*)
name	text	Configuration parameter name
setting	text	Value to be assigned to the parameter
applied	boolean	True if the value can be applied successfully
error	text	If not null, an error message indicating why this entry could not be applied

If the configuration file contains syntax errors or invalid parameter names, the server will not attempt to apply any settings from it, and therefore all the applied fields will read as false. In such a case there will be one or more rows with non-null error fields indicating the problem(s). Otherwise, individual settings will be applied if possible. If an individual setting cannot be applied (e.g., invalid value, or the setting cannot be changed after server start) it will have an appropriate message in the error field. Another way that an entry might have applied = false is that it is overridden by a later entry for the same parameter name; this case is not considered an error so nothing appears in the error field.

See Section 18.1 for more information about the various ways to change run-time parameters.

49.61. pg_group

The view pg_group exists for backwards compatibility: it emulates a catalog that existed in PostgreSQL before version 8.1. It shows the names and members of all roles that are marked as not rolcanlogin, which is an approximation to the set of roles that are being used as groups.

Table 49-62. pg_group Columns

Name	Type	References	Description
groname	name	pg_authid.rolname	Name of the group
grosysid	oid	pg_authid.oid	ID of this group
grolist	oid[]	pg_authid.oid	An array containing the IDs of the roles in this group

49.62. pg_indexes

The view pg_indexes provides access to useful information about each index in the database.

Table 49-63. pg_indexes Columns

Name	Type	References	Description
schemaname	name	pg_namespace.nspname	Name of schema containing table and index
tablename	name	pg_class.relname	Name of table the index is for
indexname	name	pg_class.relname	Name of index
tablespace	name	pg_tablespace.spcname	Name of tablespace containing index (null if default for database)
indexdef	text		Index definition (a reconstructed CREATE INDEX command)

49.63. pg_locks

The view pg_locks provides access to information about the locks held by open transactions within the database server. See Chapter 13 for more discussion of locking.

pg_locks contains one row per active lockable object, requested lock mode, and relevant transaction. Thus, the same lockable object might appear many times, if multiple transactions are holding or waiting for locks on it. However, an object that currently has no locks on it will not appear at all.

There are several distinct types of lockable objects: whole relations (e.g., tables), individual pages of relations, individual tuples of relations, transaction IDs (both virtual and permanent IDs), and general database objects (identified by class OID and object OID, in the same way as in pg_description or pg_depend). Also, the right to extend a relation is represented as a separate lockable object, as is the right to update pg_database.datfrozenxid. Also, "advisory" locks can be taken on numbers that have user-defined meanings.

Table 49-64. pg_locks Columns

Name	Type	References	Description

Name	Type	References	Description
locktype	text		Type of the lockable object: relation, extend, frozenid, page, tuple, transactionid, virtualxid, object, userlock, or advisory
database	oid	pg_database.oid	OID of the database in which the lock target exists, or zero if the target is a shared object, or null if the target is a transaction ID
relation	oid	pg_class.oid	OID of the relation targeted by the lock, or null if the target is not a relation or part of a relation
page	integer		Page number targeted by the lock within the relation, or null if the target is not a relation page or tuple
tuple	smallint		Tuple number targeted by the lock within the page, or null if the target is not a tuple
virtualxid	text		Virtual ID of the transaction targeted by the lock, or null if the target is not a virtual transaction ID
transactionid	xid		ID of the transaction targeted by the lock, or null if the target is not a transaction ID
classid	oid	pg_class.oid	OID of the system catalog containing the lock target, or null if the target is not a general database object

Name	Type	References	Description
objid	oid	any OID column	OID of the lock target within its system catalog, or null if the target is not a general database object
objsubid	smallint		Column number targeted by the lock (the classid and objid refer to the table itself), or zero if the target is some other general database object, or null if the target is not a general database object
virtualtransaction	text		Virtual ID of the transaction that is holding or awaiting this lock
pid	integer		Process ID of the server process holding or awaiting this lock, or null if the lock is held by a prepared transaction
mode	text		Name of the lock mode held or desired by this process (see Section 13.3.1 and Section 13.2.3)
granted	boolean		True if lock is held, false if lock is awaited
fastpath	boolean		True if lock was taken via fast path, false if taken via main lock table

granted is true in a row representing a lock held by the indicated transaction. False indicates that this transaction is currently waiting to acquire this lock, which implies that some other transaction is holding a conflicting lock mode on the same lockable object. The waiting transaction will sleep until the other lock is released (or a deadlock situation is detected). A single transaction can be waiting to acquire at most one lock at a time.

Every transaction holds an exclusive lock on its virtual transaction ID for its entire duration. If a permanent ID is assigned to the transaction (which normally happens only if the transaction changes the state of the database), it also holds an exclusive lock on its permanent transaction ID until it ends. When one transaction finds it necessary to wait specifically for another transaction, it does so by attempting to acquire share lock on the other transaction ID (either virtual or permanent ID depending on the situation). That

will succeed only when the other transaction terminates and releases its locks.

Although tuples are a lockable type of object, information about row-level locks is stored on disk, not in memory, and therefore row-level locks normally do not appear in this view. If a transaction is waiting for a row-level lock, it will usually appear in the view as waiting for the permanent transaction ID of the current holder of that row lock.

Advisory locks can be acquired on keys consisting of either a single `bigint` value or two integer values. A `bigint` key is displayed with its high-order half in the `classid` column, its low-order half in the `objid` column, and `objsubid` equal to 1. The original `bigint` value can be reassembled with the expression `(classid::bigint << 32) | objid::bigint`. Integer keys are displayed with the first key in the `classid` column, the second key in the `objid` column, and `objsubid` equal to 2. The actual meaning of the keys is up to the user. Advisory locks are local to each database, so the `database` column is meaningful for an advisory lock.

`pg_locks` provides a global view of all locks in the database cluster, not only those relevant to the current database. Although its `relation` column can be joined against `pg_class.oid` to identify locked relations, this will only work correctly for relations in the current database (those for which the `database` column is either the current database's OID or zero).

The `pid` column can be joined to the `pid` column of the `pg_stat_activity` view to get more information on the session holding or waiting to hold each lock, for example

```
SELECT * FROM pg_locks pl LEFT JOIN pg_stat_activity psa
    ON pl.pid = psa.pid;
```

Also, if you are using prepared transactions, the `virtualtransaction` column can be joined to the `transaction` column of the `pg_prepared_xacts` view to get more information on prepared transactions that hold locks. (A prepared transaction can never be waiting for a lock, but it continues to hold the locks it acquired while running.) For example:

```
SELECT * FROM pg_locks pl LEFT JOIN pg_prepared_xacts ppx
    ON pl.virtualtransaction = '-1/' || ppx.transaction;
```

The `pg_locks` view displays data from both the regular lock manager and the predicate lock manager, which are separate systems; in addition, the regular lock manager subdivides its locks into regular and *fast-path* locks. This data is not guaranteed to be entirely consistent. When the view is queried, data on fast-path locks (with `fastpath` = `true`) is gathered from each backend one at a time, without freezing the state of the entire lock manager, so it is possible for locks to be taken or released while information is gathered. Note, however, that these locks are known not to conflict with any other lock currently in place. After all backends have been queried for fast-path locks, the remainder of the regular lock manager is locked as a unit, and a consistent snapshot of all remaining locks is collected as an atomic action. After unlocking the regular lock manager, the predicate lock manager is similarly locked and all predicate locks are collected as an atomic action. Thus, with the exception of fast-path locks, each lock manager will deliver a consistent set of results, but as we do not lock both lock managers simultaneously, it is possible for locks to be taken or released after we interrogate the regular lock manager and before we interrogate the predicate lock manager.

Locking the regular and/or predicate lock manager could have some impact on database performance if this view is very frequently accessed. The locks are held only for the minimum amount of time neces-

sary to obtain data from the lock managers, but this does not completely eliminate the possibility of a performance impact.

49.64. pg_matviews

The view pg_matviews provides access to useful information about each materialized view in the database.

Table 49-65. pg_matviews Columns

Name	Type	References	Description
schemaname	name	pg_namespace.nspname	Name of schema containing materialized view
matviewname	name	pg_class.relname	Name of materialized view
matviewowner	name	pg_authid.rolname	Name of materialized view's owner
tablespace	name	pg_tablespace.spcname	Name of tablespace containing materialized view (null if default for database)
hasindexes	boolean		True if materialized view has (or recently had) any indexes
ispopulated	boolean		True if materialized view is currently populated
definition	text		Materialized view definition (a reconstructed SELECT query)

49.65. pg_policies

The view pg_policies provides access to useful information about each row-level security policy in the database.

Table 49-66. pg_policies Columns

Name	Type	References	Description

Name	Type	References	Description
schemaname	name	pg_namespace.nspname	Name of schema containing table policy is on
tablename	name	pg_class.relname	Name of table policy is on
policyname	name	pg_policy.polname	Name of policy
roles	name[]		The roles to which this policy applies
cmd	text		The command type to which the policy is applied
qual	text		The expression added to the security barrier qualifications for queries that this policy applies to
with_check	text		The expression added to the WITH CHECK qualifications for queries that attempt to add rows to this table

49.66. pg_prepared_statements

The pg_prepared_statements view displays all the prepared statements that are available in the current session. See PREPARE for more information about prepared statements.

pg_prepared_statements contains one row for each prepared statement. Rows are added to the view when a new prepared statement is created and removed when a prepared statement is released (for example, via the DEALLOCATE command).

Table 49-67. pg_prepared_statements Columns

Name	Type	Description
name	text	The identifier of the prepared statement

Name	Type	Description
statement	text	The query string submitted by the client to create this prepared statement. For prepared statements created via SQL, this is the PREPARE statement submitted by the client. For prepared statements created via the frontend/backend protocol, this is the text of the prepared statement itself.
prepare_time	timestamptz	The time at which the prepared statement was created
parameter_types	regtype[]	The expected parameter types for the prepared statement in the form of an array of regtype. The OID corresponding to an element of this array can be obtained by casting the regtype value to oid.
from_sql	boolean	true if the prepared statement was created via the PREPARE SQL statement; false if the statement was prepared via the frontend/backend protocol

The pg_prepared_statements view is read only.

49.67. pg_prepared_xacts

The view pg_prepared_xacts displays information about transactions that are currently prepared for two-phase commit (see PREPARE TRANSACTION for details).

pg_prepared_xacts contains one row per prepared transaction. An entry is removed when the transaction is committed or rolled back.

Table 49-68. pg_prepared_xacts Columns

Name	Type	References	Description
transaction	xid		Numeric transaction identifier of the prepared transaction

Name	Type	References	Description
gid	text		Global transaction identifier that was assigned to the transaction
prepared	timestamp with time zone		Time at which the transaction was prepared for commit
owner	name	pg_authid.rolname	Name of the user that executed the transaction
database	name	pg_database.datname	Name of the database in which the transaction was executed

When the pg_prepared_xacts view is accessed, the internal transaction manager data structures are momentarily locked, and a copy is made for the view to display. This ensures that the view produces a consistent set of results, while not blocking normal operations longer than necessary. Nonetheless there could be some impact on database performance if this view is frequently accessed.

49.68. pg_replication_origin_status

The pg_replication_origin_status view contains information about how far replay for a certain origin has progressed. For more on replication origins see Chapter 47.

Table 49-69. pg_replication_origin_status Columns

Name	Type	References	Description
local_id	Oid	pg_replication_origin.roident	internal node identifier
external_id	text	pg_replication_origin.roname	external node identifier
remote_lsn	pg_lsn		The origin node's LSN up to which data has been replicated.
local_lsn	pg_lsn		This node's LSN at which remote_lsn has been replicated. Used to flush commit records before persisting data to disk when using asynchronous commits.

49.69. pg_replication_slots

The pg_replication_slots view provides a listing of all replication slots that currently exist on the database cluster, along with their current state.

For more on replication slots, see Section 25.2.6 and Chapter 46.

Table 49-70. pg_replication_slots Columns

Name	Type	References	Description
slot_name	name		A unique, cluster-wide identifier for the replication slot
plugin	name		The base name of the shared object containing the output plugin this logical slot is using, or null for physical slots.
slot_type	text		The slot type - physical or logical
datoid	oid	pg_database.oid	The OID of the database this slot is associated with, or null. Only logical slots have an associated database.
database	text	pg_database.datname	The name of the database this slot is associated with, or null. Only logical slots have an associated database.
active	boolean		True if this slot is currently actively being used
active_pid	integer		The process ID of the session using this slot if the slot is currently actively being used. NULL if inactive.
xmin	xid		The oldest transaction that this slot needs the database to retain. VACUUM cannot remove tuples deleted by any later transaction.

Name	Type	References	Description
catalog_xmin	xid		The oldest transaction affecting the system catalogs that this slot needs the database to retain. VACUUM cannot remove catalog tuples deleted by any later transaction.
restart_lsn	pg_lsn		The address (LSN) of oldest WAL which still might be required by the consumer of this slot and thus won't be automatically removed during checkpoints.

49.70. pg_roles

The view pg_roles provides access to information about database roles. This is simply a publicly readable view of pg_authid that blanks out the password field.

This view explicitly exposes the OID column of the underlying table, since that is needed to do joins to other catalogs.

Table 49-71. pg_roles Columns

Name	Type	References	Description
rolname	name		Role name
rolsuper	bool		Role has superuser privileges
rolinherit	bool		Role automatically inherits privileges of roles it is a member of
rolcreaterole	bool		Role can create more roles
rolcreatedb	bool		Role can create databases
rolcanlogin	bool		Role can log in. That is, this role can be given as the initial session authorization identifier

Name	Type	References	Description
rolreplication	bool		Role is a replication role. That is, this role can initiate streaming replication (see Section 25.2.5) and set/unset the system backup mode using pg_start_backup and pg_stop_backup
rolconnlimit	int4		For roles that can log in, this sets maximum number of concurrent connections this role can make. -1 means no limit.
rolpassword	text		Not the password (always reads as ********)
rolvaliduntil	timestamptz		Password expiry time (only used for password authentication); null if no expiration
rolbypassrls	bool		Role bypasses every row level security policy, see Section 5.7 for more information.
rolconfig	text[]		Role-specific defaults for run-time configuration variables
oid	oid	pg_authid.oid	ID of role

49.71. pg_rules

The view pg_rules provides access to useful information about query rewrite rules.

Table 49-72. pg_rules Columns

Name	Type	References	Description
schemaname	name	pg_namespace.nspname	Name of schema containing table
tablename	name	pg_class.relname	Name of table the rule is for

Name	Type	References	Description
rulename	name	pg_rewrite.rulename	Name of rule
definition	text		Rule definition (a reconstructed creation command)

The pg_rules view excludes the ON SELECT rules of views and materialized views; those can be seen in pg_views and pg_matviews.

49.72. pg_seclabels

The view pg_seclabels provides information about security labels. It as an easier-to-query version of the pg_seclabel catalog.

Table 49-73. pg_seclabels Columns

Name	Type	References	Description
objoid	oid	any OID column	The OID of the object this security label pertains to
classoid	oid	pg_class.oid	The OID of the system catalog this object appears in
objsubid	int4		For a security label on a table column, this is the column number (the objoid and classoid refer to the table itself). For all other object types, this column is zero.
objtype	text		The type of object to which this label applies, as text.
objnamespace	oid	pg_namespace.oid	The OID of the namespace for this object, if applicable; otherwise NULL.
objname	text		The name of the object to which this label applies, as text.

Name	Type	References	Description
provider	text	pg_seclabel.provider	The label provider associated with this label.
label	text	pg_seclabel.label	The security label applied to this object.

49.73. pg_settings

The view pg_settings provides access to run-time parameters of the server. It is essentially an alternative interface to the SHOW and SET commands. It also provides access to some facts about each parameter that are not directly available from SHOW, such as minimum and maximum values.

Table 49-74. pg_settings Columns

Name	Type	Description
name	text	Run-time configuration parameter name
setting	text	Current value of the parameter
unit	text	Implicit unit of the parameter
category	text	Logical group of the parameter
short_desc	text	A brief description of the parameter
extra_desc	text	Additional, more detailed, description of the parameter
context	text	Context required to set the parameter's value (see below)
vartype	text	Parameter type (bool, enum, integer, real, or string)
source	text	Source of the current parameter value
min_val	text	Minimum allowed value of the parameter (null for non-numeric values)
max_val	text	Maximum allowed value of the parameter (null for non-numeric values)
enumvals	text[]	Allowed values of an enum parameter (null for non-enum values)

Name	Type	Description
boot_val	text	Parameter value assumed at server startup if the parameter is not otherwise set
reset_val	text	Value that RESET would reset the parameter to in the current session
sourcefile	text	Configuration file the current value was set in (null for values set from sources other than configuration files, or when examined by a non-superuser); helpful when using include directives in configuration files
sourceline	integer	Line number within the configuration file the current value was set at (null for values set from sources other than configuration files, or when examined by a non-superuser)
pending_restart	boolean	true if the value has been changed in the configuration file but needs a restart; or false otherwise.

There are several possible values of context. In order of decreasing difficulty of changing the setting, they are:

internal

These settings cannot be changed directly; they reflect internally determined values. Some of them may be adjustable by rebuilding the server with different configuration options, or by changing options supplied to initdb.

postmaster

These settings can only be applied when the server starts, so any change requires restarting the server. Values for these settings are typically stored in the postgresql.conf file, or passed on the command line when starting the server. Of course, settings with any of the lower context types can also be set at server start time.

sighup

Changes to these settings can be made in postgresql.conf without restarting the server. Send a SIGHUP signal to the postmaster to cause it to re-read postgresql.conf and apply the changes. The postmaster will also forward the SIGHUP signal to its child processes so that they all pick up the new value.

superuser-backend

Changes to these settings can be made in postgresql.conf without restarting the server. They

can also be set for a particular session in the connection request packet (for example, via libpq's PGOPTIONS environment variable), but only if the connecting user is a superuser. However, these settings never change in a session after it is started. If you change them in postgresql.conf, send a SIGHUP signal to the postmaster to cause it to re-read postgresql.conf. The new values will only affect subsequently-launched sessions.

backend

Changes to these settings can be made in postgresql.conf without restarting the server. They can also be set for a particular session in the connection request packet (for example, via libpq's PGOPTIONS environment variable); any user can make such a change for his session. However, these settings never change in a session after it is started. If you change them in postgresql.conf, send a SIGHUP signal to the postmaster to cause it to re-read postgresql.conf. The new values will only affect subsequently-launched sessions.

superuser

These settings can be set from postgresql.conf, or within a session via the SET command; but only superusers can change them via SET. Changes in postgresql.conf will affect existing sessions only if no session-local value has been established with SET.

user

These settings can be set from postgresql.conf, or within a session via the SET command. Any user is allowed to change his session-local value. Changes in postgresql.conf will affect existing sessions only if no session-local value has been established with SET.

See Section 18.1 for more information about the various ways to change these parameters.

The pg_settings view cannot be inserted into or deleted from, but it can be updated. An UPDATE applied to a row of pg_settings is equivalent to executing the SET command on that named parameter. The change only affects the value used by the current session. If an UPDATE is issued within a transaction that is later aborted, the effects of the UPDATE command disappear when the transaction is rolled back. Once the surrounding transaction is committed, the effects will persist until the end of the session, unless overridden by another UPDATE or SET.

49.74. pg_shadow

The view pg_shadow exists for backwards compatibility: it emulates a catalog that existed in PostgreSQL before version 8.1. It shows properties of all roles that are marked as rolcanlogin in pg_authid.

The name stems from the fact that this table should not be readable by the public since it contains passwords. pg_user is a publicly readable view on pg_shadow that blanks out the password field.

Table 49-75. pg_shadow Columns

Name	Type	References	Description
usename	name	pg_authid.rolname	User name
usesysid	oid	pg_authid.oid	ID of this user
usecreatedb	bool		User can create databases

Name	Type	References	Description
usesuper	bool		User is a superuser
userepl	bool		User can initiate streaming replication and put the system in and out of backup mode.
usebypassrls	bool		User bypasses every row level security policy, see Section 5.7 for more information.
passwd	text		Password (possibly encrypted); null if none. See pg_authid for details of how encrypted passwords are stored.
valuntil	abstime		Password expiry time (only used for password authentication)
useconfig	text[]		Session defaults for run-time configuration variables

49.75. pg_stats

The view pg_stats provides access to the information stored in the pg_statistic catalog. This view allows access only to rows of pg_statistic that correspond to tables the user has permission to read, and therefore it is safe to allow public read access to this view.

pg_stats is also designed to present the information in a more readable format than the underlying catalog — at the cost that its schema must be extended whenever new slot types are defined for pg_statistic.

Table 49-76. pg_stats Columns

Name	Type	References	Description
schemaname	name	pg_namespace.nspname	Name of schema containing table
tablename	name	pg_class.relname	Name of table
attname	name	pg_attribute.attname	Name of the column described by this row

Name	Type	References	Description
inherited	bool		If true, this row includes inheritance child columns, not just the values in the specified table
null_frac	real		Fraction of column entries that are null
avg_width	integer		Average width in bytes of column's entries
n_distinct	real		If greater than zero, the estimated number of distinct values in the column. If less than zero, the negative of the number of distinct values divided by the number of rows. (The negated form is used when ANALYZE believes that the number of distinct values is likely to increase as the table grows; the positive form is used when the column seems to have a fixed number of possible values.) For example, -1 indicates a unique column in which the number of distinct values is the same as the number of rows.
most_common_vals	anyarray		A list of the most common values in the column. (Null if no values seem to be more common than any others.)

Name	Type	References	Description
most_common_freqs	real[]		A list of the frequencies of the most common values, i.e., number of occurrences of each divided by total number of rows. (Null when most_common_vals is.)
histogram_bounds	anyarray		A list of values that divide the column's values into groups of approximately equal population. The values in most_common_vals, if present, are omitted from this histogram calculation. (This column is null if the column data type does not have a < operator or if the most_common_vals list accounts for the entire population.)
correlation	real		Statistical correlation between physical row ordering and logical ordering of the column values. This ranges from -1 to +1. When the value is near -1 or +1, an index scan on the column will be estimated to be cheaper than when it is near zero, due to reduction of random access to the disk. (This column is null if the column data type does not have a < operator.)
most_common_elems	anyarray		A list of non-null element values most often appearing within values of the column. (Null for scalar types.)

Name	Type	References	Description
most_common_elem_freqs	real[]		A list of the frequencies of the most common element values, i.e., the fraction of rows containing at least one instance of the given value. Two or three additional values follow the per-element frequencies; these are the minimum and maximum of the preceding per-element frequencies, and optionally the frequency of null elements. (Null when most_common_elems is.)
elem_count_histogram	real[]		A histogram of the counts of distinct non-null element values within the values of the column, followed by the average number of distinct non-null elements. (Null for scalar types.)

The maximum number of entries in the array fields can be controlled on a column-by-column basis using the ALTER TABLE SET STATISTICS command, or globally by setting the default_statistics_target run-time parameter.

49.76. pg_tables

The view pg_tables provides access to useful information about each table in the database.

Table 49-77. pg_tables Columns

Name	Type	References	Description
schemaname	name	pg_namespace.nspname	Name of schema containing table
tablename	name	pg_class.relname	Name of table
tableowner	name	pg_authid.rolname	Name of table's owner

Name	Type	References	Description
tablespace	name	pg_tablespace.spcname	Name of tablespace containing table (null if default for database)
hasindexes	boolean	pg_class.relhasindex	True if table has (or recently had) any indexes
hasrules	boolean	pg_class.relhasrules	True if table has (or once had) rules
hastriggers	boolean	pg_class.relhastriggers	True if table has (or once had) triggers
rowsecurity	boolean	pg_class.relrowsecurity	True if row security is enabled on the table

49.77. pg_timezone_abbrevs

The view pg_timezone_abbrevs provides a list of time zone abbreviations that are currently recognized by the datetime input routines. The contents of this view change when the timezone_abbreviations run-time parameter is modified.

Table 49-78. pg_timezone_abbrevs Columns

Name	Type	Description
abbrev	text	Time zone abbreviation
utc_offset	interval	Offset from UTC (positive means east of Greenwich)
is_dst	boolean	True if this is a daylight-savings abbreviation

While most timezone abbreviations represent fixed offsets from UTC, there are some that have historically varied in value (see Section B.4 for more information). In such cases this view presents their current meaning.

49.78. pg_timezone_names

The view pg_timezone_names provides a list of time zone names that are recognized by SET TIMEZONE, along with their associated abbreviations, UTC offsets, and daylight-savings status. (Technically, PostgreSQL uses UT1 rather than UTC because leap seconds are not handled.) Unlike the abbreviations shown in pg_timezone_abbrevs, many of these names imply a set of daylight-savings transition date rules. Therefore, the associated information changes across local DST boundaries. The displayed information is computed based on the current value of CURRENT_TIMESTAMP.

Table 49-79. `pg_timezone_names` Columns

Name	Type	Description
`name`	`text`	Time zone name
`abbrev`	`text`	Time zone abbreviation
`utc_offset`	`interval`	Offset from UTC (positive means east of Greenwich)
`is_dst`	`boolean`	True if currently observing daylight savings

49.79. `pg_user`

The view `pg_user` provides access to information about database users. This is simply a publicly readable view of `pg_shadow` that blanks out the password field.

Table 49-80. `pg_user` Columns

Name	Type	Description
`usename`	`name`	User name
`usesysid`	`oid`	ID of this user
`usecreatedb`	`bool`	User can create databases
`usesuper`	`bool`	User is a superuser
`userepl`	`bool`	User can initiate streaming replication and put the system in and out of backup mode.
`usebypassrls`	`bool`	User bypasses every row level security policy, see Section 5.7 for more information.
`passwd`	`text`	Not the password (always reads as `********`)
`valuntil`	`abstime`	Password expiry time (only used for password authentication)
`useconfig`	`text[]`	Session defaults for run-time configuration variables

49.80. `pg_user_mappings`

The view `pg_user_mappings` provides access to information about user mappings. This is essentially a publicly readable view of `pg_user_mapping` that leaves out the options field if the user has no rights to use it.

Table 49-81. `pg_user_mappings` Columns

Name	Type	References	Description
umid	oid	pg_user_mapping.oid	OID of the user mapping
srvid	oid	pg_foreign_server.oid	The OID of the foreign server that contains this mapping
srvname	name	pg_foreign_server.srvname	Name of the foreign server
umuser	oid	pg_authid.oid	OID of the local role being mapped, 0 if the user mapping is public
usename	name		Name of the local user to be mapped
umoptions	text[]		User mapping specific options, as "keyword=value" strings

To protect password information stored as a user mapping option, the `umoptions` column will read as null unless one of the following applies:

- current user is the user being mapped, and owns the server or holds `USAGE` privilege on it
- current user is the server owner and mapping is for `PUBLIC`
- current user is a superuser

49.81. `pg_views`

The view `pg_views` provides access to useful information about each view in the database.

Table 49-82. `pg_views` Columns

Name	Type	References	Description
schemaname	name	pg_namespace.nspname	Name of schema containing view
viewname	name	pg_class.relname	Name of view
viewowner	name	pg_authid.rolname	Name of view's owner
definition	text		View definition (a reconstructed `SELECT` query)

Chapter 50. Frontend/Backend Protocol

PostgreSQL uses a message-based protocol for communication between frontends and backends (clients and servers). The protocol is supported over TCP/IP and also over Unix-domain sockets. Port number 5432 has been registered with IANA as the customary TCP port number for servers supporting this protocol, but in practice any non-privileged port number can be used.

This document describes version 3.0 of the protocol, implemented in PostgreSQL 7.4 and later. For descriptions of the earlier protocol versions, see previous releases of the PostgreSQL documentation. A single server can support multiple protocol versions. The initial startup-request message tells the server which protocol version the client is attempting to use. If the major version requested by the client is not supported by the server, the connection will be rejected (for example, this would occur if the client requested protocol version 4.0, which does not exist as of this writing). If the minor version requested by the client is not supported by the server (e.g., the client requests version 3.1, but the server supports only 3.0), the server may either reject the connection or may respond with a NegotiateProtocolVersion message containing the highest minor protocol version which it supports. The client may then choose either to continue with the connection using the specified protocol version or to abort the connection.

In order to serve multiple clients efficiently, the server launches a new "backend" process for each client. In the current implementation, a new child process is created immediately after an incoming connection is detected. This is transparent to the protocol, however. For purposes of the protocol, the terms "backend" and "server" are interchangeable; likewise "frontend" and "client" are interchangeable.

50.1. Overview

The protocol has separate phases for startup and normal operation. In the startup phase, the frontend opens a connection to the server and authenticates itself to the satisfaction of the server. (This might involve a single message, or multiple messages depending on the authentication method being used.) If all goes well, the server then sends status information to the frontend, and finally enters normal operation. Except for the initial startup-request message, this part of the protocol is driven by the server.

During normal operation, the frontend sends queries and other commands to the backend, and the backend sends back query results and other responses. There are a few cases (such as `NOTIFY`) wherein the backend will send unsolicited messages, but for the most part this portion of a session is driven by frontend requests.

Termination of the session is normally by frontend choice, but can be forced by the backend in certain cases. In any case, when the backend closes the connection, it will roll back any open (incomplete) transaction before exiting.

Within normal operation, SQL commands can be executed through either of two sub-protocols. In the "simple query" protocol, the frontend just sends a textual query string, which is parsed and immediately executed by the backend. In the "extended query" protocol, processing of queries is separated into multiple steps: parsing, binding of parameter values, and execution. This offers flexibility and performance benefits, at the cost of extra complexity.

Normal operation has additional sub-protocols for special operations such as `COPY`.

50.1.1. Messaging Overview

All communication is through a stream of messages. The first byte of a message identifies the message type, and the next four bytes specify the length of the rest of the message (this length count includes itself, but not the message-type byte). The remaining contents of the message are determined by the message type. For historical reasons, the very first message sent by the client (the startup message) has no initial message-type byte.

To avoid losing synchronization with the message stream, both servers and clients typically read an entire message into a buffer (using the byte count) before attempting to process its contents. This allows easy recovery if an error is detected while processing the contents. In extreme situations (such as not having enough memory to buffer the message), the receiver can use the byte count to determine how much input to skip before it resumes reading messages.

Conversely, both servers and clients must take care never to send an incomplete message. This is commonly done by marshaling the entire message in a buffer before beginning to send it. If a communications failure occurs partway through sending or receiving a message, the only sensible response is to abandon the connection, since there is little hope of recovering message-boundary synchronization.

50.1.2. Extended Query Overview

In the extended-query protocol, execution of SQL commands is divided into multiple steps. The state retained between steps is represented by two types of objects: *prepared statements* and *portals*. A prepared statement represents the result of parsing and semantic analysis of a textual query string. A prepared statement is not in itself ready to execute, because it might lack specific values for *parameters*. A portal represents a ready-to-execute or already-partially-executed statement, with any missing parameter values filled in. (For `SELECT` statements, a portal is equivalent to an open cursor, but we choose to use a different term since cursors don't handle non-`SELECT` statements.)

The overall execution cycle consists of a *parse* step, which creates a prepared statement from a textual query string; a *bind* step, which creates a portal given a prepared statement and values for any needed parameters; and an *execute* step that runs a portal's query. In the case of a query that returns rows (`SELECT`, `SHOW`, etc), the execute step can be told to fetch only a limited number of rows, so that multiple execute steps might be needed to complete the operation.

The backend can keep track of multiple prepared statements and portals (but note that these exist only within a session, and are never shared across sessions). Existing prepared statements and portals are referenced by names assigned when they were created. In addition, an "unnamed" prepared statement and portal exist. Although these behave largely the same as named objects, operations on them are optimized for the case of executing a query only once and then discarding it, whereas operations on named objects are optimized on the expectation of multiple uses.

50.1.3. Formats and Format Codes

Data of a particular data type might be transmitted in any of several different *formats*. As of PostgreSQL 7.4 the only supported formats are "text" and "binary", but the protocol makes provision for future extensions. The desired format for any value is specified by a *format code*. Clients can specify a format code for each transmitted parameter value and for each column of a query result. Text has format code zero, binary has format code one, and all other format codes are reserved for future definition.

The text representation of values is whatever strings are produced and accepted by the input/output conversion functions for the particular data type. In the transmitted representation, there is no trailing null character; the frontend must add one to received values if it wants to process them as C strings. (The text format does not allow embedded nulls, by the way.)

Binary representations for integers use network byte order (most significant byte first). For other data types consult the documentation or source code to learn about the binary representation. Keep in mind that binary representations for complex data types might change across server versions; the text format is usually the more portable choice.

50.2. Message Flow

This section describes the message flow and the semantics of each message type. (Details of the exact representation of each message appear in Section 50.5.) There are several different sub-protocols depending on the state of the connection: start-up, query, function call, `COPY`, and termination. There are also special provisions for asynchronous operations (including notification responses and command cancellation), which can occur at any time after the start-up phase.

50.2.1. Start-up

To begin a session, a frontend opens a connection to the server and sends a startup message. This message includes the names of the user and of the database the user wants to connect to; it also identifies the particular protocol version to be used. (Optionally, the startup message can include additional settings for run-time parameters.) The server then uses this information and the contents of its configuration files (such as `pg_hba.conf`) to determine whether the connection is provisionally acceptable, and what additional authentication is required (if any).

The server then sends an appropriate authentication request message, to which the frontend must reply with an appropriate authentication response message (such as a password). For all authentication methods except GSSAPI and SSPI, there is at most one request and one response. In some methods, no response at all is needed from the frontend, and so no authentication request occurs. For GSSAPI and SSPI, multiple exchanges of packets may be needed to complete the authentication.

The authentication cycle ends with the server either rejecting the connection attempt (ErrorResponse), or sending AuthenticationOk.

The possible messages from the server in this phase are:

ErrorResponse

The connection attempt has been rejected. The server then immediately closes the connection.

AuthenticationOk

The authentication exchange is successfully completed.

AuthenticationKerberosV5

The frontend must now take part in a Kerberos V5 authentication dialog (not described here, part of the Kerberos specification) with the server. If this is successful, the server responds with an AuthenticationOk, otherwise it responds with an ErrorResponse. This is no longer supported.

AuthenticationCleartextPassword

The frontend must now send a PasswordMessage containing the password in clear-text form. If this is the correct password, the server responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationMD5Password

The frontend must now send a PasswordMessage containing the password (with user name) encrypted via MD5, then encrypted again using the 4-byte random salt specified in the AuthenticationMD5Password message. If this is the correct password, the server responds with an AuthenticationOk, otherwise it responds with an ErrorResponse. The actual PasswordMessage can be computed in SQL as `concat('md5', md5(concat(md5(concat(password, username)), random-salt)))`. (Keep in mind the `md5()` function returns its result as a hex string.)

AuthenticationSCMCredential

This response is only possible for local Unix-domain connections on platforms that support SCM credential messages. The frontend must issue an SCM credential message and then send a single data byte. (The contents of the data byte are uninteresting; it's only used to ensure that the server waits long enough to receive the credential message.) If the credential is acceptable, the server responds with an AuthenticationOk, otherwise it responds with an ErrorResponse. (This message type is only issued by pre-9.1 servers. It may eventually be removed from the protocol specification.)

AuthenticationGSS

The frontend must now initiate a GSSAPI negotiation. The frontend will send a PasswordMessage with the first part of the GSSAPI data stream in response to this. If further messages are needed, the server will respond with AuthenticationGSSContinue.

AuthenticationSSPI

The frontend must now initiate a SSPI negotiation. The frontend will send a PasswordMessage with the first part of the SSPI data stream in response to this. If further messages are needed, the server will respond with AuthenticationGSSContinue.

AuthenticationGSSContinue

This message contains the response data from the previous step of GSSAPI or SSPI negotiation (AuthenticationGSS, AuthenticationSSPI or a previous AuthenticationGSSContinue). If the GSSAPI or SSPI data in this message indicates more data is needed to complete the authentication, the frontend must send that data as another PasswordMessage. If GSSAPI or SSPI authentication is completed by this message, the server will next send AuthenticationOk to indicate successful authentication or ErrorResponse to indicate failure.

NegotiateProtocolVersion

The server does not support the minor protocol version requested by the client, but does support an earlier version of the protocol; this message indicates the highest supported minor version. This message will also be sent if the client requested unsupported protocol options (i.e., beginning with `_pq_`.) in the startup packet. This message will be followed by an ErrorResponse or a message indicating the success or failure of authentication.

If the frontend does not support the authentication method requested by the server, then it should immediately close the connection.

After having received AuthenticationOk, the frontend must wait for further messages from the server. In this phase a backend process is being started, and the frontend is just an interested bystander. It is still possible for the startup attempt to fail (ErrorResponse) or the server to decline support for the requested minor protocol version (NegotiateProtocolVersion), but in the normal case the backend will send some ParameterStatus messages, BackendKeyData, and finally ReadyForQuery.

During this phase the backend will attempt to apply any additional run-time parameter settings that were given in the startup message. If successful, these values become session defaults. An error causes ErrorResponse and exit.

The possible messages from the backend in this phase are:

BackendKeyData

This message provides secret-key data that the frontend must save if it wants to be able to issue cancel requests later. The frontend should not respond to this message, but should continue listening for a ReadyForQuery message.

ParameterStatus

This message informs the frontend about the current (initial) setting of backend parameters, such as client_encoding or DateStyle. The frontend can ignore this message, or record the settings for its future use; see Section 50.2.6 for more details. The frontend should not respond to this message, but should continue listening for a ReadyForQuery message.

ReadyForQuery

Start-up is completed. The frontend can now issue commands.

ErrorResponse

Start-up failed. The connection is closed after sending this message.

NoticeResponse

A warning message has been issued. The frontend should display the message but continue listening for ReadyForQuery or ErrorResponse.

The ReadyForQuery message is the same one that the backend will issue after each command cycle. Depending on the coding needs of the frontend, it is reasonable to consider ReadyForQuery as starting a command cycle, or to consider ReadyForQuery as ending the start-up phase and each subsequent command cycle.

50.2.2. Simple Query

A simple query cycle is initiated by the frontend sending a Query message to the backend. The message includes an SQL command (or commands) expressed as a text string. The backend then sends one or more response messages depending on the contents of the query command string, and finally a ReadyForQuery response message. ReadyForQuery informs the frontend that it can safely send a new command. (It is not actually necessary for the frontend to wait for ReadyForQuery before issuing another command, but

the frontend must then take responsibility for figuring out what happens if the earlier command fails and already-issued later commands succeed.)

The possible response messages from the backend are:

CommandComplete

An SQL command completed normally.

CopyInResponse

The backend is ready to copy data from the frontend to a table; see Section 50.2.5.

CopyOutResponse

The backend is ready to copy data from a table to the frontend; see Section 50.2.5.

RowDescription

Indicates that rows are about to be returned in response to a `SELECT`, `FETCH`, etc query. The contents of this message describe the column layout of the rows. This will be followed by a DataRow message for each row being returned to the frontend.

DataRow

One of the set of rows returned by a `SELECT`, `FETCH`, etc query.

EmptyQueryResponse

An empty query string was recognized.

ErrorResponse

An error has occurred.

ReadyForQuery

Processing of the query string is complete. A separate message is sent to indicate this because the query string might contain multiple SQL commands. (CommandComplete marks the end of processing one SQL command, not the whole string.) ReadyForQuery will always be sent, whether processing terminates successfully or with an error.

NoticeResponse

A warning message has been issued in relation to the query. Notices are in addition to other responses, i.e., the backend will continue processing the command.

The response to a `SELECT` query (or other queries that return row sets, such as `EXPLAIN` or `SHOW`) normally consists of RowDescription, zero or more DataRow messages, and then CommandComplete. `COPY` to or from the frontend invokes special protocol as described in Section 50.2.5. All other query types normally produce only a CommandComplete message.

Since a query string could contain several queries (separated by semicolons), there might be several such response sequences before the backend finishes processing the query string. ReadyForQuery is issued when the entire string has been processed and the backend is ready to accept a new query string.

If a completely empty (no contents other than whitespace) query string is received, the response is EmptyQueryResponse followed by ReadyForQuery.

In the event of an error, ErrorResponse is issued followed by ReadyForQuery. All further processing of the query string is aborted by ErrorResponse (even if more queries remained in it). Note that this might occur partway through the sequence of messages generated by an individual query.

In simple Query mode, the format of retrieved values is always text, except when the given command is a `FETCH` from a cursor declared with the `BINARY` option. In that case, the retrieved values are in binary format. The format codes given in the RowDescription message tell which format is being used.

A frontend must be prepared to accept ErrorResponse and NoticeResponse messages whenever it is expecting any other type of message. See also Section 50.2.6 concerning messages that the backend might generate due to outside events.

Recommended practice is to code frontends in a state-machine style that will accept any message type at any time that it could make sense, rather than wiring in assumptions about the exact sequence of messages.

50.2.3. Extended Query

The extended query protocol breaks down the above-described simple query protocol into multiple steps. The results of preparatory steps can be re-used multiple times for improved efficiency. Furthermore, additional features are available, such as the possibility of supplying data values as separate parameters instead of having to insert them directly into a query string.

In the extended protocol, the frontend first sends a Parse message, which contains a textual query string, optionally some information about data types of parameter placeholders, and the name of a destination prepared-statement object (an empty string selects the unnamed prepared statement). The response is either ParseComplete or ErrorResponse. Parameter data types can be specified by OID; if not given, the parser attempts to infer the data types in the same way as it would do for untyped literal string constants.

> **Note:** A parameter data type can be left unspecified by setting it to zero, or by making the array of parameter type OIDs shorter than the number of parameter symbols (`$n`) used in the query string. Another special case is that a parameter's type can be specified as `void` (that is, the OID of the `void` pseudotype). This is meant to allow parameter symbols to be used for function parameters that are actually OUT parameters. Ordinarily there is no context in which a `void` parameter could be used, but if such a parameter symbol appears in a function's parameter list, it is effectively ignored. For example, a function call such as `foo($1,$2,$3,$4)` could match a function with two IN and two OUT arguments, if `$3` and `$4` are specified as having type `void`.

> **Note:** The query string contained in a Parse message cannot include more than one SQL statement; else a syntax error is reported. This restriction does not exist in the simple-query protocol, but it does exist in the extended protocol, because allowing prepared statements or portals to contain multiple commands would complicate the protocol unduly.

If successfully created, a named prepared-statement object lasts till the end of the current session, unless explicitly destroyed. An unnamed prepared statement lasts only until the next Parse statement specifying the unnamed statement as destination is issued. (Note that a simple Query message also destroys the unnamed statement.) Named prepared statements must be explicitly closed before they can be redefined

by another Parse message, but this is not required for the unnamed statement. Named prepared statements can also be created and accessed at the SQL command level, using `PREPARE` and `EXECUTE`.

Once a prepared statement exists, it can be readied for execution using a Bind message. The Bind message gives the name of the source prepared statement (empty string denotes the unnamed prepared statement), the name of the destination portal (empty string denotes the unnamed portal), and the values to use for any parameter placeholders present in the prepared statement. The supplied parameter set must match those needed by the prepared statement. (If you declared any `void` parameters in the Parse message, pass NULL values for them in the Bind message.) Bind also specifies the format to use for any data returned by the query; the format can be specified overall, or per-column. The response is either BindComplete or ErrorResponse.

> **Note:** The choice between text and binary output is determined by the format codes given in Bind, regardless of the SQL command involved. The `BINARY` attribute in cursor declarations is irrelevant when using extended query protocol.

Query planning typically occurs when the Bind message is processed. If the prepared statement has no parameters, or is executed repeatedly, the server might save the created plan and re-use it during subsequent Bind messages for the same prepared statement. However, it will do so only if it finds that a generic plan can be created that is not much less efficient than a plan that depends on the specific parameter values supplied. This happens transparently so far as the protocol is concerned.

If successfully created, a named portal object lasts till the end of the current transaction, unless explicitly destroyed. An unnamed portal is destroyed at the end of the transaction, or as soon as the next Bind statement specifying the unnamed portal as destination is issued. (Note that a simple Query message also destroys the unnamed portal.) Named portals must be explicitly closed before they can be redefined by another Bind message, but this is not required for the unnamed portal. Named portals can also be created and accessed at the SQL command level, using `DECLARE CURSOR` and `FETCH`.

Once a portal exists, it can be executed using an Execute message. The Execute message specifies the portal name (empty string denotes the unnamed portal) and a maximum result-row count (zero meaning "fetch all rows"). The result-row count is only meaningful for portals containing commands that return row sets; in other cases the command is always executed to completion, and the row count is ignored. The possible responses to Execute are the same as those described above for queries issued via simple query protocol, except that Execute doesn't cause ReadyForQuery or RowDescription to be issued.

If Execute terminates before completing the execution of a portal (due to reaching a nonzero result-row count), it will send a PortalSuspended message; the appearance of this message tells the frontend that another Execute should be issued against the same portal to complete the operation. The CommandComplete message indicating completion of the source SQL command is not sent until the portal's execution is completed. Therefore, an Execute phase is always terminated by the appearance of exactly one of these messages: CommandComplete, EmptyQueryResponse (if the portal was created from an empty query string), ErrorResponse, or PortalSuspended.

At completion of each series of extended-query messages, the frontend should issue a Sync message. This parameterless message causes the backend to close the current transaction if it's not inside a `BEGIN`/`COMMIT` transaction block ("close" meaning to commit if no error, or roll back if error). Then a ReadyForQuery response is issued. The purpose of Sync is to provide a resynchronization point for error recovery. When an error is detected while processing any extended-query message, the backend issues ErrorResponse, then reads and discards messages until a Sync is reached, then issues ReadyForQuery

and returns to normal message processing. (But note that no skipping occurs if an error is detected *while* processing Sync — this ensures that there is one and only one ReadyForQuery sent for each Sync.)

> **Note:** Sync does not cause a transaction block opened with `BEGIN` to be closed. It is possible to detect this situation since the ReadyForQuery message includes transaction status information.

In addition to these fundamental, required operations, there are several optional operations that can be used with extended-query protocol.

The Describe message (portal variant) specifies the name of an existing portal (or an empty string for the unnamed portal). The response is a RowDescription message describing the rows that will be returned by executing the portal; or a NoData message if the portal does not contain a query that will return rows; or ErrorResponse if there is no such portal.

The Describe message (statement variant) specifies the name of an existing prepared statement (or an empty string for the unnamed prepared statement). The response is a ParameterDescription message describing the parameters needed by the statement, followed by a RowDescription message describing the rows that will be returned when the statement is eventually executed (or a NoData message if the statement will not return rows). ErrorResponse is issued if there is no such prepared statement. Note that since Bind has not yet been issued, the formats to be used for returned columns are not yet known to the backend; the format code fields in the RowDescription message will be zeroes in this case.

> **Tip:** In most scenarios the frontend should issue one or the other variant of Describe before issuing Execute, to ensure that it knows how to interpret the results it will get back.

The Close message closes an existing prepared statement or portal and releases resources. It is not an error to issue Close against a nonexistent statement or portal name. The response is normally CloseComplete, but could be ErrorResponse if some difficulty is encountered while releasing resources. Note that closing a prepared statement implicitly closes any open portals that were constructed from that statement.

The Flush message does not cause any specific output to be generated, but forces the backend to deliver any data pending in its output buffers. A Flush must be sent after any extended-query command except Sync, if the frontend wishes to examine the results of that command before issuing more commands. Without Flush, messages returned by the backend will be combined into the minimum possible number of packets to minimize network overhead.

> **Note:** The simple Query message is approximately equivalent to the series Parse, Bind, portal Describe, Execute, Close, Sync, using the unnamed prepared statement and portal objects and no parameters. One difference is that it will accept multiple SQL statements in the query string, automatically performing the bind/describe/execute sequence for each one in succession. Another difference is that it will not return ParseComplete, BindComplete, CloseComplete, or NoData messages.

50.2.4. Function Call

The Function Call sub-protocol allows the client to request a direct call of any function that exists in the database's `pg_proc` system catalog. The client must have execute permission for the function.

> **Note:** The Function Call sub-protocol is a legacy feature that is probably best avoided in new code. Similar results can be accomplished by setting up a prepared statement that does `SELECT function($1, ...)`. The Function Call cycle can then be replaced with Bind/Execute.

A Function Call cycle is initiated by the frontend sending a FunctionCall message to the backend. The backend then sends one or more response messages depending on the results of the function call, and finally a ReadyForQuery response message. ReadyForQuery informs the frontend that it can safely send a new query or function call.

The possible response messages from the backend are:

ErrorResponse

An error has occurred.

FunctionCallResponse

The function call was completed and returned the result given in the message. (Note that the Function Call protocol can only handle a single scalar result, not a row type or set of results.)

ReadyForQuery

Processing of the function call is complete. ReadyForQuery will always be sent, whether processing terminates successfully or with an error.

NoticeResponse

A warning message has been issued in relation to the function call. Notices are in addition to other responses, i.e., the backend will continue processing the command.

50.2.5. COPY Operations

The `COPY` command allows high-speed bulk data transfer to or from the server. Copy-in and copy-out operations each switch the connection into a distinct sub-protocol, which lasts until the operation is completed.

Copy-in mode (data transfer to the server) is initiated when the backend executes a `COPY FROM STDIN` SQL statement. The backend sends a CopyInResponse message to the frontend. The frontend should then send zero or more CopyData messages, forming a stream of input data. (The message boundaries are not required to have anything to do with row boundaries, although that is often a reasonable choice.) The frontend can terminate the copy-in mode by sending either a CopyDone message (allowing successful termination) or a CopyFail message (which will cause the `COPY` SQL statement to fail with an error). The backend then reverts to the command-processing mode it was in before the `COPY` started, which will be either simple or extended query protocol. It will next send either CommandComplete (if successful) or ErrorResponse (if not).

In the event of a backend-detected error during copy-in mode (including receipt of a CopyFail message), the backend will issue an ErrorResponse message. If the `COPY` command was issued via an extended-query message, the backend will now discard frontend messages until a Sync message is received, then it will issue ReadyForQuery and return to normal processing. If the `COPY` command was issued in a simple Query message, the rest of that message is discarded and ReadyForQuery is issued. In either case, any subsequent CopyData, CopyDone, or CopyFail messages issued by the frontend will simply be dropped.

The backend will ignore Flush and Sync messages received during copy-in mode. Receipt of any other non-copy message type constitutes an error that will abort the copy-in state as described above. (The exception for Flush and Sync is for the convenience of client libraries that always send Flush or Sync after an Execute message, without checking whether the command to be executed is a `COPY FROM STDIN`.)

Copy-out mode (data transfer from the server) is initiated when the backend executes a `COPY TO STDOUT` SQL statement. The backend sends a CopyOutResponse message to the frontend, followed by zero or more CopyData messages (always one per row), followed by CopyDone. The backend then reverts to the command-processing mode it was in before the `COPY` started, and sends CommandComplete. The frontend cannot abort the transfer (except by closing the connection or issuing a Cancel request), but it can discard unwanted CopyData and CopyDone messages.

In the event of a backend-detected error during copy-out mode, the backend will issue an ErrorResponse message and revert to normal processing. The frontend should treat receipt of ErrorResponse as terminating the copy-out mode.

It is possible for NoticeResponse and ParameterStatus messages to be interspersed between CopyData messages; frontends must handle these cases, and should be prepared for other asynchronous message types as well (see Section 50.2.6). Otherwise, any message type other than CopyData or CopyDone may be treated as terminating copy-out mode.

There is another Copy-related mode called copy-both, which allows high-speed bulk data transfer to *and* from the server. Copy-both mode is initiated when a backend in walsender mode executes a `START_REPLICATION` statement. The backend sends a CopyBothResponse message to the frontend. Both the backend and the frontend may then send CopyData messages until either end sends a CopyDone message. After the client sends a CopyDone message, the connection goes from copy-both mode to copy-out mode, and the client may not send any more CopyData messages. Similarly, when the server sends a CopyDone message, the connection goes into copy-in mode, and the server may not send any more CopyData messages. After both sides have sent a CopyDone message, the copy mode is terminated, and the backend reverts to the command-processing mode. In the event of a backend-detected error during copy-both mode, the backend will issue an ErrorResponse message, discard frontend messages until a Sync message is received, and then issue ReadyForQuery and return to normal processing. The frontend should treat receipt of ErrorResponse as terminating the copy in both directions; no CopyDone should be sent in this case. See Section 50.3 for more information on the subprotocol transmitted over copy-both mode.

The CopyInResponse, CopyOutResponse and CopyBothResponse messages include fields that inform the frontend of the number of columns per row and the format codes being used for each column. (As of the present implementation, all columns in a given `COPY` operation will use the same format, but the message design does not assume this.)

50.2.6. Asynchronous Operations

There are several cases in which the backend will send messages that are not specifically prompted by the frontend's command stream. Frontends must be prepared to deal with these messages at any time, even when not engaged in a query. At minimum, one should check for these cases before beginning to read a query response.

It is possible for NoticeResponse messages to be generated due to outside activity; for example, if the database administrator commands a "fast" database shutdown, the backend will send a NoticeResponse indicating this fact before closing the connection. Accordingly, frontends should always be prepared to accept and display NoticeResponse messages, even when the connection is nominally idle.

ParameterStatus messages will be generated whenever the active value changes for any of the parameters the backend believes the frontend should know about. Most commonly this occurs in response to a `SET` SQL command executed by the frontend, and this case is effectively synchronous — but it is also possible for parameter status changes to occur because the administrator changed a configuration file and then sent the SIGHUP signal to the server. Also, if a `SET` command is rolled back, an appropriate ParameterStatus message will be generated to report the current effective value.

At present there is a hard-wired set of parameters for which ParameterStatus will be generated: they are `server_version`, `server_encoding`, `client_encoding`, `application_name`, `is_superuser`, `session_authorization`, `DateStyle`, `IntervalStyle`, `TimeZone`, `integer_datetimes`, and `standard_conforming_strings`. (`server_encoding`, `TimeZone`, and `integer_datetimes` were not reported by releases before 8.0; `standard_conforming_strings` was not reported by releases before 8.1; `IntervalStyle` was not reported by releases before 8.4; `application_name` was not reported by releases before 9.0.) Note that `server_version`, `server_encoding` and `integer_datetimes` are pseudo-parameters that cannot change after startup. This set might change in the future, or even become configurable. Accordingly, a frontend should simply ignore ParameterStatus for parameters that it does not understand or care about.

If a frontend issues a `LISTEN` command, then the backend will send a NotificationResponse message (not to be confused with NoticeResponse!) whenever a `NOTIFY` command is executed for the same channel name.

> **Note:** At present, NotificationResponse can only be sent outside a transaction, and thus it will not occur in the middle of a command-response series, though it might occur just before ReadyForQuery. It is unwise to design frontend logic that assumes that, however. Good practice is to be able to accept NotificationResponse at any point in the protocol.

50.2.7. Canceling Requests in Progress

During the processing of a query, the frontend might request cancellation of the query. The cancel request is not sent directly on the open connection to the backend for reasons of implementation efficiency: we don't want to have the backend constantly checking for new input from the frontend during query processing. Cancel requests should be relatively infrequent, so we make them slightly cumbersome in order to avoid a penalty in the normal case.

To issue a cancel request, the frontend opens a new connection to the server and sends a CancelRequest message, rather than the StartupMessage message that would ordinarily be sent across a new connection.

The server will process this request and then close the connection. For security reasons, no direct reply is made to the cancel request message.

A CancelRequest message will be ignored unless it contains the same key data (PID and secret key) passed to the frontend during connection start-up. If the request matches the PID and secret key for a currently executing backend, the processing of the current query is aborted. (In the existing implementation, this is done by sending a special signal to the backend process that is processing the query.)

The cancellation signal might or might not have any effect — for example, if it arrives after the backend has finished processing the query, then it will have no effect. If the cancellation is effective, it results in the current command being terminated early with an error message.

The upshot of all this is that for reasons of both security and efficiency, the frontend has no direct way to tell whether a cancel request has succeeded. It must continue to wait for the backend to respond to the query. Issuing a cancel simply improves the odds that the current query will finish soon, and improves the odds that it will fail with an error message instead of succeeding.

Since the cancel request is sent across a new connection to the server and not across the regular frontend/backend communication link, it is possible for the cancel request to be issued by any process, not just the frontend whose query is to be canceled. This might provide additional flexibility when building multiple-process applications. It also introduces a security risk, in that unauthorized persons might try to cancel queries. The security risk is addressed by requiring a dynamically generated secret key to be supplied in cancel requests.

50.2.8. Termination

The normal, graceful termination procedure is that the frontend sends a Terminate message and immediately closes the connection. On receipt of this message, the backend closes the connection and terminates.

In rare cases (such as an administrator-commanded database shutdown) the backend might disconnect without any frontend request to do so. In such cases the backend will attempt to send an error or notice message giving the reason for the disconnection before it closes the connection.

Other termination scenarios arise from various failure cases, such as core dump at one end or the other, loss of the communications link, loss of message-boundary synchronization, etc. If either frontend or backend sees an unexpected closure of the connection, it should clean up and terminate. The frontend has the option of launching a new backend by recontacting the server if it doesn't want to terminate itself. Closing the connection is also advisable if an unrecognizable message type is received, since this probably indicates loss of message-boundary sync.

For either normal or abnormal termination, any open transaction is rolled back, not committed. One should note however that if a frontend disconnects while a non-`SELECT` query is being processed, the backend will probably finish the query before noticing the disconnection. If the query is outside any transaction block (`BEGIN` ... `COMMIT` sequence) then its results might be committed before the disconnection is recognized.

50.2.9. SSL Session Encryption

If PostgreSQL was built with SSL support, frontend/backend communications can be encrypted using SSL. This provides communication security in environments where attackers might be able to capture the session traffic. For more information on encrypting PostgreSQL sessions with SSL, see Section 17.9.

To initiate an SSL-encrypted connection, the frontend initially sends an SSLRequest message rather than a StartupMessage. The server then responds with a single byte containing `S` or `N`, indicating that it is willing or unwilling to perform SSL, respectively. The frontend might close the connection at this point if it is dissatisfied with the response. To continue after `S`, perform an SSL startup handshake (not described here, part of the SSL specification) with the server. If this is successful, continue with sending the usual StartupMessage. In this case the StartupMessage and all subsequent data will be SSL-encrypted. To continue after `N`, send the usual StartupMessage and proceed without encryption.

The frontend should also be prepared to handle an ErrorMessage response to SSLRequest from the server. This would only occur if the server predates the addition of SSL support to PostgreSQL. (Such servers are now very ancient, and likely do not exist in the wild anymore.) In this case the connection must be closed, but the frontend might choose to open a fresh connection and proceed without requesting SSL.

An initial SSLRequest can also be used in a connection that is being opened to send a CancelRequest message.

While the protocol itself does not provide a way for the server to force SSL encryption, the administrator can configure the server to reject unencrypted sessions as a byproduct of authentication checking.

50.3. Streaming Replication Protocol

To initiate streaming replication, the frontend sends the `replication` parameter in the startup message. A Boolean value of `true` tells the backend to go into walsender mode, wherein a small set of replication commands can be issued instead of SQL statements. Only the simple query protocol can be used in walsender mode. Replication commands are logged in the server log when log_replication_commands is enabled. Passing `database` as the value instructs walsender to connect to the database specified in the `dbname` parameter, which will allow the connection to be used for logical replication from that database.

For the purpose of testing replication commands, you can make a replication connection via psql or any other `libpq`-using tool with a connection string including the `replication` option, e.g.:

```
psql "dbname=postgres replication=database" -c "IDENTIFY_SYSTEM;"
```

However it is often more useful to use pg_receivexlog (for physical replication) or pg_recvlogical (for logical replication).

The commands accepted in walsender mode are:

IDENTIFY_SYSTEM

Requests the server to identify itself. Server replies with a result set of a single row, containing four fields:

systemid

The unique system identifier identifying the cluster. This can be used to check that the base backup used to initialize the standby came from the same cluster.

timeline

Current TimelineID. Also useful to check that the standby is consistent with the master.

xlogpos

Current xlog flush location. Useful to get a known location in the transaction log where streaming can start.

dbname

Database connected to or NULL.

TIMELINE_HISTORY *tli*

Requests the server to send over the timeline history file for timeline *tli*. Server replies with a result set of a single row, containing two fields:

filename

Filename of the timeline history file, e.g `00000002.history`.

content

Contents of the timeline history file.

CREATE_REPLICATION_SLOT *slot_name* { PHYSICAL | LOGICAL *output_plugin* }

Create a physical or logical replication slot. See Section 25.2.6 for more about replication slots.

slot_name

The name of the slot to create. Must be a valid replication slot name (see Section 25.2.6.1).

output_plugin

The name of the output plugin used for logical decoding (see Section 46.6).

START_REPLICATION [SLOT *slot_name*] [PHYSICAL] *XXX/XXX* [TIMELINE *tli*]

Instructs server to start streaming WAL, starting at WAL position *XXX/XXX*. If TIMELINE option is specified, streaming starts on timeline *tli*; otherwise, the server's current timeline is selected. The server can reply with an error, e.g., if the requested section of WAL has already been recycled. On success, server responds with a CopyBothResponse message, and then starts to stream WAL to the frontend.

If a slot's name is provided via *slot_name*, it will be updated as replication progresses so that the server knows which WAL segments, and if `hot_standby_feedback` is on which transactions, are still needed by the standby.

If the client requests a timeline that's not the latest, but is part of the history of the server, the server will stream all the WAL on that timeline starting from the requested startpoint, up to the point where the server switched to another timeline. If the client requests streaming at exactly the end of an old timeline, the server responds immediately with CommandComplete without entering COPY mode.

After streaming all the WAL on a timeline that is not the latest one, the server will end streaming by exiting the COPY mode. When the client acknowledges this by also exiting COPY mode, the server sends a result set with one row and two columns, indicating the next timeline in this server's history. The first column is the next timeline's ID, and the second column is the XLOG position where the switch happened. Usually, the switch position is the end of the WAL that was streamed, but there

are corner cases where the server can send some WAL from the old timeline that it has not itself replayed before promoting. Finally, the server sends CommandComplete message, and is ready to accept a new command.

WAL data is sent as a series of CopyData messages. (This allows other information to be intermixed; in particular the server can send an ErrorResponse message if it encounters a failure after beginning to stream.) The payload of each CopyData message from server to the client contains a message of one of the following formats:

XLogData (B)

Byte1('w')

Identifies the message as WAL data.

Int64

The starting point of the WAL data in this message.

Int64

The current end of WAL on the server.

Int64

The server's system clock at the time of transmission, as microseconds since midnight on 2000-01-01.

Byte*n*

A section of the WAL data stream.

A single WAL record is never split across two XLogData messages. When a WAL record crosses a WAL page boundary, and is therefore already split using continuation records, it can be split at the page boundary. In other words, the first main WAL record and its continuation records can be sent in different XLogData messages.

Primary keepalive message (B)

Byte1('k')

Identifies the message as a sender keepalive.

Int64

The current end of WAL on the server.

Int64

The server's system clock at the time of transmission, as microseconds since midnight on 2000-01-01.

Byte1

1 means that the client should reply to this message as soon as possible, to avoid a timeout disconnect. 0 otherwise.

The receiving process can send replies back to the sender at any time, using one of the following message formats (also in the payload of a CopyData message):

Standby status update (F)

Byte1('r')

Identifies the message as a receiver status update.

Int64

The location of the last WAL byte + 1 received and written to disk in the standby.

Int64

The location of the last WAL byte + 1 flushed to disk in the standby.

Int64

The location of the last WAL byte + 1 applied in the standby.

Int64

The client's system clock at the time of transmission, as microseconds since midnight on 2000-01-01.

Byte1

If 1, the client requests the server to reply to this message immediately. This can be used to ping the server, to test if the connection is still healthy.

Hot Standby feedback message (F)

Byte1('h')

Identifies the message as a Hot Standby feedback message.

Int64

The client's system clock at the time of transmission, as microseconds since midnight on 2000-01-01.

Int32

The standby's current xmin. This may be 0, if the standby is sending notification that Hot Standby feedback will no longer be sent on this connection. Later non-zero messages may reinitiate the feedback mechanism.

Int32

The standby's current epoch.

START_REPLICATION SLOT *slot_name* LOGICAL *XXX/XXX* [(*option_name* [*option_value*] [, ...])]

Instructs server to start streaming WAL for logical replication, starting at WAL position *XXX/XXX*. The server can reply with an error, e.g., if the requested section of WAL has already been recycled. On success, server responds with a CopyBothResponse message, and then starts to stream WAL to the frontend.

The messages inside the CopyBothResponse messages are of the same format documented for START_REPLICATION ... PHYSICAL.

The output plugin associated with the selected slot is used to process the output for streaming.

SLOT *slot_name*

The name of the slot to stream changes from. This parameter is required, and must correspond to an existing logical replication slot created with CREATE_REPLICATION_SLOT in LOGICAL mode.

XXX/XXX

The WAL position to begin streaming at.

option_name

The name of an option passed to the slot's logical decoding plugin.

option_value

Optional value, in the form of a string constant, associated with the specified option.

DROP_REPLICATION_SLOT *slot_name*

Drops a replication slot, freeing any reserved server-side resources. If the slot is currently in use by an active connection, this command fails.

slot_name

The name of the slot to drop.

BASE_BACKUP [LABEL *'label'*] [PROGRESS] [FAST] [WAL] [NOWAIT] [MAX_RATE *rate*] [TABLESPACE_MAP]

Instructs the server to start streaming a base backup. The system will automatically be put in backup mode before the backup is started, and taken out of it when the backup is complete. The following options are accepted:

LABEL *'label'*

Sets the label of the backup. If none is specified, a backup label of base backup will be used. The quoting rules for the label are the same as a standard SQL string with standard_conforming_strings turned on.

PROGRESS

Request information required to generate a progress report. This will send back an approximate size in the header of each tablespace, which can be used to calculate how far along the stream is done. This is calculated by enumerating all the file sizes once before the transfer is even started, and may as such have a negative impact on the performance - in particular it may take longer before the first data is streamed. Since the database files can change during the backup, the size

is only approximate and may both grow and shrink between the time of approximation and the sending of the actual files.

`FAST`

Request a fast checkpoint.

`WAL`

Include the necessary WAL segments in the backup. This will include all the files between start and stop backup in the `pg_xlog` directory of the base directory tar file.

`NOWAIT`

By default, the backup will wait until the last required xlog segment has been archived, or emit a warning if log archiving is not enabled. Specifying `NOWAIT` disables both the waiting and the warning, leaving the client responsible for ensuring the required log is available.

`MAX_RATE` *rate*

Limit (throttle) the maximum amount of data transferred from server to client per unit of time. The expected unit is kilobytes per second. If this option is specified, the value must either be equal to zero or it must fall within the range from 32 kB through 1 GB (inclusive). If zero is passed or the option is not specified, no restriction is imposed on the transfer.

`TABLESPACE_MAP`

Include information about symbolic links present in the directory `pg_tblspc` in a file named `tablespace_map`. The tablespace map file includes each symbolic link name as it exists in the directory `pg_tblspc/` and the full path of that symbolic link.

When the backup is started, the server will first send two ordinary result sets, followed by one or more CopyResponse results.

The first ordinary result set contains the starting position of the backup, in a single row with two columns. The first column contains the start position given in XLogRecPtr format, and the second column contains the corresponding timeline ID.

The second ordinary result set has one row for each tablespace. The fields in this row are:

spcoid

The oid of the tablespace, or `NULL` if it's the base directory.

spclocation

The full path of the tablespace directory, or `NULL` if it's the base directory.

size

The approximate size of the tablespace, if progress report has been requested; otherwise it's `NULL`.

After the second regular result set, one or more CopyResponse results will be sent, one for PGDATA and one for each additional tablespace other than `pg_default` and `pg_global`. The data in the CopyResponse results will be a tar format (following the "ustar interchange format" specified in the POSIX 1003.1-2008 standard) dump of the tablespace contents, except that the two trailing blocks of

zeroes specified in the standard are omitted. After the tar data is complete, a final ordinary result set will be sent, containing the WAL end position of the backup, in the same format as the start position.

The tar archive for the data directory and each tablespace will contain all files in the directories, regardless of whether they are PostgreSQL files or other files added to the same directory. The only excluded files are:

- `postmaster.pid`
- `postmaster.opts`
- various temporary files created during the operation of the PostgreSQL server
- `pg_xlog`, including subdirectories. If the backup is run with WAL files included, a synthesized version of `pg_xlog` will be included, but it will only contain the files necessary for the backup to work, not the rest of the contents.
- `pg_replslot` is copied as an empty directory.
- Files other than regular files and directories, such as symbolic links and special device files, are skipped. (Symbolic links in `pg_tblspc` are maintained.)

Owner, group and file mode are set if the underlying file system on the server supports it.

Once all tablespaces have been sent, a final regular result set will be sent. This result set contains the end position of the backup, given in XLogRecPtr format as a single column in a single row.

50.4. Message Data Types

This section describes the base data types used in messages.

Int*n*(*i*)

An *n*-bit integer in network byte order (most significant byte first). If *i* is specified it is the exact value that will appear, otherwise the value is variable. Eg. Int16, Int32(42).

Int*n*[*k*]

An array of *k* *n*-bit integers, each in network byte order. The array length *k* is always determined by an earlier field in the message. Eg. Int16[M].

String(*s*)

A null-terminated string (C-style string). There is no specific length limitation on strings. If *s* is specified it is the exact value that will appear, otherwise the value is variable. Eg. String, String("user").

> **Note:** *There is no predefined limit* on the length of a string that can be returned by the backend. Good coding strategy for a frontend is to use an expandable buffer so that anything that fits in memory can be accepted. If that's not feasible, read the full string and discard trailing characters that don't fit into your fixed-size buffer.

Byte*n*(*c*)

Exactly *n* bytes. If the field width *n* is not a constant, it is always determinable from an earlier field in the message. If *c* is specified it is the exact value. Eg. Byte2, Byte1('\n').

50.5. Message Formats

This section describes the detailed format of each message. Each is marked to indicate that it can be sent by a frontend (F), a backend (B), or both (F & B). Notice that although each message includes a byte count at the beginning, the message format is defined so that the message end can be found without reference to the byte count. This aids validity checking. (The CopyData message is an exception, because it forms part of a data stream; the contents of any individual CopyData message cannot be interpretable on their own.)

AuthenticationOk (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(8)

Length of message contents in bytes, including self.

Int32(0)

Specifies that the authentication was successful.

AuthenticationKerberosV5 (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(8)

Length of message contents in bytes, including self.

Int32(2)

Specifies that Kerberos V5 authentication is required.

AuthenticationCleartextPassword (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(8)

Length of message contents in bytes, including self.

Int32(3)

Specifies that a clear-text password is required.

AuthenticationMD5Password (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(12)

Length of message contents in bytes, including self.

Int32(5)

Specifies that an MD5-encrypted password is required.

Byte4

The salt to use when encrypting the password.

AuthenticationSCMCredential (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(8)

Length of message contents in bytes, including self.

Int32(6)

Specifies that an SCM credentials message is required.

AuthenticationGSS (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(8)

Length of message contents in bytes, including self.

Int32(7)

Specifies that GSSAPI authentication is required.

AuthenticationSSPI (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(8)

Length of message contents in bytes, including self.

Int32(9)

Specifies that SSPI authentication is required.

AuthenticationGSSContinue (B)

Byte1('R')

Identifies the message as an authentication request.

Int32

Length of message contents in bytes, including self.

Int32(8)

Specifies that this message contains GSSAPI or SSPI data.

Byte*n*

GSSAPI or SSPI authentication data.

BackendKeyData (B)

Byte1('K')

Identifies the message as cancellation key data. The frontend must save these values if it wishes to be able to issue CancelRequest messages later.

Int32(12)

Length of message contents in bytes, including self.

Int32

The process ID of this backend.

Int32

The secret key of this backend.

Bind (F)

Byte1('B')

Identifies the message as a Bind command.

Int32

Length of message contents in bytes, including self.

String

The name of the destination portal (an empty string selects the unnamed portal).

String

The name of the source prepared statement (an empty string selects the unnamed prepared statement).

Int16

The number of parameter format codes that follow (denoted *C* below). This can be zero to indicate that there are no parameters or that the parameters all use the default format (text); or one, in which case the specified format code is applied to all parameters; or it can equal the actual number of parameters.

Int16[*C*]

The parameter format codes. Each must presently be zero (text) or one (binary).

Int16

The number of parameter values that follow (possibly zero). This must match the number of parameters needed by the query.

Next, the following pair of fields appear for each parameter:

Int32

The length of the parameter value, in bytes (this count does not include itself). Can be zero. As a special case, -1 indicates a NULL parameter value. No value bytes follow in the NULL case.

Byte*n*

The value of the parameter, in the format indicated by the associated format code. *n* is the above length.

After the last parameter, the following fields appear:

Int16

The number of result-column format codes that follow (denoted *R* below). This can be zero to indicate that there are no result columns or that the result columns should all use the default format (text); or one, in which case the specified format code is applied to all result columns (if any); or it can equal the actual number of result columns of the query.

Int16[*R*]

The result-column format codes. Each must presently be zero (text) or one (binary).

BindComplete (B)

Byte1('2')

Identifies the message as a Bind-complete indicator.

Int32(4)

Length of message contents in bytes, including self.

CancelRequest (F)

Int32(16)

Length of message contents in bytes, including self.

Int32(80877102)

The cancel request code. The value is chosen to contain `1234` in the most significant 16 bits, and `5678` in the least significant 16 bits. (To avoid confusion, this code must not be the same as any protocol version number.)

Int32

The process ID of the target backend.

Int32

The secret key for the target backend.

Close (F)

Byte1('C')

Identifies the message as a Close command.

Int32

Length of message contents in bytes, including self.

Byte1

'`S`' to close a prepared statement; or '`P`' to close a portal.

String

The name of the prepared statement or portal to close (an empty string selects the unnamed prepared statement or portal).

CloseComplete (B)

Byte1('3')

Identifies the message as a Close-complete indicator.

Int32(4)

Length of message contents in bytes, including self.

CommandComplete (B)

Byte1('C')

Identifies the message as a command-completed response.

Int32

Length of message contents in bytes, including self.

String

The command tag. This is usually a single word that identifies which SQL command was completed.

For an `INSERT` command, the tag is `INSERT` *`oid rows`*, where *`rows`* is the number of rows inserted. *`oid`* is the object ID of the inserted row if *`rows`* is 1 and the target table has OIDs; otherwise *`oid`* is 0.

For a `DELETE` command, the tag is `DELETE` *`rows`* where *`rows`* is the number of rows deleted.

For an `UPDATE` command, the tag is `UPDATE` *`rows`* where *`rows`* is the number of rows updated.

For a `SELECT` or `CREATE TABLE AS` command, the tag is `SELECT` *`rows`* where *`rows`* is the number of rows retrieved.

For a `MOVE` command, the tag is `MOVE` *`rows`* where *`rows`* is the number of rows the cursor's position has been changed by.

For a `FETCH` command, the tag is `FETCH` *`rows`* where *`rows`* is the number of rows that have been retrieved from the cursor.

For a `COPY` command, the tag is `COPY` *`rows`* where *`rows`* is the number of rows copied. (Note: the row count appears only in PostgreSQL 8.2 and later.)

CopyData (F & B)

Byte1('d')

Identifies the message as `COPY` data.

Int32

Length of message contents in bytes, including self.

Byte*n*

Data that forms part of a `COPY` data stream. Messages sent from the backend will always correspond to single data rows, but messages sent by frontends might divide the data stream arbitrarily.

CopyDone (F & B)

Byte1('c')

Identifies the message as a `COPY`-complete indicator.

Int32(4)

Length of message contents in bytes, including self.

CopyFail (F)

Byte1('f')

Identifies the message as a `COPY`-failure indicator.

Int32

Length of message contents in bytes, including self.

String

An error message to report as the cause of failure.

CopyInResponse (B)

Byte1('G')

Identifies the message as a Start Copy In response. The frontend must now send copy-in data (if not prepared to do so, send a CopyFail message).

Int32

Length of message contents in bytes, including self.

Int8

0 indicates the overall `COPY` format is textual (rows separated by newlines, columns separated by separator characters, etc). 1 indicates the overall copy format is binary (similar to DataRow format). See COPY for more information.

Int16

The number of columns in the data to be copied (denoted *N* below).

Int16[*N*]

The format codes to be used for each column. Each must presently be zero (text) or one (binary). All must be zero if the overall copy format is textual.

CopyOutResponse (B)

Byte1('H')

Identifies the message as a Start Copy Out response. This message will be followed by copy-out data.

Int32

Length of message contents in bytes, including self.

Int8

0 indicates the overall `COPY` format is textual (rows separated by newlines, columns separated by separator characters, etc). 1 indicates the overall copy format is binary (similar to DataRow format). See COPY for more information.

Int16

The number of columns in the data to be copied (denoted *N* below).

Int16[*N*]

The format codes to be used for each column. Each must presently be zero (text) or one (binary). All must be zero if the overall copy format is textual.

CopyBothResponse (B)

Byte1('W')

Identifies the message as a Start Copy Both response. This message is used only for Streaming Replication.

Int32

Length of message contents in bytes, including self.

Int8

0 indicates the overall `COPY` format is textual (rows separated by newlines, columns separated by separator characters, etc). 1 indicates the overall copy format is binary (similar to DataRow format). See COPY for more information.

Int16

The number of columns in the data to be copied (denoted *N* below).

Int16[*N*]

The format codes to be used for each column. Each must presently be zero (text) or one (binary). All must be zero if the overall copy format is textual.

DataRow (B)

Byte1('D')

Identifies the message as a data row.

Int32

Length of message contents in bytes, including self.

Int16

The number of column values that follow (possibly zero).

Next, the following pair of fields appear for each column:

Int32

The length of the column value, in bytes (this count does not include itself). Can be zero. As a special case, -1 indicates a NULL column value. No value bytes follow in the NULL case.

Byte*n*

The value of the column, in the format indicated by the associated format code. *n* is the above length.

Describe (F)

Byte1('D')

Identifies the message as a Describe command.

Int32

Length of message contents in bytes, including self.

Byte1

'`S`' to describe a prepared statement; or '`P`' to describe a portal.

String

The name of the prepared statement or portal to describe (an empty string selects the unnamed prepared statement or portal).

EmptyQueryResponse (B)

Byte1('I')

Identifies the message as a response to an empty query string. (This substitutes for CommandComplete.)

Int32(4)

Length of message contents in bytes, including self.

ErrorResponse (B)

Byte1('E')

Identifies the message as an error.

Int32

Length of message contents in bytes, including self.

The message body consists of one or more identified fields, followed by a zero byte as a terminator. Fields can appear in any order. For each field there is the following:

Byte1

A code identifying the field type; if zero, this is the message terminator and no string follows. The presently defined field types are listed in Section 50.6. Since more field types might be added in future, frontends should silently ignore fields of unrecognized type.

String

The field value.

Execute (F)

Byte1('E')

Identifies the message as an Execute command.

Int32

Length of message contents in bytes, including self.

String

The name of the portal to execute (an empty string selects the unnamed portal).

Int32

Maximum number of rows to return, if portal contains a query that returns rows (ignored otherwise). Zero denotes “no limit”.

Flush (F)

Byte1('H')

Identifies the message as a Flush command.

Int32(4)

Length of message contents in bytes, including self.

FunctionCall (F)

Byte1('F')

Identifies the message as a function call.

Int32

Length of message contents in bytes, including self.

Int32

Specifies the object ID of the function to call.

Int16

The number of argument format codes that follow (denoted *C* below). This can be zero to indicate that there are no arguments or that the arguments all use the default format (text); or one, in which case the specified format code is applied to all arguments; or it can equal the actual number of arguments.

Int16[*C*]

The argument format codes. Each must presently be zero (text) or one (binary).

Int16

Specifies the number of arguments being supplied to the function.

Next, the following pair of fields appear for each argument:

Int32

The length of the argument value, in bytes (this count does not include itself). Can be zero. As a special case, -1 indicates a NULL argument value. No value bytes follow in the NULL case.

Byte*n*

The value of the argument, in the format indicated by the associated format code. *n* is the above length.

After the last argument, the following field appears:

Int16

The format code for the function result. Must presently be zero (text) or one (binary).

FunctionCallResponse (B)

Byte1('V')

Identifies the message as a function call result.

Int32

Length of message contents in bytes, including self.

Int32

The length of the function result value, in bytes (this count does not include itself). Can be zero. As a special case, -1 indicates a NULL function result. No value bytes follow in the NULL case.

Byte*n*

The value of the function result, in the format indicated by the associated format code. *n* is the above length.

NegotiateProtocolVersion (B)

Byte1('v')

Identifies the message as a protocol version negotiation message.

Int32

Length of message contents in bytes, including self.

Int32

Newest minor protocol version supported by the server for the major protocol version requested by the client.

Int32

Number of protocol options not recognized by the server.

Then, for protocol option not recognized by the server, there is the following:

String

The option name.

NoData (B)

Byte1('n')

Identifies the message as a no-data indicator.

Int32(4)

Length of message contents in bytes, including self.

NoticeResponse (B)

Byte1('N')

Identifies the message as a notice.

Int32

Length of message contents in bytes, including self.

The message body consists of one or more identified fields, followed by a zero byte as a terminator. Fields can appear in any order. For each field there is the following:

Byte1

A code identifying the field type; if zero, this is the message terminator and no string follows. The presently defined field types are listed in Section 50.6. Since more field types might be added in future, frontends should silently ignore fields of unrecognized type.

String

The field value.

NotificationResponse (B)

Byte1('A')

Identifies the message as a notification response.

Int32

Length of message contents in bytes, including self.

Int32

The process ID of the notifying backend process.

String

The name of the channel that the notify has been raised on.

String

The "payload" string passed from the notifying process.

ParameterDescription (B)

Byte1('t')

Identifies the message as a parameter description.

Int32

Length of message contents in bytes, including self.

Int16

The number of parameters used by the statement (can be zero).

Then, for each parameter, there is the following:

Int32

Specifies the object ID of the parameter data type.

ParameterStatus (B)

Byte1('S')

Identifies the message as a run-time parameter status report.

Int32

Length of message contents in bytes, including self.

String

The name of the run-time parameter being reported.

String

The current value of the parameter.

Parse (F)

Byte1('P')

Identifies the message as a Parse command.

Int32

Length of message contents in bytes, including self.

String

The name of the destination prepared statement (an empty string selects the unnamed prepared statement).

String

The query string to be parsed.

Int16

The number of parameter data types specified (can be zero). Note that this is not an indication of the number of parameters that might appear in the query string, only the number that the frontend wants to prespecify types for.

Then, for each parameter, there is the following:

Int32

Specifies the object ID of the parameter data type. Placing a zero here is equivalent to leaving the type unspecified.

ParseComplete (B)

Byte1('1')

Identifies the message as a Parse-complete indicator.

Int32(4)

Length of message contents in bytes, including self.

PasswordMessage (F)

Byte1('p')

Identifies the message as a password response. Note that this is also used for GSSAPI and SSPI response messages (which is really a design error, since the contained data is not a null-terminated string in that case, but can be arbitrary binary data).

Int32

Length of message contents in bytes, including self.

String

The password (encrypted, if requested).

PortalSuspended (B)

Byte1('s')

Identifies the message as a portal-suspended indicator. Note this only appears if an Execute message's row-count limit was reached.

Int32(4)

Length of message contents in bytes, including self.

Query (F)

Byte1('Q')

Identifies the message as a simple query.

Int32

Length of message contents in bytes, including self.

String

The query string itself.

ReadyForQuery (B)

Byte1('Z')

Identifies the message type. ReadyForQuery is sent whenever the backend is ready for a new query cycle.

Int32(5)

Length of message contents in bytes, including self.

Byte1

Current backend transaction status indicator. Possible values are '`I`' if idle (not in a transaction block); '`T`' if in a transaction block; or '`E`' if in a failed transaction block (queries will be rejected until block is ended).

RowDescription (B)

Byte1('T')

Identifies the message as a row description.

Int32

Length of message contents in bytes, including self.

Int16

Specifies the number of fields in a row (can be zero).

Then, for each field, there is the following:

String

The field name.

Int32

If the field can be identified as a column of a specific table, the object ID of the table; otherwise zero.

Int16

If the field can be identified as a column of a specific table, the attribute number of the column; otherwise zero.

Int32

The object ID of the field's data type.

Int16

The data type size (see `pg_type.typlen`). Note that negative values denote variable-width types.

Int32

The type modifier (see `pg_attribute.atttypmod`). The meaning of the modifier is type-specific.

Int16

The format code being used for the field. Currently will be zero (text) or one (binary). In a RowDescription returned from the statement variant of Describe, the format code is not yet known and will always be zero.

SSLRequest (F)

Int32(8)

Length of message contents in bytes, including self.

Int32(80877103)

The SSL request code. The value is chosen to contain `1234` in the most significant 16 bits, and `5679` in the least significant 16 bits. (To avoid confusion, this code must not be the same as any protocol version number.)

StartupMessage (F)

Int32

Length of message contents in bytes, including self.

Int32(196608)

The protocol version number. The most significant 16 bits are the major version number (3 for the protocol described here). The least significant 16 bits are the minor version number (0 for the protocol described here).

The protocol version number is followed by one or more pairs of parameter name and value strings. A zero byte is required as a terminator after the last name/value pair. Parameters can appear in any order. `user` is required, others are optional. Each parameter is specified as:

String

The parameter name. Currently recognized names are:

`user`

The database user name to connect as. Required; there is no default.

`database`

The database to connect to. Defaults to the user name.

`options`

Command-line arguments for the backend. (This is deprecated in favor of setting individual run-time parameters.) Spaces within this string are considered to separate arguments, unless escaped with a backslash (`\`); write `\\` to represent a literal backslash.

In addition to the above, other parameters may be listed. Parameter names beginning with `_pq_.` are reserved for use as protocol extensions, while others are treated as run-time parameters to be set at backend start time. Such settings will be applied during backend start (after parsing the command-line arguments if any) and will act as session defaults.

String

The parameter value.

Sync (F)

Byte1('S')

Identifies the message as a Sync command.

Int32(4)

Length of message contents in bytes, including self.

Terminate (F)

Byte1('X')

Identifies the message as a termination.

Int32(4)

Length of message contents in bytes, including self.

50.6. Error and Notice Message Fields

This section describes the fields that can appear in ErrorResponse and NoticeResponse messages. Each field type has a single-byte identification token. Note that any given field type should appear at most once per message.

`S`

Severity: the field contents are `ERROR`, `FATAL`, or `PANIC` (in an error message), or `WARNING`, `NOTICE`, `DEBUG`, `INFO`, or `LOG` (in a notice message), or a localized translation of one of these. Always present.

`C`

Code: the SQLSTATE code for the error (see Appendix A). Not localizable. Always present.

`M`

Message: the primary human-readable error message. This should be accurate but terse (typically one line). Always present.

`D`

Detail: an optional secondary error message carrying more detail about the problem. Might run to multiple lines.

`H`

Hint: an optional suggestion what to do about the problem. This is intended to differ from Detail in that it offers advice (potentially inappropriate) rather than hard facts. Might run to multiple lines.

`P`

Position: the field value is a decimal ASCII integer, indicating an error cursor position as an index into the original query string. The first character has index 1, and positions are measured in characters not bytes.

`p`

Internal position: this is defined the same as the `P` field, but it is used when the cursor position refers to an internally generated command rather than the one submitted by the client. The `q` field will always appear when this field appears.

`q`

Internal query: the text of a failed internally-generated command. This could be, for example, a SQL query issued by a PL/pgSQL function.

`W`

Where: an indication of the context in which the error occurred. Presently this includes a call stack traceback of active procedural language functions and internally-generated queries. The trace is one entry per line, most recent first.

`s`

Schema name: if the error was associated with a specific database object, the name of the schema containing that object, if any.

`t`

Table name: if the error was associated with a specific table, the name of the table. (Refer to the schema name field for the name of the table's schema.)

`c`

Column name: if the error was associated with a specific table column, the name of the column. (Refer to the schema and table name fields to identify the table.)

`d`

Data type name: if the error was associated with a specific data type, the name of the data type. (Refer to the schema name field for the name of the data type's schema.)

`n`

Constraint name: if the error was associated with a specific constraint, the name of the constraint. Refer to fields listed above for the associated table or domain. (For this purpose, indexes are treated as constraints, even if they weren't created with constraint syntax.)

`F`

File: the file name of the source-code location where the error was reported.

`L`

Line: the line number of the source-code location where the error was reported.

`R`

Routine: the name of the source-code routine reporting the error.

Note: The fields for schema name, table name, column name, data type name, and constraint name are supplied only for a limited number of error types; see Appendix A. Frontends should not assume that the presence of any of these fields guarantees the presence of another field. Core error sources observe the interrelationships noted above, but user-defined functions may use these fields in other ways. In the same vein, clients should not assume that these fields denote contemporary objects in the current database.

The client is responsible for formatting displayed information to meet its needs; in particular it should break long lines as needed. Newline characters appearing in the error message fields should be treated as paragraph breaks, not line breaks.

50.7. Summary of Changes since Protocol 2.0

This section provides a quick checklist of changes, for the benefit of developers trying to update existing client libraries to protocol 3.0.

The initial startup packet uses a flexible list-of-strings format instead of a fixed format. Notice that session default values for run-time parameters can now be specified directly in the startup packet. (Actually, you could do that before using the `options` field, but given the limited width of `options` and the lack of any way to quote whitespace in the values, it wasn't a very safe technique.)

All messages now have a length count immediately following the message type byte (except for startup packets, which have no type byte). Also note that PasswordMessage now has a type byte.

ErrorResponse and NoticeResponse ('`E`' and '`N`') messages now contain multiple fields, from which the client code can assemble an error message of the desired level of verbosity. Note that individual fields will typically not end with a newline, whereas the single string sent in the older protocol always did.

The ReadyForQuery ('`Z`') message includes a transaction status indicator.

The distinction between BinaryRow and DataRow message types is gone; the single DataRow message type serves for returning data in all formats. Note that the layout of DataRow has changed to make it easier to parse. Also, the representation of binary values has changed: it is no longer directly tied to the server's internal representation.

There is a new "extended query" sub-protocol, which adds the frontend message types Parse, Bind, Execute, Describe, Close, Flush, and Sync, and the backend message types ParseComplete, BindComplete, PortalSuspended, ParameterDescription, NoData, and CloseComplete. Existing clients do not have to concern themselves with this sub-protocol, but making use of it might allow improvements in performance or functionality.

`COPY` data is now encapsulated into CopyData and CopyDone messages. There is a well-defined way to recover from errors during `COPY`. The special "`\.`" last line is not needed anymore, and is not sent during `COPY OUT`. (It is still recognized as a terminator during `COPY IN`, but its use is deprecated and

will eventually be removed.) Binary `COPY` is supported. The CopyInResponse and CopyOutResponse messages include fields indicating the number of columns and the format of each column.

The layout of FunctionCall and FunctionCallResponse messages has changed. FunctionCall can now support passing NULL arguments to functions. It also can handle passing parameters and retrieving results in either text or binary format. There is no longer any reason to consider FunctionCall a potential security hole, since it does not offer direct access to internal server data representations.

The backend sends ParameterStatus (`'S'`) messages during connection startup for all parameters it considers interesting to the client library. Subsequently, a ParameterStatus message is sent whenever the active value changes for any of these parameters.

The RowDescription (`'T'`) message carries new table OID and column number fields for each column of the described row. It also shows the format code for each column.

The CursorResponse (`'P'`) message is no longer generated by the backend.

The NotificationResponse (`'A'`) message has an additional string field, which can carry a "payload" string passed from the `NOTIFY` event sender.

The EmptyQueryResponse (`'I'`) message used to include an empty string parameter; this has been removed.

Chapter 51. PostgreSQL Coding Conventions

51.1. Formatting

Source code formatting uses 4 column tab spacing, with tabs preserved (i.e., tabs are not expanded to spaces). Each logical indentation level is one additional tab stop.

Layout rules (brace positioning, etc) follow BSD conventions. In particular, curly braces for the controlled blocks of `if`, `while`, `switch`, etc go on their own lines.

Limit line lengths so that the code is readable in an 80-column window. (This doesn't mean that you must never go past 80 columns. For instance, breaking a long error message string in arbitrary places just to keep the code within 80 columns is probably not a net gain in readability.)

Do not use C++ style comments (`//` comments). Strict ANSI C compilers do not accept them. For the same reason, do not use C++ extensions such as declaring new variables mid-block.

The preferred style for multi-line comment blocks is

```
/*
 * comment text begins here
 * and continues here
 */
```

Note that comment blocks that begin in column 1 will be preserved as-is by pgindent, but it will re-flow indented comment blocks as though they were plain text. If you want to preserve the line breaks in an indented block, add dashes like this:

```
    /*----------
     * comment text begins here
     * and continues here
     *----------
     */
```

While submitted patches do not absolutely have to follow these formatting rules, it's a good idea to do so. Your code will get run through pgindent before the next release, so there's no point in making it look nice under some other set of formatting conventions. A good rule of thumb for patches is "make the new code look like the existing code around it".

The `src/tools` directory contains sample settings files that can be used with the emacs, xemacs or vim editors to help ensure that they format code according to these conventions.

The text browsing tools more and less can be invoked as:

```
more -x4
less -x4
```

to make them show tabs appropriately.

51.2. Reporting Errors Within the Server

Error, warning, and log messages generated within the server code should be created using `ereport`, or its older cousin `elog`. The use of this function is complex enough to require some explanation.

There are two required elements for every message: a severity level (ranging from `DEBUG` to `PANIC`) and a primary message text. In addition there are optional elements, the most common of which is an error identifier code that follows the SQL spec's SQLSTATE conventions. `ereport` itself is just a shell function, that exists mainly for the syntactic convenience of making message generation look like a function call in the C source code. The only parameter accepted directly by `ereport` is the severity level. The primary message text and any optional message elements are generated by calling auxiliary functions, such as `errmsg`, within the `ereport` call.

A typical call to `ereport` might look like this:

```
ereport(ERROR,
        (errcode(ERRCODE_DIVISION_BY_ZERO),
         errmsg("division by zero")));
```

This specifies error severity level `ERROR` (a run-of-the-mill error). The `errcode` call specifies the SQLSTATE error code using a macro defined in `src/include/utils/errcodes.h`. The `errmsg` call provides the primary message text. Notice the extra set of parentheses surrounding the auxiliary function calls — these are annoying but syntactically necessary.

Here is a more complex example:

```
ereport(ERROR,
        (errcode(ERRCODE_AMBIGUOUS_FUNCTION),
         errmsg("function %s is not unique",
                func_signature_string(funcname, nargs,
                                      NIL, actual_arg_types)),
         errhint("Unable to choose a best candidate function. "
                 "You might need to add explicit typecasts.")));
```

This illustrates the use of format codes to embed run-time values into a message text. Also, an optional "hint" message is provided.

If the severity level is `ERROR` or higher, `ereport` aborts the execution of the user-defined function and does not return to the caller. If the severity level is lower than `ERROR`, `ereport` returns normally.

The available auxiliary routines for `ereport` are:

- `errcode(sqlerrcode)` specifies the SQLSTATE error identifier code for the condition. If this routine is not called, the error identifier defaults to `ERRCODE_INTERNAL_ERROR` when the error severity level is `ERROR` or higher, `ERRCODE_WARNING` when the error level is `WARNING`, otherwise (for `NOTICE` and below) `ERRCODE_SUCCESSFUL_COMPLETION`. While these defaults are often convenient, always think whether they are appropriate before omitting the `errcode()` call.
- `errmsg(const char *msg, ...)` specifies the primary error message text, and possibly run-time values to insert into it. Insertions are specified by `sprintf`-style format codes. In addition to the standard format codes accepted by `sprintf`, the format code `%m` can be used to insert the error message

returned by `strerror` for the current value of `errno`. [1] `%m` does not require any corresponding entry in the parameter list for `errmsg`. Note that the message string will be run through `gettext` for possible localization before format codes are processed.

- `errmsg_internal(const char *msg, ...)` is the same as `errmsg`, except that the message string will not be translated nor included in the internationalization message dictionary. This should be used for "cannot happen" cases that are probably not worth expending translation effort on.
- `errmsg_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)` is like `errmsg`, but with support for various plural forms of the message. *`fmt_singular`* is the English singular format, *`fmt_plural`* is the English plural format, *`n`* is the integer value that determines which plural form is needed, and the remaining arguments are formatted according to the selected format string. For more information see Section 52.2.2.
- `errdetail(const char *msg, ...)` supplies an optional "detail" message; this is to be used when there is additional information that seems inappropriate to put in the primary message. The message string is processed in just the same way as for `errmsg`.
- `errdetail_internal(const char *msg, ...)` is the same as `errdetail`, except that the message string will not be translated nor included in the internationalization message dictionary. This should be used for detail messages that are not worth expending translation effort on, for instance because they are too technical to be useful to most users.
- `errdetail_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)` is like `errdetail`, but with support for various plural forms of the message. For more information see Section 52.2.2.
- `errdetail_log(const char *msg, ...)` is the same as `errdetail` except that this string goes only to the server log, never to the client. If both `errdetail` (or one of its equivalents above) and `errdetail_log` are used then one string goes to the client and the other to the log. This is useful for error details that are too security-sensitive or too bulky to include in the report sent to the client.
- `errdetail_log_plural(const char *fmt_singular, const char *fmt_plural, unsigned long n, ...)` is like `errdetail_log`, but with support for various plural forms of the message. For more information see Section 52.2.2.
- `errhint(const char *msg, ...)` supplies an optional "hint" message; this is to be used when offering suggestions about how to fix the problem, as opposed to factual details about what went wrong. The message string is processed in just the same way as for `errmsg`.
- `errcontext(const char *msg, ...)` is not normally called directly from an `ereport` message site; rather it is used in `error_context_stack` callback functions to provide information about the context in which an error occurred, such as the current location in a PL function. The message string is processed in just the same way as for `errmsg`. Unlike the other auxiliary functions, this can be called more than once per `ereport` call; the successive strings thus supplied are concatenated with separating newlines.
- `errposition(int cursorpos)` specifies the textual location of an error within a query string. Currently it is only useful for errors detected in the lexical and syntactic analysis phases of query processing.

1. That is, the value that was current when the `ereport` call was reached; changes of `errno` within the auxiliary reporting routines will not affect it. That would not be true if you were to write `strerror(errno)` explicitly in `errmsg`'s parameter list; accordingly, do not do so.

- `errtable(Relation rel)` specifies a relation whose name and schema name should be included as auxiliary fields in the error report.
- `errtablecol(Relation rel, int attnum)` specifies a column whose name, table name, and schema name should be included as auxiliary fields in the error report.
- `errtableconstraint(Relation rel, const char *conname)` specifies a table constraint whose name, table name, and schema name should be included as auxiliary fields in the error report. Indexes should be considered to be constraints for this purpose, whether or not they have an associated `pg_constraint` entry. Be careful to pass the underlying heap relation, not the index itself, as `rel`.
- `errdatatype(Oid datatypeOid)` specifies a data type whose name and schema name should be included as auxiliary fields in the error report.
- `errdomainconstraint(Oid datatypeOid, const char *conname)` specifies a domain constraint whose name, domain name, and schema name should be included as auxiliary fields in the error report.
- `errcode_for_file_access()` is a convenience function that selects an appropriate SQLSTATE error identifier for a failure in a file-access-related system call. It uses the saved `errno` to determine which error code to generate. Usually this should be used in combination with `%m` in the primary error message text.
- `errcode_for_socket_access()` is a convenience function that selects an appropriate SQLSTATE error identifier for a failure in a socket-related system call.
- `errhidestmt(bool hide_stmt)` can be called to specify suppression of the `STATEMENT:` portion of a message in the postmaster log. Generally this is appropriate if the message text includes the current statement already.

> **Note:** At most one of the functions `errtable`, `errtablecol`, `errtableconstraint`, `errdatatype`, or `errdomainconstraint` should be used in an `ereport` call. These functions exist to allow applications to extract the name of a database object associated with the error condition without having to examine the potentially-localized error message text. These functions should be used in error reports for which it's likely that applications would wish to have automatic error handling. As of PostgreSQL 9.3, complete coverage exists only for errors in SQLSTATE class 23 (integrity constraint violation), but this is likely to be expanded in future.

There is an older function `elog` that is still heavily used. An `elog` call:

```
elog(level, "format string", ...);
```

is exactly equivalent to:

```
ereport(level, (errmsg_internal("format string", ...)));
```

Notice that the SQLSTATE error code is always defaulted, and the message string is not subject to translation. Therefore, `elog` should be used only for internal errors and low-level debug logging. Any message that is likely to be of interest to ordinary users should go through `ereport`. Nonetheless, there are enough internal "cannot happen" error checks in the system that `elog` is still widely used; it is preferred for those messages for its notational simplicity.

Advice about writing good error messages can be found in Section 51.3.

51.3. Error Message Style Guide

This style guide is offered in the hope of maintaining a consistent, user-friendly style throughout all the messages generated by PostgreSQL.

51.3.1. What Goes Where

The primary message should be short, factual, and avoid reference to implementation details such as specific function names. "Short" means "should fit on one line under normal conditions". Use a detail message if needed to keep the primary message short, or if you feel a need to mention implementation details such as the particular system call that failed. Both primary and detail messages should be factual. Use a hint message for suggestions about what to do to fix the problem, especially if the suggestion might not always be applicable.

For example, instead of:

```
IpcMemoryCreate: shmget(key=%d, size=%u, 0%o) failed: %m
(plus a long addendum that is basically a hint)
```

write:

```
Primary:    could not create shared memory segment: %m
Detail:     Failed syscall was shmget(key=%d, size=%u, 0%o).
Hint:       the addendum
```

Rationale: keeping the primary message short helps keep it to the point, and lets clients lay out screen space on the assumption that one line is enough for error messages. Detail and hint messages can be relegated to a verbose mode, or perhaps a pop-up error-details window. Also, details and hints would normally be suppressed from the server log to save space. Reference to implementation details is best avoided since users don't know the details anyway.

51.3.2. Formatting

Don't put any specific assumptions about formatting into the message texts. Expect clients and the server log to wrap lines to fit their own needs. In long messages, newline characters (\n) can be used to indicate suggested paragraph breaks. Don't end a message with a newline. Don't use tabs or other formatting characters. (In error context displays, newlines are automatically added to separate levels of context such as function calls.)

Rationale: Messages are not necessarily displayed on terminal-type displays. In GUI displays or browsers these formatting instructions are at best ignored.

51.3.3. Quotation Marks

English text should use double quotes when quoting is appropriate. Text in other languages should consistently use one kind of quotes that is consistent with publishing customs and computer output of other programs.

Rationale: The choice of double quotes over single quotes is somewhat arbitrary, but tends to be the preferred use. Some have suggested choosing the kind of quotes depending on the type of object according to SQL conventions (namely, strings single quoted, identifiers double quoted). But this is a language-internal technical issue that many users aren't even familiar with, it won't scale to other kinds of quoted terms, it doesn't translate to other languages, and it's pretty pointless, too.

51.3.4. Use of Quotes

Use quotes always to delimit file names, user-supplied identifiers, and other variables that might contain words. Do not use them to mark up variables that will not contain words (for example, operator names).

There are functions in the backend that will double-quote their own output at need (for example, `format_type_be()`). Do not put additional quotes around the output of such functions.

Rationale: Objects can have names that create ambiguity when embedded in a message. Be consistent about denoting where a plugged-in name starts and ends. But don't clutter messages with unnecessary or duplicate quote marks.

51.3.5. Grammar and Punctuation

The rules are different for primary error messages and for detail/hint messages:

Primary error messages: Do not capitalize the first letter. Do not end a message with a period. Do not even think about ending a message with an exclamation point.

Detail and hint messages: Use complete sentences, and end each with a period. Capitalize the first word of sentences. Put two spaces after the period if another sentence follows (for English text; might be inappropriate in other languages).

Error context strings: Do not capitalize the first letter and do not end the string with a period. Context strings should normally not be complete sentences.

Rationale: Avoiding punctuation makes it easier for client applications to embed the message into a variety of grammatical contexts. Often, primary messages are not grammatically complete sentences anyway. (And if they're long enough to be more than one sentence, they should be split into primary and detail parts.) However, detail and hint messages are longer and might need to include multiple sentences. For consistency, they should follow complete-sentence style even when there's only one sentence.

51.3.6. Upper Case vs. Lower Case

Use lower case for message wording, including the first letter of a primary error message. Use upper case for SQL commands and key words if they appear in the message.

Rationale: It's easier to make everything look more consistent this way, since some messages are complete sentences and some not.

51.3.7. Avoid Passive Voice

Use the active voice. Use complete sentences when there is an acting subject ("A could not do B"). Use telegram style without subject if the subject would be the program itself; do not use "I" for the program.

Rationale: The program is not human. Don't pretend otherwise.

51.3.8. Present vs. Past Tense

Use past tense if an attempt to do something failed, but could perhaps succeed next time (perhaps after fixing some problem). Use present tense if the failure is certainly permanent.

There is a nontrivial semantic difference between sentences of the form:

```
could not open file "%s": %m
```

and:

```
cannot open file "%s"
```

The first one means that the attempt to open the file failed. The message should give a reason, such as "disk full" or "file doesn't exist". The past tense is appropriate because next time the disk might not be full anymore or the file in question might exist.

The second form indicates that the functionality of opening the named file does not exist at all in the program, or that it's conceptually impossible. The present tense is appropriate because the condition will persist indefinitely.

Rationale: Granted, the average user will not be able to draw great conclusions merely from the tense of the message, but since the language provides us with a grammar we should use it correctly.

51.3.9. Type of the Object

When citing the name of an object, state what kind of object it is.

Rationale: Otherwise no one will know what "foo.bar.baz" refers to.

51.3.10. Brackets

Square brackets are only to be used (1) in command synopses to denote optional arguments, or (2) to denote an array subscript.

Rationale: Anything else does not correspond to widely-known customary usage and will confuse people.

51.3.11. Assembling Error Messages

When a message includes text that is generated elsewhere, embed it in this style:

```
could not open file %s: %m
```

Rationale: It would be difficult to account for all possible error codes to paste this into a single smooth sentence, so some sort of punctuation is needed. Putting the embedded text in parentheses has also been suggested, but it's unnatural if the embedded text is likely to be the most important part of the message, as is often the case.

51.3.12. Reasons for Errors

Messages should always state the reason why an error occurred. For example:

```
BAD:    could not open file %s
BETTER: could not open file %s (I/O failure)
```

If no reason is known you better fix the code.

51.3.13. Function Names

Don't include the name of the reporting routine in the error text. We have other mechanisms for finding that out when needed, and for most users it's not helpful information. If the error text doesn't make as much sense without the function name, reword it.

```
BAD:    pg_atoi: error in "z": cannot parse "z"
BETTER: invalid input syntax for integer: "z"
```

Avoid mentioning called function names, either; instead say what the code was trying to do:

```
BAD:    open() failed: %m
BETTER: could not open file %s: %m
```

If it really seems necessary, mention the system call in the detail message. (In some cases, providing the actual values passed to the system call might be appropriate information for the detail message.)

Rationale: Users don't know what all those functions do.

51.3.14. Tricky Words to Avoid

Unable. "Unable" is nearly the passive voice. Better use "cannot" or "could not", as appropriate.

Bad. Error messages like "bad result" are really hard to interpret intelligently. It's better to write why the result is "bad", e.g., "invalid format".

Illegal. "Illegal" stands for a violation of the law, the rest is "invalid". Better yet, say why it's invalid.

Unknown. Try to avoid “unknown”. Consider “error: unknown response”. If you don’t know what the response is, how do you know it’s erroneous? “Unrecognized” is often a better choice. Also, be sure to include the value being complained of.

```
BAD:    unknown node type
BETTER: unrecognized node type: 42
```

Find vs. Exists. If the program uses a nontrivial algorithm to locate a resource (e.g., a path search) and that algorithm fails, it is fair to say that the program couldn’t “find” the resource. If, on the other hand, the expected location of the resource is known but the program cannot access it there then say that the resource doesn’t “exist”. Using “find” in this case sounds weak and confuses the issue.

May vs. Can vs. Might. “May” suggests permission (e.g., "You may borrow my rake."), and has little use in documentation or error messages. “Can” suggests ability (e.g., "I can lift that log."), and “might” suggests possibility (e.g., "It might rain today."). Using the proper word clarifies meaning and assists translation.

Contractions. Avoid contractions, like “can’t”; use “cannot” instead.

51.3.15. Proper Spelling

Spell out words in full. For instance, avoid:

- spec
- stats
- parens
- auth
- xact

Rationale: This will improve consistency.

51.3.16. Localization

Keep in mind that error message texts need to be translated into other languages. Follow the guidelines in Section 52.2.2 to avoid making life difficult for translators.

Chapter 52. Native Language Support

52.1. For the Translator

PostgreSQL programs (server and client) can issue their messages in your favorite language — if the messages have been translated. Creating and maintaining translated message sets needs the help of people who speak their own language well and want to contribute to the PostgreSQL effort. You do not have to be a programmer at all to do this. This section explains how to help.

52.1.1. Requirements

We won't judge your language skills — this section is about software tools. Theoretically, you only need a text editor. But this is only in the unlikely event that you do not want to try out your translated messages. When you configure your source tree, be sure to use the `--enable-nls` option. This will also check for the libintl library and the `msgfmt` program, which all end users will need anyway. To try out your work, follow the applicable portions of the installation instructions.

If you want to start a new translation effort or want to do a message catalog merge (described later), you will need the programs `xgettext` and `msgmerge`, respectively, in a GNU-compatible implementation. Later, we will try to arrange it so that if you use a packaged source distribution, you won't need `xgettext`. (If working from Git, you will still need it.) GNU Gettext 0.10.36 or later is currently recommended.

Your local gettext implementation should come with its own documentation. Some of that is probably duplicated in what follows, but for additional details you should look there.

52.1.2. Concepts

The pairs of original (English) messages and their (possibly) translated equivalents are kept in *message catalogs*, one for each program (although related programs can share a message catalog) and for each target language. There are two file formats for message catalogs: The first is the "PO" file (for Portable Object), which is a plain text file with special syntax that translators edit. The second is the "MO" file (for Machine Object), which is a binary file generated from the respective PO file and is used while the internationalized program is run. Translators do not deal with MO files; in fact hardly anyone does.

The extension of the message catalog file is to no surprise either `.po` or `.mo`. The base name is either the name of the program it accompanies, or the language the file is for, depending on the situation. This is a bit confusing. Examples are `psql.po` (PO file for psql) or `fr.mo` (MO file in French).

The file format of the PO files is illustrated here:

```
# comment

msgid "original string"
msgstr "translated string"
```

```
msgid "more original"
msgstr "another translated"
"string can be broken up like this"

...
```

The msgid's are extracted from the program source. (They need not be, but this is the most common way.) The msgstr lines are initially empty and are filled in with useful strings by the translator. The strings can contain C-style escape characters and can be continued across lines as illustrated. (The next line must start at the beginning of the line.)

The # character introduces a comment. If whitespace immediately follows the # character, then this is a comment maintained by the translator. There can also be automatic comments, which have a non-whitespace character immediately following the #. These are maintained by the various tools that operate on the PO files and are intended to aid the translator.

```
#. automatic comment
#: filename.c:1023
#, flags, flags
```

The #. style comments are extracted from the source file where the message is used. Possibly the programmer has inserted information for the translator, such as about expected alignment. The #: comment indicates the exact location(s) where the message is used in the source. The translator need not look at the program source, but he can if there is doubt about the correct translation. The #, comments contain flags that describe the message in some way. There are currently two flags: `fuzzy` is set if the message has possibly been outdated because of changes in the program source. The translator can then verify this and possibly remove the fuzzy flag. Note that fuzzy messages are not made available to the end user. The other flag is `c-format`, which indicates that the message is a `printf`-style format template. This means that the translation should also be a format string with the same number and type of placeholders. There are tools that can verify this, which key off the c-format flag.

52.1.3. Creating and Maintaining Message Catalogs

OK, so how does one create a "blank" message catalog? First, go into the directory that contains the program whose messages you want to translate. If there is a file `nls.mk`, then this program has been prepared for translation.

If there are already some `.po` files, then someone has already done some translation work. The files are named *`language`*`.po`, where *`language`* is the ISO 639-1 two-letter language code (in lower case)[1], e.g., `fr.po` for French. If there is really a need for more than one translation effort per language then the files can also be named *`language_region`*`.po` where *`region`* is the ISO 3166-1 two-letter country code (in upper case)[2], e.g., `pt_BR.po` for Portuguese in Brazil. If you find the language you wanted you can just start working on that file.

If you need to start a new translation effort, then first run the command:

```
make init-po
```

1. https://www.loc.gov/standards/iso639-2/php/English_list.php
2. http://www.iso.org/iso/country_names_and_code_elements

This will create a file *progname*.pot. (.pot to distinguish it from PO files that are "in production". The T stands for "template".) Copy this file to *language*.po and edit it. To make it known that the new language is available, also edit the file nls.mk and add the language (or language and country) code to the line that looks like:

```
AVAIL_LANGUAGES := de fr
```

(Other languages can appear, of course.)

As the underlying program or library changes, messages might be changed or added by the programmers. In this case you do not need to start from scratch. Instead, run the command:

```
make update-po
```

which will create a new blank message catalog file (the pot file you started with) and will merge it with the existing PO files. If the merge algorithm is not sure about a particular message it marks it "fuzzy" as explained above. The new PO file is saved with a .po.new extension.

52.1.4. Editing the PO Files

The PO files can be edited with a regular text editor. The translator should only change the area between the quotes after the msgstr directive, add comments, and alter the fuzzy flag. There is (unsurprisingly) a PO mode for Emacs, which I find quite useful.

The PO files need not be completely filled in. The software will automatically fall back to the original string if no translation (or an empty translation) is available. It is no problem to submit incomplete translations for inclusions in the source tree; that gives room for other people to pick up your work. However, you are encouraged to give priority to removing fuzzy entries after doing a merge. Remember that fuzzy entries will not be installed; they only serve as reference for what might be the right translation.

Here are some things to keep in mind while editing the translations:

- Make sure that if the original ends with a newline, the translation does, too. Similarly for tabs, etc.
- If the original is a printf format string, the translation also needs to be. The translation also needs to have the same format specifiers in the same order. Sometimes the natural rules of the language make this impossible or at least awkward. In that case you can modify the format specifiers like this:

  ```
  msgstr "Die Datei %2$s hat %1$u Zeichen."
  ```

  Then the first placeholder will actually use the second argument from the list. The *digits*$ needs to follow the % immediately, before any other format manipulators. (This feature really exists in the printf family of functions. You might not have heard of it before because there is little use for it outside of message internationalization.)
- If the original string contains a linguistic mistake, report that (or fix it yourself in the program source) and translate normally. The corrected string can be merged in when the program sources have been updated. If the original string contains a factual mistake, report that (or fix it yourself) and do not translate it. Instead, you can mark the string with a comment in the PO file.
- Maintain the style and tone of the original string. Specifically, messages that are not sentences (cannot open file %s) should probably not start with a capital letter (if your language distinguishes letter

case) or end with a period (if your language uses punctuation marks). It might help to read Section 51.3.

- If you don't know what a message means, or if it is ambiguous, ask on the developers' mailing list. Chances are that English speaking end users might also not understand it or find it ambiguous, so it's best to improve the message.

52.2. For the Programmer

52.2.1. Mechanics

This section describes how to implement native language support in a program or library that is part of the PostgreSQL distribution. Currently, it only applies to C programs.

Adding NLS Support to a Program

1. Insert this code into the start-up sequence of the program:

```
#ifdef ENABLE_NLS
#include <locale.h>
#endif

...

#ifdef ENABLE_NLS
setlocale(LC_ALL, "");
bindtextdomain("progname", LOCALEDIR);
textdomain("progname");
#endif
```

   (The *progname* can actually be chosen freely.)

2. Wherever a message that is a candidate for translation is found, a call to `gettext()` needs to be inserted. E.g.:

```
fprintf(stderr, "panic level %d\n", lvl);
```

   would be changed to:

```
fprintf(stderr, gettext("panic level %d\n"), lvl);
```

   (`gettext` is defined as a no-op if NLS support is not configured.)

   This tends to add a lot of clutter. One common shortcut is to use:

```
#define _(x) gettext(x)
```

   Another solution is feasible if the program does much of its communication through one or a few functions, such as `ereport()` in the backend. Then you make this function call `gettext` internally on all input strings.

3. Add a file `nls.mk` in the directory with the program sources. This file will be read as a makefile. The following variable assignments need to be made here:

`CATALOG_NAME`

The program name, as provided in the `textdomain()` call.

`AVAIL_LANGUAGES`

List of provided translations — initially empty.

`GETTEXT_FILES`

List of files that contain translatable strings, i.e., those marked with `gettext` or an alternative solution. Eventually, this will include nearly all source files of the program. If this list gets too long you can make the first "file" be a `+` and the second word be a file that contains one file name per line.

`GETTEXT_TRIGGERS`

The tools that generate message catalogs for the translators to work on need to know what function calls contain translatable strings. By default, only `gettext()` calls are known. If you used _ or other identifiers you need to list them here. If the translatable string is not the first argument, the item needs to be of the form `func:2` (for the second argument). If you have a function that supports pluralized messages, the item should look like `func:1,2` (identifying the singular and plural message arguments).

The build system will automatically take care of building and installing the message catalogs.

52.2.2. Message-writing Guidelines

Here are some guidelines for writing messages that are easily translatable.

- Do not construct sentences at run-time, like:

  ```
  printf("Files were %s.\n", flag ? "copied" : "removed");
  ```

  The word order within the sentence might be different in other languages. Also, even if you remember to call `gettext()` on each fragment, the fragments might not translate well separately. It's better to duplicate a little code so that each message to be translated is a coherent whole. Only numbers, file names, and such-like run-time variables should be inserted at run time into a message text.

- For similar reasons, this won't work:

  ```
  printf("copied %d file%s", n, n!=1 ? "s" : "");
  ```

  because it assumes how the plural is formed. If you figured you could solve it like this:

  ```
  if (n==1)
      printf("copied 1 file");
  else
      printf("copied %d files", n):
  ```

  then be disappointed. Some languages have more than two forms, with some peculiar rules. It's often best to design the message to avoid the issue altogether, for instance like this:

  ```
  printf("number of copied files: %d", n);
  ```

If you really want to construct a properly pluralized message, there is support for this, but it's a bit awkward. When generating a primary or detail error message in `ereport()`, you can write something like this:

```
errmsg_plural("copied %d file",
              "copied %d files",
              n,
              n)
```

The first argument is the format string appropriate for English singular form, the second is the format string appropriate for English plural form, and the third is the integer control value that determines which plural form to use. Subsequent arguments are formatted per the format string as usual. (Normally, the pluralization control value will also be one of the values to be formatted, so it has to be written twice.) In English it only matters whether *n* is 1 or not 1, but in other languages there can be many different plural forms. The translator sees the two English forms as a group and has the opportunity to supply multiple substitute strings, with the appropriate one being selected based on the run-time value of *n*.

If you need to pluralize a message that isn't going directly to an `errmsg` or `errdetail` report, you have to use the underlying function `ngettext`. See the gettext documentation.

- If you want to communicate something to the translator, such as about how a message is intended to line up with other output, precede the occurrence of the string with a comment that starts with `translator`, e.g.:

  ```
  /* translator: This message is not what it seems to be. */
  ```

  These comments are copied to the message catalog files so that the translators can see them.

Chapter 53. Writing A Procedural Language Handler

All calls to functions that are written in a language other than the current "version 1" interface for compiled languages (this includes functions in user-defined procedural languages, functions written in SQL, and functions using the version 0 compiled language interface) go through a *call handler* function for the specific language. It is the responsibility of the call handler to execute the function in a meaningful way, such as by interpreting the supplied source text. This chapter outlines how a new procedural language's call handler can be written.

The call handler for a procedural language is a "normal" function that must be written in a compiled language such as C, using the version-1 interface, and registered with PostgreSQL as taking no arguments and returning the type `language_handler`. This special pseudotype identifies the function as a call handler and prevents it from being called directly in SQL commands. For more details on C language calling conventions and dynamic loading, see Section 35.9.

The call handler is called in the same way as any other function: It receives a pointer to a `FunctionCallInfoData struct` containing argument values and information about the called function, and it is expected to return a `Datum` result (and possibly set the `isnull` field of the `FunctionCallInfoData` structure, if it wishes to return an SQL null result). The difference between a call handler and an ordinary callee function is that the `flinfo->fn_oid` field of the `FunctionCallInfoData` structure will contain the OID of the actual function to be called, not of the call handler itself. The call handler must use this field to determine which function to execute. Also, the passed argument list has been set up according to the declaration of the target function, not of the call handler.

It's up to the call handler to fetch the entry of the function from the `pg_proc` system catalog and to analyze the argument and return types of the called function. The `AS` clause from the `CREATE FUNCTION` command for the function will be found in the `prosrc` column of the `pg_proc` row. This is commonly source text in the procedural language, but in theory it could be something else, such as a path name to a file, or anything else that tells the call handler what to do in detail.

Often, the same function is called many times per SQL statement. A call handler can avoid repeated lookups of information about the called function by using the `flinfo->fn_extra` field. This will initially be `NULL`, but can be set by the call handler to point at information about the called function. On subsequent calls, if `flinfo->fn_extra` is already non-`NULL` then it can be used and the information lookup step skipped. The call handler must make sure that `flinfo->fn_extra` is made to point at memory that will live at least until the end of the current query, since an `FmgrInfo` data structure could be kept that long. One way to do this is to allocate the extra data in the memory context specified by `flinfo->fn_mcxt`; such data will normally have the same lifespan as the `FmgrInfo` itself. But the handler could also choose to use a longer-lived memory context so that it can cache function definition information across queries.

When a procedural-language function is invoked as a trigger, no arguments are passed in the usual way, but the `FunctionCallInfoData`'s `context` field points at a `TriggerData` structure, rather than being `NULL` as it is in a plain function call. A language handler should provide mechanisms for procedural-

language functions to get at the trigger information.

This is a template for a procedural-language handler written in C:

```
#include "postgres.h"
#include "executor/spi.h"
#include "commands/trigger.h"
#include "fmgr.h"
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

PG_FUNCTION_INFO_V1(plsample_call_handler);

Datum
plsample_call_handler(PG_FUNCTION_ARGS)
{
    Datum          retval;

    if (CALLED_AS_TRIGGER(fcinfo))
    {
        /*
         * Called as a trigger procedure
         */
        TriggerData    *trigdata = (TriggerData *) fcinfo->context;

        retval = ...
    }
    else
    {
        /*
         * Called as a function
         */

        retval = ...
    }

    return retval;
}
```

Only a few thousand lines of code have to be added instead of the dots to complete the call handler.

After having compiled the handler function into a loadable module (see Section 35.9.6), the following commands then register the sample procedural language:

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS 'filename'
    LANGUAGE C;
CREATE LANGUAGE plsample
```

```
    HANDLER plsample_call_handler;
```

Although providing a call handler is sufficient to create a minimal procedural language, there are two other functions that can optionally be provided to make the language more convenient to use. These are a *validator* and an *inline handler*. A validator can be provided to allow language-specific checking to be done during CREATE FUNCTION. An inline handler can be provided to allow the language to support anonymous code blocks executed via the DO command.

If a validator is provided by a procedural language, it must be declared as a function taking a single parameter of type `oid`. The validator's result is ignored, so it is customarily declared to return `void`. The validator will be called at the end of a `CREATE FUNCTION` command that has created or updated a function written in the procedural language. The passed-in OID is the OID of the function's `pg_proc` row. The validator must fetch this row in the usual way, and do whatever checking is appropriate. First, call `CheckFunctionValidatorAccess()` to diagnose explicit calls to the validator that the user could not achieve through `CREATE FUNCTION`. Typical checks then include verifying that the function's argument and result types are supported by the language, and that the function's body is syntactically correct in the language. If the validator finds the function to be okay, it should just return. If it finds an error, it should report that via the normal `ereport()` error reporting mechanism. Throwing an error will force a transaction rollback and thus prevent the incorrect function definition from being committed.

Validator functions should typically honor the check_function_bodies parameter: if it is turned off then any expensive or context-sensitive checking should be skipped. If the language provides for code execution at compilation time, the validator must suppress checks that would induce such execution. In particular, this parameter is turned off by pg_dump so that it can load procedural language functions without worrying about side effects or dependencies of the function bodies on other database objects. (Because of this requirement, the call handler should avoid assuming that the validator has fully checked the function. The point of having a validator is not to let the call handler omit checks, but to notify the user immediately if there are obvious errors in a `CREATE FUNCTION` command.) While the choice of exactly what to check is mostly left to the discretion of the validator function, note that the core `CREATE FUNCTION` code only executes `SET` clauses attached to a function when `check_function_bodies` is on. Therefore, checks whose results might be affected by GUC parameters definitely should be skipped when `check_function_bodies` is off, to avoid false failures when reloading a dump.

If an inline handler is provided by a procedural language, it must be declared as a function taking a single parameter of type `internal`. The inline handler's result is ignored, so it is customarily declared to return `void`. The inline handler will be called when a `DO` statement is executed specifying the procedural language. The parameter actually passed is a pointer to an `InlineCodeBlock` struct, which contains information about the `DO` statement's parameters, in particular the text of the anonymous code block to be executed. The inline handler should execute this code and return.

It's recommended that you wrap all these function declarations, as well as the `CREATE LANGUAGE` command itself, into an *extension* so that a simple `CREATE EXTENSION` command is sufficient to install the language. See Section 35.15 for information about writing extensions.

The procedural languages included in the standard distribution are good references when trying to write your own language handler. Look into the `src/pl` subdirectory of the source tree. The CREATE LANGUAGE reference page also has some useful details.

Chapter 54. Writing A Foreign Data Wrapper

All operations on a foreign table are handled through its foreign data wrapper, which consists of a set of functions that the core server calls. The foreign data wrapper is responsible for fetching data from the remote data source and returning it to the PostgreSQL executor. If updating foreign tables is to be supported, the wrapper must handle that, too. This chapter outlines how to write a new foreign data wrapper.

The foreign data wrappers included in the standard distribution are good references when trying to write your own. Look into the `contrib` subdirectory of the source tree. The CREATE FOREIGN DATA WRAPPER reference page also has some useful details.

> **Note:** The SQL standard specifies an interface for writing foreign data wrappers. However, PostgreSQL does not implement that API, because the effort to accommodate it into PostgreSQL would be large, and the standard API hasn't gained wide adoption anyway.

54.1. Foreign Data Wrapper Functions

The FDW author needs to implement a handler function, and optionally a validator function. Both functions must be written in a compiled language such as C, using the version-1 interface. For details on C language calling conventions and dynamic loading, see Section 35.9.

The handler function simply returns a struct of function pointers to callback functions that will be called by the planner, executor, and various maintenance commands. Most of the effort in writing an FDW is in implementing these callback functions. The handler function must be registered with PostgreSQL as taking no arguments and returning the special pseudo-type `fdw_handler`. The callback functions are plain C functions and are not visible or callable at the SQL level. The callback functions are described in Section 54.2.

The validator function is responsible for validating options given in `CREATE` and `ALTER` commands for its foreign data wrapper, as well as foreign servers, user mappings, and foreign tables using the wrapper. The validator function must be registered as taking two arguments, a text array containing the options to be validated, and an OID representing the type of object the options are associated with (in the form of the OID of the system catalog the object would be stored in, either `ForeignDataWrapperRelationId`, `ForeignServerRelationId`, `UserMappingRelationId`, or `ForeignTableRelationId`). If no validator function is supplied, options are not checked at object creation time or object alteration time.

54.2. Foreign Data Wrapper Callback Routines

The FDW handler function returns a palloc'd `FdwRoutine` struct containing pointers to the callback functions described below. The scan-related functions are required, the rest are optional.

The FdwRoutine struct type is declared in src/include/foreign/fdwapi.h, which see for additional details.

54.2.1. FDW Routines For Scanning Foreign Tables

```
void
GetForeignRelSize (PlannerInfo *root,
                   RelOptInfo *baserel,
                   Oid foreigntableid);
```

Obtain relation size estimates for a foreign table. This is called at the beginning of planning for a query that scans a foreign table. root is the planner's global information about the query; baserel is the planner's information about this table; and foreigntableid is the pg_class OID of the foreign table. (foreigntableid could be obtained from the planner data structures, but it's passed explicitly to save effort.)

This function should update baserel->rows to be the expected number of rows returned by the table scan, after accounting for the filtering done by the restriction quals. The initial value of baserel->rows is just a constant default estimate, which should be replaced if at all possible. The function may also choose to update baserel->width if it can compute a better estimate of the average result row width. (The initial value is based on column data types and on column average-width values measured by the last ANALYZE.) Also, this function may update baserel->tuples if it can compute a better estimate of the foreign table's total row count. (The initial value is from pg_class.reltuples which represents the total row count seen by the last ANALYZE.)

See Section 54.4 for additional information.

```
void
GetForeignPaths (PlannerInfo *root,
                 RelOptInfo *baserel,
                 Oid foreigntableid);
```

Create possible access paths for a scan on a foreign table. This is called during query planning. The parameters are the same as for GetForeignRelSize, which has already been called.

This function must generate at least one access path (ForeignPath node) for a scan on the foreign table and must call add_path to add each such path to baserel->pathlist. It's recommended to use create_foreignscan_path to build the ForeignPath nodes. The function can generate multiple access paths, e.g., a path which has valid pathkeys to represent a pre-sorted result. Each access path must contain cost estimates, and can contain any FDW-private information that is needed to identify the specific scan method intended.

See Section 54.4 for additional information.

```
ForeignScan *
GetForeignPlan (PlannerInfo *root,
                RelOptInfo *baserel,
                Oid foreigntableid,
                ForeignPath *best_path,
                List *tlist,
```

```
                List *scan_clauses,
                Plan *outer_plan);
```

Create a `ForeignScan` plan node from the selected foreign access path. This is called at the end of query planning. The parameters are as for `GetForeignRelSize`, plus the selected `ForeignPath` (previously produced by `GetForeignPaths` or `GetForeignJoinPaths`), the target list to be emitted by the plan node, the restriction clauses to be enforced by the plan node, and the outer subplan of the `ForeignScan`, which is used for rechecks performed by `RecheckForeignScan`. (If the path is for a join rather than a base relation, `foreigntableid` is `InvalidOid`.)

This function must create and return a `ForeignScan` plan node; it's recommended to use `make_foreignscan` to build the `ForeignScan` node.

See Section 54.4 for additional information.

```
void
BeginForeignScan (ForeignScanState *node,
                  int eflags);
```

Begin executing a foreign scan. This is called during executor startup. It should perform any initialization needed before the scan can start, but not start executing the actual scan (that should be done upon the first call to `IterateForeignScan`). The `ForeignScanState` node has already been created, but its `fdw_state` field is still NULL. Information about the table to scan is accessible through the `ForeignScanState` node (in particular, from the underlying `ForeignScan` plan node, which contains any FDW-private information provided by `GetForeignPlan`). `eflags` contains flag bits describing the executor's operating mode for this plan node.

Note that when `(eflags & EXEC_FLAG_EXPLAIN_ONLY)` is true, this function should not perform any externally-visible actions; it should only do the minimum required to make the node state valid for `ExplainForeignScan` and `EndForeignScan`.

```
TupleTableSlot *
IterateForeignScan (ForeignScanState *node);
```

Fetch one row from the foreign source, returning it in a tuple table slot (the node's `ScanTupleSlot` should be used for this purpose). Return NULL if no more rows are available. The tuple table slot infrastructure allows either a physical or virtual tuple to be returned; in most cases the latter choice is preferable from a performance standpoint. Note that this is called in a short-lived memory context that will be reset between invocations. Create a memory context in `BeginForeignScan` if you need longer-lived storage, or use the `es_query_cxt` of the node's `EState`.

The rows returned must match the `fdw_scan_tlist` target list if one was supplied, otherwise they must match the row type of the foreign table being scanned. If you choose to optimize away fetching columns that are not needed, you should insert nulls in those column positions, or else generate a `fdw_scan_tlist` list with those columns omitted.

Note that PostgreSQL's executor doesn't care whether the rows returned violate any constraints that were defined on the foreign table — but the planner does care, and may optimize queries incorrectly if there are rows visible in the foreign table that do not satisfy a declared constraint. If a constraint is violated when the user has declared that the constraint should hold true, it may be appropriate to raise an error (just as you would need to do in the case of a data type mismatch).

```
void
ReScanForeignScan (ForeignScanState *node);
```

Restart the scan from the beginning. Note that any parameters the scan depends on may have changed value, so the new scan does not necessarily return exactly the same rows.

```
void
EndForeignScan (ForeignScanState *node);
```

End the scan and release resources. It is normally not important to release palloc'd memory, but for example open files and connections to remote servers should be cleaned up.

54.2.2. FDW Routines For Scanning Foreign Joins

If an FDW supports performing foreign joins remotely (rather than by fetching both tables' data and doing the join locally), it should provide this callback function:

```
void
GetForeignJoinPaths (PlannerInfo *root,
                     RelOptInfo *joinrel,
                     RelOptInfo *outerrel,
                     RelOptInfo *innerrel,
                     JoinType jointype,
                     JoinPathExtraData *extra);
```

Create possible access paths for a join of two (or more) foreign tables that all belong to the same foreign server. This optional function is called during query planning. As with `GetForeignPaths`, this function should generate `ForeignPath` path(s) for the supplied `joinrel`, and call `add_path` to add these paths to the set of paths considered for the join. But unlike `GetForeignPaths`, it is not necessary that this function succeed in creating at least one path, since paths involving local joining are always possible.

Note that this function will be invoked repeatedly for the same join relation, with different combinations of inner and outer relations; it is the responsibility of the FDW to minimize duplicated work.

If a `ForeignPath` path is chosen for the join, it will represent the entire join process; paths generated for the component tables and subsidiary joins will not be used. Subsequent processing of the join path proceeds much as it does for a path scanning a single foreign table. One difference is that the `scanrelid` of the resulting `ForeignScan` plan node should be set to zero, since there is no single relation that it represents; instead, the `fs_relids` field of the `ForeignScan` node represents the set of relations that were joined. (The latter field is set up automatically by the core planner code, and need not be filled by the FDW.) Another difference is that, because the column list for a remote join cannot be found from the system catalogs, the FDW must fill `fdw_scan_tlist` with an appropriate list of `TargetEntry` nodes, representing the set of columns it will supply at run time in the tuples it returns.

See Section 54.4 for additional information.

54.2.3. FDW Routines For Updating Foreign Tables

If an FDW supports writable foreign tables, it should provide some or all of the following callback functions depending on the needs and capabilities of the FDW:

```
void
AddForeignUpdateTargets (Query *parsetree,
                         RangeTblEntry *target_rte,
                         Relation target_relation);
```

`UPDATE` and `DELETE` operations are performed against rows previously fetched by the table-scanning functions. The FDW may need extra information, such as a row ID or the values of primary-key columns, to ensure that it can identify the exact row to update or delete. To support that, this function can add extra hidden, or "junk", target columns to the list of columns that are to be retrieved from the foreign table during an `UPDATE` or `DELETE`.

To do that, add `TargetEntry` items to `parsetree->targetList`, containing expressions for the extra values to be fetched. Each such entry must be marked `resjunk` = `true`, and must have a distinct `resname` that will identify it at execution time. Avoid using names matching `ctid`*N*, `wholerow`, or `wholerow`*N*, as the core system can generate junk columns of these names. If the extra expressions are more complex than simple Vars, they must be run through `eval_const_expressions` before adding them to the targetlist.

Although this function is called during planning, the information provided is a bit different from that available to other planning routines. `parsetree` is the parse tree for the `UPDATE` or `DELETE` command, while `target_rte` and `target_relation` describe the target foreign table.

If the `AddForeignUpdateTargets` pointer is set to `NULL`, no extra target expressions are added. (This will make it impossible to implement `DELETE` operations, though `UPDATE` may still be feasible if the FDW relies on an unchanging primary key to identify rows.)

```
List *
PlanForeignModify (PlannerInfo *root,
                   ModifyTable *plan,
                   Index resultRelation,
                   int subplan_index);
```

Perform any additional planning actions needed for an insert, update, or delete on a foreign table. This function generates the FDW-private information that will be attached to the `ModifyTable` plan node that performs the update action. This private information must have the form of a `List`, and will be delivered to `BeginForeignModify` during the execution stage.

`root` is the planner's global information about the query. `plan` is the `ModifyTable` plan node, which is complete except for the `fdwPrivLists` field. `resultRelation` identifies the target foreign table by its range table index. `subplan_index` identifies which target of the `ModifyTable` plan node this is, counting from zero; use this if you want to index into `plan->plans` or other substructure of the `plan` node.

See Section 54.4 for additional information.

If the `PlanForeignModify` pointer is set to `NULL`, no additional plan-time actions are taken, and the `fdw_private` list delivered to `BeginForeignModify` will be NIL.

```
void
```

```
BeginForeignModify (ModifyTableState *mtstate,
                    ResultRelInfo *rinfo,
                    List *fdw_private,
                    int subplan_index,
                    int eflags);
```

Begin executing a foreign table modification operation. This routine is called during executor startup. It should perform any initialization needed prior to the actual table modifications. Subsequently, `ExecForeignInsert`, `ExecForeignUpdate` or `ExecForeignDelete` will be called for each tuple to be inserted, updated, or deleted.

`mtstate` is the overall state of the `ModifyTable` plan node being executed; global data about the plan and execution state is available via this structure. `rinfo` is the `ResultRelInfo` struct describing the target foreign table. (The `ri_FdwState` field of `ResultRelInfo` is available for the FDW to store any private state it needs for this operation.) `fdw_private` contains the private data generated by `PlanForeignModify`, if any. `subplan_index` identifies which target of the `ModifyTable` plan node this is. `eflags` contains flag bits describing the executor's operating mode for this plan node.

Note that when `(eflags & EXEC_FLAG_EXPLAIN_ONLY)` is true, this function should not perform any externally-visible actions; it should only do the minimum required to make the node state valid for `ExplainForeignModify` and `EndForeignModify`.

If the `BeginForeignModify` pointer is set to `NULL`, no action is taken during executor startup.

```
TupleTableSlot *
ExecForeignInsert (EState *estate,
                   ResultRelInfo *rinfo,
                   TupleTableSlot *slot,
                   TupleTableSlot *planSlot);
```

Insert one tuple into the foreign table. `estate` is global execution state for the query. `rinfo` is the `ResultRelInfo` struct describing the target foreign table. `slot` contains the tuple to be inserted; it will match the row-type definition of the foreign table. `planSlot` contains the tuple that was generated by the `ModifyTable` plan node's subplan; it differs from `slot` in possibly containing additional "junk" columns. (The `planSlot` is typically of little interest for `INSERT` cases, but is provided for completeness.)

The return value is either a slot containing the data that was actually inserted (this might differ from the data supplied, for example as a result of trigger actions), or NULL if no row was actually inserted (again, typically as a result of triggers). The passed-in `slot` can be re-used for this purpose.

The data in the returned slot is used only if the `INSERT` query has a `RETURNING` clause or the foreign table has an `AFTER ROW` trigger. Triggers require all columns, but the FDW could choose to optimize away returning some or all columns depending on the contents of the `RETURNING` clause. Regardless, some slot must be returned to indicate success, or the query's reported row count will be wrong.

If the `ExecForeignInsert` pointer is set to `NULL`, attempts to insert into the foreign table will fail with an error message.

```
TupleTableSlot *
ExecForeignUpdate (EState *estate,
                   ResultRelInfo *rinfo,
                   TupleTableSlot *slot,
                   TupleTableSlot *planSlot);
```

Update one tuple in the foreign table. estate is global execution state for the query. rinfo is the ResultRelInfo struct describing the target foreign table. slot contains the new data for the tuple; it will match the row-type definition of the foreign table. planSlot contains the tuple that was generated by the ModifyTable plan node's subplan; it differs from slot in possibly containing additional "junk" columns. In particular, any junk columns that were requested by AddForeignUpdateTargets will be available from this slot.

The return value is either a slot containing the row as it was actually updated (this might differ from the data supplied, for example as a result of trigger actions), or NULL if no row was actually updated (again, typically as a result of triggers). The passed-in slot can be re-used for this purpose.

The data in the returned slot is used only if the UPDATE query has a RETURNING clause or the foreign table has an AFTER ROW trigger. Triggers require all columns, but the FDW could choose to optimize away returning some or all columns depending on the contents of the RETURNING clause. Regardless, some slot must be returned to indicate success, or the query's reported row count will be wrong.

If the ExecForeignUpdate pointer is set to NULL, attempts to update the foreign table will fail with an error message.

```
TupleTableSlot *
ExecForeignDelete (EState *estate,
                   ResultRelInfo *rinfo,
                   TupleTableSlot *slot,
                   TupleTableSlot *planSlot);
```

Delete one tuple from the foreign table. estate is global execution state for the query. rinfo is the ResultRelInfo struct describing the target foreign table. slot contains nothing useful upon call, but can be used to hold the returned tuple. planSlot contains the tuple that was generated by the ModifyTable plan node's subplan; in particular, it will carry any junk columns that were requested by AddForeignUpdateTargets. The junk column(s) must be used to identify the tuple to be deleted.

The return value is either a slot containing the row that was deleted, or NULL if no row was deleted (typically as a result of triggers). The passed-in slot can be used to hold the tuple to be returned.

The data in the returned slot is used only if the DELETE query has a RETURNING clause or the foreign table has an AFTER ROW trigger. Triggers require all columns, but the FDW could choose to optimize away returning some or all columns depending on the contents of the RETURNING clause. Regardless, some slot must be returned to indicate success, or the query's reported row count will be wrong.

If the ExecForeignDelete pointer is set to NULL, attempts to delete from the foreign table will fail with an error message.

```
void
EndForeignModify (EState *estate,
                  ResultRelInfo *rinfo);
```

End the table update and release resources. It is normally not important to release palloc'd memory, but for example open files and connections to remote servers should be cleaned up.

If the EndForeignModify pointer is set to NULL, no action is taken during executor shutdown.

```
int
IsForeignRelUpdatable (Relation rel);
```

Report which update operations the specified foreign table supports. The return value should be a bit mask of rule event numbers indicating which operations are supported by the foreign table, using the `CmdType` enumeration; that is, `(1 << CMD_UPDATE) = 4` for `UPDATE`, `(1 << CMD_INSERT) = 8` for `INSERT`, and `(1 << CMD_DELETE) = 16` for `DELETE`.

If the `IsForeignRelUpdatable` pointer is set to `NULL`, foreign tables are assumed to be insertable, updatable, or deletable if the FDW provides `ExecForeignInsert`, `ExecForeignUpdate`, or `ExecForeignDelete` respectively. This function is only needed if the FDW supports some tables that are updatable and some that are not. (Even then, it's permissible to throw an error in the execution routine instead of checking in this function. However, this function is used to determine updatability for display in the `information_schema` views.)

54.2.4. FDW Routines For Row Locking

If an FDW wishes to support *late row locking* (as described in Section 54.5), it must provide the following callback functions:

```
RowMarkType
GetForeignRowMarkType (RangeTblEntry *rte,
                       LockClauseStrength strength);
```

Report which row-marking option to use for a foreign table. `rte` is the `RangeTblEntry` node for the table and `strength` describes the lock strength requested by the relevant `FOR UPDATE/SHARE` clause, if any. The result must be a member of the `RowMarkType` enum type.

This function is called during query planning for each foreign table that appears in an `UPDATE`, `DELETE`, or `SELECT FOR UPDATE/SHARE` query and is not the target of `UPDATE` or `DELETE`.

If the `GetForeignRowMarkType` pointer is set to `NULL`, the `ROW_MARK_COPY` option is always used. (This implies that `RefetchForeignRow` will never be called, so it need not be provided either.)

See Section 54.5 for more information.

```
HeapTuple
RefetchForeignRow (EState *estate,
                   ExecRowMark *erm,
                   Datum rowid,
                   bool *updated);
```

Re-fetch one tuple from the foreign table, after locking it if required. `estate` is global execution state for the query. `erm` is the `ExecRowMark` struct describing the target foreign table and the row lock type (if any) to acquire. `rowid` identifies the tuple to be fetched. `updated` is an output parameter.

This function should return a palloc'ed copy of the fetched tuple, or `NULL` if the row lock couldn't be obtained. The row lock type to acquire is defined by `erm->markType`, which is the value previously returned by `GetForeignRowMarkType`. (`ROW_MARK_REFERENCE` means to just re-fetch the tuple without acquiring any lock, and `ROW_MARK_COPY` will never be seen by this routine.)

In addition, `*updated` should be set to `true` if what was fetched was an updated version of the tuple rather than the same version previously obtained. (If the FDW cannot be sure about this, always returning `true` is recommended.)

Note that by default, failure to acquire a row lock should result in raising an error; a NULL return is only appropriate if the SKIP LOCKED option is specified by erm->waitPolicy.

The rowid is the ctid value previously read for the row to be re-fetched. Although the rowid value is passed as a Datum, it can currently only be a tid. The function API is chosen in hopes that it may be possible to allow other data types for row IDs in future.

If the RefetchForeignRow pointer is set to NULL, attempts to re-fetch rows will fail with an error message.

See Section 54.5 for more information.

```
bool
RecheckForeignScan (ForeignScanState *node, TupleTableSlot *slot);
```

Recheck that a previously-returned tuple still matches the relevant scan and join qualifiers, and possibly provide a modified version of the tuple. For foreign data wrappers which do not perform join pushdown, it will typically be more convenient to set this to NULL and instead set fdw_recheck_quals appropriately. When outer joins are pushed down, however, it isn't sufficient to reapply the checks relevant to all the base tables to the result tuple, even if all needed attributes are present, because failure to match some qualifier might result in some attributes going to NULL, rather than in no tuple being returned. RecheckForeignScan can recheck qualifiers and return true if they are still satisfied and false otherwise, but it can also store a replacement tuple into the supplied slot.

To implement join pushdown, a foreign data wrapper will typically construct an alternative local join plan which is used only for rechecks; this will become the outer subplan of the ForeignScan. When a recheck is required, this subplan can be executed and the resulting tuple can be stored in the slot. This plan need not be efficient since no base table will return more than one row; for example, it may implement all joins as nested loops.

54.2.5. FDW Routines for EXPLAIN

```
void
ExplainForeignScan (ForeignScanState *node,
                    ExplainState *es);
```

Print additional EXPLAIN output for a foreign table scan. This function can call ExplainPropertyText and related functions to add fields to the EXPLAIN output. The flag fields in es can be used to determine what to print, and the state of the ForeignScanState node can be inspected to provide run-time statistics in the EXPLAIN ANALYZE case.

If the ExplainForeignScan pointer is set to NULL, no additional information is printed during EXPLAIN.

```
void
ExplainForeignModify (ModifyTableState *mtstate,
                      ResultRelInfo *rinfo,
                      List *fdw_private,
                      int subplan_index,
                      struct ExplainState *es);
```

Print additional EXPLAIN output for a foreign table update. This function can call ExplainPropertyText and related functions to add fields to the EXPLAIN output. The flag fields in es can be used to determine what to print, and the state of the ModifyTableState node can be inspected to provide run-time statistics in the EXPLAIN ANALYZE case. The first four arguments are the same as for BeginForeignModify.

If the ExplainForeignModify pointer is set to NULL, no additional information is printed during EXPLAIN.

54.2.6. FDW Routines for ANALYZE

```
bool
AnalyzeForeignTable (Relation relation,
                     AcquireSampleRowsFunc *func,
                     BlockNumber *totalpages);
```

This function is called when ANALYZE is executed on a foreign table. If the FDW can collect statistics for this foreign table, it should return true, and provide a pointer to a function that will collect sample rows from the table in func, plus the estimated size of the table in pages in totalpages. Otherwise, return false.

If the FDW does not support collecting statistics for any tables, the AnalyzeForeignTable pointer can be set to NULL.

If provided, the sample collection function must have the signature

```
int
AcquireSampleRowsFunc (Relation relation, int elevel,
                       HeapTuple *rows, int targrows,
                       double *totalrows,
                       double *totaldeadrows);
```

A random sample of up to targrows rows should be collected from the table and stored into the caller-provided rows array. The actual number of rows collected must be returned. In addition, store estimates of the total numbers of live and dead rows in the table into the output parameters totalrows and totaldeadrows. (Set totaldeadrows to zero if the FDW does not have any concept of dead rows.)

54.2.7. FDW Routines For IMPORT FOREIGN SCHEMA

```
List *
ImportForeignSchema (ImportForeignSchemaStmt *stmt, Oid serverOid);
```

Obtain a list of foreign table creation commands. This function is called when executing IMPORT FOREIGN SCHEMA, and is passed the parse tree for that statement, as well as the OID of the foreign server to use. It should return a list of C strings, each of which must contain a CREATE FOREIGN TABLE command. These strings will be parsed and executed by the core server.

Within the ImportForeignSchemaStmt struct, remote_schema is the name of the remote schema from which tables are to be imported. list_type identifies how to filter table names:

FDW_IMPORT_SCHEMA_ALL means that all tables in the remote schema should be imported (in this case table_list is empty), FDW_IMPORT_SCHEMA_LIMIT_TO means to include only tables listed in table_list, and FDW_IMPORT_SCHEMA_EXCEPT means to exclude the tables listed in table_list. options is a list of options used for the import process. The meanings of the options are up to the FDW. For example, an FDW could use an option to define whether the NOT NULL attributes of columns should be imported. These options need not have anything to do with those supported by the FDW as database object options.

The FDW may ignore the local_schema field of the ImportForeignSchemaStmt, because the core server will automatically insert that name into the parsed CREATE FOREIGN TABLE commands.

The FDW does not have to concern itself with implementing the filtering specified by list_type and table_list, either, as the core server will automatically skip any returned commands for tables excluded according to those options. However, it's often useful to avoid the work of creating commands for excluded tables in the first place. The function IsImportableForeignTable() may be useful to test whether a given foreign-table name will pass the filter.

If the FDW does not support importing table definitions, the ImportForeignSchema pointer can be set to NULL.

54.3. Foreign Data Wrapper Helper Functions

Several helper functions are exported from the core server so that authors of foreign data wrappers can get easy access to attributes of FDW-related objects, such as FDW options. To use any of these functions, you need to include the header file foreign/foreign.h in your source file. That header also defines the struct types that are returned by these functions.

```
ForeignDataWrapper *
GetForeignDataWrapper(Oid fdwid);
```

This function returns a ForeignDataWrapper object for the foreign-data wrapper with the given OID. A ForeignDataWrapper object contains properties of the FDW (see foreign/foreign.h for details).

```
ForeignServer *
GetForeignServer(Oid serverid);
```

This function returns a ForeignServer object for the foreign server with the given OID. A ForeignServer object contains properties of the server (see foreign/foreign.h for details).

```
UserMapping *
GetUserMapping(Oid userid, Oid serverid);
```

This function returns a UserMapping object for the user mapping of the given role on the given server. (If there is no mapping for the specific user, it will return the mapping for PUBLIC, or throw error if there is none.) A UserMapping object contains properties of the user mapping (see foreign/foreign.h for details).

```
ForeignTable *
```

```
GetForeignTable(Oid relid);
```

This function returns a `ForeignTable` object for the foreign table with the given OID. A `ForeignTable` object contains properties of the foreign table (see `foreign/foreign.h` for details).

```
List *
GetForeignColumnOptions(Oid relid, AttrNumber attnum);
```

This function returns the per-column FDW options for the column with the given foreign table OID and attribute number, in the form of a list of `DefElem`. NIL is returned if the column has no options.

Some object types have name-based lookup functions in addition to the OID-based ones:

```
ForeignDataWrapper *
GetForeignDataWrapperByName(const char *name, bool missing_ok);
```

This function returns a `ForeignDataWrapper` object for the foreign-data wrapper with the given name. If the wrapper is not found, return NULL if missing_ok is true, otherwise raise an error.

```
ForeignServer *
GetForeignServerByName(const char *name, bool missing_ok);
```

This function returns a `ForeignServer` object for the foreign server with the given name. If the server is not found, return NULL if missing_ok is true, otherwise raise an error.

54.4. Foreign Data Wrapper Query Planning

The FDW callback functions `GetForeignRelSize`, `GetForeignPaths`, `GetForeignPlan`, `PlanForeignModify`, and `GetForeignJoinPaths` must fit into the workings of the PostgreSQL planner. Here are some notes about what they must do.

The information in `root` and `baserel` can be used to reduce the amount of information that has to be fetched from the foreign table (and therefore reduce the cost). `baserel->baserestrictinfo` is particularly interesting, as it contains restriction quals (`WHERE` clauses) that should be used to filter the rows to be fetched. (The FDW itself is not required to enforce these quals, as the core executor can check them instead.) `baserel->reltargetlist` can be used to determine which columns need to be fetched; but note that it only lists columns that have to be emitted by the `ForeignScan` plan node, not columns that are used in qual evaluation but not output by the query.

Various private fields are available for the FDW planning functions to keep information in. Generally, whatever you store in FDW private fields should be palloc'd, so that it will be reclaimed at the end of planning.

`baserel->fdw_private` is a `void` pointer that is available for FDW planning functions to store information relevant to the particular foreign table. The core planner does not touch it except to initialize it to NULL when the `RelOptInfo` node is created. It is useful for passing information forward from `GetForeignRelSize` to `GetForeignPaths` and/or `GetForeignPaths` to `GetForeignPlan`, thereby avoiding recalculation.

`GetForeignPaths` can identify the meaning of different access paths by storing private information in the `fdw_private` field of `ForeignPath` nodes. `fdw_private` is declared as a `List` pointer, but could actually contain anything since the core planner does not touch it. However, best practice is to use a representation that's dumpable by `nodeToString`, for use with debugging support available in the backend.

`GetForeignPlan` can examine the `fdw_private` field of the selected `ForeignPath` node, and can generate `fdw_exprs` and `fdw_private` lists to be placed in the `ForeignScan` plan node, where they will be available at execution time. Both of these lists must be represented in a form that `copyObject` knows how to copy. The `fdw_private` list has no other restrictions and is not interpreted by the core backend in any way. The `fdw_exprs` list, if not NIL, is expected to contain expression trees that are intended to be executed at run time. These trees will undergo post-processing by the planner to make them fully executable.

In `GetForeignPlan`, generally the passed-in target list can be copied into the plan node as-is. The passed `scan_clauses` list contains the same clauses as `baserel->baserestrictinfo`, but may be re-ordered for better execution efficiency. In simple cases the FDW can just strip `RestrictInfo` nodes from the `scan_clauses` list (using `extract_actual_clauses`) and put all the clauses into the plan node's qual list, which means that all the clauses will be checked by the executor at run time. More complex FDWs may be able to check some of the clauses internally, in which case those clauses can be removed from the plan node's qual list so that the executor doesn't waste time rechecking them.

As an example, the FDW might identify some restriction clauses of the form *`foreign_variable`* = *`sub_expression`*, which it determines can be executed on the remote server given the locally-evaluated value of the *`sub_expression`*. The actual identification of such a clause should happen during `GetForeignPaths`, since it would affect the cost estimate for the path. The path's `fdw_private` field would probably include a pointer to the identified clause's `RestrictInfo` node. Then `GetForeignPlan` would remove that clause from `scan_clauses`, but add the *`sub_expression`* to `fdw_exprs` to ensure that it gets massaged into executable form. It would probably also put control information into the plan node's `fdw_private` field to tell the execution functions what to do at run time. The query transmitted to the remote server would involve something like `WHERE` *`foreign_variable`* `= $1`, with the parameter value obtained at run time from evaluation of the `fdw_exprs` expression tree.

Any clauses removed from the plan node's qual list must instead be added to `fdw_recheck_quals` or rechecked by `RecheckForeignScan` in order to ensure correct behavior at the `READ COMMITTED` isolation level. When a concurrent update occurs for some other table involved in the query, the executor may need to verify that all of the original quals are still satisfied for the tuple, possibly against a different set of parameter values. Using `fdw_recheck_quals` is typically easier than implementing checks inside `RecheckForeignScan`, but this method will be insufficient when outer joins have been pushed down, since the join tuples in that case might have some fields go to NULL without rejecting the tuple entirely.

Another `ForeignScan` field that can be filled by FDWs is `fdw_scan_tlist`, which describes the tuples returned by the FDW for this plan node. For simple foreign table scans this can be set to `NIL`, implying that the returned tuples have the row type declared for the foreign table. A non-`NIL` value must be a target list (list of `TargetEntry`s) containing Vars and/or expressions representing the returned columns. This might be used, for example, to show that the FDW has omitted some columns that it noticed won't be needed for the query. Also, if the FDW can compute expressions used by the query more cheaply than can be done locally, it could add those expressions to `fdw_scan_tlist`. Note that join plans (created from paths made by `GetForeignJoinPaths`) must always supply `fdw_scan_tlist` to describe the set of columns they will return.

The FDW should always construct at least one path that depends only on the table's restriction clauses. In join queries, it might also choose to construct path(s) that depend on join clauses, for example *foreign_variable = local_variable*. Such clauses will not be found in baserel->baserestrictinfo but must be sought in the relation's join lists. A path using such a clause is called a "parameterized path". It must identify the other relations used in the selected join clause(s) with a suitable value of param_info; use get_baserel_parampathinfo to compute that value. In GetForeignPlan, the *local_variable* portion of the join clause would be added to fdw_exprs, and then at run time the case works the same as for an ordinary restriction clause.

If an FDW supports remote joins, GetForeignJoinPaths should produce ForeignPaths for potential remote joins in much the same way as GetForeignPaths works for base tables. Information about the intended join can be passed forward to GetForeignPlan in the same ways described above. However, baserestrictinfo is not relevant for join relations; instead, the relevant join clauses for a particular join are passed to GetForeignJoinPaths as a separate parameter (extra->restrictlist).

When planning an UPDATE or DELETE, PlanForeignModify can look up the RelOptInfo struct for the foreign table and make use of the baserel->fdw_private data previously created by the scan-planning functions. However, in INSERT the target table is not scanned so there is no RelOptInfo for it. The List returned by PlanForeignModify has the same restrictions as the fdw_private list of a ForeignScan plan node, that is it must contain only structures that copyObject knows how to copy.

INSERT with an ON CONFLICT clause does not support specifying the conflict target, as unique constraints or exclusion constraints on remote tables are not locally known. This in turn implies that ON CONFLICT DO UPDATE is not supported, since the specification is mandatory there.

54.5. Row Locking in Foreign Data Wrappers

If an FDW's underlying storage mechanism has a concept of locking individual rows to prevent concurrent updates of those rows, it is usually worthwhile for the FDW to perform row-level locking with as close an approximation as practical to the semantics used in ordinary PostgreSQL tables. There are multiple considerations involved in this.

One key decision to be made is whether to perform *early locking* or *late locking*. In early locking, a row is locked when it is first retrieved from the underlying store, while in late locking, the row is locked only when it is known that it needs to be locked. (The difference arises because some rows may be discarded by locally-checked restriction or join conditions.) Early locking is much simpler and avoids extra round trips to a remote store, but it can cause locking of rows that need not have been locked, resulting in reduced concurrency or even unexpected deadlocks. Also, late locking is only possible if the row to be locked can be uniquely re-identified later. Preferably the row identifier should identify a specific version of the row, as PostgreSQL TIDs do.

By default, PostgreSQL ignores locking considerations when interfacing to FDWs, but an FDW can perform early locking without any explicit support from the core code. The API functions described in Section 54.2.4, which were added in PostgreSQL 9.5, allow an FDW to use late locking if it wishes.

An additional consideration is that in READ COMMITTED isolation mode, PostgreSQL may need to re-check restriction and join conditions against an updated version of some target tuple. Rechecking join conditions requires re-obtaining copies of the non-target rows that were previously joined to the target tuple. When working with standard PostgreSQL tables, this is done by including the TIDs of the non-target tables in the column list projected through the join, and then re-fetching non-target rows when

required. This approach keeps the join data set compact, but it requires inexpensive re-fetch capability, as well as a TID that can uniquely identify the row version to be re-fetched. By default, therefore, the approach used with foreign tables is to include a copy of the entire row fetched from a foreign table in the column list projected through the join. This puts no special demands on the FDW but can result in reduced performance of merge and hash joins. An FDW that is capable of meeting the re-fetch requirements can choose to do it the first way.

For an `UPDATE` or `DELETE` on a foreign table, it is recommended that the `ForeignScan` operation on the target table perform early locking on the rows that it fetches, perhaps via the equivalent of `SELECT FOR UPDATE`. An FDW can detect whether a table is an `UPDATE`/`DELETE` target at plan time by comparing its relid to `root->parse->resultRelation`, or at execution time by using `ExecRelationIsTargetRelation()`. An alternative possibility is to perform late locking within the `ExecForeignUpdate` or `ExecForeignDelete` callback, but no special support is provided for this.

For foreign tables that are specified to be locked by a `SELECT FOR UPDATE/SHARE` command, the `ForeignScan` operation can again perform early locking by fetching tuples with the equivalent of `SELECT FOR UPDATE/SHARE`. To perform late locking instead, provide the callback functions defined in Section 54.2.4. In `GetForeignRowMarkType`, select rowmark option `ROW_MARK_EXCLUSIVE`, `ROW_MARK_NOKEYEXCLUSIVE`, `ROW_MARK_SHARE`, or `ROW_MARK_KEYSHARE` depending on the requested lock strength. (The core code will act the same regardless of which of these four options you choose.) Elsewhere, you can detect whether a foreign table was specified to be locked by this type of command by using `get_plan_rowmark` at plan time, or `ExecFindRowMark` at execution time; you must check not only whether a non-null rowmark struct is returned, but that its `strength` field is not `LCS_NONE`.

Lastly, for foreign tables that are used in an `UPDATE`, `DELETE` or `SELECT FOR UPDATE/SHARE` command but are not specified to be row-locked, you can override the default choice to copy entire rows by having `GetForeignRowMarkType` select option `ROW_MARK_REFERENCE` when it sees lock strength `LCS_NONE`. This will cause `RefetchForeignRow` to be called with that value for `markType`; it should then re-fetch the row without acquiring any new lock. (If you have a `GetForeignRowMarkType` function but don't wish to re-fetch unlocked rows, select option `ROW_MARK_COPY` for `LCS_NONE`.)

See `src/include/nodes/lockoptions.h`, the comments for `RowMarkType` and `PlanRowMark` in `src/include/nodes/plannodes.h`, and the comments for `ExecRowMark` in `src/include/nodes/execnodes.h` for additional information.

Chapter 55. Writing A Table Sampling Method

PostgreSQL's implementation of the TABLESAMPLE clause supports custom table sampling methods, in addition to the BERNOULLI and SYSTEM methods that are required by the SQL standard. The sampling method determines which rows of the table will be selected when the TABLESAMPLE clause is used.

At the SQL level, a table sampling method is represented by a single SQL function, typically implemented in C, having the signature

```
method_name(internal) RETURNS tsm_handler
```

The name of the function is the same method name appearing in the TABLESAMPLE clause. The internal argument is a dummy (always having value zero) that simply serves to prevent this function from being called directly from a SQL command. The result of the function must be a palloc'd struct of type TsmRoutine, which contains pointers to support functions for the sampling method. These support functions are plain C functions and are not visible or callable at the SQL level. The support functions are described in Section 55.1.

In addition to function pointers, the TsmRoutine struct must provide these additional fields:

`List *parameterTypes`

: This is an OID list containing the data type OIDs of the parameter(s) that will be accepted by the TABLESAMPLE clause when this sampling method is used. For example, for the built-in methods, this list contains a single item with value FLOAT4OID, which represents the sampling percentage. Custom sampling methods can have more or different parameters.

`bool repeatable_across_queries`

: If true, the sampling method can deliver identical samples across successive queries, if the same parameters and REPEATABLE seed value are supplied each time and the table contents have not changed. When this is false, the REPEATABLE clause is not accepted for use with the sampling method.

`bool repeatable_across_scans`

: If true, the sampling method can deliver identical samples across successive scans in the same query (assuming unchanging parameters, seed value, and snapshot). When this is false, the planner will not select plans that would require scanning the sampled table more than once, since that might result in inconsistent query output.

The TsmRoutine struct type is declared in src/include/access/tsmapi.h, which see for additional details.

The table sampling methods included in the standard distribution are good references when trying to write your own. Look into the src/backend/access/tablesample subdirectory of the source tree for the built-in sampling methods, and into the contrib subdirectory for add-on methods.

55.1. Sampling Method Support Functions

The TSM handler function returns a palloc'd `TsmRoutine` struct containing pointers to the support functions described below. Most of the functions are required, but some are optional, and those pointers can be NULL.

```
void
SampleScanGetSampleSize (PlannerInfo *root,
                         RelOptInfo *baserel,
                         List *paramexprs,
                         BlockNumber *pages,
                         double *tuples);
```

This function is called during planning. It must estimate the number of relation pages that will be read during a sample scan, and the number of tuples that will be selected by the scan. (For example, these might be determined by estimating the sampling fraction, and then multiplying the `baserel->pages` and `baserel->tuples` numbers by that, being sure to round the results to integral values.) The `paramexprs` list holds the expression(s) that are parameters to the `TABLESAMPLE` clause. It is recommended to use `estimate_expression_value()` to try to reduce these expressions to constants, if their values are needed for estimation purposes; but the function must provide size estimates even if they cannot be reduced, and it should not fail even if the values appear invalid (remember that they're only estimates of what the run-time values will be). The `pages` and `tuples` parameters are outputs.

```
void
InitSampleScan (SampleScanState *node,
                int eflags);
```

Initialize for execution of a SampleScan plan node. This is called during executor startup. It should perform any initialization needed before processing can start. The `SampleScanState` node has already been created, but its `tsm_state` field is NULL. The `InitSampleScan` function can palloc whatever internal state data is needed by the sampling method, and store a pointer to it in `node->tsm_state`. Information about the table to scan is accessible through other fields of the `SampleScanState` node (but note that the `node->ss.ss_currentScanDesc` scan descriptor is not set up yet). `eflags` contains flag bits describing the executor's operating mode for this plan node.

When `(eflags & EXEC_FLAG_EXPLAIN_ONLY)` is true, the scan will not actually be performed, so this function should only do the minimum required to make the node state valid for `EXPLAIN` and `EndSampleScan`.

This function can be omitted (set the pointer to NULL), in which case `BeginSampleScan` must perform all initialization needed by the sampling method.

```
void
BeginSampleScan (SampleScanState *node,
                 Datum *params,
                 int nparams,
                 uint32 seed);
```

Begin execution of a sampling scan. This is called just before the first attempt to fetch a tuple, and may be called again if the scan needs to be restarted. Information about the table to scan is accessible through fields of the `SampleScanState` node (but note that the `node->ss.ss_currentScanDesc` scan descriptor is

not set up yet). The `params` array, of length `nparams`, contains the values of the parameters supplied in the `TABLESAMPLE` clause. These will have the number and types specified in the sampling method's `parameterTypes` list, and have been checked to not be null. `seed` contains a seed to use for any random numbers generated within the sampling method; it is either a hash derived from the `REPEATABLE` value if one was given, or the result of `random()` if not.

This function may adjust the fields `node->use_bulkread` and `node->use_pagemode`. If `node->use_bulkread` is `true`, which it is by default, the scan will use a buffer access strategy that encourages recycling buffers after use. It might be reasonable to set this to `false` if the scan will visit only a small fraction of the table's pages. If `node->use_pagemode` is `true`, which it is by default, the scan will perform visibility checking in a single pass for all tuples on each visited page. It might be reasonable to set this to `false` if the scan will select only a small fraction of the tuples on each visited page. That will result in fewer tuple visibility checks being performed, though each one will be more expensive because it will require more locking.

If the sampling method is marked `repeatable_across_scans`, it must be able to select the same set of tuples during a rescan as it did originally, that is a fresh call of `BeginSampleScan` must lead to selecting the same tuples as before (if the `TABLESAMPLE` parameters and seed don't change).

```
BlockNumber
NextSampleBlock (SampleScanState *node);
```

Returns the block number of the next page to be scanned, or `InvalidBlockNumber` if no pages remain to be scanned.

This function can be omitted (set the pointer to NULL), in which case the core code will perform a sequential scan of the entire relation. Such a scan can use synchronized scanning, so that the sampling method cannot assume that the relation pages are visited in the same order on each scan.

```
OffsetNumber
NextSampleTuple (SampleScanState *node,
                 BlockNumber blockno,
                 OffsetNumber maxoffset);
```

Returns the offset number of the next tuple to be sampled on the specified page, or `InvalidOffsetNumber` if no tuples remain to be sampled. `maxoffset` is the largest offset number in use on the page.

> **Note:** `NextSampleTuple` is not explicitly told which of the offset numbers in the range `1 .. maxoffset` actually contain valid tuples. This is not normally a problem since the core code ignores requests to sample missing or invisible tuples; that should not result in any bias in the sample. However, if necessary, the function can examine `node->ss.ss_currentScanDesc->rs_vistuples[]` to identify which tuples are valid and visible. (This requires `node->use_pagemode` to be `true`.)

> **Note:** `NextSampleTuple` must *not* assume that `blockno` is the same page number returned by the most recent `NextSampleBlock` call. It was returned by some previous `NextSampleBlock` call, but the core code is allowed to call `NextSampleBlock` in advance of actually scanning pages, so as

to support prefetching. It is OK to assume that once sampling of a given page begins, successive `NextSampleTuple` calls all refer to the same page until `InvalidOffsetNumber` is returned.

```
void
EndSampleScan (SampleScanState *node);
```

End the scan and release resources. It is normally not important to release palloc'd memory, but any externally-visible resources should be cleaned up. This function can be omitted (set the pointer to NULL) in the common case where no such resources exist.

Chapter 56. Writing A Custom Scan Provider

PostgreSQL supports a set of experimental facilities which are intended to allow extension modules to add new scan types to the system. Unlike a foreign data wrapper, which is only responsible for knowing how to scan its own foreign tables, a custom scan provider can provide an alternative method of scanning any relation in the system. Typically, the motivation for writing a custom scan provider will be to allow the use of some optimization not supported by the core system, such as caching or some form of hardware acceleration. This chapter outlines how to write a new custom scan provider.

Implementing a new type of custom scan is a three-step process. First, during planning, it is necessary to generate access paths representing a scan using the proposed strategy. Second, if one of those access paths is selected by the planner as the optimal strategy for scanning a particular relation, the access path must be converted to a plan. Finally, it must be possible to execute the plan and generate the same results that would have been generated for any other access path targeting the same relation.

56.1. Creating Custom Scan Paths

A custom scan provider will typically add paths for a base relation by setting the following hook, which is called after the core code has generated what it believes to be the complete and correct set of access paths for the relation.

```
typedef void (*set_rel_pathlist_hook_type) (PlannerInfo *root,
                                            RelOptInfo *rel,
                                            Index rti,
                                            RangeTblEntry *rte);
extern PGDLLIMPORT set_rel_pathlist_hook_type set_rel_pathlist_hook;
```

Although this hook function can be used to examine, modify, or remove paths generated by the core system, a custom scan provider will typically confine itself to generating `CustomPath` objects and adding them to `rel` using `add_path`. The custom scan provider is responsible for initializing the `CustomPath` object, which is declared like this:

```
typedef struct CustomPath
{
    Path      path;
    uint32    flags;
    List     *custom_paths;
    List     *custom_private;
    const CustomPathMethods *methods;
} CustomPath;
```

`path` must be initialized as for any other path, including the row-count estimate, start and total cost, and sort ordering provided by this path. `flags` is a bit mask, which should include

CUSTOMPATH_SUPPORT_BACKWARD_SCAN if the custom path can support a backward scan and CUSTOMPATH_SUPPORT_MARK_RESTORE if it can support mark and restore. Both capabilities are optional. An optional custom_paths is a list of Path nodes used by this custom-path node; these will be transformed into Plan nodes by planner. custom_private can be used to store the custom path's private data. Private data should be stored in a form that can be handled by nodeToString, so that debugging routines that attempt to print the custom path will work as designed. methods must point to a (usually statically allocated) object implementing the required custom path methods, of which there are currently only two, as further detailed below.

A custom scan provider can also provide join paths. Just as for base relations, such a path must produce the same output as would normally be produced by the join it replaces. To do this, the join provider should set the following hook, and then within the hook function, create CustomPath path(s) for the join relation.

```
typedef void (*set_join_pathlist_hook_type) (PlannerInfo *root,
                                             RelOptInfo *joinrel,
                                             RelOptInfo *outerrel,
                                             RelOptInfo *innerrel,
                                             JoinType jointype,
                                             JoinPathExtraData *extra);
extern PGDLLIMPORT set_join_pathlist_hook_type set_join_pathlist_hook;
```

This hook will be invoked repeatedly for the same join relation, with different combinations of inner and outer relations; it is the responsibility of the hook to minimize duplicated work.

56.1.1. Custom Scan Path Callbacks

```
Plan *(*PlanCustomPath) (PlannerInfo *root,
                         RelOptInfo *rel,
                         CustomPath *best_path,
                         List *tlist,
                         List *clauses,
                         List *custom_plans);
```

Convert a custom path to a finished plan. The return value will generally be a CustomScan object, which the callback must allocate and initialize. See Section 56.2 for more details.

```
void (*TextOutCustomPath) (StringInfo str,
                           const CustomPath *node);
```

Generate additional output when nodeToString is invoked on this custom path. This callback is optional. Since nodeToString will automatically dump all fields in the structure that it can see, including custom_private, this is only useful if the CustomPath is actually embedded in a larger struct containing additional fields.

56.2. Creating Custom Scan Plans

A custom scan is represented in a finished plan tree using the following structure:

```
typedef struct CustomScan
{
    Scan      scan;
    uint32    flags;
    List     *custom_plans;
    List     *custom_exprs;
    List     *custom_private;
    List     *custom_scan_tlist;
    Bitmapset *custom_relids;
    const CustomScanMethods *methods;
} CustomScan;
```

`scan` must be initialized as for any other scan, including estimated costs, target lists, qualifications, and so on. `flags` is a bit mask with the same meaning as in `CustomPath`. `custom_plans` can be used to store child `Plan` nodes. `custom_exprs` should be used to store expression trees that will need to be fixed up by `setrefs.c` and `subselect.c`, while `custom_private` should be used to store other private data that is only used by the custom scan provider itself. `custom_scan_tlist` can be NIL when scanning a base relation, indicating that the custom scan returns scan tuples that match the base relation's row type. Otherwise it is a target list describing the actual scan tuples. `custom_scan_tlist` must be provided for joins, and could be provided for scans if the custom scan provider can compute some non-Var expressions. `custom_relids` is set by the core code to the set of relations (range table indexes) that this scan node handles; except when this scan is replacing a join, it will have only one member. `methods` must point to a (usually statically allocated) object implementing the required custom scan methods, which are further detailed below.

When a `CustomScan` scans a single relation, `scan.scanrelid` must be the range table index of the table to be scanned. When it replaces a join, `scan.scanrelid` should be zero.

Plan trees must be able to be duplicated using `copyObject`, so all the data stored within the "custom" fields must consist of nodes that that function can handle. Furthermore, custom scan providers cannot substitute a larger structure that embeds a `CustomScan` for the structure itself, as would be possible for a `CustomPath` or `CustomScanState`.

56.2.1. Custom Scan Plan Callbacks

```
Node *(*CreateCustomScanState) (CustomScan *cscan);
```

Allocate a `CustomScanState` for this `CustomScan`. The actual allocation will often be larger than required for an ordinary `CustomScanState`, because many providers will wish to embed that as the first field of a larger structure. The value returned must have the node tag and `methods` set appropriately, but other fields should be left as zeroes at this stage; after `ExecInitCustomScan` performs basic initialization, the `BeginCustomScan` callback will be invoked to give the custom scan provider a chance to do whatever else is needed.

```
void (*TextOutCustomScan) (StringInfo str,
                           const CustomScan *node);
```

Generate additional output when `nodeToString` is invoked on this custom plan node. This callback is optional. Since `nodeToString` will automatically dump all fields in the structure, including the substructure of the "custom" fields, there is usually not much need for this callback.

56.3. Executing Custom Scans

When a `CustomScan` is executed, its execution state is represented by a `CustomScanState`, which is declared as follows:

```
typedef struct CustomScanState
{
    ScanState ss;
    uint32    flags;
    const CustomExecMethods *methods;
} CustomScanState;
```

`ss` is initialized as for any other scan state, except that if the scan is for a join rather than a base relation, `ss.ss_currentRelation` is left NULL. `flags` is a bit mask with the same meaning as in `CustomPath` and `CustomScan`. `methods` must point to a (usually statically allocated) object implementing the required custom scan state methods, which are further detailed below. Typically, a `CustomScanState`, which need not support `copyObject`, will actually be a larger structure embedding the above as its first member.

56.3.1. Custom Scan Execution Callbacks

```
void (*BeginCustomScan) (CustomScanState *node,
                         EState *estate,
                         int eflags);
```

Complete initialization of the supplied `CustomScanState`. Standard fields have been initialized by `ExecInitCustomScan`, but any private fields should be initialized here.

```
TupleTableSlot *(*ExecCustomScan) (CustomScanState *node);
```

Fetch the next scan tuple. If any tuples remain, it should fill `ps_ResultTupleSlot` with the next tuple in the current scan direction, and then return the tuple slot. If not, `NULL` or an empty slot should be returned.

```
void (*EndCustomScan) (CustomScanState *node);
```

Clean up any private data associated with the `CustomScanState`. This method is required, but it does not need to do anything if there is no associated data or it will be cleaned up automatically.

```
void (*ReScanCustomScan) (CustomScanState *node);
```

Rewind the current scan to the beginning and prepare to rescan the relation.

```
void (*MarkPosCustomScan) (CustomScanState *node);
```

Save the current scan position so that it can subsequently be restored by the `RestrPosCustomScan` callback. This callback is optional, and need only be supplied if the `CUSTOMPATH_SUPPORT_MARK_RESTORE` flag is set.

```
void (*RestrPosCustomScan) (CustomScanState *node);
```

Restore the previous scan position as saved by the `MarkPosCustomScan` callback. This callback is optional, and need only be supplied if the `CUSTOMPATH_SUPPORT_MARK_RESTORE` flag is set.

```
void (*ExplainCustomScan) (CustomScanState *node,
                           List *ancestors,
                           ExplainState *es);
```

Output additional information for `EXPLAIN` of a custom-scan plan node. This callback is optional. Common data stored in the `ScanState`, such as the target list and scan relation, will be shown even without this callback, but the callback allows the display of additional, private state.

Chapter 57. Genetic Query Optimizer

Author: Written by Martin Utesch (<utesch@aut.tu-freiberg.de>) for the Institute of Automatic Control at the University of Mining and Technology in Freiberg, Germany.

57.1. Query Handling as a Complex Optimization Problem

Among all relational operators the most difficult one to process and optimize is the *join*. The number of possible query plans grows exponentially with the number of joins in the query. Further optimization effort is caused by the support of a variety of *join methods* (e.g., nested loop, hash join, merge join in PostgreSQL) to process individual joins and a diversity of *indexes* (e.g., B-tree, hash, GiST and GIN in PostgreSQL) as access paths for relations.

The normal PostgreSQL query optimizer performs a *near-exhaustive search* over the space of alternative strategies. This algorithm, first introduced in IBM's System R database, produces a near-optimal join order, but can take an enormous amount of time and memory space when the number of joins in the query grows large. This makes the ordinary PostgreSQL query optimizer inappropriate for queries that join a large number of tables.

The Institute of Automatic Control at the University of Mining and Technology, in Freiberg, Germany, encountered some problems when it wanted to use PostgreSQL as the backend for a decision support knowledge based system for the maintenance of an electrical power grid. The DBMS needed to handle large join queries for the inference machine of the knowledge based system. The number of joins in these queries made using the normal query optimizer infeasible.

In the following we describe the implementation of a *genetic algorithm* to solve the join ordering problem in a manner that is efficient for queries involving large numbers of joins.

57.2. Genetic Algorithms

The genetic algorithm (GA) is a heuristic optimization method which operates through randomized search. The set of possible solutions for the optimization problem is considered as a *population* of *individuals*. The degree of adaptation of an individual to its environment is specified by its *fitness*.

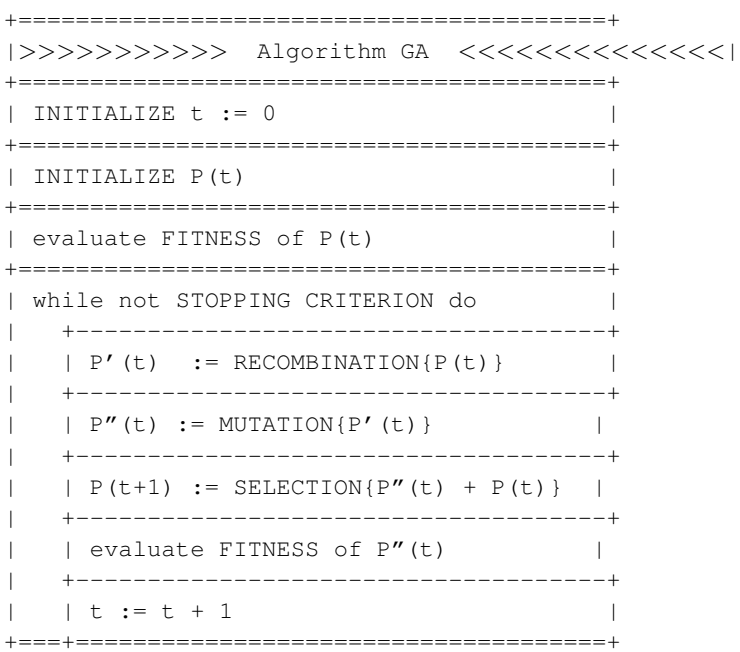
The coordinates of an individual in the search space are represented by *chromosomes*, in essence a set of character strings. A *gene* is a subsection of a chromosome which encodes the value of a single parameter being optimized. Typical encodings for a gene could be *binary* or *integer*.

Through simulation of the evolutionary operations *recombination*, *mutation*, and *selection* new generations of search points are found that show a higher average fitness than their ancestors.

According to the comp.ai.genetic FAQ it cannot be stressed too strongly that a GA is not a pure random search for a solution to a problem. A GA uses stochastic processes, but the result is distinctly non-random (better than random).

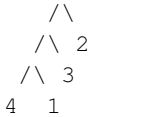
Figure 57-1. Structured Diagram of a Genetic Algorithm

P(t)	generation of ancestors at a time t
P”(t)	generation of descendants at a time t



57.3. Genetic Query Optimization (GEQO) in PostgreSQL

The GEQO module approaches the query optimization problem as though it were the well-known traveling salesman problem (TSP). Possible query plans are encoded as integer strings. Each string represents the join order from one relation of the query to the next. For example, the join tree



is encoded by the integer string ’4-1-3-2’, which means, first join relation ’4’ and ’1’, then ’3’, and then ’2’, where 1, 2, 3, 4 are relation IDs within the PostgreSQL optimizer.

Specific characteristics of the GEQO implementation in PostgreSQL are:

- Usage of a *steady state* GA (replacement of the least fit individuals in a population, not whole-generational replacement) allows fast convergence towards improved query plans. This is essential for query handling with reasonable time;
- Usage of *edge recombination crossover* which is especially suited to keep edge losses low for the solution of the TSP by means of a GA;
- Mutation as genetic operator is deprecated so that no repair mechanisms are needed to generate legal TSP tours.

Parts of the GEQO module are adapted from D. Whitley's Genitor algorithm.

The GEQO module allows the PostgreSQL query optimizer to support large join queries effectively through non-exhaustive search.

57.3.1. Generating Possible Plans with GEQO

The GEQO planning process uses the standard planner code to generate plans for scans of individual relations. Then join plans are developed using the genetic approach. As shown above, each candidate join plan is represented by a sequence in which to join the base relations. In the initial stage, the GEQO code simply generates some possible join sequences at random. For each join sequence considered, the standard planner code is invoked to estimate the cost of performing the query using that join sequence. (For each step of the join sequence, all three possible join strategies are considered; and all the initially-determined relation scan plans are available. The estimated cost is the cheapest of these possibilities.) Join sequences with lower estimated cost are considered "more fit" than those with higher cost. The genetic algorithm discards the least fit candidates. Then new candidates are generated by combining genes of more-fit candidates — that is, by using randomly-chosen portions of known low-cost join sequences to create new sequences for consideration. This process is repeated until a preset number of join sequences have been considered; then the best one found at any time during the search is used to generate the finished plan.

This process is inherently nondeterministic, because of the randomized choices made during both the initial population selection and subsequent "mutation" of the best candidates. To avoid surprising changes of the selected plan, each run of the GEQO algorithm restarts its random number generator with the current geqo_seed parameter setting. As long as `geqo_seed` and the other GEQO parameters are kept fixed, the same plan will be generated for a given query (and other planner inputs such as statistics). To experiment with different search paths, try changing `geqo_seed`.

57.3.2. Future Implementation Tasks for PostgreSQL GEQO

Work is still needed to improve the genetic algorithm parameter settings. In file `src/backend/optimizer/geqo/geqo_main.c`, routines `gimme_pool_size` and `gimme_number_generations`, we have to find a compromise for the parameter settings to satisfy two competing demands:

- Optimality of the query plan
- Computing time

In the current implementation, the fitness of each candidate join sequence is estimated by running the standard planner's join selection and cost estimation code from scratch. To the extent that different candidates use similar sub-sequences of joins, a great deal of work will be repeated. This could be made significantly faster by retaining cost estimates for sub-joins. The problem is to avoid expending unreasonable amounts of memory on retaining that state.

At a more basic level, it is not clear that solving query optimization with a GA algorithm designed for TSP is appropriate. In the TSP case, the cost associated with any substring (partial tour) is independent of the rest of the tour, but this is certainly not true for query optimization. Thus it is questionable whether edge recombination crossover is the most effective mutation procedure.

57.4. Further Reading

The following resources contain additional information about genetic algorithms:

- The Hitch-Hiker's Guide to Evolutionary Computation[1], (FAQ for news://comp.ai.genetic)
- Evolutionary Computation and its application to art and design[2], by Craig Reynolds
- *Fundamentals of Database Systems*
- *The design and implementation of the POSTGRES query optimizer*

1. http://www.faqs.org/faqs/ai-faq/genetic/part1/
2. https://www.red3d.com/cwr/evolve.html

Chapter 58. Index Access Method Interface Definition

This chapter defines the interface between the core PostgreSQL system and *index access methods*, which manage individual index types. The core system knows nothing about indexes beyond what is specified here, so it is possible to develop entirely new index types by writing add-on code.

All indexes in PostgreSQL are what are known technically as *secondary indexes*; that is, the index is physically separate from the table file that it describes. Each index is stored as its own physical *relation* and so is described by an entry in the `pg_class` catalog. The contents of an index are entirely under the control of its index access method. In practice, all index access methods divide indexes into standard-size pages so that they can use the regular storage manager and buffer manager to access the index contents. (All the existing index access methods furthermore use the standard page layout described in Section 63.6, and they all use the same format for index tuple headers; but these decisions are not forced on an access method.)

An index is effectively a mapping from some data key values to *tuple identifiers*, or TIDs, of row versions (tuples) in the index's parent table. A TID consists of a block number and an item number within that block (see Section 63.6). This is sufficient information to fetch a particular row version from the table. Indexes are not directly aware that under MVCC, there might be multiple extant versions of the same logical row; to an index, each tuple is an independent object that needs its own index entry. Thus, an update of a row always creates all-new index entries for the row, even if the key values did not change. (HOT tuples are an exception to this statement; but indexes do not deal with those, either.) Index entries for dead tuples are reclaimed (by vacuuming) when the dead tuples themselves are reclaimed.

58.1. Catalog Entries for Indexes

Each index access method is described by a row in the `pg_am` system catalog (see Section 49.3). The principal contents of a `pg_am` row are references to `pg_proc` entries that identify the index access functions supplied by the access method. The APIs for these functions are defined later in this chapter. In addition, the `pg_am` row specifies a few fixed properties of the access method, such as whether it can support multicolumn indexes. There is not currently any special support for creating or deleting `pg_am` entries; anyone able to write a new access method is expected to be competent to insert an appropriate row for themselves.

To be useful, an index access method must also have one or more *operator families* and *operator classes* defined in `pg_opfamily`, `pg_opclass`, `pg_amop`, and `pg_amproc`. These entries allow the planner to determine what kinds of query qualifications can be used with indexes of this access method. Operator families and classes are described in Section 35.14, which is prerequisite material for reading this chapter.

An individual index is defined by a `pg_class` entry that describes it as a physical relation, plus a `pg_index` entry that shows the logical content of the index — that is, the set of index columns it has and the semantics of those columns, as captured by the associated operator classes. The index columns (key values) can be either simple columns of the underlying table or expressions over the table rows. The index access method normally has no interest in where the index key values come from (it is al-

ways handed precomputed key values) but it will be very interested in the operator class information in pg_index. Both of these catalog entries can be accessed as part of the Relation data structure that is passed to all operations on the index.

Some of the flag columns of pg_am have nonobvious implications. The requirements of amcanunique are discussed in Section 58.5. The amcanmulticol flag asserts that the access method supports multicolumn indexes, while amoptionalkey asserts that it allows scans where no indexable restriction clause is given for the first index column. When amcanmulticol is false, amoptionalkey essentially says whether the access method supports full-index scans without any restriction clause. Access methods that support multiple index columns *must* support scans that omit restrictions on any or all of the columns after the first; however they are permitted to require some restriction to appear for the first index column, and this is signaled by setting amoptionalkey false. One reason that an index AM might set amoptionalkey false is if it doesn't index null values. Since most indexable operators are strict and hence cannot return true for null inputs, it is at first sight attractive to not store index entries for null values: they could never be returned by an index scan anyway. However, this argument fails when an index scan has no restriction clause for a given index column. In practice this means that indexes that have amoptionalkey true must index nulls, since the planner might decide to use such an index with no scan keys at all. A related restriction is that an index access method that supports multiple index columns *must* support indexing null values in columns after the first, because the planner will assume the index can be used for queries that do not restrict these columns. For example, consider an index on (a,b) and a query with WHERE a = 4. The system will assume the index can be used to scan for rows with a = 4, which is wrong if the index omits rows where b is null. It is, however, OK to omit rows where the first indexed column is null. An index access method that does index nulls may also set amsearchnulls, indicating that it supports IS NULL and IS NOT NULL clauses as search conditions.

58.2. Index Access Method Functions

The index construction and maintenance functions that an index access method must provide are:

```
IndexBuildResult *
ambuild (Relation heapRelation,
         Relation indexRelation,
         IndexInfo *indexInfo);
```

Build a new index. The index relation has been physically created, but is empty. It must be filled in with whatever fixed data the access method requires, plus entries for all tuples already existing in the table. Ordinarily the ambuild function will call IndexBuildHeapScan() to scan the table for existing tuples and compute the keys that need to be inserted into the index. The function must return a palloc'd struct containing statistics about the new index.

```
void
ambuildempty (Relation indexRelation);
```

Build an empty index, and write it to the initialization fork (INIT_FORKNUM) of the given relation. This method is called only for unlogged tables; the empty index written to the initialization fork will be copied over the main relation fork on each server restart.

```
bool
```

```
aminsert (Relation indexRelation,
          Datum *values,
          bool *isnull,
          ItemPointer heap_tid,
          Relation heapRelation,
          IndexUniqueCheck checkUnique);
```

Insert a new tuple into an existing index. The `values` and `isnull` arrays give the key values to be indexed, and `heap_tid` is the TID to be indexed. If the access method supports unique indexes (its `pg_am.amcanunique` flag is true) then `checkUnique` indicates the type of uniqueness check to perform. This varies depending on whether the unique constraint is deferrable; see Section 58.5 for details. Normally the access method only needs the `heapRelation` parameter when performing uniqueness checking (since then it will have to look into the heap to verify tuple liveness).

The function's Boolean result value is significant only when `checkUnique` is `UNIQUE_CHECK_PARTIAL`. In this case a TRUE result means the new entry is known unique, whereas FALSE means it might be non-unique (and a deferred uniqueness check must be scheduled). For other cases a constant FALSE result is recommended.

Some indexes might not index all tuples. If the tuple is not to be indexed, `aminsert` should just return without doing anything.

```
IndexBulkDeleteResult *
ambulkdelete (IndexVacuumInfo *info,
              IndexBulkDeleteResult *stats,
              IndexBulkDeleteCallback callback,
              void *callback_state);
```

Delete tuple(s) from the index. This is a "bulk delete" operation that is intended to be implemented by scanning the whole index and checking each entry to see if it should be deleted. The passed-in `callback` function must be called, in the style `callback(`*`TID`*`, callback_state) returns bool`, to determine whether any particular index entry, as identified by its referenced TID, is to be deleted. Must return either NULL or a palloc'd struct containing statistics about the effects of the deletion operation. It is OK to return NULL if no information needs to be passed on to `amvacuumcleanup`.

Because of limited `maintenance_work_mem`, `ambulkdelete` might need to be called more than once when many tuples are to be deleted. The `stats` argument is the result of the previous call for this index (it is NULL for the first call within a `VACUUM` operation). This allows the AM to accumulate statistics across the whole operation. Typically, `ambulkdelete` will modify and return the same struct if the passed `stats` is not null.

```
IndexBulkDeleteResult *
amvacuumcleanup (IndexVacuumInfo *info,
                 IndexBulkDeleteResult *stats);
```

Clean up after a `VACUUM` operation (zero or more `ambulkdelete` calls). This does not have to do anything beyond returning index statistics, but it might perform bulk cleanup such as reclaiming empty index pages. `stats` is whatever the last `ambulkdelete` call returned, or NULL if `ambulkdelete` was not called because no tuples needed to be deleted. If the result is not NULL it must be a palloc'd struct. The statistics it contains will be used to update `pg_class`, and will be reported by `VACUUM` if `VERBOSE` is given. It is

OK to return NULL if the index was not changed at all during the VACUUM operation, but otherwise correct stats should be returned.

As of PostgreSQL 8.4, amvacuumcleanup will also be called at completion of an ANALYZE operation. In this case stats is always NULL and any return value will be ignored. This case can be distinguished by checking info->analyze_only. It is recommended that the access method do nothing except post-insert cleanup in such a call, and that only in an autovacuum worker process.

```
bool
amcanreturn (Relation indexRelation, int attno);
```

Check whether the index can support *index-only scans* on the given column, by returning the indexed column values for an index entry in the form of an IndexTuple. The attribute number is 1-based, i.e., the first columns attno is 1. Returns TRUE if supported, else FALSE. If the access method does not support index-only scans at all, the amcanreturn field in its pg_am row can be set to zero.

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation);
```

Estimate the costs of an index scan. This function is described fully in Section 58.6, below.

```
bytea *
amoptions (ArrayType *reloptions,
           bool validate);
```

Parse and validate the reloptions array for an index. This is called only when a non-null reloptions array exists for the index. reloptions is a text array containing entries of the form *name*=*value*. The function should construct a bytea value, which will be copied into the rd_options field of the index's relcache entry. The data contents of the bytea value are open for the access method to define; most of the standard access methods use struct StdRdOptions. When validate is true, the function should report a suitable error message if any of the options are unrecognized or have invalid values; when validate is false, invalid entries should be silently ignored. (validate is false when loading options already stored in pg_catalog; an invalid entry could only be found if the access method has changed its rules for options, and in that case ignoring obsolete entries is appropriate.) It is OK to return NULL if default behavior is wanted.

The purpose of an index, of course, is to support scans for tuples matching an indexable WHERE condition, often called a *qualifier* or *scan key*. The semantics of index scanning are described more fully in Section 58.3, below. An index access method can support "plain" index scans, "bitmap" index scans, or both. The scan-related functions that an index access method must or may provide are:

```
IndexScanDesc
ambeginscan (Relation indexRelation,
             int nkeys,
             int norderbys);
```

Prepare for an index scan. The `nkeys` and `norderbys` parameters indicate the number of quals and ordering operators that will be used in the scan; these may be useful for space allocation purposes. Note that the actual values of the scan keys aren't provided yet. The result must be a palloc'd struct. For implementation reasons the index access method *must* create this struct by calling `RelationGetIndexScan()`. In most cases `ambeginscan` does little beyond making that call and perhaps acquiring locks; the interesting parts of index-scan startup are in `amrescan`.

```
void
amrescan (IndexScanDesc scan,
          ScanKey keys,
          int nkeys,
          ScanKey orderbys,
          int norderbys);
```

Start or restart an index scan, possibly with new scan keys. (To restart using previously-passed keys, NULL is passed for `keys` and/or `orderbys`.) Note that it is not allowed for the number of keys or order-by operators to be larger than what was passed to `ambeginscan`. In practice the restart feature is used when a new outer tuple is selected by a nested-loop join and so a new key comparison value is needed, but the scan key structure remains the same.

```
boolean
amgettuple (IndexScanDesc scan,
            ScanDirection direction);
```

Fetch the next tuple in the given scan, moving in the given direction (forward or backward in the index). Returns TRUE if a tuple was obtained, FALSE if no matching tuples remain. In the TRUE case the tuple TID is stored into the `scan` structure. Note that "success" means only that the index contains an entry that matches the scan keys, not that the tuple necessarily still exists in the heap or will pass the caller's snapshot test. On success, `amgettuple` must also set `scan->xs_recheck` to TRUE or FALSE. FALSE means it is certain that the index entry matches the scan keys. TRUE means this is not certain, and the conditions represented by the scan keys must be rechecked against the heap tuple after fetching it. This provision supports "lossy" index operators. Note that rechecking will extend only to the scan conditions; a partial index predicate (if any) is never rechecked by `amgettuple` callers.

If the index supports index-only scans (i.e., `amcanreturn` returns TRUE for it), then on success the AM must also check `scan->xs_want_itup`, and if that is true it must return the original indexed data for the index entry, in the form of an `IndexTuple` pointer stored at `scan->xs_itup`, with tuple descriptor `scan->xs_itupdesc`. (Management of the data referenced by the pointer is the access method's responsibility. The data must remain good at least until the next `amgettuple`, `amrescan`, or `amendscan` call for the scan.)

The `amgettuple` function need only be provided if the access method supports "plain" index scans. If it doesn't, the `amgettuple` field in its `pg_am` row must be set to zero.

```
int64
amgetbitmap (IndexScanDesc scan,
             TIDBitmap *tbm);
```

Fetch all tuples in the given scan and add them to the caller-supplied `TIDBitmap` (that is, OR the set of tuple IDs into whatever set is already in the bitmap). The number of tuples fetched is returned (this might

be just an approximate count, for instance some AMs do not detect duplicates). While inserting tuple IDs into the bitmap, `amgetbitmap` can indicate that rechecking of the scan conditions is required for specific tuple IDs. This is analogous to the `xs_recheck` output parameter of `amgettuple`. Note: in the current implementation, support for this feature is conflated with support for lossy storage of the bitmap itself, and therefore callers recheck both the scan conditions and the partial index predicate (if any) for recheckable tuples. That might not always be true, however. `amgetbitmap` and `amgettuple` cannot be used in the same index scan; there are other restrictions too when using `amgetbitmap`, as explained in Section 58.3.

The `amgetbitmap` function need only be provided if the access method supports "bitmap" index scans. If it doesn't, the `amgetbitmap` field in its `pg_am` row must be set to zero.

```
void
amendscan (IndexScanDesc scan);
```

End a scan and release resources. The `scan` struct itself should not be freed, but any locks or pins taken internally by the access method must be released, as well as any other memory allocated by `ambeginscan` and other scan-related functions.

```
void
ammarkpos (IndexScanDesc scan);
```

Mark current scan position. The access method need only support one remembered scan position per scan.

```
void
amrestrpos (IndexScanDesc scan);
```

Restore the scan to the most recently marked position.

By convention, the `pg_proc` entry for an index access method function should show the correct number of arguments, but declare them all as type `internal` (since most of the arguments have types that are not known to SQL, and we don't want users calling the functions directly anyway). The return type is declared as `void`, `internal`, or `boolean` as appropriate. The only exception is `amoptions`, which should be correctly declared as taking `text[]` and `bool` and returning `bytea`. This provision allows client code to execute `amoptions` to test validity of options settings.

58.3. Index Scanning

In an index scan, the index access method is responsible for regurgitating the TIDs of all the tuples it has been told about that match the *scan keys*. The access method is *not* involved in actually fetching those tuples from the index's parent table, nor in determining whether they pass the scan's time qualification test or other conditions.

A scan key is the internal representation of a `WHERE` clause of the form *`index_key operator constant`*, where the index key is one of the columns of the index and the operator is one of the members of the operator family associated with that index column. An index scan has zero or more scan keys, which are implicitly ANDed — the returned tuples are expected to satisfy all the indicated conditions.

The access method can report that the index is *lossy*, or requires rechecks, for a particular query. This implies that the index scan will return all the entries that pass the scan key, plus possibly additional entries

that do not. The core system's index-scan machinery will then apply the index conditions again to the heap tuple to verify whether or not it really should be selected. If the recheck option is not specified, the index scan must return exactly the set of matching entries.

Note that it is entirely up to the access method to ensure that it correctly finds all and only the entries passing all the given scan keys. Also, the core system will simply hand off all the `WHERE` clauses that match the index keys and operator families, without any semantic analysis to determine whether they are redundant or contradictory. As an example, given `WHERE x > 4 AND x > 14` where `x` is a b-tree indexed column, it is left to the b-tree `amrescan` function to realize that the first scan key is redundant and can be discarded. The extent of preprocessing needed during `amrescan` will depend on the extent to which the index access method needs to reduce the scan keys to a "normalized" form.

Some access methods return index entries in a well-defined order, others do not. There are actually two different ways that an access method can support sorted output:

- Access methods that always return entries in the natural ordering of their data (such as btree) should set `pg_am.amcanorder` to true. Currently, such access methods must use btree-compatible strategy numbers for their equality and ordering operators.
- Access methods that support ordering operators should set `pg_am.amcanorderbyop` to true. This indicates that the index is capable of returning entries in an order satisfying `ORDER BY` *`index_key operator constant`*. Scan modifiers of that form can be passed to `amrescan` as described previously.

The `amgettuple` function has a `direction` argument, which can be either `ForwardScanDirection` (the normal case) or `BackwardScanDirection`. If the first call after `amrescan` specifies `BackwardScanDirection`, then the set of matching index entries is to be scanned back-to-front rather than in the normal front-to-back direction, so `amgettuple` must return the last matching tuple in the index, rather than the first one as it normally would. (This will only occur for access methods that set `amcanorder` to true.) After the first call, `amgettuple` must be prepared to advance the scan in either direction from the most recently returned entry. (But if `pg_am.amcanbackward` is false, all subsequent calls will have the same direction as the first one.)

Access methods that support ordered scans must support "marking" a position in a scan and later returning to the marked position. The same position might be restored multiple times. However, only one position need be remembered per scan; a new `ammarkpos` call overrides the previously marked position. An access method that does not support ordered scans should still provide mark and restore functions in `pg_am`, but it is sufficient to have them throw errors if called.

Both the scan position and the mark position (if any) must be maintained consistently in the face of concurrent insertions or deletions in the index. It is OK if a freshly-inserted entry is not returned by a scan that would have found the entry if it had existed when the scan started, or for the scan to return such an entry upon rescanning or backing up even though it had not been returned the first time through. Similarly, a concurrent delete might or might not be reflected in the results of a scan. What is important is that insertions or deletions not cause the scan to miss or multiply return entries that were not themselves being inserted or deleted.

If the index stores the original indexed data values (and not some lossy representation of them), it is useful to support index-only scans, in which the index returns the actual data not just the TID of the heap tuple. This will only work if the visibility map shows that the TID is on an all-visible page; else the heap tuple must be visited anyway to check MVCC visibility. But that is no concern of the access method's.

Instead of using amgettuple, an index scan can be done with amgetbitmap to fetch all tuples in one call. This can be noticeably more efficient than amgettuple because it allows avoiding lock/unlock cycles within the access method. In principle amgetbitmap should have the same effects as repeated amgettuple calls, but we impose several restrictions to simplify matters. First of all, amgetbitmap returns all tuples at once and marking or restoring scan positions isn't supported. Secondly, the tuples are returned in a bitmap which doesn't have any specific ordering, which is why amgetbitmap doesn't take a direction argument. (Ordering operators will never be supplied for such a scan, either.) Also, there is no provision for index-only scans with amgetbitmap, since there is no way to return the contents of index tuples. Finally, amgetbitmap does not guarantee any locking of the returned tuples, with implications spelled out in Section 58.4.

Note that it is permitted for an access method to implement only amgetbitmap and not amgettuple, or vice versa, if its internal implementation is unsuited to one API or the other.

58.4. Index Locking Considerations

Index access methods must handle concurrent updates of the index by multiple processes. The core PostgreSQL system obtains AccessShareLock on the index during an index scan, and RowExclusiveLock when updating the index (including plain VACUUM). Since these lock types do not conflict, the access method is responsible for handling any fine-grained locking it might need. An exclusive lock on the index as a whole will be taken only during index creation, destruction, or REINDEX.

Building an index type that supports concurrent updates usually requires extensive and subtle analysis of the required behavior. For the b-tree and hash index types, you can read about the design decisions involved in src/backend/access/nbtree/README and src/backend/access/hash/README.

Aside from the index's own internal consistency requirements, concurrent updates create issues about consistency between the parent table (the *heap*) and the index. Because PostgreSQL separates accesses and updates of the heap from those of the index, there are windows in which the index might be inconsistent with the heap. We handle this problem with the following rules:

- A new heap entry is made before making its index entries. (Therefore a concurrent index scan is likely to fail to see the heap entry. This is okay because the index reader would be uninterested in an uncommitted row anyway. But see Section 58.5.)
- When a heap entry is to be deleted (by VACUUM), all its index entries must be removed first.
- An index scan must maintain a pin on the index page holding the item last returned by amgettuple, and ambulkdelete cannot delete entries from pages that are pinned by other backends. The need for this rule is explained below.

Without the third rule, it is possible for an index reader to see an index entry just before it is removed by VACUUM, and then to arrive at the corresponding heap entry after that was removed by VACUUM. This creates no serious problems if that item number is still unused when the reader reaches it, since an empty item slot will be ignored by heap_fetch(). But what if a third backend has already re-used the item slot for something else? When using an MVCC-compliant snapshot, there is no problem because the new occupant of the slot is certain to be too new to pass the snapshot test. However, with a non-MVCC-compliant snapshot (such as SnapshotAny), it would be possible to accept and return a row that does not in fact match the scan keys. We could defend against this scenario by requiring the scan keys to be rechecked against the heap row in all cases, but that is too expensive. Instead, we use a pin on an index

page as a proxy to indicate that the reader might still be "in flight" from the index entry to the matching heap entry. Making `ambulkdelete` block on such a pin ensures that `VACUUM` cannot delete the heap entry before the reader is done with it. This solution costs little in run time, and adds blocking overhead only in the rare cases where there actually is a conflict.

This solution requires that index scans be "synchronous": we have to fetch each heap tuple immediately after scanning the corresponding index entry. This is expensive for a number of reasons. An "asynchronous" scan in which we collect many TIDs from the index, and only visit the heap tuples sometime later, requires much less index locking overhead and can allow a more efficient heap access pattern. Per the above analysis, we must use the synchronous approach for non-MVCC-compliant snapshots, but an asynchronous scan is workable for a query using an MVCC snapshot.

In an `amgetbitmap` index scan, the access method does not keep an index pin on any of the returned tuples. Therefore it is only safe to use such scans with MVCC-compliant snapshots.

When the `ampredlocks` flag is not set, any scan using that index access method within a serializable transaction will acquire a nonblocking predicate lock on the full index. This will generate a read-write conflict with the insert of any tuple into that index by a concurrent serializable transaction. If certain patterns of read-write conflicts are detected among a set of concurrent serializable transactions, one of those transactions may be canceled to protect data integrity. When the flag is set, it indicates that the index access method implements finer-grained predicate locking, which will tend to reduce the frequency of such transaction cancellations.

58.5. Index Uniqueness Checks

PostgreSQL enforces SQL uniqueness constraints using *unique indexes*, which are indexes that disallow multiple entries with identical keys. An access method that supports this feature sets `pg_am.amcanunique` true. (At present, only b-tree supports it.)

Because of MVCC, it is always necessary to allow duplicate entries to exist physically in an index: the entries might refer to successive versions of a single logical row. The behavior we actually want to enforce is that no MVCC snapshot could include two rows with equal index keys. This breaks down into the following cases that must be checked when inserting a new row into a unique index:

- If a conflicting valid row has been deleted by the current transaction, it's okay. (In particular, since an UPDATE always deletes the old row version before inserting the new version, this will allow an UPDATE on a row without changing the key.)
- If a conflicting row has been inserted by an as-yet-uncommitted transaction, the would-be inserter must wait to see if that transaction commits. If it rolls back then there is no conflict. If it commits without deleting the conflicting row again, there is a uniqueness violation. (In practice we just wait for the other transaction to end and then redo the visibility check in toto.)
- Similarly, if a conflicting valid row has been deleted by an as-yet-uncommitted transaction, the would-be inserter must wait for that transaction to commit or abort, and then repeat the test.

Furthermore, immediately before reporting a uniqueness violation according to the above rules, the access method must recheck the liveness of the row being inserted. If it is committed dead then no violation should be reported. (This case cannot occur during the ordinary scenario of inserting a row that's just

been created by the current transaction. It can happen during `CREATE UNIQUE INDEX CONCURRENTLY`, however.)

We require the index access method to apply these tests itself, which means that it must reach into the heap to check the commit status of any row that is shown to have a duplicate key according to the index contents. This is without a doubt ugly and non-modular, but it saves redundant work: if we did a separate probe then the index lookup for a conflicting row would be essentially repeated while finding the place to insert the new row's index entry. What's more, there is no obvious way to avoid race conditions unless the conflict check is an integral part of insertion of the new index entry.

If the unique constraint is deferrable, there is additional complexity: we need to be able to insert an index entry for a new row, but defer any uniqueness-violation error until end of statement or even later. To avoid unnecessary repeat searches of the index, the index access method should do a preliminary uniqueness check during the initial insertion. If this shows that there is definitely no conflicting live tuple, we are done. Otherwise, we schedule a recheck to occur when it is time to enforce the constraint. If, at the time of the recheck, both the inserted tuple and some other tuple with the same key are live, then the error must be reported. (Note that for this purpose, "live" actually means "any tuple in the index entry's HOT chain is live".) To implement this, the `aminsert` function is passed a `checkUnique` parameter having one of the following values:

- `UNIQUE_CHECK_NO` indicates that no uniqueness checking should be done (this is not a unique index).
- `UNIQUE_CHECK_YES` indicates that this is a non-deferrable unique index, and the uniqueness check must be done immediately, as described above.
- `UNIQUE_CHECK_PARTIAL` indicates that the unique constraint is deferrable. PostgreSQL will use this mode to insert each row's index entry. The access method must allow duplicate entries into the index, and report any potential duplicates by returning FALSE from `aminsert`. For each row for which FALSE is returned, a deferred recheck will be scheduled.

  The access method must identify any rows which might violate the unique constraint, but it is not an error for it to report false positives. This allows the check to be done without waiting for other transactions to finish; conflicts reported here are not treated as errors and will be rechecked later, by which time they may no longer be conflicts.
- `UNIQUE_CHECK_EXISTING` indicates that this is a deferred recheck of a row that was reported as a potential uniqueness violation. Although this is implemented by calling `aminsert`, the access method must *not* insert a new index entry in this case. The index entry is already present. Rather, the access method must check to see if there is another live index entry. If so, and if the target row is also still live, report error.

  It is recommended that in a `UNIQUE_CHECK_EXISTING` call, the access method further verify that the target row actually does have an existing entry in the index, and report error if not. This is a good idea because the index tuple values passed to `aminsert` will have been recomputed. If the index definition involves functions that are not really immutable, we might be checking the wrong area of the index. Checking that the target row is found in the recheck verifies that we are scanning for the same tuple values as were used in the original insertion.

58.6. Index Cost Estimation Functions

The `amcostestimate` function is given information describing a possible index scan, including lists of WHERE and ORDER BY clauses that have been determined to be usable with the index. It must return estimates of the cost of accessing the index and the selectivity of the WHERE clauses (that is, the fraction of parent-table rows that will be retrieved during the index scan). For simple cases, nearly all the work of the cost estimator can be done by calling standard routines in the optimizer; the point of having an `amcostestimate` function is to allow index access methods to provide index-type-specific knowledge, in case it is possible to improve on the standard estimates.

Each `amcostestimate` function must have the signature:

```
void
amcostestimate (PlannerInfo *root,
                IndexPath *path,
                double loop_count,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity,
                double *indexCorrelation);
```

The first three parameters are inputs:

`root`

The planner's information about the query being processed.

`path`

The index access path being considered. All fields except cost and selectivity values are valid.

`loop_count`

The number of repetitions of the index scan that should be factored into the cost estimates. This will typically be greater than one when considering a parameterized scan for use in the inside of a nestloop join. Note that the cost estimates should still be for just one scan; a larger `loop_count` means that it may be appropriate to allow for some caching effects across multiple scans.

The last four parameters are pass-by-reference outputs:

`*indexStartupCost`

Set to cost of index start-up processing

`*indexTotalCost`

Set to total cost of index processing

`*indexSelectivity`

Set to index selectivity

`*indexCorrelation`

Set to correlation coefficient between index scan order and underlying table's order

Note that cost estimate functions must be written in C, not in SQL or any available procedural language, because they must access internal data structures of the planner/optimizer.

The index access costs should be computed using the parameters used by `src/backend/optimizer/path/costsize.c`: a sequential disk block fetch has cost `seq_page_cost`, a nonsequential fetch has cost `random_page_cost`, and the cost of processing one index row should usually be taken as `cpu_index_tuple_cost`. In addition, an appropriate multiple of `cpu_operator_cost` should be charged for any comparison operators invoked during index processing (especially evaluation of the indexquals themselves).

The access costs should include all disk and CPU costs associated with scanning the index itself, but *not* the costs of retrieving or processing the parent-table rows that are identified by the index.

The "start-up cost" is the part of the total scan cost that must be expended before we can begin to fetch the first row. For most indexes this can be taken as zero, but an index type with a high start-up cost might want to set it nonzero.

The `indexSelectivity` should be set to the estimated fraction of the parent table rows that will be retrieved during the index scan. In the case of a lossy query, this will typically be higher than the fraction of rows that actually pass the given qual conditions.

The `indexCorrelation` should be set to the correlation (ranging between -1.0 and 1.0) between the index order and the table order. This is used to adjust the estimate for the cost of fetching rows from the parent table.

When `loop_count` is greater than one, the returned numbers should be averages expected for any one scan of the index.

Cost Estimation

A typical cost estimator will proceed as follows:

1. Estimate and return the fraction of parent-table rows that will be visited based on the given qual conditions. In the absence of any index-type-specific knowledge, use the standard optimizer function `clauselist_selectivity()`:

   ```
   *indexSelectivity = clauselist_selectivity(root, path->indexquals,
                                              path->indexinfo->rel->relid,
                                              JOIN_INNER, NULL);
   ```

2. Estimate the number of index rows that will be visited during the scan. For many index types this is the same as `indexSelectivity` times the number of rows in the index, but it might be more. (Note that the index's size in pages and rows is available from the `path->indexinfo` struct.)

3. Estimate the number of index pages that will be retrieved during the scan. This might be just `indexSelectivity` times the index's size in pages.

4. Compute the index access cost. A generic estimator might do this:

   ```
   /*
    * Our generic assumption is that the index pages will be read
    * sequentially, so they cost seq_page_cost each, not random_page_cost.
    * Also, we charge for evaluation of the indexquals at each index row.
    * All the costs are assumed to be paid incrementally during the scan.
    */
   cost_qual_eval(&index_qual_cost, path->indexquals, root);
   ```

```
*indexStartupCost = index_qual_cost.startup;
*indexTotalCost = seq_page_cost * numIndexPages +
    (cpu_index_tuple_cost + index_qual_cost.per_tuple) * numIndexTuples;
```

However, the above does not account for amortization of index reads across repeated index scans.

5. Estimate the index correlation. For a simple ordered index on a single field, this can be retrieved from pg_statistic. If the correlation is not known, the conservative estimate is zero (no correlation).

Examples of cost estimator functions can be found in `src/backend/utils/adt/selfuncs.c`.

Chapter 59. GiST Indexes

59.1. Introduction

GiST stands for Generalized Search Tree. It is a balanced, tree-structured access method, that acts as a base template in which to implement arbitrary indexing schemes. B-trees, R-trees and many other indexing schemes can be implemented in GiST.

One advantage of GiST is that it allows the development of custom data types with the appropriate access methods, by an expert in the domain of the data type, rather than a database expert.

Some of the information here is derived from the University of California at Berkeley's GiST Indexing Project web site[1] and Marcel Kornacker's thesis, Access Methods for Next-Generation Database Systems[2]. The GiST implementation in PostgreSQL is primarily maintained by Teodor Sigaev and Oleg Bartunov, and there is more information on their web site[3].

59.2. Built-in Operator Classes

The core PostgreSQL distribution includes the GiST operator classes shown in Table 59-1. (Some of the optional modules described in Appendix F provide additional GiST operator classes.)

Table 59-1. Built-in GiST Operator Classes

Name	Indexed Data Type	Indexable Operators	Ordering Operators
box_ops	box	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~=	
circle_ops	circle	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~=	<->
inet_ops	inet, cidr	&& >> >>= > >= <> << <<= < <= =	
point_ops	point	>> >^ << <@ <@ <@ <^ ~=	<->
poly_ops	polygon	&& &> &< &<| >> << <<| <@ @> @ |&> |>> ~ ~=	<->

1. http://gist.cs.berkeley.edu/
2. http://www.sai.msu.su/~megera/postgres/gist/papers/concurrency/access-methods-for-next-generation.pdf.gz
3. http://www.sai.msu.su/~megera/postgres/gist/

Name	Indexed Data Type	Indexable Operators	Ordering Operators
range_ops	any range type	&& &> &< >> << <@ -|- = @> @>	
tsquery_ops	tsquery	<@ @>	
tsvector_ops	tsvector	@@	

For historical reasons, the inet_ops operator class is not the default class for types inet and cidr. To use it, mention the class name in CREATE INDEX, for example

```
CREATE INDEX ON my_table USING GIST (my_inet_column inet_ops);
```

59.3. Extensibility

Traditionally, implementing a new index access method meant a lot of difficult work. It was necessary to understand the inner workings of the database, such as the lock manager and Write-Ahead Log. The GiST interface has a high level of abstraction, requiring the access method implementer only to implement the semantics of the data type being accessed. The GiST layer itself takes care of concurrency, logging and searching the tree structure.

This extensibility should not be confused with the extensibility of the other standard search trees in terms of the data they can handle. For example, PostgreSQL supports extensible B-trees and hash indexes. That means that you can use PostgreSQL to build a B-tree or hash over any data type you want. But B-trees only support range predicates (<, =, >), and hash indexes only support equality queries.

So if you index, say, an image collection with a PostgreSQL B-tree, you can only issue queries such as "is imagex equal to imagey", "is imagex less than imagey" and "is imagex greater than imagey". Depending on how you define "equals", "less than" and "greater than" in this context, this could be useful. However, by using a GiST based index, you could create ways to ask domain-specific questions, perhaps "find all images of horses" or "find all over-exposed images".

All it takes to get a GiST access method up and running is to implement several user-defined methods, which define the behavior of keys in the tree. Of course these methods have to be pretty fancy to support fancy queries, but for all the standard queries (B-trees, R-trees, etc.) they're relatively straightforward. In short, GiST combines extensibility along with generality, code reuse, and a clean interface.

There are seven methods that an index operator class for GiST must provide, and two that are optional. Correctness of the index is ensured by proper implementation of the same, consistent and union methods, while efficiency (size and speed) of the index will depend on the penalty and picksplit methods. The remaining two basic methods are compress and decompress, which allow an index to have internal tree data of a different type than the data it indexes. The leaves are to be of the indexed data type, while the other tree nodes can be of any C struct (but you still have to follow PostgreSQL data type rules here, see about varlena for variable sized data). If the tree's internal data type exists at the SQL level, the STORAGE option of the CREATE OPERATOR CLASS command can be used. The optional eighth method is distance, which is needed if the operator class wishes to support ordered scans (nearest-neighbor searches). The optional ninth method fetch is needed if the operator class wishes to support index-only scans.

`consistent`

Given an index entry `p` and a query value `q`, this function determines whether the index entry is "consistent" with the query; that is, could the predicate "*`indexed_column indexable_operator` q*" be true for any row represented by the index entry? For a leaf index entry this is equivalent to testing the indexable condition, while for an internal tree node this determines whether it is necessary to scan the subtree of the index represented by the tree node. When the result is `true`, a `recheck` flag must also be returned. This indicates whether the predicate is certainly true or only possibly true. If `recheck` = `false` then the index has tested the predicate condition exactly, whereas if `recheck` = `true` the row is only a candidate match. In that case the system will automatically evaluate the *`indexable_operator`* against the actual row value to see if it is really a match. This convention allows GiST to support both lossless and lossy index structures.

The SQL declaration of the function must look like this:

```
CREATE OR REPLACE FUNCTION my_consistent(internal, data_type, smallint, oid, internal)
RETURNS bool
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

And the matching code in the C module could then follow this skeleton:

```
PG_FUNCTION_INFO_V1(my_consistent);

Datum
my_consistent(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type  *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    bool       *recheck = (bool *) PG_GETARG_POINTER(4);
    data_type  *key = DatumGetDataType(entry->key);
    bool        retval;

    /*
     * determine return value as a function of strategy, key and query.
     *
     * Use GIST_LEAF(entry) to know where you're called in the index tree,
     * which comes handy when supporting the = operator for example (you could
     * check for non empty union() in non-leaf nodes and equality in leaf
     * nodes).
     */

    *recheck = true;        /* or false if check is exact */

    PG_RETURN_BOOL(retval);
}
```

Here, `key` is an element in the index and `query` the value being looked up in the index. The `StrategyNumber` parameter indicates which operator of your operator class is being applied — it matches one of the operator numbers in the `CREATE OPERATOR CLASS` command. Depending on what operators you have included in the class, the data type of `query` could vary with the operator, but the above skeleton assumes it doesn't.

`union`

This method consolidates information in the tree. Given a set of entries, this function generates a new index entry that represents all the given entries.

The SQL declaration of the function must look like this:

```
CREATE OR REPLACE FUNCTION my_union(internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

And the matching code in the C module could then follow this skeleton:

```
PG_FUNCTION_INFO_V1(my_union);

Datum
my_union(PG_FUNCTION_ARGS)
{
    GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
    GISTENTRY  *ent = entryvec->vector;
    data_type  *out,
               *tmp,
               *old;
    int         numranges,
                i = 0;

    numranges = entryvec->n;
    tmp = DatumGetDataType(ent[0].key);
    out = tmp;

    if (numranges == 1)
    {
        out = data_type_deep_copy(tmp);

        PG_RETURN_DATA_TYPE_P(out);
    }

    for (i = 1; i < numranges; i++)
    {
        old = out;
        tmp = DatumGetDataType(ent[i].key);
        out = my_union_implementation(out, tmp);
    }

    PG_RETURN_DATA_TYPE_P(out);
}
```

As you can see, in this skeleton we're dealing with a data type where `union(X, Y, Z) = union(union(X, Y), Z)`. It's easy enough to support data types where this is not the case, by implementing the proper union algorithm in this GiST support method.

The `union` implementation function should return a pointer to newly `palloc()`ed memory. You can't just return whatever the input is.

`compress`

Converts the data item into a format suitable for physical storage in an index page.

The SQL declaration of the function must look like this:

```
CREATE OR REPLACE FUNCTION my_compress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

And the matching code in the C module could then follow this skeleton:

```
PG_FUNCTION_INFO_V1(my_compress);

Datum
my_compress(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY  *retval;

    if (entry->leafkey)
    {
        /* replace entry->key with a compressed version */
        compressed_data_type *compressed_data = palloc(sizeof(compressed_data_type));

        /* fill *compressed_data from entry->key ... */

        retval = palloc(sizeof(GISTENTRY));
        gistentryinit(*retval, PointerGetDatum(compressed_data),
                      entry->rel, entry->page, entry->offset, FALSE);
    }
    else
    {
        /* typically we needn't do anything with non-leaf entries */
        retval = entry;
    }

    PG_RETURN_POINTER(retval);
}
```

You have to adapt *`compressed_data_type`* to the specific type you're converting to in order to compress your leaf nodes, of course.

`decompress`

The reverse of the `compress` method. Converts the index representation of the data item into a format that can be manipulated by the other GiST methods in the operator class.

The SQL declaration of the function must look like this:

```
CREATE OR REPLACE FUNCTION my_decompress(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

And the matching code in the C module could then follow this skeleton:

```
PG_FUNCTION_INFO_V1(my_decompress);
```

```
Datum
my_decompress(PG_FUNCTION_ARGS)
{
    PG_RETURN_POINTER(PG_GETARG_POINTER(0));
}
```

The above skeleton is suitable for the case where no decompression is needed.

`penalty`

Returns a value indicating the "cost" of inserting the new entry into a particular branch of the tree. Items will be inserted down the path of least `penalty` in the tree. Values returned by `penalty` should be non-negative. If a negative value is returned, it will be treated as zero.

The SQL declaration of the function must look like this:

```
CREATE OR REPLACE FUNCTION my_penalty(internal, internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;  -- in some cases penalty functions need not be strict
```

And the matching code in the C module could then follow this skeleton:

```
PG_FUNCTION_INFO_V1(my_penalty);

Datum
my_penalty(PG_FUNCTION_ARGS)
{
    GISTENTRY  *origentry = (GISTENTRY *) PG_GETARG_POINTER(0);
    GISTENTRY  *newentry = (GISTENTRY *) PG_GETARG_POINTER(1);
    float      *penalty = (float *) PG_GETARG_POINTER(2);
    data_type  *orig = DatumGetDataType(origentry->key);
    data_type  *new = DatumGetDataType(newentry->key);

    *penalty = my_penalty_implementation(orig, new);
    PG_RETURN_POINTER(penalty);
}
```

The `penalty` function is crucial to good performance of the index. It'll get used at insertion time to determine which branch to follow when choosing where to add the new entry in the tree. At query time, the more balanced the index, the quicker the lookup.

`picksplit`

When an index page split is necessary, this function decides which entries on the page are to stay on the old page, and which are to move to the new page.

The SQL declaration of the function must look like this:

```
CREATE OR REPLACE FUNCTION my_picksplit(internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

And the matching code in the C module could then follow this skeleton:

```
PG_FUNCTION_INFO_V1(my_picksplit);

Datum
my_picksplit(PG_FUNCTION_ARGS)
{
```

```
GistEntryVector *entryvec = (GistEntryVector *) PG_GETARG_POINTER(0);
OffsetNumber maxoff = entryvec->n - 1;
GISTENTRY  *ent = entryvec->vector;
GIST_SPLITVEC *v = (GIST_SPLITVEC *) PG_GETARG_POINTER(1);
int         i,
            nbytes;
OffsetNumber *left,
           *right;
data_type  *tmp_union;
data_type  *unionL;
data_type  *unionR;
GISTENTRY **raw_entryvec;

maxoff = entryvec->n - 1;
nbytes = (maxoff + 1) * sizeof(OffsetNumber);

v->spl_left = (OffsetNumber *) palloc(nbytes);
left = v->spl_left;
v->spl_nleft = 0;

v->spl_right = (OffsetNumber *) palloc(nbytes);
right = v->spl_right;
v->spl_nright = 0;

unionL = NULL;
unionR = NULL;

/* Initialize the raw entry vector. */
raw_entryvec = (GISTENTRY **) malloc(entryvec->n * sizeof(void *));
for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
    raw_entryvec[i] = &(entryvec->vector[i]);

for (i = FirstOffsetNumber; i <= maxoff; i = OffsetNumberNext(i))
{
    int         real_index = raw_entryvec[i] - entryvec->vector;

    tmp_union = DatumGetDataType(entryvec->vector[real_index].key);
    Assert(tmp_union != NULL);

    /*
     * Choose where to put the index entries and update unionL and unionR
     * accordingly. Append the entries to either v->spl_left or
     * v->spl_right, and care about the counters.
     */

    if (my_choice_is_left(unionL, curl, unionR, curr))
    {
        if (unionL == NULL)
            unionL = tmp_union;
        else
            unionL = my_union_implementation(unionL, tmp_union);

        *left = real_index;
```

```
                ++left;
                ++(v->spl_nleft);
            }
            else
            {
                /*
                 * Same on the right
                 */
            }
        }

        v->spl_ldatum = DataTypeGetDatum(unionL);
        v->spl_rdatum = DataTypeGetDatum(unionR);
        PG_RETURN_POINTER(v);
    }
```

Like `penalty`, the `picksplit` function is crucial to good performance of the index. Designing suitable `penalty` and `picksplit` implementations is where the challenge of implementing well-performing GiST indexes lies.

`same`

Returns true if two index entries are identical, false otherwise.

The SQL declaration of the function must look like this:

```
CREATE OR REPLACE FUNCTION my_same(internal, internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

And the matching code in the C module could then follow this skeleton:

```
PG_FUNCTION_INFO_V1(my_same);

Datum
my_same(PG_FUNCTION_ARGS)
{
    prefix_range *v1 = PG_GETARG_PREFIX_RANGE_P(0);
    prefix_range *v2 = PG_GETARG_PREFIX_RANGE_P(1);
    bool       *result = (bool *) PG_GETARG_POINTER(2);

    *result = my_eq(v1, v2);
    PG_RETURN_POINTER(result);
}
```

For historical reasons, the `same` function doesn't just return a Boolean result; instead it has to store the flag at the location indicated by the third argument.

`distance`

Given an index entry `p` and a query value `q`, this function determines the index entry's "distance" from the query value. This function must be supplied if the operator class contains any ordering operators. A query using the ordering operator will be implemented by returning index entries with the smallest "distance" values first, so the results must be consistent with the operator's semantics. For a leaf index entry the result just represents the distance to the index entry; for an internal tree node, the result must be the smallest distance that any child entry could have.

The SQL declaration of the function must look like this:

```
CREATE OR REPLACE FUNCTION my_distance(internal, data_type, smallint, oid)
RETURNS float8
AS 'MODULE_PATHNAME'
LANGUAGE C STRICT;
```

And the matching code in the C module could then follow this skeleton:

```
PG_FUNCTION_INFO_V1(my_distance);

Datum
my_distance(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    data_type  *query = PG_GETARG_DATA_TYPE_P(1);
    StrategyNumber strategy = (StrategyNumber) PG_GETARG_UINT16(2);
    /* Oid subtype = PG_GETARG_OID(3); */
    /* bool *recheck = (bool *) PG_GETARG_POINTER(4); */
    data_type  *key = DatumGetDataType(entry->key);
    double      retval;

    /*
     * determine return value as a function of strategy, key and query.
     */

    PG_RETURN_FLOAT8(retval);
}
```

The arguments to the `distance` function are identical to the arguments of the `consistent` function.

Some approximation is allowed when determining the distance, so long as the result is never greater than the entry's actual distance. Thus, for example, distance to a bounding box is usually sufficient in geometric applications. For an internal tree node, the distance returned must not be greater than the distance to any of the child nodes. If the returned distance is not exact, the function must set `*recheck` to true. (This is not necessary for internal tree nodes; for them, the calculation is always assumed to be inexact.) In this case the executor will calculate the accurate distance after fetching the tuple from the heap, and reorder the tuples if necessary.

If the distance function returns `*recheck = true` for any leaf node, the original ordering operator's return type must be `float8` or `float4`, and the distance function's result values must be comparable to those of the original ordering operator, since the executor will sort using both distance function results and recalculated ordering-operator results. Otherwise, the distance function's result values can be any finite `float8` values, so long as the relative order of the result values matches the order returned by the ordering operator. (Infinity and minus infinity are used internally to handle cases such as nulls, so it is not recommended that `distance` functions return these values.)

`fetch`

Converts the compressed index representation of the data item into the original data type, for index-only scans. The returned data must be an exact, non-lossy copy of the originally indexed value.

The SQL declaration of the function must look like this:

```
CREATE OR REPLACE FUNCTION my_fetch(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
```

```
LANGUAGE C STRICT;
```

The argument is a pointer to a `GISTENTRY` struct. On entry, its 'key' field contains a non-NULL leaf datum in its compressed form. The return value is another `GISTENTRY` struct, whose 'key' field contains the same datum in the original, uncompressed form. If the opclass' compress function does nothing for leaf entries, the fetch method can return the argument as is.

The matching code in the C module could then follow this skeleton:

```
PG_FUNCTION_INFO_V1(my_fetch);

Datum
my_fetch(PG_FUNCTION_ARGS)
{
    GISTENTRY  *entry = (GISTENTRY *) PG_GETARG_POINTER(0);
    input_data_type *in = DatumGetPointer(entry->key);
    fetched_data_type *fetched_data;
    GISTENTRY  *retval;

    retval = palloc(sizeof(GISTENTRY));
    fetched_data = palloc(sizeof(fetched_data_type));

    /*
     * Convert 'fetched_data' into the a Datum of the original datatype.
     */

    /* fill *retval from fetched_data. */
    gistentryinit(*retval, PointerGetDatum(converted_datum),
                  entry->rel, entry->page, entry->offset, FALSE);

    PG_RETURN_POINTER(retval);
}
```

If the compress method is lossy for leaf entries, the operator class cannot support index-only scans, and must not define a 'fetch' function.

All the GiST support methods are normally called in short-lived memory contexts; that is, `CurrentMemoryContext` will get reset after each tuple is processed. It is therefore not very important to worry about pfree'ing everything you palloc. However, in some cases it's useful for a support method to cache data across repeated calls. To do that, allocate the longer-lived data in `fcinfo->flinfo->fn_mcxt`, and keep a pointer to it in `fcinfo->flinfo->fn_extra`. Such data will survive for the life of the index operation (e.g., a single GiST index scan, index build, or index tuple insertion). Be careful to pfree the previous value when replacing a `fn_extra` value, or the leak will accumulate for the duration of the operation.

59.4. Implementation

59.4.1. GiST buffering build

Building large GiST indexes by simply inserting all the tuples tends to be slow, because if the index tuples are scattered across the index and the index is large enough to not fit in cache, the insertions need to

perform a lot of random I/O. Beginning in version 9.2, PostgreSQL supports a more efficient method to build GiST indexes based on buffering, which can dramatically reduce the number of random I/Os needed for non-ordered data sets. For well-ordered data sets the benefit is smaller or non-existent, because only a small number of pages receive new tuples at a time, and those pages fit in cache even if the index as whole does not.

However, buffering index build needs to call the `penalty` function more often, which consumes some extra CPU resources. Also, the buffers used in the buffering build need temporary disk space, up to the size of the resulting index. Buffering can also influence the quality of the resulting index, in both positive and negative directions. That influence depends on various factors, like the distribution of the input data and the operator class implementation.

By default, a GiST index build switches to the buffering method when the index size reaches effective_cache_size. It can be manually turned on or off by the `buffering` parameter to the CREATE INDEX command. The default behavior is good for most cases, but turning buffering off might speed up the build somewhat if the input data is ordered.

59.5. Examples

The PostgreSQL source distribution includes several examples of index methods implemented using GiST. The core system currently provides text search support (indexing for `tsvector` and `tsquery`) as well as R-Tree equivalent functionality for some of the built-in geometric data types (see `src/backend/access/gist/gistproc.c`). The following `contrib` modules also contain GiST operator classes:

`btree_gist`

B-tree equivalent functionality for several data types

`cube`

Indexing for multidimensional cubes

`hstore`

Module for storing (key, value) pairs

`intarray`

RD-Tree for one-dimensional array of int4 values

`ltree`

Indexing for tree-like structures

`pg_trgm`

Text similarity using trigram matching

`seg`

Indexing for “float ranges”

Chapter 60. SP-GiST Indexes

60.1. Introduction

SP-GiST is an abbreviation for space-partitioned GiST. SP-GiST supports partitioned search trees, which facilitate development of a wide range of different non-balanced data structures, such as quad-trees, k-d trees, and radix trees (tries). The common feature of these structures is that they repeatedly divide the search space into partitions that need not be of equal size. Searches that are well matched to the partitioning rule can be very fast.

These popular data structures were originally developed for in-memory usage. In main memory, they are usually designed as a set of dynamically allocated nodes linked by pointers. This is not suitable for direct storing on disk, since these chains of pointers can be rather long which would require too many disk accesses. In contrast, disk-based data structures should have a high fanout to minimize I/O. The challenge addressed by SP-GiST is to map search tree nodes to disk pages in such a way that a search need access only a few disk pages, even if it traverses many nodes.

Like GiST, SP-GiST is meant to allow the development of custom data types with the appropriate access methods, by an expert in the domain of the data type, rather than a database expert.

Some of the information here is derived from Purdue University's SP-GiST Indexing Project web site[1]. The SP-GiST implementation in PostgreSQL is primarily maintained by Teodor Sigaev and Oleg Bartunov, and there is more information on their web site[2].

60.2. Built-in Operator Classes

The core PostgreSQL distribution includes the SP-GiST operator classes shown in Table 60-1.

Table 60-1. Built-in SP-GiST Operator Classes

Name	Indexed Data Type	Indexable Operators
kd_point_ops	point	<< <@ <^ >> >^ ~=
quad_point_ops	point	<< <@ <^ >> >^ ~=
range_ops	any range type	&& &< &> -|- << <@ = >> @>
text_ops	text	< <= = > >= ~<=~ ~<~ ~>=~ ~>~

Of the two operator classes for type point, quad_point_ops is the default. kd_point_ops supports the same operators but uses a different index data structure that may offer better performance in some

1. http://www.cs.purdue.edu/spgist/
2. http://www.sai.msu.su/~megera/wiki/spgist_dev

applications.

60.3. Extensibility

SP-GiST offers an interface with a high level of abstraction, requiring the access method developer to implement only methods specific to a given data type. The SP-GiST core is responsible for efficient disk mapping and searching the tree structure. It also takes care of concurrency and logging considerations.

Leaf tuples of an SP-GiST tree contain values of the same data type as the indexed column. Leaf tuples at the root level will always contain the original indexed data value, but leaf tuples at lower levels might contain only a compressed representation, such as a suffix. In that case the operator class support functions must be able to reconstruct the original value using information accumulated from the inner tuples that are passed through to reach the leaf level.

Inner tuples are more complex, since they are branching points in the search tree. Each inner tuple contains a set of one or more *nodes*, which represent groups of similar leaf values. A node contains a downlink that leads to either another, lower-level inner tuple, or a short list of leaf tuples that all lie on the same index page. Each node has a *label* that describes it; for example, in a radix tree the node label could be the next character of the string value. Optionally, an inner tuple can have a *prefix* value that describes all its members. In a radix tree this could be the common prefix of the represented strings. The prefix value is not necessarily really a prefix, but can be any data needed by the operator class; for example, in a quad-tree it can store the central point that the four quadrants are measured with respect to. A quad-tree inner tuple would then also contain four nodes corresponding to the quadrants around this central point.

Some tree algorithms require knowledge of level (or depth) of the current tuple, so the SP-GiST core provides the possibility for operator classes to manage level counting while descending the tree. There is also support for incrementally reconstructing the represented value when that is needed.

> **Note:** The SP-GiST core code takes care of null entries. Although SP-GiST indexes do store entries for nulls in indexed columns, this is hidden from the index operator class code: no null index entries or search conditions will ever be passed to the operator class methods. (It is assumed that SP-GiST operators are strict and so cannot succeed for null values.) Null values are therefore not discussed further here.

There are five user-defined methods that an index operator class for SP-GiST must provide. All five follow the convention of accepting two `internal` arguments, the first of which is a pointer to a C struct containing input values for the support method, while the second argument is a pointer to a C struct where output values must be placed. Four of the methods just return `void`, since all their results appear in the output struct; but `leaf_consistent` additionally returns a `boolean` result. The methods must not modify any fields of their input structs. In all cases, the output struct is initialized to zeroes before calling the user-defined method.

The five user-defined methods are:

`config`

Returns static information about the index implementation, including the data type OIDs of the prefix and node label data types.

The SQL declaration of the function must look like this:

```
CREATE FUNCTION my_config(internal, internal) RETURNS void ...
```

The first argument is a pointer to a `spgConfigIn` C struct, containing input data for the function. The second argument is a pointer to a `spgConfigOut` C struct, which the function must fill with result data.

```
typedef struct spgConfigIn
{
    Oid         attType;        /* Data type to be indexed */
} spgConfigIn;

typedef struct spgConfigOut
{
    Oid         prefixType;     /* Data type of inner-tuple prefixes */
    Oid         labelType;      /* Data type of inner-tuple node labels */
    bool        canReturnData;  /* Opclass can reconstruct original data */
    bool        longValuesOK;   /* Opclass can cope with values > 1 page */
} spgConfigOut;
```

`attType` is passed in order to support polymorphic index operator classes; for ordinary fixed-data-type operator classes, it will always have the same value and so can be ignored.

For operator classes that do not use prefixes, `prefixType` can be set to `VOIDOID`. Likewise, for operator classes that do not use node labels, `labelType` can be set to `VOIDOID`. `canReturnData` should be set true if the operator class is capable of reconstructing the originally-supplied index value. `longValuesOK` should be set true only when the `attType` is of variable length and the operator class is capable of segmenting long values by repeated suffixing (see Section 60.4.1).

`choose`

Chooses a method for inserting a new value into an inner tuple.

The SQL declaration of the function must look like this:

```
CREATE FUNCTION my_choose(internal, internal) RETURNS void ...
```

The first argument is a pointer to a `spgChooseIn` C struct, containing input data for the function. The second argument is a pointer to a `spgChooseOut` C struct, which the function must fill with result data.

```
typedef struct spgChooseIn
{
    Datum       datum;          /* original datum to be indexed */
    Datum       leafDatum;      /* current datum to be stored at leaf */
    int         level;          /* current level (counting from zero) */

    /* Data from current inner tuple */
    bool        allTheSame;     /* tuple is marked all-the-same? */
    bool        hasPrefix;      /* tuple has a prefix? */
    Datum       prefixDatum;    /* if so, the prefix value */
    int         nNodes;         /* number of nodes in the inner tuple */
    Datum      *nodeLabels;     /* node label values (NULL if none) */
} spgChooseIn;

typedef enum spgChooseResultType
{
    spgMatchNode = 1,           /* descend into existing node */
```

```
    spgAddNode,                 /* add a node to the inner tuple */
    spgSplitTuple               /* split inner tuple (change its prefix) */
} spgChooseResultType;

typedef struct spgChooseOut
{
    spgChooseResultType resultType;     /* action code, see above */
    union
    {
        struct                  /* results for spgMatchNode */
        {
            int         nodeN;      /* descend to this node (index from 0) */
            int         levelAdd;   /* increment level by this much */
            Datum       restDatum;  /* new leaf datum */
        }           matchNode;
        struct                  /* results for spgAddNode */
        {
            Datum       nodeLabel;  /* new node's label */
            int         nodeN;      /* where to insert it (index from 0) */
        }           addNode;
        struct                  /* results for spgSplitTuple */
        {
            /* Info to form new inner tuple with one node */
            bool        prefixHasPrefix;    /* tuple should have a prefix? */
            Datum       prefixPrefixDatum;  /* if so, its value */
            Datum       nodeLabel;          /* node's label */

            /* Info to form new lower-level inner tuple with all old nodes */
            bool        postfixHasPrefix;   /* tuple should have a prefix? */
            Datum       postfixPrefixDatum; /* if so, its value */
        }           splitTuple;
    }           result;
} spgChooseOut;
```

`datum` is the original datum that was to be inserted into the index. `leafDatum` is initially the same as `datum`, but can change at lower levels of the tree if the `choose` or `picksplit` methods change it. When the insertion search reaches a leaf page, the current value of `leafDatum` is what will be stored in the newly created leaf tuple. `level` is the current inner tuple's level, starting at zero for the root level. `allTheSame` is true if the current inner tuple is marked as containing multiple equivalent nodes (see Section 60.4.3). `hasPrefix` is true if the current inner tuple contains a prefix; if so, `prefixDatum` is its value. `nNodes` is the number of child nodes contained in the inner tuple, and `nodeLabels` is an array of their label values, or NULL if there are no labels.

The `choose` function can determine either that the new value matches one of the existing child nodes, or that a new child node must be added, or that the new value is inconsistent with the tuple prefix and so the inner tuple must be split to create a less restrictive prefix.

If the new value matches one of the existing child nodes, set `resultType` to `spgMatchNode`. Set `nodeN` to the index (from zero) of that node in the node array. Set `levelAdd` to the increment in `level` caused by descending through that node, or leave it as zero if the operator class does not use levels. Set `restDatum` to equal `datum` if the operator class does not modify datums from one level to the next, or otherwise set it to the modified value to be used as `leafDatum` at the next level.

If a new child node must be added, set `resultType` to `spgAddNode`. Set `nodeLabel` to the label to be used for the new node, and set `nodeN` to the index (from zero) at which to insert the node in the node array. After the node has been added, the `choose` function will be called again with the modified inner tuple; that call should result in an `spgMatchNode` result.

If the new value is inconsistent with the tuple prefix, set `resultType` to `spgSplitTuple`. This action moves all the existing nodes into a new lower-level inner tuple, and replaces the existing inner tuple with a tuple having a single node that links to the new lower-level inner tuple. Set `prefixHasPrefix` to indicate whether the new upper tuple should have a prefix, and if so set `prefixPrefixDatum` to the prefix value. This new prefix value must be sufficiently less restrictive than the original to accept the new value to be indexed, and it should be no longer than the original prefix. Set `nodeLabel` to the label to be used for the node that will point to the new lower-level inner tuple. Set `postfixHasPrefix` to indicate whether the new lower-level inner tuple should have a prefix, and if so set `postfixPrefixDatum` to the prefix value. The combination of these two prefixes and the additional label must have the same meaning as the original prefix, because there is no opportunity to alter the node labels that are moved to the new lower-level tuple, nor to change any child index entries. After the node has been split, the `choose` function will be called again with the replacement inner tuple. That call will usually result in an `spgAddNode` result, since presumably the node label added in the split step will not match the new value; so after that, there will be a third call that finally returns `spgMatchNode` and allows the insertion to descend to the leaf level.

`picksplit`

Decides how to create a new inner tuple over a set of leaf tuples.

The SQL declaration of the function must look like this:

```
CREATE FUNCTION my_picksplit(internal, internal) RETURNS void ...
```

The first argument is a pointer to a `spgPickSplitIn` C struct, containing input data for the function. The second argument is a pointer to a `spgPickSplitOut` C struct, which the function must fill with result data.

```
typedef struct spgPickSplitIn
{
    int          nTuples;        /* number of leaf tuples */
    Datum       *datums;         /* their datums (array of length nTuples) */
    int          level;          /* current level (counting from zero) */
} spgPickSplitIn;

typedef struct spgPickSplitOut
{
    bool         hasPrefix;      /* new inner tuple should have a prefix? */
    Datum        prefixDatum;    /* if so, its value */

    int          nNodes;         /* number of nodes for new inner tuple */
    Datum       *nodeLabels;     /* their labels (or NULL for no labels) */

    int         *mapTuplesToNodes;   /* node index for each leaf tuple */
    Datum       *leafTupleDatums;    /* datum to store in each new leaf tuple */
} spgPickSplitOut;
```

`nTuples` is the number of leaf tuples provided. `datums` is an array of their datum values. `level` is the current level that all the leaf tuples share, which will become the level of the new inner tuple.

Set `hasPrefix` to indicate whether the new inner tuple should have a prefix, and if so set `prefixDatum` to the prefix value. Set `nNodes` to indicate the number of nodes that the new inner tuple will contain, and set `nodeLabels` to an array of their label values. (If the nodes do not require labels, set `nodeLabels` to NULL; see Section 60.4.2 for details.) Set `mapTuplesToNodes` to an array that gives the index (from zero) of the node that each leaf tuple should be assigned to. Set `leafTupleDatums` to an array of the values to be stored in the new leaf tuples (these will be the same as the input `datums` if the operator class does not modify datums from one level to the next). Note that the `picksplit` function is responsible for palloc'ing the `nodeLabels`, `mapTuplesToNodes` and `leafTupleDatums` arrays.

If more than one leaf tuple is supplied, it is expected that the `picksplit` function will classify them into more than one node; otherwise it is not possible to split the leaf tuples across multiple pages, which is the ultimate purpose of this operation. Therefore, if the `picksplit` function ends up placing all the leaf tuples in the same node, the core SP-GiST code will override that decision and generate an inner tuple in which the leaf tuples are assigned at random to several identically-labeled nodes. Such a tuple is marked `allTheSame` to signify that this has happened. The `choose` and `inner_consistent` functions must take suitable care with such inner tuples. See Section 60.4.3 for more information.

`picksplit` can be applied to a single leaf tuple only in the case that the `config` function set `longValuesOK` to true and a larger-than-a-page input value has been supplied. In this case the point of the operation is to strip off a prefix and produce a new, shorter leaf datum value. The call will be repeated until a leaf datum short enough to fit on a page has been produced. See Section 60.4.1 for more information.

`inner_consistent`

Returns set of nodes (branches) to follow during tree search.

The SQL declaration of the function must look like this:

```
CREATE FUNCTION my_inner_consistent(internal, internal) RETURNS void ...
```

The first argument is a pointer to a `spgInnerConsistentIn` C struct, containing input data for the function. The second argument is a pointer to a `spgInnerConsistentOut` C struct, which the function must fill with result data.

```
typedef struct spgInnerConsistentIn
{
    ScanKey     scankeys;       /* array of operators and comparison values */
    int         nkeys;          /* length of array */

    Datum       reconstructedValue;     /* value reconstructed at parent */
    int         level;          /* current level (counting from zero) */
    bool        returnData;     /* original data must be returned? */

    /* Data from current inner tuple */
    bool        allTheSame;     /* tuple is marked all-the-same? */
    bool        hasPrefix;      /* tuple has a prefix? */
    Datum       prefixDatum;    /* if so, the prefix value */
    int         nNodes;         /* number of nodes in the inner tuple */
    Datum      *nodeLabels;     /* node label values (NULL if none) */
} spgInnerConsistentIn;

typedef struct spgInnerConsistentOut
```

```
{
    int         nNodes;         /* number of child nodes to be visited */
    int        *nodeNumbers;    /* their indexes in the node array */
    int        *levelAdds;      /* increment level by this much for each */
    Datum      *reconstructedValues;    /* associated reconstructed values */
} spgInnerConsistentOut;
```

The array `scankeys`, of length `nkeys`, describes the index search condition(s). These conditions are combined with AND — only index entries that satisfy all of them are interesting. (Note that `nkeys` = 0 implies that all index entries satisfy the query.) Usually the consistent function only cares about the `sk_strategy` and `sk_argument` fields of each array entry, which respectively give the indexable operator and comparison value. In particular it is not necessary to check `sk_flags` to see if the comparison value is NULL, because the SP-GiST core code will filter out such conditions. `reconstructedValue` is the value reconstructed for the parent tuple; it is `(Datum) 0` at the root level or if the `inner_consistent` function did not provide a value at the parent level. `level` is the current inner tuple's level, starting at zero for the root level. `returnData` is `true` if reconstructed data is required for this query; this will only be so if the `config` function asserted `canReturnData`. `allTheSame` is true if the current inner tuple is marked "all-the-same"; in this case all the nodes have the same label (if any) and so either all or none of them match the query (see Section 60.4.3). `hasPrefix` is true if the current inner tuple contains a prefix; if so, `prefixDatum` is its value. `nNodes` is the number of child nodes contained in the inner tuple, and `nodeLabels` is an array of their label values, or NULL if the nodes do not have labels.

`nNodes` must be set to the number of child nodes that need to be visited by the search, and `nodeNumbers` must be set to an array of their indexes. If the operator class keeps track of levels, set `levelAdds` to an array of the level increments required when descending to each node to be visited. (Often these increments will be the same for all the nodes, but that's not necessarily so, so an array is used.) If value reconstruction is needed, set `reconstructedValues` to an array of the values reconstructed for each child node to be visited; otherwise, leave `reconstructedValues` as NULL. Note that the `inner_consistent` function is responsible for palloc'ing the `nodeNumbers`, `levelAdds` and `reconstructedValues` arrays.

`leaf_consistent`

Returns true if a leaf tuple satisfies a query.

The SQL declaration of the function must look like this:

```
CREATE FUNCTION my_leaf_consistent(internal, internal) RETURNS bool ...
```

The first argument is a pointer to a `spgLeafConsistentIn` C struct, containing input data for the function. The second argument is a pointer to a `spgLeafConsistentOut` C struct, which the function must fill with result data.

```
typedef struct spgLeafConsistentIn
{
    ScanKey     scankeys;       /* array of operators and comparison values */
    int         nkeys;          /* length of array */

    Datum       reconstructedValue;     /* value reconstructed at parent */
    int         level;          /* current level (counting from zero) */
    bool        returnData;     /* original data must be returned? */

    Datum       leafDatum;      /* datum in leaf tuple */
} spgLeafConsistentIn;
```

```
typedef struct spgLeafConsistentOut
{
    Datum       leafValue;      /* reconstructed original data, if any */
    bool        recheck;        /* set true if operator must be rechecked */
} spgLeafConsistentOut;
```

The array `scankeys`, of length `nkeys`, describes the index search condition(s). These conditions are combined with AND — only index entries that satisfy all of them satisfy the query. (Note that `nkeys` = 0 implies that all index entries satisfy the query.) Usually the consistent function only cares about the `sk_strategy` and `sk_argument` fields of each array entry, which respectively give the indexable operator and comparison value. In particular it is not necessary to check `sk_flags` to see if the comparison value is NULL, because the SP-GiST core code will filter out such conditions. `reconstructedValue` is the value reconstructed for the parent tuple; it is `(Datum) 0` at the root level or if the `inner_consistent` function did not provide a value at the parent level. `level` is the current leaf tuple's level, starting at zero for the root level. `returnData` is `true` if reconstructed data is required for this query; this will only be so if the `config` function asserted `canReturnData`. `leafDatum` is the key value stored in the current leaf tuple.

The function must return `true` if the leaf tuple matches the query, or `false` if not. In the `true` case, if `returnData` is `true` then `leafValue` must be set to the value originally supplied to be indexed for this leaf tuple. Also, `recheck` may be set to `true` if the match is uncertain and so the operator(s) must be re-applied to the actual heap tuple to verify the match.

All the SP-GiST support methods are normally called in a short-lived memory context; that is, `CurrentMemoryContext` will be reset after processing of each tuple. It is therefore not very important to worry about pfree'ing everything you palloc. (The `config` method is an exception: it should try to avoid leaking memory. But usually the `config` method need do nothing but assign constants into the passed parameter struct.)

If the indexed column is of a collatable data type, the index collation will be passed to all the support methods, using the standard `PG_GET_COLLATION()` mechanism.

60.4. Implementation

This section covers implementation details and other tricks that are useful for implementers of SP-GiST operator classes to know.

60.4.1. SP-GiST Limits

Individual leaf tuples and inner tuples must fit on a single index page (8KB by default). Therefore, when indexing values of variable-length data types, long values can only be supported by methods such as radix trees, in which each level of the tree includes a prefix that is short enough to fit on a page, and the final leaf level includes a suffix also short enough to fit on a page. The operator class should set `longValuesOK` to TRUE only if it is prepared to arrange for this to happen. Otherwise, the SP-GiST core will reject any request to index a value that is too large to fit on an index page.

Likewise, it is the operator class's responsibility that inner tuples do not grow too large to fit on an index page; this limits the number of child nodes that can be used in one inner tuple, as well as the maximum size of a prefix value.

Another limitation is that when an inner tuple's node points to a set of leaf tuples, those tuples must all be in the same index page. (This is a design decision to reduce seeking and save space in the links that chain such tuples together.) If the set of leaf tuples grows too large for a page, a split is performed and an intermediate inner tuple is inserted. For this to fix the problem, the new inner tuple *must* divide the set of leaf values into more than one node group. If the operator class's `picksplit` function fails to do that, the SP-GiST core resorts to extraordinary measures described in Section 60.4.3.

60.4.2. SP-GiST Without Node Labels

Some tree algorithms use a fixed set of nodes for each inner tuple; for example, in a quad-tree there are always exactly four nodes corresponding to the four quadrants around the inner tuple's centroid point. In such a case the code typically works with the nodes by number, and there is no need for explicit node labels. To suppress node labels (and thereby save some space), the `picksplit` function can return NULL for the `nodeLabels` array. This will in turn result in `nodeLabels` being NULL during subsequent calls to `choose` and `inner_consistent`. In principle, node labels could be used for some inner tuples and omitted for others in the same index.

When working with an inner tuple having unlabeled nodes, it is an error for `choose` to return `spgAddNode`, since the set of nodes is supposed to be fixed in such cases. Also, there is no provision for generating an unlabeled node in `spgSplitTuple` actions, since it is expected that an `spgAddNode` action will be needed as well.

60.4.3. "All-the-same" Inner Tuples

The SP-GiST core can override the results of the operator class's `picksplit` function when `picksplit` fails to divide the supplied leaf values into at least two node categories. When this happens, the new inner tuple is created with multiple nodes that each have the same label (if any) that `picksplit` gave to the one node it did use, and the leaf values are divided at random among these equivalent nodes. The `allTheSame` flag is set on the inner tuple to warn the `choose` and `inner_consistent` functions that the tuple does not have the node set that they might otherwise expect.

When dealing with an `allTheSame` tuple, a `choose` result of `spgMatchNode` is interpreted to mean that the new value can be assigned to any of the equivalent nodes; the core code will ignore the supplied `nodeN` value and descend into one of the nodes at random (so as to keep the tree balanced). It is an error for `choose` to return `spgAddNode`, since that would make the nodes not all equivalent; the `spgSplitTuple` action must be used if the value to be inserted doesn't match the existing nodes.

When dealing with an `allTheSame` tuple, the `inner_consistent` function should return either all or none of the nodes as targets for continuing the index search, since they are all equivalent. This may or may not require any special-case code, depending on how much the `inner_consistent` function normally assumes about the meaning of the nodes.

60.5. Examples

The PostgreSQL source distribution includes several examples of index operator classes for SP-GiST. The

core system currently provides radix trees over text columns and two types of trees over points: quad-tree and k-d tree. Look into `src/backend/access/spgist/` to see the code.

Chapter 61. GIN Indexes

61.1. Introduction

GIN stands for Generalized Inverted Index. GIN is designed for handling cases where the items to be indexed are composite values, and the queries to be handled by the index need to search for element values that appear within the composite items. For example, the items could be documents, and the queries could be searches for documents containing specific words.

We use the word *item* to refer to a composite value that is to be indexed, and the word *key* to refer to an element value. GIN always stores and searches for keys, not item values per se.

A GIN index stores a set of (key, posting list) pairs, where a *posting list* is a set of row IDs in which the key occurs. The same row ID can appear in multiple posting lists, since an item can contain more than one key. Each key value is stored only once, so a GIN index is very compact for cases where the same key appears many times.

GIN is generalized in the sense that the GIN access method code does not need to know the specific operations that it accelerates. Instead, it uses custom strategies defined for particular data types. The strategy defines how keys are extracted from indexed items and query conditions, and how to determine whether a row that contains some of the key values in a query actually satisfies the query.

One advantage of GIN is that it allows the development of custom data types with the appropriate access methods, by an expert in the domain of the data type, rather than a database expert. This is much the same advantage as using GiST.

The GIN implementation in PostgreSQL is primarily maintained by Teodor Sigaev and Oleg Bartunov. There is more information about GIN on their website[1].

61.2. Built-in Operator Classes

The core PostgreSQL distribution includes the GIN operator classes shown in Table 61-1. (Some of the optional modules described in Appendix F provide additional GIN operator classes.)

Table 61-1. Built-in GIN Operator Classes

Name	Indexed Data Type	Indexable Operators
_abstime_ops	abstime[]	&& <@ = @>
_bit_ops	bit[]	&& <@ = @>
_bool_ops	boolean[]	&& <@ = @>
_bpchar_ops	character[]	&& <@ = @>
_bytea_ops	bytea[]	&& <@ = @>

1. http://www.sai.msu.su/~megera/wiki/Gin

Name	Indexed Data Type	Indexable Operators
_char_ops	"char"[]	&& <@ = @>
_cidr_ops	cidr[]	&& <@ = @>
_date_ops	date[]	&& <@ = @>
_float4_ops	float4[]	&& <@ = @>
_float8_ops	float8[]	&& <@ = @>
_inet_ops	inet[]	&& <@ = @>
_int2_ops	smallint[]	&& <@ = @>
_int4_ops	integer[]	&& <@ = @>
_int8_ops	bigint[]	&& <@ = @>
_interval_ops	interval[]	&& <@ = @>
_macaddr_ops	macaddr[]	&& <@ = @>
_money_ops	money[]	&& <@ = @>
_name_ops	name[]	&& <@ = @>
_numeric_ops	numeric[]	&& <@ = @>
_oid_ops	oid[]	&& <@ = @>
_oidvector_ops	oidvector[]	&& <@ = @>
_reltime_ops	reltime[]	&& <@ = @>
_text_ops	text[]	&& <@ = @>
_time_ops	time[]	&& <@ = @>
_timestamp_ops	timestamp[]	&& <@ = @>
_timestamptz_ops	timestamp with time zone[]	&& <@ = @>
_timetz_ops	time with time zone[]	&& <@ = @>
_tinterval_ops	tinterval[]	&& <@ = @>
_varbit_ops	bit varying[]	&& <@ = @>
_varchar_ops	character varying[]	&& <@ = @>
jsonb_ops	jsonb	? ?& ?| @>
jsonb_path_ops	jsonb	@>
tsvector_ops	tsvector	@@ @@@

Of the two operator classes for type jsonb, jsonb_ops is the default. jsonb_path_ops supports fewer operators but offers better performance for those operators. See Section 8.14.4 for details.

61.3. Extensibility

The GIN interface has a high level of abstraction, requiring the access method implementer only to implement the semantics of the data type being accessed. The GIN layer itself takes care of concurrency, logging and searching the tree structure.

All it takes to get a GIN access method working is to implement a few user-defined methods, which define

the behavior of keys in the tree and the relationships between keys, indexed items, and indexable queries. In short, GIN combines extensibility with generality, code reuse, and a clean interface.

There are three methods that an operator class for GIN must provide:

`int compare(Datum a, Datum b)`

Compares two keys (not indexed items!) and returns an integer less than zero, zero, or greater than zero, indicating whether the first key is less than, equal to, or greater than the second. Null keys are never passed to this function.

`Datum *extractValue(Datum itemValue, int32 *nkeys, bool **nullFlags)`

Returns a palloc'd array of keys given an item to be indexed. The number of returned keys must be stored into `*nkeys`. If any of the keys can be null, also palloc an array of `*nkeys` `bool` fields, store its address at `*nullFlags`, and set these null flags as needed. `*nullFlags` can be left `NULL` (its initial value) if all keys are non-null. The return value can be `NULL` if the item contains no keys.

`Datum *extractQuery(Datum query, int32 *nkeys, StrategyNumber n, bool **pmatch, Pointer **extra_data, bool **nullFlags, int32 *searchMode)`

Returns a palloc'd array of keys given a value to be queried; that is, `query` is the value on the right-hand side of an indexable operator whose left-hand side is the indexed column. `n` is the strategy number of the operator within the operator class (see Section 35.14.2). Often, `extractQuery` will need to consult `n` to determine the data type of `query` and the method it should use to extract key values. The number of returned keys must be stored into `*nkeys`. If any of the keys can be null, also palloc an array of `*nkeys` `bool` fields, store its address at `*nullFlags`, and set these null flags as needed. `*nullFlags` can be left `NULL` (its initial value) if all keys are non-null. The return value can be `NULL` if the `query` contains no keys.

`searchMode` is an output argument that allows `extractQuery` to specify details about how the search will be done. If `*searchMode` is set to `GIN_SEARCH_MODE_DEFAULT` (which is the value it is initialized to before call), only items that match at least one of the returned keys are considered candidate matches. If `*searchMode` is set to `GIN_SEARCH_MODE_INCLUDE_EMPTY`, then in addition to items containing at least one matching key, items that contain no keys at all are considered candidate matches. (This mode is useful for implementing is-subset-of operators, for example.) If `*searchMode` is set to `GIN_SEARCH_MODE_ALL`, then all non-null items in the index are considered candidate matches, whether they match any of the returned keys or not. (This mode is much slower than the other two choices, since it requires scanning essentially the entire index, but it may be necessary to implement corner cases correctly. An operator that needs this mode in most cases is probably not a good candidate for a GIN operator class.) The symbols to use for setting this mode are defined in `access/gin.h`.

`pmatch` is an output argument for use when partial match is supported. To use it, `extractQuery` must allocate an array of `*nkeys` booleans and store its address at `*pmatch`. Each element of the array should be set to TRUE if the corresponding key requires partial match, FALSE if not. If `*pmatch` is set to `NULL` then GIN assumes partial match is not required. The variable is initialized to `NULL` before call, so this argument can simply be ignored by operator classes that do not support partial match.

`extra_data` is an output argument that allows `extractQuery` to pass additional data to the `consistent` and `comparePartial` methods. To use it, `extractQuery` must allocate an array of `*nkeys` pointers and store its address at `*extra_data`, then store whatever it wants to into the individual pointers. The variable is initialized to `NULL` before call, so this argument can simply be

ignored by operator classes that do not require extra data. If `*extra_data` is set, the whole array is passed to the `consistent` method, and the appropriate element to the `comparePartial` method.

An operator class must also provide a function to check if an indexed item matches the query. It comes in two flavors, a boolean `consistent` function, and a ternary `triConsistent` function. `triConsistent` covers the functionality of both, so providing triConsistent alone is sufficient. However, if the boolean variant is significantly cheaper to calculate, it can be advantageous to provide both. If only the boolean variant is provided, some optimizations that depend on refuting index items before fetching all the keys are disabled.

```
bool consistent(bool check[], StrategyNumber n, Datum query, int32 nkeys,
Pointer extra_data[], bool *recheck, Datum queryKeys[], bool nullFlags[])
```

Returns TRUE if an indexed item satisfies the query operator with strategy number `n` (or might satisfy it, if the recheck indication is returned). This function does not have direct access to the indexed item's value, since GIN does not store items explicitly. Rather, what is available is knowledge about which key values extracted from the query appear in a given indexed item. The `check` array has length `nkeys`, which is the same as the number of keys previously returned by `extractQuery` for this `query` datum. Each element of the `check` array is TRUE if the indexed item contains the corresponding query key, i.e., if (check[i] == TRUE) the i-th key of the `extractQuery` result array is present in the indexed item. The original `query` datum is passed in case the `consistent` method needs to consult it, and so are the `queryKeys[]` and `nullFlags[]` arrays previously returned by `extractQuery`. `extra_data` is the extra-data array returned by `extractQuery`, or `NULL` if none.

When `extractQuery` returns a null key in `queryKeys[]`, the corresponding `check[]` element is TRUE if the indexed item contains a null key; that is, the semantics of `check[]` are like `IS NOT DISTINCT FROM`. The `consistent` function can examine the corresponding `nullFlags[]` element if it needs to tell the difference between a regular value match and a null match.

On success, `*recheck` should be set to TRUE if the heap tuple needs to be rechecked against the query operator, or FALSE if the index test is exact. That is, a FALSE return value guarantees that the heap tuple does not match the query; a TRUE return value with `*recheck` set to FALSE guarantees that the heap tuple does match the query; and a TRUE return value with `*recheck` set to TRUE means that the heap tuple might match the query, so it needs to be fetched and rechecked by evaluating the query operator directly against the originally indexed item.

```
GinTernaryValue triConsistent(GinTernaryValue check[], StrategyNumber n,
Datum query, int32 nkeys, Pointer extra_data[], Datum queryKeys[], bool
nullFlags[])
```

`triConsistent` is similar to `consistent`, but instead of a boolean `check[]`, there are three possible values for each key: `GIN_TRUE`, `GIN_FALSE` and `GIN_MAYBE`. `GIN_FALSE` and `GIN_TRUE` have the same meaning as regular boolean values. `GIN_MAYBE` means that the presence of that key is not known. When `GIN_MAYBE` values are present, the function should only return GIN_TRUE if the item matches whether or not the index item contains the corresponding query keys. Likewise, the function must return GIN_FALSE only if the item does not match, whether or not it contains the GIN_MAYBE keys. If the result depends on the GIN_MAYBE entries, i.e., the match cannot be confirmed or refuted based on the known query keys, the function must return GIN_MAYBE.

When there are no GIN_MAYBE values in the `check` vector, `GIN_MAYBE` return value is equivalent of setting `recheck` flag in the boolean `consistent` function.

Optionally, an operator class for GIN can supply the following method:

`int comparePartial(Datum partial_key, Datum key, StrategyNumber n, Pointer extra_data)`

Compare a partial-match query key to an index key. Returns an integer whose sign indicates the result: less than zero means the index key does not match the query, but the index scan should continue; zero means that the index key does match the query; greater than zero indicates that the index scan should stop because no more matches are possible. The strategy number `n` of the operator that generated the partial match query is provided, in case its semantics are needed to determine when to end the scan. Also, `extra_data` is the corresponding element of the extra-data array made by `extractQuery`, or `NULL` if none. Null keys are never passed to this function.

To support "partial match" queries, an operator class must provide the `comparePartial` method, and its `extractQuery` method must set the `pmatch` parameter when a partial-match query is encountered. See Section 61.4.2 for details.

The actual data types of the various `Datum` values mentioned above vary depending on the operator class. The item values passed to `extractValue` are always of the operator class's input type, and all key values must be of the class's `STORAGE` type. The type of the `query` argument passed to `extractQuery`, `consistent` and `triConsistent` is whatever is specified as the right-hand input type of the class member operator identified by the strategy number. This need not be the same as the item type, so long as key values of the correct type can be extracted from it.

61.4. Implementation

Internally, a GIN index contains a B-tree index constructed over keys, where each key is an element of one or more indexed items (a member of an array, for example) and where each tuple in a leaf page contains either a pointer to a B-tree of heap pointers (a "posting tree"), or a simple list of heap pointers (a "posting list") when the list is small enough to fit into a single index tuple along with the key value.

As of PostgreSQL 9.1, null key values can be included in the index. Also, placeholder nulls are included in the index for indexed items that are null or contain no keys according to `extractValue`. This allows searches that should find empty items to do so.

Multicolumn GIN indexes are implemented by building a single B-tree over composite values (column number, key value). The key values for different columns can be of different types.

61.4.1. GIN Fast Update Technique

Updating a GIN index tends to be slow because of the intrinsic nature of inverted indexes: inserting or updating one heap row can cause many inserts into the index (one for each key extracted from the indexed item). As of PostgreSQL 8.4, GIN is capable of postponing much of this work by inserting new tuples into a temporary, unsorted list of pending entries. When the table is vacuumed, or if the pending list becomes larger than gin_pending_list_limit, the entries are moved to the main GIN data structure using the same bulk insert techniques used during initial index creation. This greatly improves GIN index update speed, even counting the additional vacuum overhead. Moreover the overhead work can be done by a background process instead of in foreground query processing.

The main disadvantage of this approach is that searches must scan the list of pending entries in addition to searching the regular index, and so a large list of pending entries will slow searches significantly. Another disadvantage is that, while most updates are fast, an update that causes the pending list to become "too large" will incur an immediate cleanup cycle and thus be much slower than other updates. Proper use of autovacuum can minimize both of these problems.

If consistent response time is more important than update speed, use of pending entries can be disabled by turning off the `fastupdate` storage parameter for a GIN index. See CREATE INDEX for details.

61.4.2. Partial Match Algorithm

GIN can support "partial match" queries, in which the query does not determine an exact match for one or more keys, but the possible matches fall within a reasonably narrow range of key values (within the key sorting order determined by the `compare` support method). The `extractQuery` method, instead of returning a key value to be matched exactly, returns a key value that is the lower bound of the range to be searched, and sets the `pmatch` flag true. The key range is then scanned using the `comparePartial` method. `comparePartial` must return zero for a matching index key, less than zero for a non-match that is still within the range to be searched, or greater than zero if the index key is past the range that could match.

61.5. GIN Tips and Tricks

Create vs. insert

Insertion into a GIN index can be slow due to the likelihood of many keys being inserted for each item. So, for bulk insertions into a table it is advisable to drop the GIN index and recreate it after finishing bulk insertion.

As of PostgreSQL 8.4, this advice is less necessary since delayed indexing is used (see Section 61.4.1 for details). But for very large updates it may still be best to drop and recreate the index.

maintenance_work_mem

Build time for a GIN index is very sensitive to the `maintenance_work_mem` setting; it doesn't pay to skimp on work memory during index creation.

gin_pending_list_limit

During a series of insertions into an existing GIN index that has `fastupdate` enabled, the system will clean up the pending-entry list whenever the list grows larger than `gin_pending_list_limit`. To avoid fluctuations in observed response time, it's desirable to have pending-list cleanup occur in the background (i.e., via autovacuum). Foreground cleanup operations can be avoided by increasing `gin_pending_list_limit` or making autovacuum more aggressive. However, enlarging the threshold of the cleanup operation means that if a foreground cleanup does occur, it will take even longer.

`gin_pending_list_limit` can be overridden for individual GIN indexes by changing storage parameters, which allows each GIN index to have its own cleanup threshold. For example, it's possible

to increase the threshold only for the GIN index which can be updated heavily, and decrease it otherwise.

gin_fuzzy_search_limit

The primary goal of developing GIN indexes was to create support for highly scalable full-text search in PostgreSQL, and there are often situations when a full-text search returns a very large set of results. Moreover, this often happens when the query contains very frequent words, so that the large result set is not even useful. Since reading many tuples from the disk and sorting them could take a lot of time, this is unacceptable for production. (Note that the index search itself is very fast.)

To facilitate controlled execution of such queries, GIN has a configurable soft upper limit on the number of rows returned: the `gin_fuzzy_search_limit` configuration parameter. It is set to 0 (meaning no limit) by default. If a non-zero limit is set, then the returned set is a subset of the whole result set, chosen at random.

"Soft" means that the actual number of returned results could differ somewhat from the specified limit, depending on the query and the quality of the system's random number generator.

From experience, values in the thousands (e.g., 5000 — 20000) work well.

61.6. Limitations

GIN assumes that indexable operators are strict. This means that `extractValue` will not be called at all on a null item value (instead, a placeholder index entry is created automatically), and `extractQuery` will not be called on a null query value either (instead, the query is presumed to be unsatisfiable). Note however that null key values contained within a non-null composite item or query value are supported.

61.7. Examples

The PostgreSQL source distribution includes GIN operator classes for `tsvector` and for one-dimensional arrays of all internal types. Prefix searching in `tsvector` is implemented using the GIN partial match feature. The following `contrib` modules also contain GIN operator classes:

`btree_gin`

B-tree equivalent functionality for several data types

`hstore`

Module for storing (key, value) pairs

`intarray`

Enhanced support for `int[]`

`pg_trgm`

Text similarity using trigram matching

Chapter 62. BRIN Indexes

62.1. Introduction

BRIN stands for Block Range Index. BRIN is designed for handling very large tables in which certain columns have some natural correlation with their physical location within the table. A *block range* is a group of pages that are physically adjacent in the table; for each block range, some summary info is stored by the index. For example, a table storing a store's sale orders might have a date column on which each order was placed, and most of the time the entries for earlier orders will appear earlier in the table as well; a table storing a ZIP code column might have all codes for a city grouped together naturally.

BRIN indexes can satisfy queries via regular bitmap index scans, and will return all tuples in all pages within each range if the summary info stored by the index is *consistent* with the query conditions. The query executor is in charge of rechecking these tuples and discarding those that do not match the query conditions — in other words, these indexes are lossy. Because a BRIN index is very small, scanning the index adds little overhead compared to a sequential scan, but may avoid scanning large parts of the table that are known not to contain matching tuples.

The specific data that a BRIN index will store, as well as the specific queries that the index will be able to satisfy, depend on the operator class selected for each column of the index. Data types having a linear sort order can have operator classes that store the minimum and maximum value within each block range, for instance; geometrical types might store the bounding box for all the objects in the block range.

The size of the block range is determined at index creation time by the `pages_per_range` storage parameter. The number of index entries will be equal to the size of the relation in pages divided by the selected value for `pages_per_range`. Therefore, the smaller the number, the larger the index becomes (because of the need to store more index entries), but at the same time the summary data stored can be more precise and more data blocks can be skipped during an index scan.

62.1.1. Index Maintenance

At the time of creation, all existing index pages are scanned and a summary index tuple is created for each range, including the possibly-incomplete range at the end. As new pages are filled with data, page ranges that are already summarized will cause the summary information to be updated with data from the new tuples. When a new page is created that does not fall within the last summarized range, that range does not automatically acquire a summary tuple; those tuples remain unsummarized until a summarization run is invoked later, creating initial summaries. This process can be invoked manually using the `brin_summarize_new_values(regclass)` function, or automatically when `VACUUM` processes the table.

62.2. Built-in Operator Classes

The core PostgreSQL distribution includes the BRIN operator classes shown in Table 62-1.

The *minmax* operator classes store the minimum and the maximum values appearing in the indexed column within the range. The *inclusion* operator classes store a value which includes the values in the indexed column within the range.

Table 62-1. Built-in BRIN Operator Classes

Name	Indexed Data Type	Indexable Operators
abstime_minmax_ops	abstime	< <= = >= >
int8_minmax_ops	bigint	< <= = >= >
bit_minmax_ops	bit	< <= = >= >
varbit_minmax_ops	bit varying	< <= = >= >
box_inclusion_ops	box	<< &< && &> >> ~= @> <@ &<| <<| |>> |&>
bytea_minmax_ops	bytea	< <= = >= >
bpchar_minmax_ops	character	< <= = >= >
char_minmax_ops	"char"	< <= = >= >
date_minmax_ops	date	< <= = >= >
float8_minmax_ops	double precision	< <= = >= >
inet_minmax_ops	inet	< <= = >= >
network_inclusion_ops	inet	&& >>= <<= = >> <<
int4_minmax_ops	integer	< <= = >= >
interval_minmax_ops	interval	< <= = >= >
macaddr_minmax_ops	macaddr	< <= = >= >
name_minmax_ops	name	< <= = >= >
numeric_minmax_ops	numeric	< <= = >= >
pg_lsn_minmax_ops	pg_lsn	< <= = >= >
oid_minmax_ops	oid	< <= = >= >
range_inclusion_ops	any range type	<< &< && &> >> @> <@ -|- = < <= = > >=
float4_minmax_ops	real	< <= = >= >
reltime_minmax_ops	reltime	< <= = >= >
int2_minmax_ops	smallint	< <= = >= >
text_minmax_ops	text	< <= = >= >
tid_minmax_ops	tid	< <= = >= >
timestamp_minmax_ops	timestamp without time zone	< <= = >= >
timestamptz_minmax_ops	timestamp with time zone	< <= = >= >
time_minmax_ops	time without time zone	< <= = >= >

Name	Indexed Data Type	Indexable Operators
timetz_minmax_ops	time with time zone	< <= = >= >
uuid_minmax_ops	uuid	< <= = >= >

62.3. Extensibility

The BRIN interface has a high level of abstraction, requiring the access method implementer only to implement the semantics of the data type being accessed. The BRIN layer itself takes care of concurrency, logging and searching the index structure.

All it takes to get a BRIN access method working is to implement a few user-defined methods, which define the behavior of summary values stored in the index and the way they interact with scan keys. In short, BRIN combines extensibility with generality, code reuse, and a clean interface.

There are four methods that an operator class for BRIN must provide:

`BrinOpcInfo *opcInfo(Oid type_oid)`

Returns internal information about the indexed columns' summary data. The return value must point to a palloc'd `BrinOpcInfo`, which has this definition:

```
typedef struct BrinOpcInfo
{
    /* Number of columns stored in an index column of this opclass */
    uint16      oi_nstored;

    /* Opaque pointer for the opclass' private use */
    void       *oi_opaque;

    /* Type cache entries of the stored columns */
    TypeCacheEntry *oi_typcache[FLEXIBLE_ARRAY_MEMBER];
} BrinOpcInfo;
```

`BrinOpcInfo.oi_opaque` can be used by the operator class routines to pass information between support procedures during an index scan.

`bool consistent(BrinDesc *bdesc, BrinValues *column, ScanKey key)`

Returns whether the ScanKey is consistent with the given indexed values for a range. The attribute number to use is passed as part of the scan key.

`bool addValue(BrinDesc *bdesc, BrinValues *column, Datum newval, bool isnull)`

Given an index tuple and an indexed value, modifies the indicated attribute of the tuple so that it additionally represents the new value. If any modification was done to the tuple, `true` is returned.

`bool unionTuples(BrinDesc *bdesc, BrinValues *a, BrinValues *b)`

Consolidates two index tuples. Given two index tuples, modifies the indicated attribute of the first of them so that it represents both tuples. The second tuple is not modified.

The core distribution includes support for two types of operator classes: minmax and inclusion. Operator class definitions using them are shipped for in-core data types as appropriate. Additional operator classes can be defined by the user for other data types using equivalent definitions, without having to

write any source code; appropriate catalog entries being declared is enough. Note that assumptions about the semantics of operator strategies are embedded in the support procedures' source code.

Operator classes that implement completely different semantics are also possible, provided implementations of the four main support procedures described above are written. Note that backwards compatibility across major releases is not guaranteed: for example, additional support procedures might be required in later releases.

To write an operator class for a data type that implements a totally ordered set, it is possible to use the minmax support procedures alongside the corresponding operators, as shown in Table 62-2. All operator class members (procedures and operators) are mandatory.

Table 62-2. Procedure and Support Numbers for Minmax Operator Classes

Operator class member	Object
Support Procedure 1	internal function `brin_minmax_opcinfo()`
Support Procedure 2	internal function `brin_minmax_add_value()`
Support Procedure 3	internal function `brin_minmax_consistent()`
Support Procedure 4	internal function `brin_minmax_union()`
Operator Strategy 1	operator less-than
Operator Strategy 2	operator less-than-or-equal-to
Operator Strategy 3	operator equal-to
Operator Strategy 4	operator greater-than-or-equal-to
Operator Strategy 5	operator greater-than

To write an operator class for a complex datatype which has values included within another type, it's possible to use the inclusion support procedures alongside the corresponding operators, as shown in Table 62-3. It requires only a single additional function, which can be written in any language. More functions can be defined for additional functionality. All operators are optional. Some operators require other operators, as shown as dependencies on the table.

Table 62-3. Procedure and Support Numbers for Inclusion Operator Classes

Operator class member	Object	Dependency
Support Procedure 1	internal function `brin_inclusion_opcinfo()`	
Support Procedure 2	internal function `brin_inclusion_add_value()`	
Support Procedure 3	internal function `brin_inclusion_consistent()`	
Support Procedure 4	internal function `brin_inclusion_union()`	
Support Procedure 11	function to merge two elements	

Operator class member	Object	Dependency
Support Procedure 12	optional function to check whether two elements are mergeable	
Support Procedure 13	optional function to check if an element is contained within another	
Support Procedure 14	optional function to check whether an element is empty	
Operator Strategy 1	operator left-of	Operator Strategy 4
Operator Strategy 2	operator does-not-extend-to-the-right-of	Operator Strategy 5
Operator Strategy 3	operator overlaps	
Operator Strategy 4	operator right-of	Operator Strategy 2
Operator Strategy 5	operator does-not-extend-to-the-right-of	Operator Strategy 1
Operator Strategy 6, 18	operator same-as-or-equal-to	Operator Strategy 7
Operator Strategy 7, 13, 16, 24, 25	operator contains-or-equal-to	
Operator Strategy 8, 14, 26, 27	operator is-contained-by-or-equal-to	Operator Strategy 3
Operator Strategy 9	operator does-not-extend-above	Operator Strategy 11
Operator Strategy 10	operator is-below	Operator Strategy 12
Operator Strategy 11	operator is-above	Operator Strategy 9
Operator Strategy 12	operator does-not-extend-below	Operator Strategy 10
Operator Strategy 20	operator less-than	Operator Strategy 4
Operator Strategy 21	operator less-than-or-equal-to	Operator Strategy 4
Operator Strategy 22	operator greater-than	Operator Strategy 1
Operator Strategy 23	operator greater-than-or-equal-to	Operator Strategy 1

Support procedure numbers 1-10 are reserved for the BRIN internal functions, so the SQL level functions start with number 11. Support function number 11 is the main function required to build the index. It should accept two arguments with the same datatype as the opclass, and return the union of them. The inclusion opclass can store union values with different datatypes if it is defined with the `STORAGE` parameter. The return value of the union function should match the `STORAGE` datatype.

Support procedure numbers 12 and 14 are provided to support irregularities of built-in datatypes. Procedure number 12 is used to support network addresses from different families which are not mergeable. Procedure number 14 is used to support empty ranges. Procedure number 13 is an optional but recommended one, which allows the new value to be checked before it is passed to the union function. As the BRIN framework can shortcut some operations when the union is not changed, using this function can improve index performance.

Both minmax and inclusion opclasses support cross-datatype operators, though with these the dependencies become more complicated. The minmax opclass requires a full set of operators to be defined with

both arguments having the same datatype. It allows additional datatypes to be supported by defining extra sets of operators. Inclusion opclass operator strategies are dependent on another operator strategy as shown in Table 62-3, or the same operator strategy as themselves. They require the dependency operator to be defined with the `STORAGE` datatype as the left-hand-side argument and the other supported datatype to be the right-hand-side argument of the supported operator. See `float4_minmax_ops` as an example of minmax, and `box_inclusion_ops` as an example of inclusion.

Chapter 63. Database Physical Storage

This chapter provides an overview of the physical storage format used by PostgreSQL databases.

63.1. Database File Layout

This section describes the storage format at the level of files and directories.

Traditionally, the configuration and data files used by a database cluster are stored together within the cluster's data directory, commonly referred to as PGDATA (after the name of the environment variable that can be used to define it). A common location for PGDATA is /var/lib/pgsql/data. Multiple clusters, managed by different server instances, can exist on the same machine.

The PGDATA directory contains several subdirectories and control files, as shown in Table 63-1. In addition to these required items, the cluster configuration files postgresql.conf, pg_hba.conf, and pg_ident.conf are traditionally stored in PGDATA, although it is possible to place them elsewhere.

Table 63-1. Contents of PGDATA

Item	Description
PG_VERSION	A file containing the major version number of PostgreSQL
base	Subdirectory containing per-database subdirectories
global	Subdirectory containing cluster-wide tables, such as pg_database
pg_commit_ts	Subdirectory containing transaction commit timestamp data
pg_clog	Subdirectory containing transaction commit status data
pg_dynshmem	Subdirectory containing files used by the dynamic shared memory subsystem
pg_logical	Subdirectory containing status data for logical decoding
pg_multixact	Subdirectory containing multitransaction status data (used for shared row locks)
pg_notify	Subdirectory containing LISTEN/NOTIFY status data
pg_replslot	Subdirectory containing replication slot data
pg_serial	Subdirectory containing information about committed serializable transactions
pg_snapshots	Subdirectory containing exported snapshots

Item	Description
pg_stat	Subdirectory containing permanent files for the statistics subsystem
pg_stat_tmp	Subdirectory containing temporary files for the statistics subsystem
pg_subtrans	Subdirectory containing subtransaction status data
pg_tblspc	Subdirectory containing symbolic links to tablespaces
pg_twophase	Subdirectory containing state files for prepared transactions
pg_xlog	Subdirectory containing WAL (Write Ahead Log) files
postgresql.auto.conf	A file used for storing configuration parameters that are set by ALTER SYSTEM
postmaster.opts	A file recording the command-line options the server was last started with
postmaster.pid	A lock file recording the current postmaster process ID (PID), cluster data directory path, postmaster start timestamp, port number, Unix-domain socket directory path (empty on Windows), first valid listen_address (IP address or *, or empty if not listening on TCP), and shared memory segment ID (this file is not present after server shutdown)

For each database in the cluster there is a subdirectory within PGDATA/base, named after the database's OID in pg_database. This subdirectory is the default location for the database's files; in particular, its system catalogs are stored there.

Each table and index is stored in a separate file. For ordinary relations, these files are named after the table or index's *filenode* number, which can be found in pg_class.relfilenode. But for temporary relations, the file name is of the form t*BBB*_*FFF*, where *BBB* is the backend ID of the backend which created the file, and *FFF* is the filenode number. In either case, in addition to the main file (a/k/a main fork), each table and index has a *free space map* (see Section 63.3), which stores information about free space available in the relation. The free space map is stored in a file named with the filenode number plus the suffix _fsm. Tables also have a *visibility map*, stored in a fork with the suffix _vm, to track which pages are known to have no dead tuples. The visibility map is described further in Section 63.4. Unlogged tables and indexes have a third fork, known as the initialization fork, which is stored in a fork with the suffix _init (see Section 63.5).

> **Caution**
>
> Note that while a table's filenode often matches its OID, this is *not* necessarily the case; some operations, like `TRUNCATE`, `REINDEX`, `CLUSTER` and some forms of `ALTER TABLE`, can change the filenode while preserving the OID. Avoid assuming that filenode and table OID are the same. Also, for certain system catalogs including `pg_class` itself, `pg_class.relfilenode` contains zero. The actual filenode number of these catalogs is stored in a lower-level data structure, and can be obtained using the `pg_relation_filenode()` function.

When a table or index exceeds 1 GB, it is divided into gigabyte-sized *segments*. The first segment's file name is the same as the filenode; subsequent segments are named filenode.1, filenode.2, etc. This arrangement avoids problems on platforms that have file size limitations. (Actually, 1 GB is just the default segment size. The segment size can be adjusted using the configuration option `--with-segsize` when building PostgreSQL.) In principle, free space map and visibility map forks could require multiple segments as well, though this is unlikely to happen in practice.

A table that has columns with potentially large entries will have an associated *TOAST* table, which is used for out-of-line storage of field values that are too large to keep in the table rows proper. `pg_class.reltoastrelid` links from a table to its TOAST table, if any. See Section 63.2 for more information.

The contents of tables and indexes are discussed further in Section 63.6.

Tablespaces make the scenario more complicated. Each user-defined tablespace has a symbolic link inside the `PGDATA/pg_tblspc` directory, which points to the physical tablespace directory (i.e., the location specified in the tablespace's `CREATE TABLESPACE` command). This symbolic link is named after the tablespace's OID. Inside the physical tablespace directory there is a subdirectory with a name that depends on the PostgreSQL server version, such as `PG_9.0_201008051`. (The reason for using this subdirectory is so that successive versions of the database can use the same `CREATE TABLESPACE` location value without conflicts.) Within the version-specific subdirectory, there is a subdirectory for each database that has elements in the tablespace, named after the database's OID. Tables and indexes are stored within that directory, using the filenode naming scheme. The `pg_default` tablespace is not accessed through `pg_tblspc`, but corresponds to `PGDATA/base`. Similarly, the `pg_global` tablespace is not accessed through `pg_tblspc`, but corresponds to `PGDATA/global`.

The `pg_relation_filepath()` function shows the entire path (relative to `PGDATA`) of any relation. It is often useful as a substitute for remembering many of the above rules. But keep in mind that this function just gives the name of the first segment of the main fork of the relation — you may need to append a segment number and/or `_fsm`, `_vm`, or `_init` to find all the files associated with the relation.

Temporary files (for operations such as sorting more data than can fit in memory) are created within `PGDATA/base/pgsql_tmp`, or within a `pgsql_tmp` subdirectory of a tablespace directory if a tablespace other than `pg_default` is specified for them. The name of a temporary file has the form `pgsql_tmp`*`PPP`*`.`*`NNN`*, where *`PPP`* is the PID of the owning backend and *`NNN`* distinguishes different temporary files of that backend.

63.2. TOAST

This section provides an overview of TOAST (The Oversized-Attribute Storage Technique).

PostgreSQL uses a fixed page size (commonly 8 kB), and does not allow tuples to span multiple pages. Therefore, it is not possible to store very large field values directly. To overcome this limitation, large field values are compressed and/or broken up into multiple physical rows. This happens transparently to the user, with only small impact on most of the backend code. The technique is affectionately known as TOAST (or "the best thing since sliced bread"). The TOAST infrastructure is also used to improve handling of large data values in-memory.

Only certain data types support TOAST — there is no need to impose the overhead on data types that cannot produce large field values. To support TOAST, a data type must have a variable-length (*varlena*) representation, in which, ordinarily, the first four-byte word of any stored value contains the total length of the value in bytes (including itself). TOAST does not constrain the rest of the data type's representation. The special representations collectively called *TOASTed values* work by modifying or reinterpreting this initial length word. Therefore, the C-level functions supporting a TOAST-able data type must be careful about how they handle potentially TOASTed input values: an input might not actually consist of a four-byte length word and contents until after it's been *detoasted*. (This is normally done by invoking `PG_DETOAST_DATUM` before doing anything with an input value, but in some cases more efficient approaches are possible. See Section 35.11.1 for more detail.)

TOAST usurps two bits of the varlena length word (the high-order bits on big-endian machines, the low-order bits on little-endian machines), thereby limiting the logical size of any value of a TOAST-able data type to 1 GB (2^{30} - 1 bytes). When both bits are zero, the value is an ordinary un-TOASTed value of the data type, and the remaining bits of the length word give the total datum size (including length word) in bytes. When the highest-order or lowest-order bit is set, the value has only a single-byte header instead of the normal four-byte header, and the remaining bits of that byte give the total datum size (including length byte) in bytes. This alternative supports space-efficient storage of values shorter than 127 bytes, while still allowing the data type to grow to 1 GB at need. Values with single-byte headers aren't aligned on any particular boundary, whereas values with four-byte headers are aligned on at least a four-byte boundary; this omission of alignment padding provides additional space savings that is significant compared to short values. As a special case, if the remaining bits of a single-byte header are all zero (which would be impossible for a self-inclusive length), the value is a pointer to out-of-line data, with several possible alternatives as described below. The type and size of such a *TOAST pointer* are determined by a code stored in the second byte of the datum. Lastly, when the highest-order or lowest-order bit is clear but the adjacent bit is set, the content of the datum has been compressed and must be decompressed before use. In this case the remaining bits of the four-byte length word give the total size of the compressed datum, not the original data. Note that compression is also possible for out-of-line data but the varlena header does not tell whether it has occurred — the content of the TOAST pointer tells that, instead.

As mentioned, there are multiple types of TOAST pointer datums. The oldest and most common type is a pointer to out-of-line data stored in a *TOAST table* that is separate from, but associated with, the table containing the TOAST pointer datum itself. These *on-disk* pointer datums are created by the TOAST management code (in `access/heap/tuptoaster.c`) when a tuple to be stored on disk is too large to be stored as-is. Further details appear in Section 63.2.1. Alternatively, a TOAST pointer datum can contain a pointer to out-of-line data that appears elsewhere in memory. Such datums are necessarily short-lived, and will never appear on-disk, but they are very useful for avoiding copying and redundant processing of large data values. Further details appear in Section 63.2.2.

The compression technique used for either in-line or out-of-line compressed data is a fairly simple and very fast member of the LZ family of compression techniques. See `src/common/pg_lzcompress.c` for the details.

63.2.1. Out-of-line, on-disk TOAST storage

If any of the columns of a table are TOAST-able, the table will have an associated TOAST table, whose OID is stored in the table's `pg_class.reltoastrelid` entry. On-disk TOASTed values are kept in the TOAST table, as described in more detail below.

Out-of-line values are divided (after compression if used) into chunks of at most `TOAST_MAX_CHUNK_SIZE` bytes (by default this value is chosen so that four chunk rows will fit on a page, making it about 2000 bytes). Each chunk is stored as a separate row in the TOAST table belonging to the owning table. Every TOAST table has the columns `chunk_id` (an OID identifying the particular TOASTed value), `chunk_seq` (a sequence number for the chunk within its value), and `chunk_data` (the actual data of the chunk). A unique index on `chunk_id` and `chunk_seq` provides fast retrieval of the values. A pointer datum representing an out-of-line on-disk TOASTed value therefore needs to store the OID of the TOAST table in which to look and the OID of the specific value (its `chunk_id`). For convenience, pointer datums also store the logical datum size (original uncompressed data length) and physical stored size (different if compression was applied). Allowing for the varlena header bytes, the total size of an on-disk TOAST pointer datum is therefore 18 bytes regardless of the actual size of the represented value.

The TOAST management code is triggered only when a row value to be stored in a table is wider than `TOAST_TUPLE_THRESHOLD` bytes (normally 2 kB). The TOAST code will compress and/or move field values out-of-line until the row value is shorter than `TOAST_TUPLE_TARGET` bytes (also normally 2 kB) or no more gains can be had. During an UPDATE operation, values of unchanged fields are normally preserved as-is; so an UPDATE of a row with out-of-line values incurs no TOAST costs if none of the out-of-line values change.

The TOAST management code recognizes four different strategies for storing TOAST-able columns on disk:

- `PLAIN` prevents either compression or out-of-line storage; furthermore it disables use of single-byte headers for varlena types. This is the only possible strategy for columns of non-TOAST-able data types.
- `EXTENDED` allows both compression and out-of-line storage. This is the default for most TOAST-able data types. Compression will be attempted first, then out-of-line storage if the row is still too big.
- `EXTERNAL` allows out-of-line storage but not compression. Use of `EXTERNAL` will make substring operations on wide `text` and `bytea` columns faster (at the penalty of increased storage space) because these operations are optimized to fetch only the required parts of the out-of-line value when it is not compressed.
- `MAIN` allows compression but not out-of-line storage. (Actually, out-of-line storage will still be performed for such columns, but only as a last resort when there is no other way to make the row small enough to fit on a page.)

Each TOAST-able data type specifies a default strategy for columns of that data type, but the strategy for a given table column can be altered with `ALTER TABLE SET STORAGE`.

This scheme has a number of advantages compared to a more straightforward approach such as allowing row values to span pages. Assuming that queries are usually qualified by comparisons against relatively small key values, most of the work of the executor will be done using the main row entry. The big values of TOASTed attributes will only be pulled out (if selected at all) at the time the result set is sent to the client. Thus, the main table is much smaller and more of its rows fit in the shared buffer cache than would be the case without any out-of-line storage. Sort sets shrink also, and sorts will more often be done entirely

in memory. A little test showed that a table containing typical HTML pages and their URLs was stored in about half of the raw data size including the TOAST table, and that the main table contained only about 10% of the entire data (the URLs and some small HTML pages). There was no run time difference compared to an un-TOASTed comparison table, in which all the HTML pages were cut down to 7 kB to fit.

63.2.2. Out-of-line, in-memory TOAST storage

TOAST pointers can point to data that is not on disk, but is elsewhere in the memory of the current server process. Such pointers obviously cannot be long-lived, but they are nonetheless useful. There are currently two sub-cases: pointers to *indirect* data and pointers to *expanded* data.

Indirect TOAST pointers simply point at a non-indirect varlena value stored somewhere in memory. This case was originally created merely as a proof of concept, but it is currently used during logical decoding to avoid possibly having to create physical tuples exceeding 1 GB (as pulling all out-of-line field values into the tuple might do). The case is of limited use since the creator of the pointer datum is entirely responsible that the referenced data survives for as long as the pointer could exist, and there is no infrastructure to help with this.

Expanded TOAST pointers are useful for complex data types whose on-disk representation is not especially suited for computational purposes. As an example, the standard varlena representation of a PostgreSQL array includes dimensionality information, a nulls bitmap if there are any null elements, then the values of all the elements in order. When the element type itself is variable-length, the only way to find the `N`'th element is to scan through all the preceding elements. This representation is appropriate for on-disk storage because of its compactness, but for computations with the array it's much nicer to have an "expanded" or "deconstructed" representation in which all the element starting locations have been identified. The TOAST pointer mechanism supports this need by allowing a pass-by-reference Datum to point to either a standard varlena value (the on-disk representation) or a TOAST pointer that points to an expanded representation somewhere in memory. The details of this expanded representation are up to the data type, though it must have a standard header and meet the other API requirements given in `src/include/utils/expandeddatum.h`. C-level functions working with the data type can choose to handle either representation. Functions that do not know about the expanded representation, but simply apply `PG_DETOAST_DATUM` to their inputs, will automatically receive the traditional varlena representation; so support for an expanded representation can be introduced incrementally, one function at a time.

TOAST pointers to expanded values are further broken down into *read-write* and *read-only* pointers. The pointed-to representation is the same either way, but a function that receives a read-write pointer is allowed to modify the referenced value in-place, whereas one that receives a read-only pointer must not; it must first create a copy if it wants to make a modified version of the value. This distinction and some associated conventions make it possible to avoid unnecessary copying of expanded values during query execution.

For all types of in-memory TOAST pointer, the TOAST management code ensures that no such pointer datum can accidentally get stored on disk. In-memory TOAST pointers are automatically expanded to normal in-line varlena values before storage — and then possibly converted to on-disk TOAST pointers, if the containing tuple would otherwise be too big.

63.3. Free Space Map

Each heap and index relation, except for hash indexes, has a Free Space Map (FSM) to keep track of available space in the relation. It's stored alongside the main relation data in a separate relation fork, named after the filenode number of the relation, plus a `_fsm` suffix. For example, if the filenode of a relation is 12345, the FSM is stored in a file called `12345_fsm`, in the same directory as the main relation file.

The Free Space Map is organized as a tree of FSM pages. The bottom level FSM pages store the free space available on each heap (or index) page, using one byte to represent each such page. The upper levels aggregate information from the lower levels.

Within each FSM page is a binary tree, stored in an array with one byte per node. Each leaf node represents a heap page, or a lower level FSM page. In each non-leaf node, the higher of its children's values is stored. The maximum value in the leaf nodes is therefore stored at the root.

See `src/backend/storage/freespace/README` for more details on how the FSM is structured, and how it's updated and searched. The pg_freespacemap module can be used to examine the information stored in free space maps.

63.4. Visibility Map

Each heap relation has a Visibility Map (VM) to keep track of which pages contain only tuples that are known to be visible to all active transactions. It's stored alongside the main relation data in a separate relation fork, named after the filenode number of the relation, plus a `_vm` suffix. For example, if the filenode of a relation is 12345, the VM is stored in a file called `12345_vm`, in the same directory as the main relation file. Note that indexes do not have VMs.

The visibility map simply stores one bit per heap page. A set bit means that all tuples on the page are known to be visible to all transactions. This means that the page does not contain any tuples that need to be vacuumed. This information can also be used by *index-only scans* to answer queries using only the index tuple.

The map is conservative in the sense that we make sure that whenever a bit is set, we know the condition is true, but if a bit is not set, it might or might not be true. Visibility map bits are only set by vacuum, but are cleared by any data-modifying operations on a page.

63.5. The Initialization Fork

Each unlogged table, and each index on an unlogged table, has an initialization fork. The initialization fork is an empty table or index of the appropriate type. When an unlogged table must be reset to empty due to a crash, the initialization fork is copied over the main fork, and any other forks are erased (they will be recreated automatically as needed).

63.6. Database Page Layout

This section provides an overview of the page format used within PostgreSQL tables and indexes.[1] Sequences and TOAST tables are formatted just like a regular table.

In the following explanation, a *byte* is assumed to contain 8 bits. In addition, the term *item* refers to an individual data value that is stored on a page. In a table, an item is a row; in an index, an item is an index entry.

Every table and index is stored as an array of *pages* of a fixed size (usually 8 kB, although a different page size can be selected when compiling the server). In a table, all the pages are logically equivalent, so a particular item (row) can be stored in any page. In indexes, the first page is generally reserved as a *metapage* holding control information, and there can be different types of pages within the index, depending on the index access method.

Table 63-2 shows the overall layout of a page. There are five parts to each page.

Table 63-2. Overall Page Layout

Item	Description
PageHeaderData	24 bytes long. Contains general information about the page, including free space pointers.
ItemIdData	Array of item identifiers pointing to the actual items. Each entry is an (offset,length) pair. 4 bytes per item.
Free space	The unallocated space. New item identifiers are allocated from the start of this area, new items from the end.
Items	The actual items themselves.
Special space	Index access method specific data. Different methods store different data. Empty in ordinary tables.

The first 24 bytes of each page consists of a page header (`PageHeaderData`). Its format is detailed in Table 63-3. The first field tracks the most recent WAL entry related to this page. The second field contains the page checksum if data checksums are enabled. Next is a 2-byte field containing flag bits. This is followed by three 2-byte integer fields (`pd_lower`, `pd_upper`, and `pd_special`). These contain byte offsets from the page start to the start of unallocated space, to the end of unallocated space, and to the start of the special space. The next 2 bytes of the page header, `pd_pagesize_version`, store both the page size and a version indicator. Beginning with PostgreSQL 8.3 the version number is 4; PostgreSQL 8.1 and 8.2 used version number 3; PostgreSQL 8.0 used version number 2; PostgreSQL 7.3 and 7.4 used version number 1; prior releases used version number 0. (The basic page layout and header format has not changed in most of these versions, but the layout of heap row headers has.) The page size is basically only present as a cross-check; there is no support for having more than one page size in an installation. The last field is a hint that shows whether pruning the page is likely to be profitable: it tracks the oldest un-pruned XMAX on the page.

1. Actually, index access methods need not use this page format. All the existing index methods do use this basic format, but the data kept on index metapages usually doesn't follow the item layout rules.

Table 63-3. PageHeaderData Layout

Field	Type	Length	Description
pd_lsn	PageXLogRecPtr	8 bytes	LSN: next byte after last byte of xlog record for last change to this page
pd_checksum	uint16	2 bytes	Page checksum
pd_flags	uint16	2 bytes	Flag bits
pd_lower	LocationIndex	2 bytes	Offset to start of free space
pd_upper	LocationIndex	2 bytes	Offset to end of free space
pd_special	LocationIndex	2 bytes	Offset to start of special space
pd_pagesize_version	uint16	2 bytes	Page size and layout version number information
pd_prune_xid	TransactionId	4 bytes	Oldest unpruned XMAX on page, or zero if none

All the details can be found in `src/include/storage/bufpage.h`.

Following the page header are item identifiers (`ItemIdData`), each requiring four bytes. An item identifier contains a byte-offset to the start of an item, its length in bytes, and a few attribute bits which affect its interpretation. New item identifiers are allocated as needed from the beginning of the unallocated space. The number of item identifiers present can be determined by looking at `pd_lower`, which is increased to allocate a new identifier. Because an item identifier is never moved until it is freed, its index can be used on a long-term basis to reference an item, even when the item itself is moved around on the page to compact free space. In fact, every pointer to an item (`ItemPointer`, also known as `CTID`) created by PostgreSQL consists of a page number and the index of an item identifier.

The items themselves are stored in space allocated backwards from the end of unallocated space. The exact structure varies depending on what the table is to contain. Tables and sequences both use a structure named `HeapTupleHeaderData`, described below.

The final section is the "special section" which can contain anything the access method wishes to store. For example, b-tree indexes store links to the page's left and right siblings, as well as some other data relevant to the index structure. Ordinary tables do not use a special section at all (indicated by setting `pd_special` to equal the page size).

All table rows are structured in the same way. There is a fixed-size header (occupying 23 bytes on most machines), followed by an optional null bitmap, an optional object ID field, and the user data. The header is detailed in Table 63-4. The actual user data (columns of the row) begins at the offset indicated by `t_hoff`, which must always be a multiple of the MAXALIGN distance for the platform. The null bitmap is only present if the *HEAP_HASNULL* bit is set in `t_infomask`. If it is present it begins just after the fixed header and occupies enough bytes to have one bit per data column (that is, `t_natts` bits altogether). In this list of bits, a 1 bit indicates not-null, a 0 bit is a null. When the bitmap is not present, all columns are assumed not-null. The object ID is only present if the *HEAP_HASOID* bit is set in `t_infomask`. If

present, it appears just before the t_hoff boundary. Any padding needed to make t_hoff a MAXALIGN multiple will appear between the null bitmap and the object ID. (This in turn ensures that the object ID is suitably aligned.)

Table 63-4. HeapTupleHeaderData Layout

Field	Type	Length	Description
t_xmin	TransactionId	4 bytes	insert XID stamp
t_xmax	TransactionId	4 bytes	delete XID stamp
t_cid	CommandId	4 bytes	insert and/or delete CID stamp (overlays with t_xvac)
t_xvac	TransactionId	4 bytes	XID for VACUUM operation moving a row version
t_ctid	ItemPointerData	6 bytes	current TID of this or newer row version
t_infomask2	uint16	2 bytes	number of attributes, plus various flag bits
t_infomask	uint16	2 bytes	various flag bits
t_hoff	uint8	1 byte	offset to user data

All the details can be found in src/include/access/htup_details.h.

Interpreting the actual data can only be done with information obtained from other tables, mostly pg_attribute. The key values needed to identify field locations are attlen and attalign. There is no way to directly get a particular attribute, except when there are only fixed width fields and no null values. All this trickery is wrapped up in the functions *heap_getattr*, *fastgetattr* and *heap_getsysattr*.

To read the data you need to examine each attribute in turn. First check whether the field is NULL according to the null bitmap. If it is, go to the next. Then make sure you have the right alignment. If the field is a fixed width field, then all the bytes are simply placed. If it's a variable length field (attlen = -1) then it's a bit more complicated. All variable-length data types share the common header structure struct varlena, which includes the total length of the stored value and some flag bits. Depending on the flags, the data can be either inline or in a TOAST table; it might be compressed, too (see Section 63.2).

Chapter 64. BKI Backend Interface

Backend Interface (BKI) files are scripts in a special language that is understood by the PostgreSQL backend when running in the "bootstrap" mode. The bootstrap mode allows system catalogs to be created and filled from scratch, whereas ordinary SQL commands require the catalogs to exist already. BKI files can therefore be used to create the database system in the first place. (And they are probably not useful for anything else.)

initdb uses a BKI file to do part of its job when creating a new database cluster. The input file used by initdb is created as part of building and installing PostgreSQL by a program named `genbki.pl`, which reads some specially formatted C header files in the `src/include/catalog/` directory of the source tree. The created BKI file is called `postgres.bki` and is normally installed in the `share` subdirectory of the installation tree.

Related information can be found in the documentation for initdb.

64.1. BKI File Format

This section describes how the PostgreSQL backend interprets BKI files. This description will be easier to understand if the `postgres.bki` file is at hand as an example.

BKI input consists of a sequence of commands. Commands are made up of a number of tokens, depending on the syntax of the command. Tokens are usually separated by whitespace, but need not be if there is no ambiguity. There is no special command separator; the next token that syntactically cannot belong to the preceding command starts a new one. (Usually you would put a new command on a new line, for clarity.) Tokens can be certain key words, special characters (parentheses, commas, etc.), numbers, or double-quoted strings. Everything is case sensitive.

Lines starting with `#` are ignored.

64.2. BKI Commands

`create` *`tablename`* *`tableoid`* [`bootstrap`] [`shared_relation`] [`without_oids`] [`rowtype_oid` *`oid`*] (*`name1`* = *`type1`* [FORCE NOT NULL | FORCE NULL] [, *`name2`* = *`type2`* [FORCE NOT NULL | FORCE NULL], ...])

Create a table named *`tablename`*, and having the OID *`tableoid`*, with the columns given in parentheses.

The following column types are supported directly by `bootstrap.c`: `bool`, `bytea`, `char` (1 byte), `name`, `int2`, `int4`, `regproc`, `regclass`, `regtype`, `text`, `oid`, `tid`, `xid`, `cid`, `int2vector`, `oidvector`, `_int4` (array), `_text` (array), `_oid` (array), `_char` (array), `_aclitem` (array). Although it is possible to create tables containing columns of other types, this cannot be done until after `pg_type` has been created and filled with appropriate entries. (That effectively means that only

these column types can be used in bootstrapped tables, but non-bootstrap catalogs can contain any built-in type.)

When `bootstrap` is specified, the table will only be created on disk; nothing is entered into `pg_class`, `pg_attribute`, etc, for it. Thus the table will not be accessible by ordinary SQL operations until such entries are made the hard way (with `insert` commands). This option is used for creating `pg_class` etc themselves.

The table is created as shared if `shared_relation` is specified. It will have OIDs unless `without_oids` is specified. The table's row type OID (`pg_type` OID) can optionally be specified via the `rowtype_oid` clause; if not specified, an OID is automatically generated for it. (The `rowtype_oid` clause is useless if `bootstrap` is specified, but it can be provided anyway for documentation.)

`open` *`tablename`*

Open the table named *`tablename`* for insertion of data. Any currently open table is closed.

`close` [*`tablename`*]

Close the open table. The name of the table can be given as a cross-check, but this is not required.

`insert` [`OID =` *`oid_value`*] `(` *`value1 value2`* ... `)`

Insert a new row into the open table using *`value1`*, *`value2`*, etc., for its column values and *`oid_value`* for its OID. If *`oid_value`* is zero (0) or the clause is omitted, and the table has OIDs, then the next available OID is assigned.

NULL values can be specified using the special key word `_null_`. Values containing spaces must be double quoted.

`declare` [`unique`] `index` *`indexname indexoid`* `on` *`tablename`* `using` *`amname`* `(` *`opclass1 name1`* [, ...] `)`

Create an index named *`indexname`*, having OID *`indexoid`*, on the table named *`tablename`*, using the *`amname`* access method. The fields to index are called *`name1`*, *`name2`* etc., and the operator classes to use are *`opclass1`*, *`opclass2`* etc., respectively. The index file is created and appropriate catalog entries are made for it, but the index contents are not initialized by this command.

`declare toast` *`toasttableoid toastindexoid`* `on` *`tablename`*

Create a TOAST table for the table named *`tablename`*. The TOAST table is assigned OID *`toasttableoid`* and its index is assigned OID *`toastindexoid`*. As with `declare index`, filling of the index is postponed.

`build indices`

Fill in the indices that have previously been declared.

64.3. Structure of the Bootstrap BKI File

The `open` command cannot be used until the tables it uses exist and have entries for the table that is to be opened. (These minimum tables are `pg_class`, `pg_attribute`, `pg_proc`, and `pg_type`.) To allow those tables themselves to be filled, `create` with the `bootstrap` option implicitly opens the created table for data insertion.

Also, the `declare index` and `declare toast` commands cannot be used until the system catalogs they need have been created and filled in.

Thus, the structure of the `postgres.bki` file has to be:

1. `create bootstrap` one of the critical tables
2. `insert` data describing at least the critical tables
3. `close`
4. Repeat for the other critical tables.
5. `create` (without `bootstrap`) a noncritical table
6. `open`
7. `insert` desired data
8. `close`
9. Repeat for the other noncritical tables.
10. Define indexes and toast tables.
11. `build indices`

There are doubtless other, undocumented ordering dependencies.

64.4. Example

The following sequence of commands will create the table `test_table` with OID 420, having two columns `cola` and `colb` of type `int4` and `text`, respectively, and insert two rows into the table:

```
create test_table 420 (cola = int4, colb = text)
open test_table
insert OID=421 ( 1 "value1" )
insert OID=422 ( 2 _null_ )
close test_table
```

Chapter 65. How the Planner Uses Statistics

This chapter builds on the material covered in Section 14.1 and Section 14.2 to show some additional details about how the planner uses the system statistics to estimate the number of rows each part of a query might return. This is a significant part of the planning process, providing much of the raw material for cost calculation.

The intent of this chapter is not to document the code in detail, but to present an overview of how it works. This will perhaps ease the learning curve for someone who subsequently wishes to read the code.

65.1. Row Estimation Examples

The examples shown below use tables in the PostgreSQL regression test database. The outputs shown are taken from version 8.3. The behavior of earlier (or later) versions might vary. Note also that since `ANALYZE` uses random sampling while producing statistics, the results will change slightly after any new `ANALYZE`.

Let's start with a very simple query:

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

How the planner determines the cardinality of `tenk1` is covered in Section 14.2, but is repeated here for completeness. The number of pages and rows is looked up in `pg_class`:

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';

 relpages | reltuples
----------+-----------
      358 |     10000
```

These numbers are current as of the last `VACUUM` or `ANALYZE` on the table. The planner then fetches the actual current number of pages in the table (this is a cheap operation, not requiring a table scan). If that is different from `relpages` then `reltuples` is scaled accordingly to arrive at a current number-of-rows estimate. In the example above, the value of `relpages` is up-to-date so the rows estimate is the same as `reltuples`.

Let's move on to an example with a range condition in its `WHERE` clause:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;

                                 QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=24.06..394.64 rows=1007 width=244)
   Recheck Cond: (unique1 < 1000)
```

```
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

The planner examines the `WHERE` clause condition and looks up the selectivity function for the operator `<` in `pg_operator`. This is held in the column `oprrest`, and the entry in this case is `scalarltsel`. The `scalarltsel` function retrieves the histogram for `unique1` from `pg_statistics`. For manual queries it is more convenient to look in the simpler `pg_stats` view:

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='unique1';

                    histogram_bounds
------------------------------------------------------
 {0,993,1997,3050,4040,5036,5957,7057,8029,9016,9995}
```

Next the fraction of the histogram occupied by "< 1000" is worked out. This is the selectivity. The histogram divides the range into equal frequency buckets, so all we have to do is locate the bucket that our value is in and count *part* of it and *all* of the ones before. The value 1000 is clearly in the second bucket (993-1997). Assuming a linear distribution of values inside each bucket, we can calculate the selectivity as:

```
selectivity = (1 + (1000 - bucket[2].min)/(bucket[2].max - bucket[2].min))/num_buckets
            = (1 + (1000 - 993)/(1997 - 993))/10
            = 0.100697
```

that is, one whole bucket plus a linear fraction of the second, divided by the number of buckets. The estimated number of rows can now be calculated as the product of the selectivity and the cardinality of `tenk1`:

```
rows = rel_cardinality * selectivity
     = 10000 * 0.100697
     = 1007  (rounding off)
```

Next let's consider an example with an equality condition in its `WHERE` clause:

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'CRAAAA';

                        QUERY PLAN
----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=30 width=244)
   Filter: (stringu1 = 'CRAAAA'::name)
```

Again the planner examines the `WHERE` clause condition and looks up the selectivity function for =, which is `eqsel`. For equality estimation the histogram is not useful; instead the list of *most common values* (MCVs) is used to determine the selectivity. Let's have a look at the MCVs, with some additional columns that will be useful later:

```
SELECT null_frac, n_distinct, most_common_vals, most_common_freqs FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

null_frac         | 0
```

```
n_distinct        | 676
most_common_vals  | {EJAAAA,BBAAAA,CRAAAA,FCAAAA,FEAAAA,GSAAAA,JOAAAA,MCAAAA,NAAAAA,WGAAAA}
most_common_freqs | {0.00333333,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003}
```

Since `CRAAAA` appears in the list of MCVs, the selectivity is merely the corresponding entry in the list of most common frequencies (MCFs):

```
selectivity = mcf[3]
            = 0.003
```

As before, the estimated number of rows is just the product of this with the cardinality of `tenk1`:

```
rows = 10000 * 0.003
     = 30
```

Now consider the same query, but with a constant that is not in the MCV list:

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'xxx';

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=15 width=244)
   Filter: (stringu1 = 'xxx'::name)
```

This is quite a different problem: how to estimate the selectivity when the value is *not* in the MCV list. The approach is to use the fact that the value is not in the list, combined with the knowledge of the frequencies for all of the MCVs:

```
selectivity = (1 - sum(mvf))/(num_distinct - num_mcv)
            = (1 - (0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 +
                    0.003 + 0.003 + 0.003 + 0.003))/(676 - 10)
            = 0.0014559
```

That is, add up all the frequencies for the MCVs and subtract them from one, then divide by the number of *other* distinct values. This amounts to assuming that the fraction of the column that is not any of the MCVs is evenly distributed among all the other distinct values. Notice that there are no null values so we don't have to worry about those (otherwise we'd subtract the null fraction from the numerator as well). The estimated number of rows is then calculated as usual:

```
rows = 10000 * 0.0014559
     = 15  (rounding off)
```

The previous example with `unique1 < 1000` was an oversimplification of what `scalarltsel` really does; now that we have seen an example of the use of MCVs, we can fill in some more detail. The example was correct as far as it went, because since `unique1` is a unique column it has no MCVs (obviously, no value is any more common than any other value). For a non-unique column, there will normally be both a histogram and an MCV list, and *the histogram does not include the portion of the column population represented by the MCVs*. We do things this way because it allows more precise estimation. In this situation `scalarltsel` directly applies the condition (e.g., "< 1000") to each value of the MCV list, and adds up

the frequencies of the MCVs for which the condition is true. This gives an exact estimate of the selectivity within the portion of the table that is MCVs. The histogram is then used in the same way as above to estimate the selectivity in the portion of the table that is not MCVs, and then the two numbers are combined to estimate the overall selectivity. For example, consider

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 < 'IAAAAA';

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=3077 width=244)
   Filter: (stringu1 < 'IAAAAA'::name)
```

We already saw the MCV information for `stringu1`, and here is its histogram:

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

                                histogram_bounds
------------------------------------------------------------------------------
 {AAAAAA,CQAAAA,FRAAAA,IBAAAA,KRAAAA,NFAAAA,PSAAAA,SGAAAA,VAAAAA,XLAAAA,ZZAAAA}
```

Checking the MCV list, we find that the condition `stringu1 < 'IAAAAA'` is satisfied by the first six entries and not the last four, so the selectivity within the MCV part of the population is

```
selectivity = sum(relevant mvfs)
            = 0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003
            = 0.01833333
```

Summing all the MCFs also tells us that the total fraction of the population represented by MCVs is 0.03033333, and therefore the fraction represented by the histogram is 0.96966667 (again, there are no nulls, else we'd have to exclude them here). We can see that the value `IAAAAA` falls nearly at the end of the third histogram bucket. Using some rather cheesy assumptions about the frequency of different characters, the planner arrives at the estimate 0.298387 for the portion of the histogram population that is less than `IAAAAA`. We then combine the estimates for the MCV and non-MCV populations:

```
selectivity = mcv_selectivity + histogram_selectivity * histogram_fraction
            = 0.01833333 + 0.298387 * 0.96966667
            = 0.307669

rows        = 10000 * 0.307669
            = 3077  (rounding off)
```

In this particular example, the correction from the MCV list is fairly small, because the column distribution is actually quite flat (the statistics showing these particular values as being more common than others are mostly due to sampling error). In a more typical case where some values are significantly more common than others, this complicated process gives a useful improvement in accuracy because the selectivity for the most common values is found exactly.

Now let's consider a case with more than one condition in the `WHERE` clause:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000 AND stringu1 = 'xxx';

                                    QUERY PLAN
```

```
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=23.80..396.91 rows=1 width=244)
   Recheck Cond: (unique1 < 1000)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

The planner assumes that the two conditions are independent, so that the individual selectivities of the clauses can be multiplied together:

```
selectivity = selectivity(unique1 < 1000) * selectivity(stringu1 = 'xxx')
            = 0.100697 * 0.0014559
            = 0.0001466

rows        = 10000 * 0.0001466
            = 1  (rounding off)
```

Notice that the number of rows estimated to be returned from the bitmap index scan reflects only the condition used with the index; this is important since it affects the cost estimate for the subsequent heap fetches.

Finally we will examine a query that involves a join:

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
------------------------------------------------------------------------------------
 Nested Loop  (cost=4.64..456.23 rows=50 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.64..142.17 rows=50 width=244)
         Recheck Cond: (unique1 < 50)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.63 rows=50 width=0)
               Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..6.27 rows=1 width=244)
         Index Cond: (unique2 = t1.unique2)
```

The restriction on `tenk1`, `unique1 < 50`, is evaluated before the nested-loop join. This is handled analogously to the previous range example. This time the value 50 falls into the first bucket of the `unique1` histogram:

```
selectivity = (0 + (50 - bucket[1].min)/(bucket[1].max - bucket[1].min))/num_buckets
            = (0 + (50 - 0)/(993 - 0))/10
            = 0.005035

rows        = 10000 * 0.005035
            = 50  (rounding off)
```

The restriction for the join is `t2.unique2 = t1.unique2`. The operator is just our familiar `=`, however the selectivity function is obtained from the `oprjoin` column of `pg_operator`, and is `eqjoinsel`. `eqjoinsel` looks up the statistical information for both `tenk2` and `tenk1`:

```
SELECT tablename, null_frac,n_distinct, most_common_vals FROM pg_stats
WHERE tablename IN ('tenk1', 'tenk2') AND attname='unique2';
```

```
tablename  | null_frac | n_distinct | most_common_vals
-----------+-----------+------------+------------------
 tenk1     |         0 |         -1 |
 tenk2     |         0 |         -1 |
```

In this case there is no MCV information for `unique2` because all the values appear to be unique, so we use an algorithm that relies only on the number of distinct values for both relations together with their null fractions:

```
selectivity = (1 - null_frac1) * (1 - null_frac2) * min(1/num_distinct1, 1/num_distinct2)
            = (1 - 0) * (1 - 0) / max(10000, 10000)
            = 0.0001
```

This is, subtract the null fraction from one for each of the relations, and divide by the maximum of the numbers of distinct values. The number of rows that the join is likely to emit is calculated as the cardinality of the Cartesian product of the two inputs, multiplied by the selectivity:

```
rows = (outer_cardinality * inner_cardinality) * selectivity
     = (50 * 10000) * 0.0001
     = 50
```

Had there been MCV lists for the two columns, `eqjoinsel` would have used direct comparison of the MCV lists to determine the join selectivity within the part of the column populations represented by the MCVs. The estimate for the remainder of the populations follows the same approach shown here.

Notice that we showed `inner_cardinality` as 10000, that is, the unmodified size of `tenk2`. It might appear from inspection of the `EXPLAIN` output that the estimate of join rows comes from 50 * 1, that is, the number of outer rows times the estimated number of rows obtained by each inner index scan on `tenk2`. But this is not the case: the join relation size is estimated before any particular join plan has been considered. If everything is working well then the two ways of estimating the join size will produce about the same answer, but due to round-off error and other factors they sometimes diverge significantly.

For those interested in further details, estimation of the size of a table (before any `WHERE` clauses) is done in `src/backend/optimizer/util/plancat.c`. The generic logic for clause selectivities is in `src/backend/optimizer/path/clausesel.c`. The operator-specific selectivity functions are mostly found in `src/backend/utils/adt/selfuncs.c`.

65.2. Planner Statistics and Security

Access to the table `pg_statistic` is restricted to superusers, so that ordinary users cannot learn about the contents of the tables of other users from it. Some selectivity estimation functions will use a user-provided operator (either the operator appearing in the query or a related operator) to analyze the stored statistics. For example, in order to determine whether a stored most common value is applicable, the selectivity estimator will have to run the appropriate = operator to compare the constant in the query to the stored value. Thus the data in `pg_statistic` is potentially passed to user-defined operators. An appropriately crafted operator can intentionally leak the passed operands (for example, by logging them or writing them

to a different table), or accidentally leak them by showing their values in error messages, in either case possibly exposing data from `pg_statistic` to a user who should not be able to see it.

In order to prevent this, the following applies to all built-in selectivity estimation functions. When planning a query, in order to be able to use stored statistics, the current user must either have `SELECT` privilege on the table or the involved columns, or the operator used must be `LEAKPROOF` (more accurately, the function that the operator is based on). If not, then the selectivity estimator will behave as if no statistics are available, and the planner will proceed with default or fall-back assumptions.

If a user does not have the required privilege on the table or columns, then in many cases the query will ultimately receive a permission-denied error, in which case this mechanism is invisible in practice. But if the user is reading from a security-barrier view, then the planner might wish to check the statistics of an underlying table that is otherwise inaccessible to the user. In that case, the operator should be leak-proof or the statistics will not be used. There is no direct feedback about that, except that the plan might be suboptimal. If one suspects that this is the case, one could try running the query as a more privileged user, to see if a different plan results.

This restriction applies only to cases where the planner would need to execute a user-defined operator on one or more values from `pg_statistic`. Thus the planner is permitted to use generic statistical information, such as the fraction of null values or the number of distinct values in a column, regardless of access privileges.

Selectivity estimation functions contained in third-party extensions that potentially operate on statistics with user-defined operators should follow the same security rules. Consult the PostgreSQL source code for guidance.

VIII. Appendixes

Appendix A. PostgreSQL Error Codes

All messages emitted by the PostgreSQL server are assigned five-character error codes that follow the SQL standard's conventions for "SQLSTATE" codes. Applications that need to know which error condition has occurred should usually test the error code, rather than looking at the textual error message. The error codes are less likely to change across PostgreSQL releases, and also are not subject to change due to localization of error messages. Note that some, but not all, of the error codes produced by PostgreSQL are defined by the SQL standard; some additional error codes for conditions not defined by the standard have been invented or borrowed from other databases.

According to the standard, the first two characters of an error code denote a class of errors, while the last three characters indicate a specific condition within that class. Thus, an application that does not recognize the specific error code might still be able to infer what to do from the error class.

Table A-1 lists all the error codes defined in PostgreSQL 9.5.24. (Some are not actually used at present, but are defined by the SQL standard.) The error classes are also shown. For each error class there is a "standard" error code having the last three characters `000`. This code is used only for error conditions that fall within the class but do not have any more-specific code assigned.

The symbol shown in the column "Condition Name" is the condition name to use in PL/pgSQL. Condition names can be written in either upper or lower case. (Note that PL/pgSQL does not recognize warning, as opposed to error, condition names; those are classes 00, 01, and 02.)

For some types of errors, the server reports the name of a database object (a table, table column, data type, or constraint) associated with the error; for example, the name of the unique constraint that caused a `unique_violation` error. Such names are supplied in separate fields of the error report message so that applications need not try to extract them from the possibly-localized human-readable text of the message. As of PostgreSQL 9.3, complete coverage for this feature exists only for errors in SQLSTATE class 23 (integrity constraint violation), but this is likely to be expanded in future.

Table A-1. PostgreSQL Error Codes

Error Code	Condition Name
Class 00 — Successful Completion	
00000	successful_completion
Class 01 — Warning	
01000	warning
0100C	dynamic_result_sets_returned
01008	implicit_zero_bit_padding
01003	null_value_eliminated_in_set_function
01007	privilege_not_granted
01006	privilege_not_revoked
01004	string_data_right_truncation
01P01	deprecated_feature
Class 02 — No Data (this is also a warning class per the SQL standard)	
02000	no_data

Error Code	Condition Name
02001	no_additional_dynamic_result_sets_returned
Class 03 — SQL Statement Not Yet Complete	
03000	sql_statement_not_yet_complete
Class 08 — Connection Exception	
08000	connection_exception
08003	connection_does_not_exist
08006	connection_failure
08001	sqlclient_unable_to_establish_sqlconnection
08004	sqlserver_rejected_establishment_of_sqlconnection
08007	transaction_resolution_unknown
08P01	protocol_violation
Class 09 — Triggered Action Exception	
09000	triggered_action_exception
Class 0A — Feature Not Supported	
0A000	feature_not_supported
Class 0B — Invalid Transaction Initiation	
0B000	invalid_transaction_initiation
Class 0F — Locator Exception	
0F000	locator_exception
0F001	invalid_locator_specification
Class 0L — Invalid Grantor	
0L000	invalid_grantor
0LP01	invalid_grant_operation
Class 0P — Invalid Role Specification	
0P000	invalid_role_specification
Class 0Z — Diagnostics Exception	
0Z000	diagnostics_exception
0Z002	stacked_diagnostics_accessed_without_active_handler
Class 20 — Case Not Found	
20000	case_not_found
Class 21 — Cardinality Violation	
21000	cardinality_violation
Class 22 — Data Exception	
22000	data_exception
2202E	array_subscript_error

Error Code	Condition Name
22021	character_not_in_repertoire
22008	datetime_field_overflow
22012	division_by_zero
22005	error_in_assignment
2200B	escape_character_conflict
22022	indicator_overflow
22015	interval_field_overflow
2201E	invalid_argument_for_logarithm
22014	invalid_argument_for_ntile_function
22016	invalid_argument_for_nth_value_function
2201F	invalid_argument_for_power_function
2201G	invalid_argument_for_width_bucket_function
22018	invalid_character_value_for_cast
22007	invalid_datetime_format
22019	invalid_escape_character
2200D	invalid_escape_octet
22025	invalid_escape_sequence
22P06	nonstandard_use_of_escape_character
22010	invalid_indicator_parameter_value
22023	invalid_parameter_value
2201B	invalid_regular_expression
2201W	invalid_row_count_in_limit_clause
2201X	invalid_row_count_in_result_offset_clause
2202H	invalid_tablesample_argument
2202G	invalid_tablesample_repeat
22009	invalid_time_zone_displacement_value
2200C	invalid_use_of_escape_character
2200G	most_specific_type_mismatch
22004	null_value_not_allowed
22002	null_value_no_indicator_parameter
22003	numeric_value_out_of_range
22026	string_data_length_mismatch
22001	string_data_right_truncation
22011	substring_error
22027	trim_error
22024	unterminated_c_string

Error Code	Condition Name
2200F	zero_length_character_string
22P01	floating_point_exception
22P02	invalid_text_representation
22P03	invalid_binary_representation
22P04	bad_copy_file_format
22P05	untranslatable_character
2200L	not_an_xml_document
2200M	invalid_xml_document
2200N	invalid_xml_content
2200S	invalid_xml_comment
2200T	invalid_xml_processing_instruction
Class 23 — Integrity Constraint Violation	
23000	integrity_constraint_violation
23001	restrict_violation
23502	not_null_violation
23503	foreign_key_violation
23505	unique_violation
23514	check_violation
23P01	exclusion_violation
Class 24 — Invalid Cursor State	
24000	invalid_cursor_state
Class 25 — Invalid Transaction State	
25000	invalid_transaction_state
25001	active_sql_transaction
25002	branch_transaction_already_active
25008	held_cursor_requires_same_isolation_level
25003	inappropriate_access_mode_for_branch_transaction
25004	inappropriate_isolation_level_for_branch_transactior
25005	no_active_sql_transaction_for_branch_transaction
25006	read_only_sql_transaction
25007	schema_and_data_statement_mixing_not_supported
25P01	no_active_sql_transaction
25P02	in_failed_sql_transaction
Class 26 — Invalid SQL Statement Name	

Error Code	Condition Name
26000	invalid_sql_statement_name
Class 27 — Triggered Data Change Violation	
27000	triggered_data_change_violation
Class 28 — Invalid Authorization Specification	
28000	invalid_authorization_specification
28P01	invalid_password
Class 2B — Dependent Privilege Descriptors Still Exist	
2B000	dependent_privilege_descriptors_still_exist
2BP01	dependent_objects_still_exist
Class 2D — Invalid Transaction Termination	
2D000	invalid_transaction_termination
Class 2F — SQL Routine Exception	
2F000	sql_routine_exception
2F005	function_executed_no_return_statement
2F002	modifying_sql_data_not_permitted
2F003	prohibited_sql_statement_attempted
2F004	reading_sql_data_not_permitted
Class 34 — Invalid Cursor Name	
34000	invalid_cursor_name
Class 38 — External Routine Exception	
38000	external_routine_exception
38001	containing_sql_not_permitted
38002	modifying_sql_data_not_permitted
38003	prohibited_sql_statement_attempted
38004	reading_sql_data_not_permitted
Class 39 — External Routine Invocation Exception	
39000	external_routine_invocation_exception
39001	invalid_sqlstate_returned
39004	null_value_not_allowed
39P01	trigger_protocol_violated
39P02	srf_protocol_violated
39P03	event_trigger_protocol_violated
Class 3B — Savepoint Exception	
3B000	savepoint_exception
3B001	invalid_savepoint_specification
Class 3D — Invalid Catalog Name	
3D000	invalid_catalog_name
Class 3F — Invalid Schema Name	

Error Code	Condition Name
3F000	invalid_schema_name
Class 40 — Transaction Rollback	
40000	transaction_rollback
40002	transaction_integrity_constraint_violation
40001	serialization_failure
40003	statement_completion_unknown
40P01	deadlock_detected
Class 42 — Syntax Error or Access Rule Violation	
42000	syntax_error_or_access_rule_violation
42601	syntax_error
42501	insufficient_privilege
42846	cannot_coerce
42803	grouping_error
42P20	windowing_error
42P19	invalid_recursion
42830	invalid_foreign_key
42602	invalid_name
42622	name_too_long
42939	reserved_name
42804	datatype_mismatch
42P18	indeterminate_datatype
42P21	collation_mismatch
42P22	indeterminate_collation
42809	wrong_object_type
42703	undefined_column
42883	undefined_function
42P01	undefined_table
42P02	undefined_parameter
42704	undefined_object
42701	duplicate_column
42P03	duplicate_cursor
42P04	duplicate_database
42723	duplicate_function
42P05	duplicate_prepared_statement
42P06	duplicate_schema
42P07	duplicate_table
42712	duplicate_alias
42710	duplicate_object

Error Code	Condition Name
42702	ambiguous_column
42725	ambiguous_function
42P08	ambiguous_parameter
42P09	ambiguous_alias
42P10	invalid_column_reference
42611	invalid_column_definition
42P11	invalid_cursor_definition
42P12	invalid_database_definition
42P13	invalid_function_definition
42P14	invalid_prepared_statement_definition
42P15	invalid_schema_definition
42P16	invalid_table_definition
42P17	invalid_object_definition
Class 44 — WITH CHECK OPTION Violation	
44000	with_check_option_violation
Class 53 — Insufficient Resources	
53000	insufficient_resources
53100	disk_full
53200	out_of_memory
53300	too_many_connections
53400	configuration_limit_exceeded
Class 54 — Program Limit Exceeded	
54000	program_limit_exceeded
54001	statement_too_complex
54011	too_many_columns
54023	too_many_arguments
Class 55 — Object Not In Prerequisite State	
55000	object_not_in_prerequisite_state
55006	object_in_use
55P02	cant_change_runtime_param
55P03	lock_not_available
Class 57 — Operator Intervention	
57000	operator_intervention
57014	query_canceled
57P01	admin_shutdown
57P02	crash_shutdown
57P03	cannot_connect_now
57P04	database_dropped

Error Code	Condition Name
Class 58 — System Error (errors external to PostgreSQL itself)	
58000	system_error
58030	io_error
58P01	undefined_file
58P02	duplicate_file
Class F0 — Configuration File Error	
F0000	config_file_error
F0001	lock_file_exists
Class HV — Foreign Data Wrapper Error (SQL/MED)	
HV000	fdw_error
HV005	fdw_column_name_not_found
HV002	fdw_dynamic_parameter_value_needed
HV010	fdw_function_sequence_error
HV021	fdw_inconsistent_descriptor_information
HV024	fdw_invalid_attribute_value
HV007	fdw_invalid_column_name
HV008	fdw_invalid_column_number
HV004	fdw_invalid_data_type
HV006	fdw_invalid_data_type_descriptors
HV091	fdw_invalid_descriptor_field_identifier
HV00B	fdw_invalid_handle
HV00C	fdw_invalid_option_index
HV00D	fdw_invalid_option_name
HV090	fdw_invalid_string_length_or_buffer_length
HV00A	fdw_invalid_string_format
HV009	fdw_invalid_use_of_null_pointer
HV014	fdw_too_many_handles
HV001	fdw_out_of_memory
HV00P	fdw_no_schemas
HV00J	fdw_option_name_not_found
HV00K	fdw_reply_handle
HV00Q	fdw_schema_not_found
HV00R	fdw_table_not_found
HV00L	fdw_unable_to_create_execution
HV00M	fdw_unable_to_create_reply
HV00N	fdw_unable_to_establish_connection

Error Code	Condition Name
Class P0 — PL/pgSQL Error	
P0000	plpgsql_error
P0001	raise_exception
P0002	no_data_found
P0003	too_many_rows
P0004	assert_failure
Class XX — Internal Error	
XX000	internal_error
XX001	data_corrupted
XX002	index_corrupted

Appendix B. Date/Time Support

PostgreSQL uses an internal heuristic parser for all date/time input support. Dates and times are input as strings, and are broken up into distinct fields with a preliminary determination of what kind of information can be in the field. Each field is interpreted and either assigned a numeric value, ignored, or rejected. The parser contains internal lookup tables for all textual fields, including months, days of the week, and time zones.

This appendix includes information on the content of these lookup tables and describes the steps used by the parser to decode dates and times.

B.1. Date/Time Input Interpretation

Date/time input strings are decoded using the following procedure.

1. Break the input string into tokens and categorize each token as a string, time, time zone, or number.

    a. If the numeric token contains a colon (`:`), this is a time string. Include all subsequent digits and colons.

    b. If the numeric token contains a dash (`-`), slash (`/`), or two or more dots (`.`), this is a date string which might have a text month. If a date token has already been seen, it is instead interpreted as a time zone name (e.g., `America/New_York`).

    c. If the token is numeric only, then it is either a single field or an ISO 8601 concatenated date (e.g., `19990113` for January 13, 1999) or time (e.g., `141516` for 14:15:16).

    d. If the token starts with a plus (`+`) or minus (`-`), then it is either a numeric time zone or a special field.

2. If the token is an alphabetic string, match up with possible strings:

    a. See if the token matches any known time zone abbreviation. These abbreviations are supplied by the configuration file described in Section B.4.

    b. If not found, search an internal table to match the token as either a special string (e.g., `today`), day (e.g., `Thursday`), month (e.g., `January`), or noise word (e.g., `at`, `on`).

    c. If still not found, throw an error.

3. When the token is a number or number field:

    a. If there are eight or six digits, and if no other date fields have been previously read, then interpret as a "concatenated date" (e.g., `19990118` or `990118`). The interpretation is `YYYYMMDD` or `YYMMDD`.

    b. If the token is three digits and a year has already been read, then interpret as day of year.

    c. If four or six digits and a year has already been read, then interpret as a time (`HHMM` or `HHMMSS`).

   d. If three or more digits and no date fields have yet been found, interpret as a year (this forces yy-mm-dd ordering of the remaining date fields).

   e. Otherwise the date field ordering is assumed to follow the `DateStyle` setting: mm-dd-yy, dd-mm-yy, or yy-mm-dd. Throw an error if a month or day field is found to be out of range.

4. If BC has been specified, negate the year and add one for internal storage. (There is no year zero in the Gregorian calendar, so numerically 1 BC becomes year zero.)

5. If BC was not specified, and if the year field was two digits in length, then adjust the year to four digits. If the field is less than 70, then add 2000, otherwise add 1900.

Tip: Gregorian years AD 1-99 can be entered by using 4 digits with leading zeros (e.g., `0099` is AD 99).

B.2. Handling of Invalid or Ambiguous Timestamps

Ordinarily, if a date/time string is syntactically valid but contains out-of-range field values, an error will be thrown. For example, input specifying the 31st of February will be rejected.

During a daylight-savings-time transition, it is possible for a seemingly valid timestamp string to represent a nonexistent or ambiguous timestamp. Such cases are not rejected; the ambiguity is resolved by determining which UTC offset to apply. For example, supposing that the TimeZone parameter is set to `America/New_York`, consider

```
=> SELECT '2018-03-11 02:30'::timestamptz;
      timestamptz
------------------------
 2018-03-11 03:30:00-04
(1 row)
```

Because that day was a spring-forward transition date in that time zone, there was no civil time instant 2:30AM; clocks jumped forward from 2AM EST to 3AM EDT. PostgreSQL interprets the given time as if it were standard time (UTC-5), which then renders as 3:30AM EDT (UTC-4).

Conversely, consider the behavior during a fall-back transition:

```
=> SELECT '2018-11-04 02:30'::timestamptz;
      timestamptz
------------------------
 2018-11-04 02:30:00-05
(1 row)
```

On that date, there were two possible interpretations of 2:30AM; there was 2:30AM EDT, and then an hour later after the reversion to standard time, there was 2:30AM EST. Again, PostgreSQL interprets the given time as if it were standard time (UTC-5). We can force the matter by specifying daylight-savings time:

```
=> SELECT '2018-11-04 02:30 EDT'::timestamptz;
      timestamptz
------------------------
 2018-11-04 01:30:00-05
(1 row)
```

This timestamp could validly be rendered as either 2:30 UTC-4 or 1:30 UTC-5; the timestamp output code chooses the latter.

The precise rule that is applied in such cases is that an invalid timestamp that appears to fall within a jump-forward daylight savings transition is assigned the UTC offset that prevailed in the time zone just before the transition, while an ambiguous timestamp that could fall on either side of a jump-back transition is assigned the UTC offset that prevailed just after the transition. In most time zones this is equivalent to saying that "the standard-time interpretation is preferred when in doubt".

In all cases, the UTC offset associated with a timestamp can be specified explicitly, using either a numeric UTC offset or a time zone abbreviation that corresponds to a fixed UTC offset. The rule just given applies only when it is necessary to infer a UTC offset for a time zone in which the offset varies.

B.3. Date/Time Key Words

Table B-1 shows the tokens that are recognized as names of months.

Table B-1. Month Names

Month	Abbreviations
January	Jan
February	Feb
March	Mar
April	Apr
May	
June	Jun
July	Jul
August	Aug
September	Sep, Sept
October	Oct
November	Nov
December	Dec

Table B-2 shows the tokens that are recognized as names of days of the week.

Table B-2. Day of the Week Names

Day	Abbreviations
Sunday	Sun

Day	Abbreviations
Monday	Mon
Tuesday	Tue, Tues
Wednesday	Wed, Weds
Thursday	Thu, Thur, Thurs
Friday	Fri
Saturday	Sat

Table B-3 shows the tokens that serve various modifier purposes.

Table B-3. Date/Time Field Modifiers

Identifier	Description
`AM`	Time is before 12:00
`AT`	Ignored
`JULIAN`, `JD`, `J`	Next field is Julian Date
`ON`	Ignored
`PM`	Time is on or after 12:00
`T`	Next field is time

B.4. Date/Time Configuration Files

Since timezone abbreviations are not well standardized, PostgreSQL provides a means to customize the set of abbreviations accepted by the server. The timezone_abbreviations run-time parameter determines the active set of abbreviations. While this parameter can be altered by any database user, the possible values for it are under the control of the database administrator — they are in fact names of configuration files stored in `.../share/timezonesets/` of the installation directory. By adding or altering files in that directory, the administrator can set local policy for timezone abbreviations.

`timezone_abbreviations` can be set to any file name found in `.../share/timezonesets/`, if the file's name is entirely alphabetic. (The prohibition against non-alphabetic characters in `timezone_abbreviations` prevents reading files outside the intended directory, as well as reading editor backup files and other extraneous files.)

A timezone abbreviation file can contain blank lines and comments beginning with `#`. Non-comment lines must have one of these formats:

```
zone_abbreviation offset
zone_abbreviation offset D
zone_abbreviation time_zone_name
@INCLUDE file_name
@OVERRIDE
```

A *zone_abbreviation* is just the abbreviation being defined. An *offset* is an integer giving the equivalent offset in seconds from UTC, positive being east from Greenwich and negative being west. For example, -18000 would be five hours west of Greenwich, or North American east coast standard time. D indicates that the zone name represents local daylight-savings time rather than standard time.

Alternatively, a *time_zone_name* can be given, referencing a zone name defined in the IANA timezone database. The zone's definition is consulted to see whether the abbreviation is or has been in use in that zone, and if so, the appropriate meaning is used — that is, the meaning that was currently in use at the timestamp whose value is being determined, or the meaning in use immediately before that if it wasn't current at that time, or the oldest meaning if it was used only after that time. This behavior is essential for dealing with abbreviations whose meaning has historically varied. It is also allowed to define an abbreviation in terms of a zone name in which that abbreviation does not appear; then using the abbreviation is just equivalent to writing out the zone name.

> **Tip:** Using a simple integer *offset* is preferred when defining an abbreviation whose offset from UTC has never changed, as such abbreviations are much cheaper to process than those that require consulting a time zone definition.

The `@INCLUDE` syntax allows inclusion of another file in the `.../share/timezonesets/` directory. Inclusion can be nested, to a limited depth.

The `@OVERRIDE` syntax indicates that subsequent entries in the file can override previous entries (typically, entries obtained from included files). Without this, conflicting definitions of the same timezone abbreviation are considered an error.

In an unmodified installation, the file `Default` contains all the non-conflicting time zone abbreviations for most of the world. Additional files `Australia` and `India` are provided for those regions: these files first include the `Default` file and then add or modify abbreviations as needed.

For reference purposes, a standard installation also contains files `Africa.txt`, `America.txt`, etc, containing information about every time zone abbreviation known to be in use according to the IANA timezone database. The zone name definitions found in these files can be copied and pasted into a custom configuration file as needed. Note that these files cannot be directly referenced as `timezone_abbreviations` settings, because of the dot embedded in their names.

> **Note:** If an error occurs while reading the time zone abbreviation set, no new value is applied and the old set is kept. If the error occurs while starting the database, startup fails.

Caution

Time zone abbreviations defined in the configuration file override non-timezone meanings built into PostgreSQL. For example, the `Australia` configuration file defines `SAT` (for South Australian Standard Time). When this file is active, `SAT` will not be recognized as an abbreviation for Saturday.

> **Caution**
>
> If you modify files in `.../share/timezonesets/`, it is up to you to make backups — a normal database dump will not include this directory.

B.5. POSIX Time Zone Specifications

PostgreSQL can accept time zone specifications that are written according to the POSIX standard's rules for the `TZ` environment variable. POSIX time zone specifications are inadequate to deal with the complexity of real-world time zone history, but there are sometimes reasons to use them.

A POSIX time zone specification has the form

```
STD offset [ DST [ dstoffset ] [ , rule ] ]
```

(For readability, we show spaces between the fields, but spaces should not be used in practice.) The fields are:

- *`STD`* is the zone abbreviation to be used for standard time.
- *`offset`* is the zone's standard-time offset from UTC.
- *`DST`* is the zone abbreviation to be used for daylight-savings time. If this field and the following ones are omitted, the zone uses a fixed UTC offset with no daylight-savings rule.
- *`dstoffset`* is the daylight-savings offset from UTC. This field is typically omitted, since it defaults to one hour less than the standard-time *`offset`*, which is usually the right thing.
- *`rule`* defines the rule for when daylight savings is in effect, as described below.

In this syntax, a zone abbreviation can be a string of letters, such as `EST`, or an arbitrary string surrounded by angle brackets, such as `<UTC-05>`. Note that the zone abbreviations given here are only used for output, and even then only in some timestamp output formats. The zone abbreviations recognized in timestamp input are determined as explained in Section B.4.

The offset fields specify the hours, and optionally minutes and seconds, difference from UTC. They have the format *`hh`*[`:`*`mm`*[`:`*`ss`*]] optionally with a leading sign (`+` or `-`). The positive sign is used for zones *west* of Greenwich. (Note that this is the opposite of the ISO-8601 sign convention used elsewhere in PostgreSQL.) *`hh`* can have one or two digits; *`mm`* and *`ss`* (if used) must have two.

The daylight-savings transition *`rule`* has the format

```
dstdate [ / dsttime ] , stddate [ / stdtime ]
```

(As before, spaces should not be included in practice.) The *`dstdate`* and *`dsttime`* fields define when daylight-savings time starts, while *`stddate`* and *`stdtime`* define when standard time starts. (In some cases, notably in zones south of the equator, the former might be later in the year than the latter.) The date fields have one of these formats:

n

A plain integer denotes a day of the year, counting from zero to 364, or to 365 in leap years.

Jn

In this form, *n* counts from 1 to 365, and February 29 is not counted even if it is present. (Thus, a transition occurring on February 29 could not be specified this way. However, days after February have the same numbers whether it's a leap year or not, so that this form is usually more useful than the plain-integer form for transitions on fixed dates.)

M*m*.*n*.*d*

This form specifies a transition that always happens during the same month and on the same day of the week. *m* identifies the month, from 1 to 12. *n* specifies the *n*'th occurrence of the weekday identified by *d*. *n* is a number between 1 and 4, or 5 meaning the last occurrence of that weekday in the month (which could be the fourth or the fifth). *d* is a number between 0 and 6, with 0 indicating Sunday. For example, `M3.2.0` means "the second Sunday in March".

Note: The `M` format is sufficient to describe many common daylight-savings transition laws. But note that none of these variants can deal with daylight-savings law changes, so in practice the historical data stored for named time zones (in the IANA time zone database) is necessary to interpret past time stamps correctly.

The time fields in a transition rule have the same format as the offset fields described previously, except that they cannot contain signs. They define the current local time at which the change to the other time occurs. If omitted, they default to `02:00:00`.

If a daylight-savings abbreviation is given but the transition *rule* field is omitted, PostgreSQL attempts to determine the transition times by consulting the `posixrules` file in the IANA time zone database. This file has the same format as a full time zone entry, but only its transition timing rules are used, not its UTC offsets. Typically, this file has the same contents as the `US/Eastern` file, so that POSIX-style time zone specifications follow USA daylight-savings rules. If needed, you can adjust this behavior by replacing the `posixrules` file.

Note: The facility to consult a `posixrules` file has been deprecated by IANA, and it is likely to go away in the future. One bug in this feature, which is unlikely to be fixed before it disappears, is that it fails to apply DST rules to dates after 2038.

If the `posixrules` file is not present, the fallback behavior is to use the rule `M3.2.0,M11.1.0`, which corresponds to USA practice as of 2020 (that is, spring forward on the second Sunday of March, fall back on the first Sunday of November, both transitions occurring at 2AM prevailing time).

As an example, `CET-1CEST,M3.5.0,M10.5.0/3` describes current (as of 2020) timekeeping practice in Paris. This specification says that standard time has the abbreviation `CET` and is one hour ahead (east) of UTC; daylight savings time has the abbreviation `CEST` and is implicitly two hours ahead of UTC; daylight savings time begins on the last Sunday in March at 2AM CET and ends on the last Sunday in October at 3AM CEST.

The four timezone names `EST5EDT`, `CST6CDT`, `MST7MDT`, and `PST8PDT` look like they are POSIX zone specifications. However, they actually are treated as named time zones because (for historical reasons) there are files by those names in the IANA time zone database. The practical implication of this is that these zone names will produce valid historical USA daylight-savings transitions, even when a plain POSIX specification would not due to lack of a suitable `posixrules` file.

One should be wary that it is easy to misspell a POSIX-style time zone specification, since there is no check on the reasonableness of the zone abbreviation(s). For example, `SET TIMEZONE TO FOOBAR0` will work, leaving the system effectively using a rather peculiar abbreviation for UTC.

B.6. History of Units

The SQL standard states that "Within the definition of a 'datetime literal', the 'datetime values' are constrained by the natural rules for dates and times according to the Gregorian calendar". PostgreSQL follows the SQL standard's lead by counting dates exclusively in the Gregorian calendar, even for years before that calendar was in use. This rule is known as the *proleptic Gregorian calendar*.

The Julian calendar was introduced by Julius Caesar in 45 BC. It was in common use in the Western world until the year 1582, when countries started changing to the Gregorian calendar. In the Julian calendar, the tropical year is approximated as 365 1/4 days = 365.25 days. This gives an error of about 1 day in 128 years.

The accumulating calendar error prompted Pope Gregory XIII to reform the calendar in accordance with instructions from the Council of Trent. In the Gregorian calendar, the tropical year is approximated as 365 + 97 / 400 days = 365.2425 days. Thus it takes approximately 3300 years for the tropical year to shift one day with respect to the Gregorian calendar.

The approximation 365+97/400 is achieved by having 97 leap years every 400 years, using the following rules:

Every year divisible by 4 is a leap year.
However, every year divisible by 100 is not a leap year.
However, every year divisible by 400 is a leap year after all.

So, 1700, 1800, 1900, 2100, and 2200 are not leap years. But 1600, 2000, and 2400 are leap years. By contrast, in the older Julian calendar all years divisible by 4 are leap years.

The papal bull of February 1582 decreed that 10 days should be dropped from October 1582 so that 15 October should follow immediately after 4 October. This was observed in Italy, Poland, Portugal, and Spain. Other Catholic countries followed shortly after, but Protestant countries were reluctant to change, and the Greek Orthodox countries didn't change until the start of the 20th century. The reform was observed by Great Britain and its dominions (including what is now the USA) in 1752. Thus 2 September 1752 was followed by 14 September 1752. This is why Unix systems that have the `cal` program produce the following:

```
$ cal 9 1752
   September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

But, of course, this calendar is only valid for Great Britain and dominions, not other places. Since it would be difficult and confusing to try to track the actual calendars that were in use in various places at various times, PostgreSQL does not try, but rather follows the Gregorian calendar rules for all dates, even though this method is not historically accurate.

Different calendars have been developed in various parts of the world, many predating the Gregorian system. For example, the beginnings of the Chinese calendar can be traced back to the 14th century BC. Legend has it that the Emperor Huangdi invented that calendar in 2637 BC. The People's Republic of China uses the Gregorian calendar for civil purposes. The Chinese calendar is used for determining festivals.

The *Julian Date* system is another type of calendar, unrelated to the Julian calendar though it is confusingly named similarly to that calendar. The Julian Date system was invented by the French scholar Joseph Justus Scaliger (1540-1609) and probably takes its name from Scaliger's father, the Italian scholar Julius Caesar Scaliger (1484-1558). In the Julian Date system, each day has a sequential number, starting from JD 0 (which is sometimes called *the* Julian Date). JD 0 corresponds to 1 January 4713 BC in the Julian calendar, or 24 November 4714 BC in the Gregorian calendar. Julian Date counting is most often used by astronomers for labeling their nightly observations, and therefore a date runs from noon UTC to the next noon UTC, rather than from midnight to midnight: JD 0 designates the 24 hours from noon UTC on 24 November 4714 BC to noon UTC on 25 November 4714 BC.

Although PostgreSQL supports Julian Date notation for input and output of dates (and also uses Julian dates for some internal datetime calculations), it does not observe the nicety of having dates run from noon to noon. PostgreSQL treats a Julian Date as running from midnight to midnight.

Appendix C. SQL Key Words

Table C-1 lists all tokens that are key words in the SQL standard and in PostgreSQL 9.5.24. Background information can be found in Section 4.1.1. (For space reasons, only the latest two versions of the SQL standard, and SQL-92 for historical comparison, are included. The differences between those and the other intermediate standard versions are small.)

SQL distinguishes between *reserved* and *non-reserved* key words. According to the standard, reserved key words are the only real key words; they are never allowed as identifiers. Non-reserved key words only have a special meaning in particular contexts and can be used as identifiers in other contexts. Most non-reserved key words are actually the names of built-in tables and functions specified by SQL. The concept of non-reserved key words essentially only exists to declare that some predefined meaning is attached to a word in some contexts.

In the PostgreSQL parser life is a bit more complicated. There are several different classes of tokens ranging from those that can never be used as an identifier to those that have absolutely no special status in the parser as compared to an ordinary identifier. (The latter is usually the case for functions specified by SQL.) Even reserved key words are not completely reserved in PostgreSQL, but can be used as column labels (for example, `SELECT 55 AS CHECK`, even though `CHECK` is a reserved key word).

In Table C-1 in the column for PostgreSQL we classify as "non-reserved" those key words that are explicitly known to the parser but are allowed as column or table names. Some key words that are otherwise non-reserved cannot be used as function or data type names and are marked accordingly. (Most of these words represent built-in functions or data types with special syntax. The function or type is still available but it cannot be redefined by the user.) Labeled "reserved" are those tokens that are not allowed as column or table names. Some reserved key words are allowable as names for functions or data types; this is also shown in the table. If not so marked, a reserved key word is only allowed as an "AS" column label name.

As a general rule, if you get spurious parser errors for commands that contain any of the listed key words as an identifier you should try to quote the identifier to see if the problem goes away.

It is important to understand before studying Table C-1 that the fact that a key word is not reserved in PostgreSQL does not mean that the feature related to the word is not implemented. Conversely, the presence of a key word does not indicate the existence of a feature.

Table C-1. SQL Key Words

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
A		non-reserved	non-reserved	
ABORT	non-reserved			
ABS		reserved	reserved	
ABSENT		non-reserved	non-reserved	
ABSOLUTE	non-reserved	non-reserved	non-reserved	reserved
ACCESS	non-reserved			
ACCORDING		non-reserved	non-reserved	

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
ACTION	non-reserved	non-reserved	non-reserved	reserved
ADA		non-reserved	non-reserved	non-reserved
ADD	non-reserved	non-reserved	non-reserved	reserved
ADMIN	non-reserved	non-reserved	non-reserved	
AFTER	non-reserved	non-reserved	non-reserved	
AGGREGATE	non-reserved			
ALL	reserved	reserved	reserved	reserved
ALLOCATE		reserved	reserved	reserved
ALSO	non-reserved			
ALTER	non-reserved	reserved	reserved	reserved
ALWAYS	non-reserved	non-reserved	non-reserved	
ANALYSE	reserved			
ANALYZE	reserved			
AND	reserved	reserved	reserved	reserved
ANY	reserved	reserved	reserved	reserved
ARE		reserved	reserved	reserved
ARRAY	reserved	reserved	reserved	
ARRAY_AGG		reserved	reserved	
ARRAY_MAX_CARDINALITY		reserved		
AS	reserved	reserved	reserved	reserved
ASC	reserved	non-reserved	non-reserved	reserved
ASENSITIVE		reserved	reserved	
ASSERTION	non-reserved	non-reserved	non-reserved	reserved
ASSIGNMENT	non-reserved	non-reserved	non-reserved	
ASYMMETRIC	reserved	reserved	reserved	
AT	non-reserved	reserved	reserved	reserved
ATOMIC		reserved	reserved	
ATTRIBUTE	non-reserved	non-reserved	non-reserved	
ATTRIBUTES		non-reserved	non-reserved	
AUTHORIZATION	reserved (can be function or type)	reserved	reserved	reserved
AVG		reserved	reserved	reserved
BACKWARD	non-reserved			
BASE64		non-reserved	non-reserved	
BEFORE	non-reserved	non-reserved	non-reserved	
BEGIN	non-reserved	reserved	reserved	reserved
BEGIN_FRAME		reserved		

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
BEGIN_PARTITION		reserved		
BERNOULLI		non-reserved	non-reserved	
BETWEEN	non-reserved (cannot be function or type)	reserved	reserved	reserved
BIGINT	non-reserved (cannot be function or type)	reserved	reserved	
BINARY	reserved (can be function or type)	reserved	reserved	
BIT	non-reserved (cannot be function or type)			reserved
BIT_LENGTH				reserved
BLOB		reserved	reserved	
BLOCKED		non-reserved	non-reserved	
BOM		non-reserved	non-reserved	
BOOLEAN	non-reserved (cannot be function or type)	reserved	reserved	
BOTH	reserved	reserved	reserved	reserved
BREADTH		non-reserved	non-reserved	
BY	non-reserved	reserved	reserved	reserved
C		non-reserved	non-reserved	non-reserved
CACHE	non-reserved			
CALL		reserved	reserved	
CALLED	non-reserved	reserved	reserved	
CARDINALITY		reserved	reserved	
CASCADE	non-reserved	non-reserved	non-reserved	reserved
CASCADED	non-reserved	reserved	reserved	reserved
CASE	reserved	reserved	reserved	reserved
CAST	reserved	reserved	reserved	reserved
CATALOG	non-reserved	non-reserved	non-reserved	reserved
CATALOG_NAME		non-reserved	non-reserved	non-reserved
CEIL		reserved	reserved	
CEILING		reserved	reserved	
CHAIN	non-reserved	non-reserved	non-reserved	

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
CHAR	non-reserved (cannot be function or type)	reserved	reserved	reserved
CHARACTER	non-reserved (cannot be function or type)	reserved	reserved	reserved
CHARACTERISTICS	non-reserved	non-reserved	non-reserved	
CHARACTERS		non-reserved	non-reserved	
CHARACTER_LENGTH		reserved	reserved	reserved
CHARACTER_SET_CATALOG		non-reserved	non-reserved	non-reserved
CHARACTER_SET_NAME		non-reserved	non-reserved	non-reserved
CHARACTER_SET_SCHEMA		non-reserved	non-reserved	non-reserved
CHAR_LENGTH		reserved	reserved	reserved
CHECK	reserved	reserved	reserved	reserved
CHECKPOINT	non-reserved			
CLASS	non-reserved			
CLASS_ORIGIN		non-reserved	non-reserved	non-reserved
CLOB		reserved	reserved	
CLOSE	non-reserved	reserved	reserved	reserved
CLUSTER	non-reserved			
COALESCE	non-reserved (cannot be function or type)	reserved	reserved	reserved
COBOL		non-reserved	non-reserved	non-reserved
COLLATE	reserved	reserved	reserved	reserved
COLLATION	reserved (can be function or type)	non-reserved	non-reserved	reserved
COLLATION_CATALOG		non-reserved	non-reserved	non-reserved
COLLATION_NAME		non-reserved	non-reserved	non-reserved
COLLATION_SCHEMA		non-reserved	non-reserved	non-reserved
COLLECT		reserved	reserved	
COLUMN	reserved	reserved	reserved	reserved
COLUMNS		non-reserved	non-reserved	
COLUMN_NAME		non-reserved	non-reserved	non-reserved

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
COMMAND_FUNCTION		non-reserved	non-reserved	non-reserved
COMMAND_FUNCTION_CODE		non-reserved	non-reserved	
COMMENT	non-reserved			
COMMENTS	non-reserved			
COMMIT	non-reserved	reserved	reserved	reserved
COMMITTED	non-reserved	non-reserved	non-reserved	non-reserved
CONCURRENTLY	reserved (can be function or type)			
CONDITION		reserved	reserved	
CONDITION_NUMBER		non-reserved	non-reserved	non-reserved
CONFIGURATION	non-reserved			
CONFLICT	non-reserved			
CONNECT		reserved	reserved	reserved
CONNECTION	non-reserved	non-reserved	non-reserved	reserved
CONNECTION_NAME		non-reserved	non-reserved	non-reserved
CONSTRAINT	reserved	reserved	reserved	reserved
CONSTRAINTS	non-reserved	non-reserved	non-reserved	reserved
CONSTRAINT_CATALOG		non-reserved	non-reserved	non-reserved
CONSTRAINT_NAME		non-reserved	non-reserved	non-reserved
CONSTRAINT_SCHEMA		non-reserved	non-reserved	non-reserved
CONSTRUCTOR		non-reserved	non-reserved	
CONTAINS		reserved	non-reserved	
CONTENT	non-reserved	non-reserved	non-reserved	
CONTINUE	non-reserved	non-reserved	non-reserved	reserved
CONTROL		non-reserved	non-reserved	
CONVERSION	non-reserved			
CONVERT		reserved	reserved	reserved
COPY	non-reserved			
CORR		reserved	reserved	
CORRESPONDING		reserved	reserved	reserved
COST	non-reserved			
COUNT		reserved	reserved	reserved
COVAR_POP		reserved	reserved	

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
COVAR_SAMP		reserved	reserved	
CREATE	reserved	reserved	reserved	reserved
CROSS	reserved (can be function or type)	reserved	reserved	reserved
CSV	non-reserved			
CUBE	non-reserved	reserved	reserved	
CUME_DIST		reserved	reserved	
CURRENT	non-reserved	reserved	reserved	reserved
CURRENT_CATALOG	reserved	reserved	reserved	
CURRENT_DATE	reserved	reserved	reserved	reserved
CURRENT_DEFAULT_TRANSFORM_GROUP		reserved	reserved	
CURRENT_PATH		reserved	reserved	
CURRENT_ROLE	reserved	reserved	reserved	
CURRENT_ROW		reserved		
CURRENT_SCHEMA	reserved (can be function or type)	reserved	reserved	
CURRENT_TIME	reserved	reserved	reserved	reserved
CURRENT_TIMESTAMP	reserved	reserved	reserved	reserved
CURRENT_TRANSFORM_GROUP_FOR_TYPE		reserved	reserved	
CURRENT_USER	reserved	reserved	reserved	reserved
CURSOR	non-reserved	reserved	reserved	reserved
CURSOR_NAME		non-reserved	non-reserved	non-reserved
CYCLE	non-reserved	reserved	reserved	
DATA	non-reserved	non-reserved	non-reserved	non-reserved
DATABASE	non-reserved			
DATALINK		reserved	reserved	
DATE		reserved	reserved	reserved
DATETIME_INTERVAL_CODE		non-reserved	non-reserved	non-reserved
DATETIME_INTERVAL_PRECISION		non-reserved	non-reserved	non-reserved
DAY	non-reserved	reserved	reserved	reserved
DB		non-reserved	non-reserved	
DEALLOCATE	non-reserved	reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
DEC	non-reserved (cannot be function or type)	reserved	reserved	reserved
DECIMAL	non-reserved (cannot be function or type)	reserved	reserved	reserved
DECLARE	non-reserved	reserved	reserved	reserved
DEFAULT	reserved	reserved	reserved	reserved
DEFAULTS	non-reserved	non-reserved	non-reserved	
DEFERRABLE	reserved	non-reserved	non-reserved	reserved
DEFERRED	non-reserved	non-reserved	non-reserved	reserved
DEFINED		non-reserved	non-reserved	
DEFINER	non-reserved	non-reserved	non-reserved	
DEGREE		non-reserved	non-reserved	
DELETE	non-reserved	reserved	reserved	reserved
DELIMITER	non-reserved			
DELIMITERS	non-reserved			
DENSE_RANK		reserved	reserved	
DEPTH		non-reserved	non-reserved	
DEREF		reserved	reserved	
DERIVED		non-reserved	non-reserved	
DESC	reserved	non-reserved	non-reserved	reserved
DESCRIBE		reserved	reserved	reserved
DESCRIPTOR		non-reserved	non-reserved	reserved
DETERMINISTIC		reserved	reserved	
DIAGNOSTICS		non-reserved	non-reserved	reserved
DICTIONARY	non-reserved			
DISABLE	non-reserved			
DISCARD	non-reserved			
DISCONNECT		reserved	reserved	reserved
DISPATCH		non-reserved	non-reserved	
DISTINCT	reserved	reserved	reserved	reserved
DLNEWCOPY		reserved	reserved	
DLPREVIOUSCOPY		reserved	reserved	
DLURLCOMPLETE		reserved	reserved	
DLURLCOMPLETEONLY		reserved	reserved	
DLURLCOMPLETEWRITE		reserved	reserved	

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
DLURLPATH		reserved	reserved	
DLURLPATHONLY		reserved	reserved	
DLURLPATHWRITE		reserved	reserved	
DLURLSCHEME		reserved	reserved	
DLURLSERVER		reserved	reserved	
DLVALUE		reserved	reserved	
DO	reserved			
DOCUMENT	non-reserved	non-reserved	non-reserved	
DOMAIN	non-reserved	non-reserved	non-reserved	reserved
DOUBLE	non-reserved	reserved	reserved	reserved
DROP	non-reserved	reserved	reserved	reserved
DYNAMIC		reserved	reserved	
DYNAMIC_FUNCTION		non-reserved	non-reserved	non-reserved
DYNAMIC_FUNCTION_CODE		non-reserved	non-reserved	
EACH	non-reserved	reserved	reserved	
ELEMENT		reserved	reserved	
ELSE	reserved	reserved	reserved	reserved
EMPTY		non-reserved	non-reserved	
ENABLE	non-reserved			
ENCODING	non-reserved	non-reserved	non-reserved	
ENCRYPTED	non-reserved			
END	reserved	reserved	reserved	reserved
END-EXEC		reserved	reserved	reserved
END_FRAME		reserved		
END_PARTITION		reserved		
ENFORCED		non-reserved		
ENUM	non-reserved			
EQUALS		reserved	non-reserved	
ESCAPE	non-reserved	reserved	reserved	reserved
EVENT	non-reserved			
EVERY		reserved	reserved	
EXCEPT	reserved	reserved	reserved	reserved
EXCEPTION				reserved
EXCLUDE	non-reserved	non-reserved	non-reserved	
EXCLUDING	non-reserved	non-reserved	non-reserved	
EXCLUSIVE	non-reserved			
EXEC		reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
EXECUTE	non-reserved	reserved	reserved	reserved
EXISTS	non-reserved (cannot be function or type)	reserved	reserved	reserved
EXP		reserved	reserved	
EXPLAIN	non-reserved			
EXPRESSION		non-reserved		
EXTENSION	non-reserved			
EXTERNAL	non-reserved	reserved	reserved	reserved
EXTRACT	non-reserved (cannot be function or type)	reserved	reserved	reserved
FALSE	reserved	reserved	reserved	reserved
FAMILY	non-reserved			
FETCH	reserved	reserved	reserved	reserved
FILE		non-reserved	non-reserved	
FILTER	non-reserved	reserved	reserved	
FINAL		non-reserved	non-reserved	
FIRST	non-reserved	non-reserved	non-reserved	reserved
FIRST_VALUE		reserved	reserved	
FLAG		non-reserved	non-reserved	
FLOAT	non-reserved (cannot be function or type)	reserved	reserved	reserved
FLOOR		reserved	reserved	
FOLLOWING	non-reserved	non-reserved	non-reserved	
FOR	reserved	reserved	reserved	reserved
FORCE	non-reserved			
FOREIGN	reserved	reserved	reserved	reserved
FORTRAN		non-reserved	non-reserved	non-reserved
FORWARD	non-reserved			
FOUND		non-reserved	non-reserved	reserved
FRAME_ROW		reserved		
FREE		reserved	reserved	
FREEZE	reserved (can be function or type)			
FROM	reserved	reserved	reserved	reserved
FS		non-reserved	non-reserved	
FULL	reserved (can be function or type)	reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
FUNCTION	non-reserved	reserved	reserved	
FUNCTIONS	non-reserved			
FUSION		reserved	reserved	
G		non-reserved	non-reserved	
GENERAL		non-reserved	non-reserved	
GENERATED		non-reserved	non-reserved	
GET		reserved	reserved	reserved
GLOBAL	non-reserved	reserved	reserved	reserved
GO		non-reserved	non-reserved	reserved
GOTO		non-reserved	non-reserved	reserved
GRANT	reserved	reserved	reserved	reserved
GRANTED	non-reserved	non-reserved	non-reserved	
GREATEST	non-reserved (cannot be function or type)			
GROUP	reserved	reserved	reserved	reserved
GROUPING	non-reserved (cannot be function or type)	reserved	reserved	
GROUPS		reserved		
HANDLER	non-reserved			
HAVING	reserved	reserved	reserved	reserved
HEADER	non-reserved			
HEX		non-reserved	non-reserved	
HIERARCHY		non-reserved	non-reserved	
HOLD	non-reserved	reserved	reserved	
HOUR	non-reserved	reserved	reserved	reserved
ID		non-reserved	non-reserved	
IDENTITY	non-reserved	reserved	reserved	reserved
IF	non-reserved			
IGNORE		non-reserved	non-reserved	
ILIKE	reserved (can be function or type)			
IMMEDIATE	non-reserved	non-reserved	non-reserved	reserved
IMMEDIATELY		non-reserved		
IMMUTABLE	non-reserved			
IMPLEMENTATION		non-reserved	non-reserved	
IMPLICIT	non-reserved			
IMPORT	non-reserved	reserved	reserved	

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
IN	reserved	reserved	reserved	reserved
INCLUDING	non-reserved	non-reserved	non-reserved	
INCREMENT	non-reserved	non-reserved	non-reserved	
INDENT		non-reserved	non-reserved	
INDEX	non-reserved			
INDEXES	non-reserved			
INDICATOR		reserved	reserved	reserved
INHERIT	non-reserved			
INHERITS	non-reserved			
INITIALLY	reserved	non-reserved	non-reserved	reserved
INLINE	non-reserved			
INNER	reserved (can be function or type)	reserved	reserved	reserved
INOUT	non-reserved (cannot be function or type)	reserved	reserved	
INPUT	non-reserved	non-reserved	non-reserved	reserved
INSENSITIVE	non-reserved	reserved	reserved	reserved
INSERT	non-reserved	reserved	reserved	reserved
INSTANCE		non-reserved	non-reserved	
INSTANTIABLE		non-reserved	non-reserved	
INSTEAD	non-reserved	non-reserved	non-reserved	
INT	non-reserved (cannot be function or type)	reserved	reserved	reserved
INTEGER	non-reserved (cannot be function or type)	reserved	reserved	reserved
INTEGRITY		non-reserved	non-reserved	
INTERSECT	reserved	reserved	reserved	reserved
INTERSECTION		reserved	reserved	
INTERVAL	non-reserved (cannot be function or type)	reserved	reserved	reserved
INTO	reserved	reserved	reserved	reserved
INVOKER	non-reserved	non-reserved	non-reserved	
IS	reserved (can be function or type)	reserved	reserved	reserved
ISNULL	reserved (can be function or type)			

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
ISOLATION	non-reserved	non-reserved	non-reserved	reserved
JOIN	reserved (can be function or type)	reserved	reserved	reserved
K		non-reserved	non-reserved	
KEY	non-reserved	non-reserved	non-reserved	reserved
KEY_MEMBER		non-reserved	non-reserved	
KEY_TYPE		non-reserved	non-reserved	
LABEL	non-reserved			
LAG		reserved	reserved	
LANGUAGE	non-reserved	reserved	reserved	reserved
LARGE	non-reserved	reserved	reserved	
LAST	non-reserved	non-reserved	non-reserved	reserved
LAST_VALUE		reserved	reserved	
LATERAL	reserved	reserved	reserved	
LEAD		reserved	reserved	
LEADING	reserved	reserved	reserved	reserved
LEAKPROOF	non-reserved			
LEAST	non-reserved (cannot be function or type)			
LEFT	reserved (can be function or type)	reserved	reserved	reserved
LENGTH		non-reserved	non-reserved	non-reserved
LEVEL	non-reserved	non-reserved	non-reserved	reserved
LIBRARY		non-reserved	non-reserved	
LIKE	reserved (can be function or type)	reserved	reserved	reserved
LIKE_REGEX		reserved	reserved	
LIMIT	reserved	non-reserved	non-reserved	
LINK		non-reserved	non-reserved	
LISTEN	non-reserved			
LN		reserved	reserved	
LOAD	non-reserved			
LOCAL	non-reserved	reserved	reserved	reserved
LOCALTIME	reserved	reserved	reserved	
LOCALTIMESTAMP	reserved	reserved	reserved	
LOCATION	non-reserved	non-reserved	non-reserved	
LOCATOR		non-reserved	non-reserved	
LOCK	non-reserved			

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
LOCKED	non-reserved			
LOGGED	non-reserved			
LOWER		reserved	reserved	reserved
M		non-reserved	non-reserved	
MAP		non-reserved	non-reserved	
MAPPING	non-reserved	non-reserved	non-reserved	
MATCH	non-reserved	reserved	reserved	reserved
MATCHED		non-reserved	non-reserved	
MATERIALIZED	non-reserved			
MAX		reserved	reserved	reserved
MAXVALUE	non-reserved	non-reserved	non-reserved	
MAX_CARDINALITY			reserved	
MEMBER		reserved	reserved	
MERGE		reserved	reserved	
MESSAGE_LENGTH		non-reserved	non-reserved	non-reserved
MESSAGE_OCTET_LENGTH		non-reserved	non-reserved	non-reserved
MESSAGE_TEXT		non-reserved	non-reserved	non-reserved
METHOD		reserved	reserved	
MIN		reserved	reserved	reserved
MINUTE	non-reserved	reserved	reserved	reserved
MINVALUE	non-reserved	non-reserved	non-reserved	
MOD		reserved	reserved	
MODE	non-reserved			
MODIFIES		reserved	reserved	
MODULE		reserved	reserved	reserved
MONTH	non-reserved	reserved	reserved	reserved
MORE		non-reserved	non-reserved	non-reserved
MOVE	non-reserved			
MULTISET		reserved	reserved	
MUMPS		non-reserved	non-reserved	non-reserved
NAME	non-reserved	non-reserved	non-reserved	non-reserved
NAMES	non-reserved	non-reserved	non-reserved	reserved
NAMESPACE		non-reserved	non-reserved	
NATIONAL	non-reserved (cannot be function or type)	reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
NATURAL	reserved (can be function or type)	reserved	reserved	reserved
NCHAR	non-reserved (cannot be function or type)	reserved	reserved	reserved
NCLOB		reserved	reserved	
NESTING		non-reserved	non-reserved	
NEW		reserved	reserved	
NEXT	non-reserved	non-reserved	non-reserved	reserved
NFC		non-reserved	non-reserved	
NFD		non-reserved	non-reserved	
NFKC		non-reserved	non-reserved	
NFKD		non-reserved	non-reserved	
NIL		non-reserved	non-reserved	
NO	non-reserved	reserved	reserved	reserved
NONE	non-reserved (cannot be function or type)	reserved	reserved	
NORMALIZE		reserved	reserved	
NORMALIZED		non-reserved	non-reserved	
NOT	reserved	reserved	reserved	reserved
NOTHING	non-reserved			
NOTIFY	non-reserved			
NOTNULL	reserved (can be function or type)			
NOWAIT	non-reserved			
NTH_VALUE		reserved	reserved	
NTILE		reserved	reserved	
NULL	reserved	reserved	reserved	reserved
NULLABLE		non-reserved	non-reserved	non-reserved
NULLIF	non-reserved (cannot be function or type)	reserved	reserved	reserved
NULLS	non-reserved	non-reserved	non-reserved	
NUMBER		non-reserved	non-reserved	non-reserved
NUMERIC	non-reserved (cannot be function or type)	reserved	reserved	reserved
OBJECT	non-reserved	non-reserved	non-reserved	

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
OCCURRENCES_REGEX		reserved	reserved	
OCTETS		non-reserved	non-reserved	
OCTET_LENGTH		reserved	reserved	reserved
OF	non-reserved	reserved	reserved	reserved
OFF	non-reserved	non-reserved	non-reserved	
OFFSET	reserved	reserved	reserved	
OIDS	non-reserved			
OLD		reserved	reserved	
ON	reserved	reserved	reserved	reserved
ONLY	reserved	reserved	reserved	reserved
OPEN		reserved	reserved	reserved
OPERATOR	non-reserved			
OPTION	non-reserved	non-reserved	non-reserved	reserved
OPTIONS	non-reserved	non-reserved	non-reserved	
OR	reserved	reserved	reserved	reserved
ORDER	reserved	reserved	reserved	reserved
ORDERING		non-reserved	non-reserved	
ORDINALITY	non-reserved	non-reserved	non-reserved	
OTHERS		non-reserved	non-reserved	
OUT	non-reserved (cannot be function or type)	reserved	reserved	
OUTER	reserved (can be function or type)	reserved	reserved	reserved
OUTPUT		non-reserved	non-reserved	reserved
OVER	non-reserved	reserved	reserved	
OVERLAPS	reserved (can be function or type)	reserved	reserved	reserved
OVERLAY	non-reserved (cannot be function or type)	reserved	reserved	
OVERRIDING		non-reserved	non-reserved	
OWNED	non-reserved			
OWNER	non-reserved			
P		non-reserved	non-reserved	
PAD		non-reserved	non-reserved	reserved
PARAMETER		reserved	reserved	
PARAMETER_MODE		non-reserved	non-reserved	
PARAMETER_NAME		non-reserved	non-reserved	

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
PARAMETER_ORDINAL_POSITION		non-reserved	non-reserved	
PARAMETER_SPECIFIC_CATALOG		non-reserved	non-reserved	
PARAMETER_SPECIFIC_NAME		non-reserved	non-reserved	
PARAMETER_SPECIFIC_SCHEMA		non-reserved	non-reserved	
PARSER	non-reserved			
PARTIAL	non-reserved	non-reserved	non-reserved	reserved
PARTITION	non-reserved	reserved	reserved	
PASCAL		non-reserved	non-reserved	non-reserved
PASSING	non-reserved	non-reserved	non-reserved	
PASSTHROUGH		non-reserved	non-reserved	
PASSWORD	non-reserved			
PATH		non-reserved	non-reserved	
PERCENT		reserved		
PERCENTILE_CONT		reserved	reserved	
PERCENTILE_DISC		reserved	reserved	
PERCENT_RANK		reserved	reserved	
PERIOD		reserved		
PERMISSION		non-reserved	non-reserved	
PLACING	reserved	non-reserved	non-reserved	
PLANS	non-reserved			
PLI		non-reserved	non-reserved	non-reserved
POLICY	non-reserved			
PORTION		reserved		
POSITION	non-reserved (cannot be function or type)	reserved	reserved	reserved
POSITION_REGEX		reserved	reserved	
POWER		reserved	reserved	
PRECEDES		reserved		
PRECEDING	non-reserved	non-reserved	non-reserved	
PRECISION	non-reserved (cannot be function or type)	reserved	reserved	reserved
PREPARE	non-reserved	reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
PREPARED	non-reserved			
PRESERVE	non-reserved	non-reserved	non-reserved	reserved
PRIMARY	reserved	reserved	reserved	reserved
PRIOR	non-reserved	non-reserved	non-reserved	reserved
PRIVILEGES	non-reserved	non-reserved	non-reserved	reserved
PROCEDURAL	non-reserved			
PROCEDURE	non-reserved	reserved	reserved	reserved
PROGRAM	non-reserved			
PUBLIC		non-reserved	non-reserved	reserved
QUOTE	non-reserved			
RANGE	non-reserved	reserved	reserved	
RANK		reserved	reserved	
READ	non-reserved	non-reserved	non-reserved	reserved
READS		reserved	reserved	
REAL	non-reserved (cannot be function or type)	reserved	reserved	reserved
REASSIGN	non-reserved			
RECHECK	non-reserved			
RECOVERY		non-reserved	non-reserved	
RECURSIVE	non-reserved	reserved	reserved	
REF	non-reserved	reserved	reserved	
REFERENCES	reserved	reserved	reserved	reserved
REFERENCING		reserved	reserved	
REFRESH	non-reserved			
REGR_AVGX		reserved	reserved	
REGR_AVGY		reserved	reserved	
REGR_COUNT		reserved	reserved	
REGR_INTERCEPT		reserved	reserved	
REGR_R2		reserved	reserved	
REGR_SLOPE		reserved	reserved	
REGR_SXX		reserved	reserved	
REGR_SXY		reserved	reserved	
REGR_SYY		reserved	reserved	
REINDEX	non-reserved			
RELATIVE	non-reserved	non-reserved	non-reserved	reserved
RELEASE	non-reserved	reserved	reserved	
RENAME	non-reserved			
REPEATABLE	non-reserved	non-reserved	non-reserved	non-reserved

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
REPLACE	non-reserved			
REPLICA	non-reserved			
REQUIRING		non-reserved	non-reserved	
RESET	non-reserved			
RESPECT		non-reserved	non-reserved	
RESTART	non-reserved	non-reserved	non-reserved	
RESTORE		non-reserved	non-reserved	
RESTRICT	non-reserved	non-reserved	non-reserved	reserved
RESULT		reserved	reserved	
RETURN		reserved	reserved	
RETURNED_CARDINALITY		non-reserved	non-reserved	
RETURNED_LENGTH		non-reserved	non-reserved	non-reserved
RETURNED_OCTET_LENGTH		non-reserved	non-reserved	non-reserved
RETURNED_SQLSTATE		non-reserved	non-reserved	non-reserved
RETURNING	reserved	non-reserved	non-reserved	
RETURNS	non-reserved	reserved	reserved	
REVOKE	non-reserved	reserved	reserved	reserved
RIGHT	reserved (can be function or type)	reserved	reserved	reserved
ROLE	non-reserved	non-reserved	non-reserved	
ROLLBACK	non-reserved	reserved	reserved	reserved
ROLLUP	non-reserved	reserved	reserved	
ROUTINE		non-reserved	non-reserved	
ROUTINE_CATALOG		non-reserved	non-reserved	
ROUTINE_NAME		non-reserved	non-reserved	
ROUTINE_SCHEMA		non-reserved	non-reserved	
ROW	non-reserved (cannot be function or type)	reserved	reserved	
ROWS	non-reserved	reserved	reserved	reserved
ROW_COUNT		non-reserved	non-reserved	non-reserved
ROW_NUMBER		reserved	reserved	
RULE	non-reserved			
SAVEPOINT	non-reserved	reserved	reserved	
SCALE		non-reserved	non-reserved	non-reserved

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
SCHEMA	non-reserved	non-reserved	non-reserved	reserved
SCHEMA_NAME		non-reserved	non-reserved	non-reserved
SCOPE		reserved	reserved	
SCOPE_CATALOG		non-reserved	non-reserved	
SCOPE_NAME		non-reserved	non-reserved	
SCOPE_SCHEMA		non-reserved	non-reserved	
SCROLL	non-reserved	reserved	reserved	reserved
SEARCH	non-reserved	reserved	reserved	
SECOND	non-reserved	reserved	reserved	reserved
SECTION		non-reserved	non-reserved	reserved
SECURITY	non-reserved	non-reserved	non-reserved	
SELECT	reserved	reserved	reserved	reserved
SELECTIVE		non-reserved	non-reserved	
SELF		non-reserved	non-reserved	
SENSITIVE		reserved	reserved	
SEQUENCE	non-reserved	non-reserved	non-reserved	
SEQUENCES	non-reserved			
SERIALIZABLE	non-reserved	non-reserved	non-reserved	non-reserved
SERVER	non-reserved	non-reserved	non-reserved	
SERVER_NAME		non-reserved	non-reserved	non-reserved
SESSION	non-reserved	non-reserved	non-reserved	reserved
SESSION_USER	reserved	reserved	reserved	reserved
SET	non-reserved	reserved	reserved	reserved
SETOF	non-reserved (cannot be function or type)			
SETS	non-reserved	non-reserved	non-reserved	
SHARE	non-reserved			
SHOW	non-reserved			
SIMILAR	reserved (can be function or type)	reserved	reserved	
SIMPLE	non-reserved	non-reserved	non-reserved	
SIZE		non-reserved	non-reserved	reserved
SKIP	non-reserved			
SMALLINT	non-reserved (cannot be function or type)	reserved	reserved	reserved
SNAPSHOT	non-reserved			
SOME	reserved	reserved	reserved	reserved

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
SOURCE		non-reserved	non-reserved	
SPACE		non-reserved	non-reserved	reserved
SPECIFIC		reserved	reserved	
SPECIFICTYPE		reserved	reserved	
SPECIFIC_NAME		non-reserved	non-reserved	
SQL	non-reserved	reserved	reserved	reserved
SQLCODE				reserved
SQLERROR				reserved
SQLEXCEPTION		reserved	reserved	
SQLSTATE		reserved	reserved	reserved
SQLWARNING		reserved	reserved	
SQRT		reserved	reserved	
STABLE	non-reserved			
STANDALONE	non-reserved	non-reserved	non-reserved	
START	non-reserved	reserved	reserved	
STATE		non-reserved	non-reserved	
STATEMENT	non-reserved	non-reserved	non-reserved	
STATIC		reserved	reserved	
STATISTICS	non-reserved			
STDDEV_POP		reserved	reserved	
STDDEV_SAMP		reserved	reserved	
STDIN	non-reserved			
STDOUT	non-reserved			
STORAGE	non-reserved			
STRICT	non-reserved			
STRIP	non-reserved	non-reserved	non-reserved	
STRUCTURE		non-reserved	non-reserved	
STYLE		non-reserved	non-reserved	
SUBCLASS_ORIGIN		non-reserved	non-reserved	non-reserved
SUBMULTISET		reserved	reserved	
SUBSTRING	non-reserved (cannot be function or type)	reserved	reserved	reserved
SUBSTRING_REGEX		reserved	reserved	
SUCCEEDS		reserved		
SUM		reserved	reserved	reserved
SYMMETRIC	reserved	reserved	reserved	

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
SYSID	non-reserved			
SYSTEM	non-reserved	reserved	reserved	
SYSTEM_TIME		reserved		
SYSTEM_USER		reserved	reserved	reserved
T		non-reserved	non-reserved	
TABLE	reserved	reserved	reserved	reserved
TABLES	non-reserved			
TABLESAMPLE	reserved (can be function or type)	reserved	reserved	
TABLESPACE	non-reserved			
TABLE_NAME		non-reserved	non-reserved	non-reserved
TEMP	non-reserved			
TEMPLATE	non-reserved			
TEMPORARY	non-reserved	non-reserved	non-reserved	reserved
TEXT	non-reserved			
THEN	reserved	reserved	reserved	reserved
TIES		non-reserved	non-reserved	
TIME	non-reserved (cannot be function or type)	reserved	reserved	reserved
TIMESTAMP	non-reserved (cannot be function or type)	reserved	reserved	reserved
TIMEZONE_HOUR		reserved	reserved	reserved
TIMEZONE_MINUTE		reserved	reserved	reserved
TO	reserved	reserved	reserved	reserved
TOKEN		non-reserved	non-reserved	
TOP_LEVEL_COUNT		non-reserved	non-reserved	
TRAILING	reserved	reserved	reserved	reserved
TRANSACTION	non-reserved	non-reserved	non-reserved	reserved
TRANSACTIONS_COMMITTED		non-reserved	non-reserved	
TRANSACTIONS_ROLLED_BACK		non-reserved	non-reserved	
TRANSACTION_ACTIVE		non-reserved	non-reserved	
TRANSFORM	non-reserved	non-reserved	non-reserved	
TRANSFORMS		non-reserved	non-reserved	

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
TRANSLATE		reserved	reserved	reserved
TRANSLATE_REGEX		reserved	reserved	
TRANSLATION		reserved	reserved	reserved
TREAT	non-reserved (cannot be function or type)	reserved	reserved	
TRIGGER	non-reserved	reserved	reserved	
TRIGGER_CATALOG		non-reserved	non-reserved	
TRIGGER_NAME		non-reserved	non-reserved	
TRIGGER_SCHEMA		non-reserved	non-reserved	
TRIM	non-reserved (cannot be function or type)	reserved	reserved	reserved
TRIM_ARRAY		reserved	reserved	
TRUE	reserved	reserved	reserved	reserved
TRUNCATE	non-reserved	reserved	reserved	
TRUSTED	non-reserved			
TYPE	non-reserved	non-reserved	non-reserved	non-reserved
TYPES	non-reserved			
UESCAPE		reserved	reserved	
UNBOUNDED	non-reserved	non-reserved	non-reserved	
UNCOMMITTED	non-reserved	non-reserved	non-reserved	non-reserved
UNDER		non-reserved	non-reserved	
UNENCRYPTED	non-reserved			
UNION	reserved	reserved	reserved	reserved
UNIQUE	reserved	reserved	reserved	reserved
UNKNOWN	non-reserved	reserved	reserved	reserved
UNLINK		non-reserved	non-reserved	
UNLISTEN	non-reserved			
UNLOGGED	non-reserved			
UNNAMED		non-reserved	non-reserved	non-reserved
UNNEST		reserved	reserved	
UNTIL	non-reserved			
UNTYPED		non-reserved	non-reserved	
UPDATE	non-reserved	reserved	reserved	reserved
UPPER		reserved	reserved	reserved
URI		non-reserved	non-reserved	

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
USAGE		non-reserved	non-reserved	reserved
USER	reserved	reserved	reserved	reserved
USER_DEFINED_TYPE_CATALOG		non-reserved	non-reserved	
USER_DEFINED_TYPE_CODE		non-reserved	non-reserved	
USER_DEFINED_TYPE_NAME		non-reserved	non-reserved	
USER_DEFINED_TYPE_SCHEMA		non-reserved	non-reserved	
USING	reserved	reserved	reserved	reserved
VACUUM	non-reserved			
VALID	non-reserved	non-reserved	non-reserved	
VALIDATE	non-reserved			
VALIDATOR	non-reserved			
VALUE	non-reserved	reserved	reserved	reserved
VALUES	non-reserved (cannot be function or type)	reserved	reserved	reserved
VALUE_OF		reserved		
VARBINARY		reserved	reserved	
VARCHAR	non-reserved (cannot be function or type)	reserved	reserved	reserved
VARIADIC	reserved			
VARYING	non-reserved	reserved	reserved	reserved
VAR_POP		reserved	reserved	
VAR_SAMP		reserved	reserved	
VERBOSE	reserved (can be function or type)			
VERSION	non-reserved	non-reserved	non-reserved	
VERSIONING		reserved		
VIEW	non-reserved	non-reserved	non-reserved	reserved
VIEWS	non-reserved			
VOLATILE	non-reserved			
WHEN	reserved	reserved	reserved	reserved
WHENEVER		reserved	reserved	reserved
WHERE	reserved	reserved	reserved	reserved
WHITESPACE	non-reserved	non-reserved	non-reserved	
WIDTH_BUCKET		reserved	reserved	

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
WINDOW	reserved	reserved	reserved	
WITH	reserved	reserved	reserved	reserved
WITHIN	non-reserved	reserved	reserved	
WITHOUT	non-reserved	reserved	reserved	
WORK	non-reserved	non-reserved	non-reserved	reserved
WRAPPER	non-reserved	non-reserved	non-reserved	
WRITE	non-reserved	non-reserved	non-reserved	reserved
XML	non-reserved	reserved	reserved	
XMLAGG		reserved	reserved	
XMLATTRIBUTES	non-reserved (cannot be function or type)	reserved	reserved	
XMLBINARY		reserved	reserved	
XMLCAST		reserved	reserved	
XMLCOMMENT		reserved	reserved	
XMLCONCAT	non-reserved (cannot be function or type)	reserved	reserved	
XMLDECLARATION		non-reserved	non-reserved	
XMLDOCUMENT		reserved	reserved	
XMLELEMENT	non-reserved (cannot be function or type)	reserved	reserved	
XMLEXISTS	non-reserved (cannot be function or type)	reserved	reserved	
XMLFOREST	non-reserved (cannot be function or type)	reserved	reserved	
XMLITERATE		reserved	reserved	
XMLNAMESPACES		reserved	reserved	
XMLPARSE	non-reserved (cannot be function or type)	reserved	reserved	
XMLPI	non-reserved (cannot be function or type)	reserved	reserved	
XMLQUERY		reserved	reserved	
XMLROOT	non-reserved (cannot be function or type)			

Key Word	PostgreSQL	SQL:2011	SQL:2008	SQL-92
XMLSCHEMA		non-reserved	non-reserved	
XMLSERIALIZE	non-reserved (cannot be function or type)	reserved	reserved	
XMLTABLE		reserved	reserved	
XMLTEXT		reserved	reserved	
XMLVALIDATE		reserved	reserved	
YEAR	non-reserved	reserved	reserved	reserved
YES	non-reserved	non-reserved	non-reserved	
ZONE	non-reserved	non-reserved	non-reserved	reserved

Appendix D. SQL Conformance

This section attempts to outline to what extent PostgreSQL conforms to the current SQL standard. The following information is not a full statement of conformance, but it presents the main topics in as much detail as is both reasonable and useful for users.

The formal name of the SQL standard is ISO/IEC 9075 “Database Language SQL”. A revised version of the standard is released from time to time; the most recent update appearing in 2011. The 2011 version is referred to as ISO/IEC 9075:2011, or simply as SQL:2011. The versions prior to that were SQL:2008, SQL:2003, SQL:1999, and SQL-92. Each version replaces the previous one, so claims of conformance to earlier versions have no official merit. PostgreSQL development aims for conformance with the latest official version of the standard where such conformance does not contradict traditional features or common sense. Many of the features required by the SQL standard are supported, though sometimes with slightly differing syntax or function. Further moves towards conformance can be expected over time.

SQL-92 defined three feature sets for conformance: Entry, Intermediate, and Full. Most database management systems claiming SQL standard conformance were conforming at only the Entry level, since the entire set of features in the Intermediate and Full levels was either too voluminous or in conflict with legacy behaviors.

Starting with SQL:1999, the SQL standard defines a large set of individual features rather than the ineffectively broad three levels found in SQL-92. A large subset of these features represents the “Core” features, which every conforming SQL implementation must supply. The rest of the features are purely optional. Some optional features are grouped together to form “packages”, which SQL implementations can claim conformance to, thus claiming conformance to particular groups of features.

The standard versions beginning with SQL:2003 are also split into a number of parts. Each is known by a shorthand name. Note that these parts are not consecutively numbered.

- ISO/IEC 9075-1 Framework (SQL/Framework)
- ISO/IEC 9075-2 Foundation (SQL/Foundation)
- ISO/IEC 9075-3 Call Level Interface (SQL/CLI)
- ISO/IEC 9075-4 Persistent Stored Modules (SQL/PSM)
- ISO/IEC 9075-9 Management of External Data (SQL/MED)
- ISO/IEC 9075-10 Object Language Bindings (SQL/OLB)
- ISO/IEC 9075-11 Information and Definition Schemas (SQL/Schemata)
- ISO/IEC 9075-13 Routines and Types using the Java Language (SQL/JRT)
- ISO/IEC 9075-14 XML-related specifications (SQL/XML)

The PostgreSQL core covers parts 1, 2, 9, 11, and 14. Part 3 is covered by the ODBC driver, and part 13 is covered by the PL/Java plug-in, but exact conformance is currently not being verified for these components. There are currently no implementations of parts 4 and 10 for PostgreSQL.

PostgreSQL supports most of the major features of SQL:2011. Out of 179 mandatory features required for full Core conformance, PostgreSQL conforms to at least 160. In addition, there is a long list of supported optional features. It might be worth noting that at the time of writing, no current version of any database management system claims full conformance to Core SQL:2011.

In the following two sections, we provide a list of those features that PostgreSQL supports, followed by a list of the features defined in SQL:2011 which are not yet supported in PostgreSQL. Both of these lists are approximate: There might be minor details that are nonconforming for a feature that is listed as supported, and large parts of an unsupported feature might in fact be implemented. The main body of the documentation always contains the most accurate information about what does and does not work.

Note: Feature codes containing a hyphen are subfeatures. Therefore, if a particular subfeature is not supported, the main feature is listed as unsupported even if some other subfeatures are supported.

D.1. Supported Features

Identifier	Package	Description	Comment
B012		Embedded C	
B021		Direct SQL	
E011	Core	Numeric data types	
E011-01	Core	INTEGER and SMALLINT data types	
E011-02	Core	REAL, DOUBLE PRECISION, and FLOAT data types	
E011-03	Core	DECIMAL and NUMERIC data types	
E011-04	Core	Arithmetic operators	
E011-05	Core	Numeric comparison	
E011-06	Core	Implicit casting among the numeric data types	
E021	Core	Character data types	
E021-01	Core	CHARACTER data type	
E021-02	Core	CHARACTER VARYING data type	
E021-03	Core	Character literals	
E021-04	Core	CHARACTER_LENGTH function	trims trailing spaces from CHARACTER values before counting

Identifier	Package	Description	Comment
E021-05	Core	OCTET_LENGTH function	
E021-06	Core	SUBSTRING function	
E021-07	Core	Character concatenation	
E021-08	Core	UPPER and LOWER functions	
E021-09	Core	TRIM function	
E021-10	Core	Implicit casting among the character string types	
E021-11	Core	POSITION function	
E021-12	Core	Character comparison	
E031	Core	Identifiers	
E031-01	Core	Delimited identifiers	
E031-02	Core	Lower case identifiers	
E031-03	Core	Trailing underscore	
E051	Core	Basic query specification	
E051-01	Core	SELECT DISTINCT	
E051-02	Core	GROUP BY clause	
E051-04	Core	GROUP BY can contain columns not in <select list>	
E051-05	Core	Select list items can be renamed	
E051-06	Core	HAVING clause	
E051-07	Core	Qualified * in select list	
E051-08	Core	Correlation names in the FROM clause	
E051-09	Core	Rename columns in the FROM clause	
E061	Core	Basic predicates and search conditions	
E061-01	Core	Comparison predicate	
E061-02	Core	BETWEEN predicate	
E061-03	Core	IN predicate with list of values	
E061-04	Core	LIKE predicate	
E061-05	Core	LIKE predicate ESCAPE clause	
E061-06	Core	NULL predicate	

Identifier	Package	Description	Comment
E061-07	Core	Quantified comparison predicate	
E061-08	Core	EXISTS predicate	
E061-09	Core	Subqueries in comparison predicate	
E061-11	Core	Subqueries in IN predicate	
E061-12	Core	Subqueries in quantified comparison predicate	
E061-13	Core	Correlated subqueries	
E061-14	Core	Search condition	
E071	Core	Basic query expressions	
E071-01	Core	UNION DISTINCT table operator	
E071-02	Core	UNION ALL table operator	
E071-03	Core	EXCEPT DISTINCT table operator	
E071-05	Core	Columns combined via table operators need not have exactly the same data type	
E071-06	Core	Table operators in subqueries	
E081	Core	Basic Privileges	
E081-01	Core	SELECT privilege	
E081-02	Core	DELETE privilege	
E081-03	Core	INSERT privilege at the table level	
E081-04	Core	UPDATE privilege at the table level	
E081-05	Core	UPDATE privilege at the column level	
E081-06	Core	REFERENCES privilege at the table level	
E081-07	Core	REFERENCES privilege at the column level	
E081-08	Core	WITH GRANT OPTION	
E081-09	Core	USAGE privilege	

Identifier	Package	Description	Comment
E081-10	Core	EXECUTE privilege	
E091	Core	Set functions	
E091-01	Core	AVG	
E091-02	Core	COUNT	
E091-03	Core	MAX	
E091-04	Core	MIN	
E091-05	Core	SUM	
E091-06	Core	ALL quantifier	
E091-07	Core	DISTINCT quantifier	
E101	Core	Basic data manipulation	
E101-01	Core	INSERT statement	
E101-03	Core	Searched UPDATE statement	
E101-04	Core	Searched DELETE statement	
E111	Core	Single row SELECT statement	
E121	Core	Basic cursor support	
E121-01	Core	DECLARE CURSOR	
E121-02	Core	ORDER BY columns need not be in select list	
E121-03	Core	Value expressions in ORDER BY clause	
E121-04	Core	OPEN statement	
E121-06	Core	Positioned UPDATE statement	
E121-07	Core	Positioned DELETE statement	
E121-08	Core	CLOSE statement	
E121-10	Core	FETCH statement implicit NEXT	
E121-17	Core	WITH HOLD cursors	
E131	Core	Null value support (nulls in lieu of values)	
E141	Core	Basic integrity constraints	
E141-01	Core	NOT NULL constraints	
E141-02	Core	UNIQUE constraints of NOT NULL columns	
E141-03	Core	PRIMARY KEY constraints	

Identifier	Package	Description	Comment
E141-04	Core	Basic FOREIGN KEY constraint with the NO ACTION default for both referential delete action and referential update action	
E141-06	Core	CHECK constraints	
E141-07	Core	Column defaults	
E141-08	Core	NOT NULL inferred on PRIMARY KEY	
E141-10	Core	Names in a foreign key can be specified in any order	
E151	Core	Transaction support	
E151-01	Core	COMMIT statement	
E151-02	Core	ROLLBACK statement	
E152	Core	Basic SET TRANSACTION statement	
E152-01	Core	SET TRANSACTION statement: ISOLATION LEVEL SERIALIZABLE clause	
E152-02	Core	SET TRANSACTION statement: READ ONLY and READ WRITE clauses	
E153	Core	Updatable queries with subqueries	
E161	Core	SQL comments using leading double minus	
E171	Core	SQLSTATE support	
F021	Core	Basic information schema	
F021-01	Core	COLUMNS view	
F021-02	Core	TABLES view	
F021-03	Core	VIEWS view	
F021-04	Core	TABLE_CONSTRAINTS view	
F021-05	Core	REFERENTIAL_CONSTRAINTS view	

Identifier	Package	Description	Comment
F021-06	Core	CHECK_CONSTRAINTS view	
F031	Core	Basic schema manipulation	
F031-01	Core	CREATE TABLE statement to create persistent base tables	
F031-02	Core	CREATE VIEW statement	
F031-03	Core	GRANT statement	
F031-04	Core	ALTER TABLE statement: ADD COLUMN clause	
F031-13	Core	DROP TABLE statement: RESTRICT clause	
F031-16	Core	DROP VIEW statement: RESTRICT clause	
F031-19	Core	REVOKE statement: RESTRICT clause	
F032		CASCADE drop behavior	
F033		ALTER TABLE statement: DROP COLUMN clause	
F034		Extended REVOKE statement	
F034-01		REVOKE statement performed by other than the owner of a schema object	
F034-02		REVOKE statement: GRANT OPTION FOR clause	
F034-03		REVOKE statement to revoke a privilege that the grantee has WITH GRANT OPTION	
F041	Core	Basic joined table	
F041-01	Core	Inner join (but not necessarily the INNER keyword)	
F041-02	Core	INNER keyword	

Identifier	Package	Description	Comment
F041-03	Core	LEFT OUTER JOIN	
F041-04	Core	RIGHT OUTER JOIN	
F041-05	Core	Outer joins can be nested	
F041-07	Core	The inner table in a left or right outer join can also be used in an inner join	
F041-08	Core	All comparison operators are supported (rather than just =)	
F051	Core	Basic date and time	
F051-01	Core	DATE data type (including support of DATE literal)	
F051-02	Core	TIME data type (including support of TIME literal) with fractional seconds precision of at least 0	
F051-03	Core	TIMESTAMP data type (including support of TIMESTAMP literal) with fractional seconds precision of at least 0 and 6	
F051-04	Core	Comparison predicate on DATE, TIME, and TIMESTAMP data types	
F051-05	Core	Explicit CAST between datetime types and character string types	
F051-06	Core	CURRENT_DATE	
F051-07	Core	LOCALTIME	
F051-08	Core	LOCALTIMESTAMP	
F052	Enhanced datetime facilities	Intervals and datetime arithmetic	
F053		OVERLAPS predicate	
F081	Core	UNION and EXCEPT in views	
F111		Isolation levels other than SERIALIZABLE	

Identifier	Package	Description	Comment
F111-01		READ UNCOMMITTED isolation level	
F111-02		READ COMMITTED isolation level	
F111-03		REPEATABLE READ isolation level	
F131	Core	Grouped operations	
F131-01	Core	WHERE, GROUP BY, and HAVING clauses supported in queries with grouped views	
F131-02	Core	Multiple tables supported in queries with grouped views	
F131-03	Core	Set functions supported in queries with grouped views	
F131-04	Core	Subqueries with GROUP BY and HAVING clauses and grouped views	
F131-05	Core	Single row SELECT with GROUP BY and HAVING clauses and grouped views	
F171		Multiple schemas per user	
F191	Enhanced integrity management	Referential delete actions	
F200		TRUNCATE TABLE statement	
F201	Core	CAST function	
F221	Core	Explicit defaults	
F222		INSERT statement: DEFAULT VALUES clause	
F231		Privilege tables	
F231-01		TABLE_PRIVILEGES view	
F231-02		COLUMN_PRIVILEGES view	

Identifier	Package	Description	Comment
F231-03		USAGE_PRIVILEGES view	
F251		Domain support	
F261	Core	CASE expression	
F261-01	Core	Simple CASE	
F261-02	Core	Searched CASE	
F261-03	Core	NULLIF	
F261-04	Core	COALESCE	
F262		Extended CASE expression	
F271		Compound character literals	
F281		LIKE enhancements	
F302		INTERSECT table operator	
F302-01		INTERSECT DISTINCT table operator	
F302-02		INTERSECT ALL table operator	
F304		EXCEPT ALL table operator	
F311-01	Core	CREATE SCHEMA	
F311-02	Core	CREATE TABLE for persistent base tables	
F311-03	Core	CREATE VIEW	
F311-04	Core	CREATE VIEW: WITH CHECK OPTION	
F311-05	Core	GRANT statement	
F321		User authorization	
F361		Subprogram support	
F381		Extended schema manipulation	
F381-01		ALTER TABLE statement: ALTER COLUMN clause	
F381-02		ALTER TABLE statement: ADD CONSTRAINT clause	
F381-03		ALTER TABLE statement: DROP CONSTRAINT clause	

Identifier	Package	Description	Comment
F382		Alter column data type	
F383		Set column not null clause	
F391		Long identifiers	
F392		Unicode escapes in identifiers	
F393		Unicode escapes in literals	
F401		Extended joined table	
F401-01		NATURAL JOIN	
F401-02		FULL OUTER JOIN	
F401-04		CROSS JOIN	
F402		Named column joins for LOBs, arrays, and multisets	
F411	Enhanced datetime facilities	Time zone specification	differences regarding literal interpretation
F421		National character	
F431		Read-only scrollable cursors	
F431-01		FETCH with explicit NEXT	
F431-02		FETCH FIRST	
F431-03		FETCH LAST	
F431-04		FETCH PRIOR	
F431-05		FETCH ABSOLUTE	
F431-06		FETCH RELATIVE	
F441		Extended set function support	
F442		Mixed column references in set functions	
F471	Core	Scalar subquery values	
F481	Core	Expanded NULL predicate	
F491	Enhanced integrity management	Constraint management	
F501	Core	Features and conformance views	
F501-01	Core	SQL_FEATURES view	
F501-02	Core	SQL_SIZING view	

Identifier	Package	Description	Comment
F501-03	Core	SQL_LANGUAGES view	
F502		Enhanced documentation tables	
F502-01		SQL_SIZING_PROFILES view	
F502-02		SQL_IMPLEMENTATION_INFO view	
F502-03		SQL_PACKAGES view	
F531		Temporary tables	
F555	Enhanced datetime facilities	Enhanced seconds precision	
F561		Full value expressions	
F571		Truth value tests	
F591		Derived tables	
F611		Indicator data types	
F641		Row and table constructors	
F651		Catalog name qualifiers	
F661		Simple tables	
F672		Retrospective check constraints	
F690		Collation support	but no character set support
F692		Extended collation support	
F701	Enhanced integrity management	Referential update actions	
F711		ALTER domain	
F731		INSERT column privileges	
F751		View CHECK enhancements	
F761		Session management	
F762		CURRENT_CATALOG	
F763		CURRENT_SCHEMA	
F771		Connection management	
F781		Self-referencing operations	

Identifier	Package	Description	Comment
F791		Insensitive cursors	
F801		Full set function	
F850		Top-level <order by clause> in <query expression>	
F851		<order by clause> in subqueries	
F852		Top-level <order by clause> in views	
F855		Nested <order by clause> in <query expression>	
F856		Nested <fetch first clause> in <query expression>	
F857		Top-level <fetch first clause> in <query expression>	
F858		<fetch first clause> in subqueries	
F859		Top-level <fetch first clause> in views	
F860		<fetch first row count> in <fetch first clause>	
F861		Top-level <result offset clause> in <query expression>	
F862		<result offset clause> in subqueries	
F863		Nested <result offset clause> in <query expression>	
F864		Top-level <result offset clause> in views	
F865		<offset row count> in <result offset clause>	
S071	Enhanced object support	SQL paths in function and type name resolution	
S092		Arrays of user-defined types	
S095		Array constructors by query	

Identifier	Package	Description	Comment
S096		Optional array bounds	
S098		ARRAY_AGG	
S111	Enhanced object support	ONLY in query expressions	
S201		SQL-invoked routines on arrays	
S201-01		Array parameters	
S201-02		Array as result type of functions	
S211	Enhanced object support	User-defined cast functions	
S301		Enhanced UNNEST	
T031		BOOLEAN data type	
T071		BIGINT data type	
T121		WITH (excluding RECURSIVE) in query expression	
T122		WITH (excluding RECURSIVE) in subquery	
T131		Recursive query	
T132		Recursive query in subquery	
T141		SIMILAR predicate	
T151		DISTINCT predicate	
T152		DISTINCT predicate with negation	
T171		LIKE clause in table definition	
T172		AS subquery clause in table definition	
T173		Extended LIKE clause in table definition	
T191	Enhanced integrity management	Referential action RESTRICT	
T201	Enhanced integrity management	Comparable data types for referential constraints	
T211-01	Active database, Enhanced integrity management	Triggers activated on UPDATE, INSERT, or DELETE of one base table	

Identifier	Package	Description	Comment
T211-02	Active database, Enhanced integrity management	BEFORE triggers	
T211-03	Active database, Enhanced integrity management	AFTER triggers	
T211-04	Active database, Enhanced integrity management	FOR EACH ROW triggers	
T211-05	Active database, Enhanced integrity management	Ability to specify a search condition that must be true before the trigger is invoked	
T211-07	Active database, Enhanced integrity management	TRIGGER privilege	
T212	Enhanced integrity management	Enhanced trigger capability	
T213		INSTEAD OF triggers	
T231		Sensitive cursors	
T241		START TRANSACTION statement	
T271		Savepoints	
T281		SELECT privilege with column granularity	
T312		OVERLAY function	
T321-01	Core	User-defined functions with no overloading	
T321-03	Core	Function invocation	
T321-06	Core	ROUTINES view	
T321-07	Core	PARAMETERS view	
T323		Explicit security for external routines	
T325		Qualified SQL parameter references	
T331		Basic roles	
T341		Overloading of SQL-invoked functions and procedures	

Identifier	Package	Description	Comment
T351		Bracketed SQL comments (/*...*/ comments)	
T431	OLAP	Extended grouping capabilities	
T432		Nested and concatenated GROUPING SETS	
T433		Multiargument GROUPING function	
T441		ABS and MOD functions	
T461		Symmetric BETWEEN predicate	
T491		LATERAL derived table	
T501		Enhanced EXISTS predicate	
T551		Optional key words for default syntax	
T581		Regular expression substring function	
T591		UNIQUE constraints of possibly null columns	
T611	OLAP	Elementary OLAP operations	
T613		Sampling	
T614		NTILE function	
T615		LEAD and LAG functions	
T617		FIRST_VALUE and LAST_VALUE function	
T621		Enhanced numeric functions	
T631	Core	IN predicate with one list element	
T651		SQL-schema statements in SQL routines	
T655		Cyclically dependent routines	
X010		XML type	
X011		Arrays of XML type	

Identifier	Package	Description	Comment
X016		Persistent XML values	
X020		XMLConcat	
X031		XMLElement	
X032		XMLForest	
X034		XMLAgg	
X035		XMLAgg: ORDER BY option	
X036		XMLComment	
X037		XMLPI	
X040		Basic table mapping	
X041		Basic table mapping: nulls absent	
X042		Basic table mapping: null as nil	
X043		Basic table mapping: table as forest	
X044		Basic table mapping: table as element	
X045		Basic table mapping: with target namespace	
X046		Basic table mapping: data mapping	
X047		Basic table mapping: metadata mapping	
X048		Basic table mapping: base64 encoding of binary strings	
X049		Basic table mapping: hex encoding of binary strings	
X050		Advanced table mapping	
X051		Advanced table mapping: nulls absent	
X052		Advanced table mapping: null as nil	
X053		Advanced table mapping: table as forest	
X054		Advanced table mapping: table as element	

Identifier	Package	Description	Comment
X055		Advanced table mapping: with target namespace	
X056		Advanced table mapping: data mapping	
X057		Advanced table mapping: metadata mapping	
X058		Advanced table mapping: base64 encoding of binary strings	
X059		Advanced table mapping: hex encoding of binary strings	
X060		XMLParse: character string input and CONTENT option	
X061		XMLParse: character string input and DOCUMENT option	
X070		XMLSerialize: character string serialization and CONTENT option	
X071		XMLSerialize: character string serialization and DOCUMENT option	
X072		XMLSerialize: character string serialization	
X090		XML document predicate	
X120		XML parameters in SQL routines	
X121		XML parameters in external routines	
X400		Name and identifier mapping	
X410		Alter column data type: XML type	

D.2. Unsupported Features

The following features defined in SQL:2011 are not implemented in this release of PostgreSQL. In a few cases, equivalent functionality is available.

Identifier	Package	Description	Comment
B011		Embedded Ada	
B013		Embedded COBOL	
B014		Embedded Fortran	
B015		Embedded MUMPS	
B016		Embedded Pascal	
B017		Embedded PL/I	
B031		Basic dynamic SQL	
B032		Extended dynamic SQL	
B032-01		<describe input statement>	
B033		Untyped SQL-invoked function arguments	
B034		Dynamic specification of cursor attributes	
B035		Non-extended descriptor names	
B041		Extensions to embedded SQL exception declarations	
B051		Enhanced execution rights	
B111		Module language Ada	
B112		Module language C	
B113		Module language COBOL	
B114		Module language Fortran	
B115		Module language MUMPS	
B116		Module language Pascal	
B117		Module language PL/I	
B121		Routine language Ada	
B122		Routine language C	
B123		Routine language COBOL	

Identifier	Package	Description	Comment
B124		Routine language Fortran	
B125		Routine language MUMPS	
B126		Routine language Pascal	
B127		Routine language PL/I	
B128		Routine language SQL	
B211		Module language Ada: VARCHAR and NUMERIC support	
B221		Routine language Ada: VARCHAR and NUMERIC support	
E182	Core	Module language	
F054		TIMESTAMP in DATE type precedence list	
F121		Basic diagnostics management	
F121-01		GET DIAGNOSTICS statement	
F121-02		SET TRANSACTION statement: DIAGNOSTICS SIZE clause	
F122		Enhanced diagnostics management	
F123		All diagnostics	
F181	Core	Multiple module support	
F202		TRUNCATE TABLE: identity column restart option	
F263		Comma-separated predicates in simple CASE expression	
F291		UNIQUE predicate	
F301		CORRESPONDING in query expressions	
F311	Core	Schema definition statement	

Identifier	Package	Description	Comment
F312		MERGE statement	consider INSERT ... ON CONFLICT DO UPDATE
F313		Enhanced MERGE statement	
F314		MERGE statement with DELETE branch	
F341		Usage tables	no ROUTINE_*_USAGE tables
F384		Drop identity property clause	
F385		Drop column generation expression clause	
F386		Set identity column generation clause	
F394		Optional normal form specification	
F403		Partitioned joined tables	
F451		Character set definition	
F461		Named character sets	
F492		Optional table constraint enforcement	
F521	Enhanced integrity management	Assertions	
F671	Enhanced integrity management	Subqueries in CHECK	intentionally omitted
F693		SQL-session and client module collations	
F695		Translation support	
F696		Additional translation documentation	
F721		Deferrable constraints	foreign and unique keys only
F741		Referential MATCH types	no partial match yet
F812	Core	Basic flagging	
F813		Extended flagging	
F821		Local table references	
F831		Full cursor update	
F831-01		Updatable scrollable cursors	

Identifier	Package	Description	Comment
F831-02		Updatable ordered cursors	
F841		LIKE_REGEX predicate	
F842		OCCURENCES_REGEX function	
F843		POSITION_REGEX function	
F844		SUBSTRING_REGEX function	
F845		TRANSLATE_REGEX function	
F846		Octet support in regular expression operators	
F847		Nonconstant regular expressions	
F866		FETCH FIRST clause: PERCENT option	
F867		FETCH FIRST clause: WITH TIES option	
S011	Core	Distinct data types	
S011-01	Core	USER_DEFINED_TYPES view	
S023	Basic object support	Basic structured types	
S024	Enhanced object support	Enhanced structured types	
S025		Final structured types	
S026		Self-referencing structured types	
S027		Create method by specific method name	
S028		Permutable UDT options list	
S041	Basic object support	Basic reference types	
S043	Enhanced object support	Enhanced reference types	
S051	Basic object support	Create table of type	partially supported
S081	Enhanced object support	Subtables	
S091		Basic array support	partially supported
S091-01		Arrays of built-in data types	

Identifier	Package	Description	Comment
S091-02		Arrays of distinct types	
S091-03		Array expressions	
S094		Arrays of reference types	
S097		Array element assignment	
S151	Basic object support	Type predicate	
S161	Enhanced object support	Subtype treatment	
S162		Subtype treatment for references	
S202		SQL-invoked routines on multisets	
S231	Enhanced object support	Structured type locators	
S232		Array locators	
S233		Multiset locators	
S241		Transform functions	
S242		Alter transform statement	
S251		User-defined orderings	
S261		Specific type method	
S271		Basic multiset support	
S272		Multisets of user-defined types	
S274		Multisets of reference types	
S275		Advanced multiset support	
S281		Nested collection types	
S291		Unique constraint on entire row	
S401		Distinct types based on array types	
S402		Distinct types based on distinct types	
S403		ARRAY_MAX_CARDINALITY	
S404		TRIM_ARRAY	
T011		Timestamp in Information Schema	

Identifier	Package	Description	Comment
T021		BINARY and VARBINARY data types	
T022		Advanced support for BINARY and VARBINARY data types	
T023		Compound binary literal	
T024		Spaces in binary literals	
T041	Basic object support	Basic LOB data type support	
T041-01	Basic object support	BLOB data type	
T041-02	Basic object support	CLOB data type	
T041-03	Basic object support	POSITION, LENGTH, LOWER, TRIM, UPPER, and SUBSTRING functions for LOB data types	
T041-04	Basic object support	Concatenation of LOB data types	
T041-05	Basic object support	LOB locator: non-holdable	
T042		Extended LOB data type support	
T043		Multiplier T	
T044		Multiplier P	
T051		Row types	
T052		MAX and MIN for row types	
T053		Explicit aliases for all-fields reference	
T061		UCS support	
T101		Enhanced nullability determination	
T111		Updatable joins, unions, and columns	
T174		Identity columns	
T175		Generated columns	
T176		Sequence generator support	

Identifier	Package	Description	Comment
T177		Sequence generator support: simple restart option	
T178		Identity columns: simple restart option	
T180		System-versioned tables	
T181		Application-time period tables	
T211	Active database, Enhanced integrity management	Basic trigger capability	
T211-06	Active database, Enhanced integrity management	Support for run-time rules for the interaction of triggers and constraints	
T211-08	Active database, Enhanced integrity management	Multiple triggers for the same event are executed in the order in which they were created in the catalog	intentionally omitted
T251		SET TRANSACTION statement: LOCAL option	
T261		Chained transactions	
T272		Enhanced savepoint management	
T285		Enhanced derived column names	
T301		Functional dependencies	partially supported
T321	Core	Basic SQL-invoked routines	
T321-02	Core	User-defined stored procedures with no overloading	
T321-04	Core	CALL statement	
T321-05	Core	RETURN statement	
T322	PSM	Declared data type attributes	
T324		Explicit security for SQL routines	
T326		Table functions	

Identifier	Package	Description	Comment
T332		Extended roles	mostly supported
T434		GROUP BY DISTINCT	
T471		Result sets return value	
T472		DESCRIBE CURSOR	
T495		Combined data change and retrieval	different syntax
T502		Period predicates	
T511		Transaction counts	
T521		Named arguments in CALL statement	
T522		Default values for IN parameters of SQL-invoked procedures	supported except DEFAULT key word in invocation
T561		Holdable locators	
T571		Array-returning external SQL-invoked functions	
T572		Multiset-returning external SQL-invoked functions	
T601		Local cursor references	
T612		Advanced OLAP operations	some forms supported
T616		Null treatment option for LEAD and LAG functions	
T618		NTH_VALUE function	function exists, but some options missing
T619		Nested window functions	
T620		WINDOW clause: GROUPS option	
T641		Multiple column assignment	only some syntax variants supported
T652		SQL-dynamic statements in SQL routines	
T653		SQL-schema statements in external routines	

Identifier	Package	Description	Comment
T654		SQL-dynamic statements in external routines	
M001		Datalinks	
M002		Datalinks via SQL/CLI	
M003		Datalinks via Embedded SQL	
M004		Foreign data support	partially supported
M005		Foreign schema support	
M006		GetSQLString routine	
M007		TransmitRequest	
M009		GetOpts and GetStatistics routines	
M010		Foreign data wrapper support	different API
M011		Datalinks via Ada	
M012		Datalinks via C	
M013		Datalinks via COBOL	
M014		Datalinks via Fortran	
M015		Datalinks via M	
M016		Datalinks via Pascal	
M017		Datalinks via PL/I	
M018		Foreign data wrapper interface routines in Ada	
M019		Foreign data wrapper interface routines in C	different API
M020		Foreign data wrapper interface routines in COBOL	
M021		Foreign data wrapper interface routines in Fortran	
M022		Foreign data wrapper interface routines in MUMPS	
M023		Foreign data wrapper interface routines in Pascal	
M024		Foreign data wrapper interface routines in PL/I	

Identifier	Package	Description	Comment
M030		SQL-server foreign data support	
M031		Foreign data wrapper general routines	
X012		Multisets of XML type	
X013		Distinct types of XML type	
X014		Attributes of XML type	
X015		Fields of XML type	
X025		XMLCast	
X030		XMLDocument	
X038		XMLText	
X065		XMLParse: BLOB input and CONTENT option	
X066		XMLParse: BLOB input and DOCUMENT option	
X068		XMLSerialize: BOM	
X069		XMLSerialize: INDENT	
X073		XMLSerialize: BLOB serialization and CONTENT option	
X074		XMLSerialize: BLOB serialization and DOCUMENT option	
X075		XMLSerialize: BLOB serialization	
X076		XMLSerialize: VERSION	
X077		XMLSerialize: explicit ENCODING option	
X078		XMLSerialize: explicit XML declaration	
X080		Namespaces in XML publishing	
X081		Query-level XML namespace declarations	
X082		XML namespace declarations in DML	
X083		XML namespace declarations in DDL	

Identifier	Package	Description	Comment
X084		XML namespace declarations in compound statements	
X085		Predefined namespace prefixes	
X086		XML namespace declarations in XMLTable	
X091		XML content predicate	
X096		XMLExists	
X100		Host language support for XML: CONTENT option	
X101		Host language support for XML: DOCUMENT option	
X110		Host language support for XML: VARCHAR mapping	
X111		Host language support for XML: CLOB mapping	
X112		Host language support for XML: BLOB mapping	
X113		Host language support for XML: STRIP WHITESPACE option	
X114		Host language support for XML: PRESERVE WHITESPACE option	
X131		Query-level XMLBINARY clause	
X132		XMLBINARY clause in DML	
X133		XMLBINARY clause in DDL	
X134		XMLBINARY clause in compound statements	
X135		XMLBINARY clause in subqueries	
X141		IS VALID predicate: data-driven case	

Identifier	Package	Description	Comment
X142		IS VALID predicate: ACCORDING TO clause	
X143		IS VALID predicate: ELEMENT clause	
X144		IS VALID predicate: schema location	
X145		IS VALID predicate outside check constraints	
X151		IS VALID predicate with DOCUMENT option	
X152		IS VALID predicate with CONTENT option	
X153		IS VALID predicate with SEQUENCE option	
X155		IS VALID predicate: NAMESPACE without ELEMENT clause	
X157		IS VALID predicate: NO NAMESPACE with ELEMENT clause	
X160		Basic Information Schema for registered XML Schemas	
X161		Advanced Information Schema for registered XML Schemas	
X170		XML null handling options	
X171		NIL ON NO CONTENT option	
X181		XML(DOCUMENT(UNTYPED)) type	
X182		XML(DOCUMENT(ANY)) type	
X190		XML(SEQUENCE) type	
X191		XML(DOCUMENT(XMLSCHEMA)) type	

Identifier	Package	Description	Comment
X192		XML(CONTENT(XMLSCHEMA)) type	
X200		XMLQuery	
X201		XMLQuery: RETURNING CONTENT	
X202		XMLQuery: RETURNING SEQUENCE	
X203		XMLQuery: passing a context item	
X204		XMLQuery: initializing an XQuery variable	
X205		XMLQuery: EMPTY ON EMPTY option	
X206		XMLQuery: NULL ON EMPTY option	
X211		XML 1.1 support	
X221		XML passing mechanism BY VALUE	
X222		XML passing mechanism BY REF	
X231		XML(CONTENT(UNTYPED)) type	
X232		XML(CONTENT(ANY)) type	
X241		RETURNING CONTENT in XML publishing	
X242		RETURNING SEQUENCE in XML publishing	
X251		Persistent XML values of XML(DOCUMENT(UNTYPED)) type	
X252		Persistent XML values of XML(DOCUMENT(ANY)) type	

Identifier	Package	Description	Comment
X253		Persistent XML values of XML(CONTENT(UNTYPED)) type	
X254		Persistent XML values of XML(CONTENT(ANY)) type	
X255		Persistent XML values of XML(SEQUENCE) type	
X256		Persistent XML values of XML(DOCUMENT(XMLSCHEMA)) type	
X257		Persistent XML values of XML(CONTENT(XMLSCHEMA)) type	
X260		XML type: ELEMENT clause	
X261		XML type: NAMESPACE without ELEMENT clause	
X263		XML type: NO NAMESPACE with ELEMENT clause	
X264		XML type: schema location	
X271		XMLValidate: data-driven case	
X272		XMLValidate: ACCORDING TO clause	
X273		XMLValidate: ELEMENT clause	
X274		XMLValidate: schema location	
X281		XMLValidate: with DOCUMENT option	
X282		XMLValidate with CONTENT option	

Identifier	Package	Description	Comment
X283		XMLValidate with SEQUENCE option	
X284		XMLValidate NAMESPACE without ELEMENT clause	
X286		XMLValidate: NO NAMESPACE with ELEMENT clause	
X300		XMLTable	
X301		XMLTable: derived column list option	
X302		XMLTable: ordinality column option	
X303		XMLTable: column default option	
X304		XMLTable: passing a context item	
X305		XMLTable: initializing an XQuery variable	

Appendix E. Release Notes

The release notes contain the significant changes in each PostgreSQL release, with major features and migration issues listed at the top. The release notes do not contain changes that affect only a few users or changes that are internal and therefore not user-visible. For example, the optimizer is improved in almost every release, but the improvements are usually observed by users as simply faster queries.

A complete list of changes for each release can be obtained by viewing the Git logs for each release. The `pgsql-committers` email list[1] records all source code changes as well. There is also a web interface[2] that shows changes to specific files.

The name appearing next to each item represents the major developer for that item. Of course all changes involve community discussion and patch review, so each item is truly a community effort.

E.1. Release 9.5.24

Release date: 2020-11-12

This release contains a variety of fixes from 9.5.23. For information about new features in the 9.5 major release, see Section E.25.

The PostgreSQL community will stop releasing updates for the 9.5.X release series in February 2021. Users are encouraged to update to a newer release branch soon.

E.1.1. Migration to Version 9.5.24

A dump/restore is not required for those running 9.5.X.

However, if you are upgrading from a version earlier than 9.5.20, see Section E.5.

E.1.2. Changes

- Block `DECLARE CURSOR ... WITH HOLD` and firing of deferred triggers within index expressions and materialized view queries (Noah Misch)

  This is essentially a leak in the "security restricted operation" sandbox mechanism. An attacker having permission to create non-temporary SQL objects could parlay this leak to execute arbitrary SQL code as a superuser.

  The PostgreSQL Project thanks Etienne Stalmans for reporting this problem. (CVE-2020-25695)

- Fix usage of complex connection-string parameters in pg_dump, pg_restore, clusterdb, reindexdb, and vacuumdb (Tom Lane)

1. https://archives.postgresql.org/pgsql-committers/
2. https://git.postgresql.org/gitweb?p=postgresql.git;a=summary

The `-d` parameter of pg_dump and pg_restore, or the `--maintenance-db` parameter of the other programs mentioned, can be a "connection string" containing multiple connection parameters rather than just a database name. In cases where these programs need to initiate additional connections, such as parallel processing or processing of multiple databases, the connection string was forgotten and just the basic connection parameters (database name, host, port, and username) were used for the additional connections. This could lead to connection failures if the connection string included any other essential information, such as non-default SSL or GSS parameters. Worse, the connection might succeed but not be encrypted as intended, or be vulnerable to man-in-the-middle attacks that the intended connection parameters would have prevented. (CVE-2020-25694)

- When psql's `\connect` command re-uses connection parameters, ensure that all non-overridden parameters from a previous connection string are re-used (Tom Lane)

  This avoids cases where reconnection might fail due to omission of relevant parameters, such as non-default SSL or GSS options. Worse, the reconnection might succeed but not be encrypted as intended, or be vulnerable to man-in-the-middle attacks that the intended connection parameters would have prevented. This is largely the same problem as just cited for pg_dump et al, although psql's behavior is more complex since the user may intentionally override some connection parameters. (CVE-2020-25694)

- Prevent psql's `\gset` command from modifying specially-treated variables (Noah Misch)

  `\gset` without a prefix would overwrite whatever variables the server told it to. Thus, a compromised server could set specially-treated variables such as `PROMPT1`, giving the ability to execute arbitrary shell code in the user's session.

  The PostgreSQL Project thanks Nick Cleaton for reporting this problem. (CVE-2020-25696)

- Prevent possible data loss from concurrent truncations of SLRU logs (Noah Misch)

  This rare problem would manifest in later "apparent wraparound" or "could not access status of transaction" errors.

- Ensure that SLRU directories are properly fsync'd during checkpoints (Thomas Munro)

  This prevents possible data loss in a subsequent operating system crash.

- Fix `ALTER ROLE` for users with the `BYPASSRLS` attribute (Tom Lane, Stephen Frost)

  The `BYPASSRLS` attribute is only allowed to be changed by superusers, but other `ALTER ROLE` operations, such as password changes, should be allowed with only ordinary permission checks. The previous coding erroneously restricted all changes on such a role to superusers.

- Fix handling of expressions in `CREATE TABLE LIKE` with inheritance (Tom Lane)

  If a `CREATE TABLE` command uses both `LIKE` and traditional inheritance, column references in `CHECK` constraints and expression indexes that came from a `LIKE` parent table tended to get mis-numbered, resulting in wrong answers and/or bizarre error messages. The same could happen in `GENERATED` expressions, in branches that have that feature.

- Fix off-by-one conversion of negative years to BC dates in `to_date()` and `to_timestamp()` (Dar Alathar-Yemen, Tom Lane)

  Also, arrange for the combination of a negative year and an explicit "BC" marker to cancel out and produce AD.

- Ensure that standby servers will archive WAL timeline history files when `archive_mode` is set to `always` (Grigory Smolkin, Fujii Masao)

This oversight could lead to failure of subsequent PITR recovery attempts.

- Avoid running atexit handlers when exiting due to SIGQUIT (Kyotaro Horiguchi, Tom Lane)

  Most server processes followed this practice already, but the archiver process was overlooked. Backends that were still waiting for a client startup packet got it wrong, too.

- Avoid misoptimization of subquery qualifications that reference apparently-constant grouping columns (Tom Lane)

  A "constant" subquery output column isn't really constant if it is a grouping column that appears in only some of the grouping sets.

- Avoid failure when SQL function inlining changes the shape of a potentially-hashable subplan comparison expression (Tom Lane)

- While building or re-building an index, tolerate the appearance of new HOT chains due to concurrent updates (Anastasia Lubennikova, Álvaro Herrera)

  This oversight could lead to "failed to find parent tuple for heap-only tuple" errors.

- Ensure that data is detoasted before being inserted into a BRIN index (Tomas Vondra)

  Index entries are not supposed to contain out-of-line TOAST pointers, but BRIN didn't get that memo. This could lead to errors like "missing chunk number 0 for toast value NNN". (If you are faced with such an error from an existing index, `REINDEX` should be enough to fix it.)

- Handle concurrent desummarization correctly during BRIN index scans (Alexander Lakhin, Álvaro Herrera)

  Previously, if a page range was desummarized at just the wrong time, an index scan might falsely raise an error indicating index corruption.

- Fix rare "lost saved point in index" errors in scans of multicolumn GIN indexes (Tom Lane)

- Fix use-after-free hazard when an event trigger monitors an `ALTER TABLE` operation (Jehan-Guillaume de Rorthais)

- Fix incorrect error message about inconsistent moving-aggregate data types (Jeff Janes)

- Avoid lockup when a parallel worker reports a very long error message (Vignesh C)

- Avoid unnecessary failure when transferring very large payloads through shared memory queues (Markus Wanner)

- Fix relation cache memory leaks with RLS policies (Tom Lane)

- Fix small memory leak when SIGHUP processing decides that a new GUC variable value cannot be applied without a restart (Tom Lane)

- Make libpq support arbitrary-length lines in `.pgpass` files (Tom Lane)

  This is mostly useful to allow using very long security tokens as passwords.

- In libpq for Windows, call `WSAStartup()` once per process and `WSACleanup()` not at all (Tom Lane, Alexander Lakhin)

  Previously, libpq invoked `WSAStartup()` at connection start and `WSACleanup()` at connection cleanup. However, it appears that calling `WSACleanup()` can interfere with other program operations; notably, we have observed rare failures to emit expected output to stdout. There appear to be no ill

effects from omitting the call, so do that. (This also eliminates a performance issue from repeated DLL loads and unloads when a program performs a series of database connections.)

- Fix ecpg library's per-thread initialization logic for Windows (Tom Lane, Alexander Lakhin)

  Multi-threaded ecpg applications could suffer rare misbehavior due to incorrect locking.

- On Windows, make psql read the output of a backtick command in text mode, not binary mode (Tom Lane)

  This ensures proper handling of newlines.

- Ensure that pg_dump collects per-column information about extension configuration tables (Fabrízio de Royes Mello, Tom Lane)

  Failure to do this led to crashes when specifying `--inserts`, or underspecified (though usually correct) `COPY` commands when using `COPY` to reload the tables' data.

- Make pg_upgrade check for pre-existence of tablespace directories in the target cluster (Bruce Momjian)

- Fix potential memory leak in `contrib/pgcrypto` (Michael Paquier)

- Add check for an unlikely failure case in `contrib/pgcrypto` (Daniel Gustafsson)

- Use `return` not `exit()` in configure's test programs (Peter Eisentraut)

  This avoids failures with pickier compilers.

- Update time zone data files to tzdata release 2020d for DST law changes in Fiji, Morocco, Palestine, the Canadian Yukon, Macquarie Island, and Casey Station (Antarctica); plus historical corrections for France, Hungary, Monaco, and Palestine.

- Sync our copy of the timezone library with IANA tzcode release 2020d (Tom Lane)

  This absorbs upstream's change of zic's default output option from "fat" to "slim". That's just cosmetic for our purposes, as we continue to select the "fat" mode in pre-v13 branches. This change also ensures that `strftime()` does not change `errno` unless it fails.

E.2. Release 9.5.23

Release date: 2020-08-13

This release contains a variety of fixes from 9.5.22. For information about new features in the 9.5 major release, see Section E.25.

The PostgreSQL community will stop releasing updates for the 9.5.X release series in February 2021. Users are encouraged to update to a newer release branch soon.

E.2.1. Migration to Version 9.5.23

A dump/restore is not required for those running 9.5.X.

However, if you are upgrading from a version earlier than 9.5.20, see Section E.5.

E.2.2. Changes

- Make contrib modules' installation scripts more secure (Tom Lane)

  Attacks similar to those described in CVE-2018-1058 could be carried out against an extension installation script, if the attacker can create objects in either the extension's target schema or the schema of some prerequisite extension. Since extensions often require superuser privilege to install, this can open a path to obtaining superuser privilege. To mitigate this risk, be more careful about the `search_path` used to run an installation script; disable `check_function_bodies` within the script; and fix catalog-adjustment queries used in some contrib modules to ensure they are secure. Also provide documentation to help third-party extension authors make their installation scripts secure. This is not a complete solution; extensions that depend on other extensions can still be at risk if installed carelessly. (CVE-2020-14350)

- In logical replication walsender, fix failure to send feedback messages after sending a keepalive message (Álvaro Herrera)

  This is a relatively minor problem when using built-in logical replication, because the built-in walreceiver will send a feedback reply (which clears the incorrect state) fairly frequently anyway. But with some other replication systems, such as pglogical, it causes significant performance issues.

- Ensure the `repeat()` function can be interrupted by query cancel (Joe Conway)

- Undo double-quoting of index names in `EXPLAIN`'s non-text output formats (Tom Lane, Euler Taveira)

- Fix timing of constraint revalidation in `ALTER TABLE` (David Rowley)

  If `ALTER TABLE` needs to fully rewrite the table's contents (for example, due to change of a column's data type) and also needs to scan the table to re-validate foreign keys or `CHECK` constraints, it sometimes did things in the wrong order, leading to odd errors such as "could not read block 0 in file "base/nnnnn/nnnnn": read only 0 of 8192 bytes".

- Cope with `LATERAL` references in restriction clauses attached to an un-flattened sub-`SELECT` in the `FROM` clause (Tom Lane)

  This oversight could result in assertion failures or crashes at query execution.

- Avoid believing that a never-analyzed foreign table has zero tuples (Tom Lane)

  This primarily affected the planner's estimate of the number of groups that would be obtained by `GROUP BY`.

- Improve error handling in the server's `buffile` module (Thomas Munro)

  Fix some cases where I/O errors were indistinguishable from reaching EOF, or were not reported at all. Also add details such as block numbers and byte counts where appropriate.

- Fix conflict-checking anomalies in `SERIALIZABLE` isolation mode (Peter Geoghegan)

  If a concurrently-inserted tuple was updated by a different concurrent transaction, and neither tuple version was visible to the current transaction's snapshot, serialization conflict checking could draw the wrong conclusions about whether the tuple was relevant to the results of the current transaction. This could allow a serializable transaction to commit when it should have failed with a serialization error.

- Avoid repeated marking of dead btree index entries as dead (Masahiko Sawada)

While functionally harmless, this led to useless WAL traffic when checksums are enabled or `wal_log_hints` is on.

- Fix failure of some code paths to acquire the correct lock before modifying `pg_control` (Nathan Bossart, Fujii Masao)

  This oversight could allow `pg_control` to be written out with an inconsistent checksum, possibly causing trouble later, including inability to restart the database if it crashed before the next `pg_control` update.

- Fix errors in `currtid()` and `currtid2()` (Michael Paquier)

  These functions (which are undocumented and used only by ancient versions of the ODBC driver) contained coding errors that could result in crashes, or in confusing error messages such as "could not open file" when applied to a relation having no storage.

- Avoid calling `elog()` or `palloc()` while holding a spinlock (Michael Paquier, Tom Lane)

  Logic associated with replication slots had several violations of this coding rule. While the odds of trouble are quite low, an error in the called function would lead to a stuck spinlock.

- Report out-of-disk-space errors properly in pg_dump and pg_basebackup (Justin Pryzby, Tom Lane, Álvaro Herrera)

  Some code paths could produce silly reports like "could not write file: Success".

- Fix parallel restore of tables having both table-level privileges and per-column privileges (Tom Lane)

  The table-level privilege grants have to be applied first, but a parallel restore did not reliably order them that way; this could lead to "tuple concurrently updated" errors, or to disappearance of some per-column privilege grants. The fix for this is to include dependency links between such entries in the archive file, meaning that a new dump has to be taken with a corrected pg_dump to ensure that the problem will not recur.

- Ensure that pg_upgrade runs with `vacuum_defer_cleanup_age` set to zero in the target cluster (Bruce Momjian)

  If the target cluster's configuration has been modified to set `vacuum_defer_cleanup_age` to a nonzero value, that prevented freezing of the system catalogs from working properly, which caused the upgrade to fail in confusing ways. Ensure that any such setting is overridden for the duration of the upgrade.

- Fix pg_recvlogical to drain pending messages before exiting (Noah Misch)

  Without this, the replication sender might detect a send failure and exit without making the expected final update to the replication slot's LSN position. That led to re-transmitting data after the next connection. It was also possible to miss error messages sent after the last data that pg_recvlogical wants to consume.

- Fix pg_rewind's handling of just-deleted files in the source data directory (Justin Pryzby, Michael Paquier)

  When working with an on-line source database, concurrent file deletions are possible, but pg_rewind would get confused if deletion happened between seeing a file's directory entry and examining it with `stat()`.

- Make pg_test_fsync use binary I/O mode on Windows (Michael Paquier)

Previously it wrote the test file in text mode, which is not an accurate reflection of PostgreSQL's actual usage.

- Fix failure to initialize local state correctly in `contrib/dblink` (Joe Conway)

  With the right combination of circumstances, this could lead to `dblink_close()` issuing an unexpected remote `COMMIT`.

- Fix `contrib/pgcrypto`'s misuse of `deflate()` (Tom Lane)

  The `pgp_sym_encrypt` functions could produce incorrect compressed data due to mishandling of zlib's API requirements. We have no reports of this error manifesting with stock zlib, but it can be seen when using IBM's zlibNX implementation.

- Fix corner case in decompression logic in `contrib/pgcrypto`'s `pgp_sym_decrypt` functions (Kyotaro Horiguchi, Michael Paquier)

  A compressed stream can validly end with an empty packet, but the decompressor failed to handle this and would complain about corrupt data.

- Use POSIX-standard `strsignal()` in place of the BSD-ish `sys_siglist[]` (Tom Lane)

  This avoids build failures with very recent versions of glibc.

- Support building our NLS code with Microsoft Visual Studio 2015 or later (Juan José Santamaría Flecha, Davinder Singh, Amit Kapila)

- Avoid possible failure of our MSVC install script when there is a file named `configure` several levels above the source code tree (Arnold Müller)

  This could confuse some logic that looked for `configure` to identify the top level of the source tree.

E.3. Release 9.5.22

Release date: 2020-05-14

This release contains a variety of fixes from 9.5.21. For information about new features in the 9.5 major release, see Section E.25.

The PostgreSQL community will stop releasing updates for the 9.5.X release series in February 2021. Users are encouraged to update to a newer release branch soon.

E.3.1. Migration to Version 9.5.22

A dump/restore is not required for those running 9.5.X.

However, if you are upgrading from a version earlier than 9.5.20, see Section E.5.

E.3.2. Changes

- Preserve the `indisclustered` setting of indexes rewritten by `ALTER TABLE` (Amit Langote, Justin Pryzby)

  Previously, `ALTER TABLE` lost track of which index had been used for `CLUSTER`.

- Preserve the replica identity properties of indexes rewritten by `ALTER TABLE` (Quan Zongliang, Peter Eisentraut)

- Lock objects sooner during `DROP OWNED BY` (Álvaro Herrera)

  This avoids failures in race-condition cases where another session is deleting some of the same objects.

- Fix error-case processing for `CREATE ROLE ... IN ROLE` (Andrew Gierth)

  Some error cases would be reported as "unexpected node type" or the like, instead of the intended message.

- Fix bugs in `gin_fuzzy_search_limit` processing (Adé Heyward, Tom Lane)

  A small value of `gin_fuzzy_search_limit` could result in unexpected slowness due to unintentionally rescanning the same index page many times. Another code path failed to apply the intended filtering at all, possibly returning too many values.

- Allow input of type `circle` to accept the format "`(x,y),r`" as the documentation says it does (David Zhang)

- Make the `get_bit()` and `set_bit()` functions cope with `bytea` strings longer than 256MB (Movead Li)

  Since the bit number argument is only `int4`, it's impossible to use these functions to access bits beyond the first 256MB of a long `bytea`. We'll widen the argument to `int8` in v13, but in the meantime, allow these functions to work on the initial substring of a long `bytea`.

- Avoid possibly leaking an open-file descriptor for a directory in `pg_ls_dir()`, `pg_timezone_names()`, `pg_tablespace_databases()`, and allied functions (Justin Pryzby)

- Fix polymorphic-function type resolution to correctly infer the actual type of an `anyarray` output when given only an `anyrange` input (Tom Lane)

- Avoid unlikely crash when `REINDEX` is terminated by a session-shutdown signal (Tom Lane)

- Prevent printout of possibly-incorrect hash join table statistics in `EXPLAIN` (Konstantin Knizhnik, Tom Lane, Thomas Munro)

- Fix reporting of elapsed time for heap truncation steps in `VACUUM VERBOSE` (Tatsuhito Kasahara)

- Avoid possibly showing "waiting" twice in a process's PS status (Masahiko Sawada)

- Avoid premature recycling of WAL segments during crash recovery (Jehan-Guillaume de Rorthais)

  WAL segments that become ready to be archived during crash recovery were potentially recycled without being archived.

- Avoid scanning irrelevant timelines during archive recovery (Kyotaro Horiguchi)

  This can eliminate many attempts to fetch non-existent WAL files from archive storage, which is helpful if archive access is slow.

- Remove bogus "subtransaction logged without previous top-level txn record" error check in logical decoding (Arseny Sher, Amit Kapila)

  This condition is legitimately reachable in various scenarios, so remove the check.

- Ensure that a replication slot's `io_in_progress_lock` is released in failure code paths (Pavan Deolasee)

  This could result in a walsender later becoming stuck waiting for the lock.

- Ensure `nextXid` can't go backwards on a standby server (Eka Palamadai)

  This race condition could allow incorrect hot standby feedback messages to be sent back to the primary server, potentially allowing `VACUUM` to run too soon on the primary.

- Add missing SQLSTATE values to a few error reports (Sawada Masahiko)

- Fix PL/pgSQL to reliably refuse to execute an event trigger function as a plain function (Tom Lane)

- Fix memory leak in libpq when using `sslmode=verify-full` (Roman Peshkurov)

  Certificate verification during connection startup could leak some memory. This would become an issue if a client process opened many database connections during its lifetime.

- Fix ecpg to treat an argument of just "–" as meaning "read from stdin" on all platforms (Tom Lane)

- Fix pg_dump to dump comments on RLS policy objects (Tom Lane)

- In pg_dump, postpone restore of event triggers till the end (Fabrízio de Royes Mello, Hamid Akhtar, Tom Lane)

  This minimizes the risk that an event trigger could interfere with the restoration of other objects.

- Fix quoting of `--encoding`, `--lc-ctype` and `--lc-collate` values in createdb utility (Michael Paquier)

- `contrib/lo`'s `lo_manage()` function crashed if called directly rather than as a trigger (Tom Lane)

- In `contrib/ltree`, protect against overflow of `ltree` and `lquery` length fields (Nikita Glukhov)

- Fix cache reference leak in `contrib/sepgsql` (Michael Luo)

- Avoid failures when dealing with Unix-style locale names on Windows (Juan José Santamaría Flecha)

- In MSVC builds, cope with spaces in the path name for Python (Victor Wagner)

- In MSVC builds, fix detection of Visual Studio version to work with more language settings (Andrew Dunstan)

- In MSVC builds, use `-Wno-deprecated` with bison versions newer than 3.0, as non-Windows builds already do (Andrew Dunstan)

- Update time zone data files to tzdata release 2020a for DST law changes in Morocco and the Canadian Yukon, plus historical corrections for Shanghai.

  The America/Godthab zone has been renamed to America/Nuuk to reflect current English usage; however, the old name remains available as a compatibility link.

  Also, update initdb's list of known Windows time zone names to include recent additions, improving the odds that it will correctly translate the system time zone setting on that platform.

E.4. Release 9.5.21

Release date: 2020-02-13

This release contains a variety of fixes from 9.5.20. For information about new features in the 9.5 major release, see Section E.25.

E.4.1. Migration to Version 9.5.21

A dump/restore is not required for those running 9.5.X.

However, if you are upgrading from a version earlier than 9.5.20, see Section E.5.

E.4.2. Changes

- Avoid failure in logical decoding when a large transaction must be spilled into many separate temporary files (Amit Khandekar)
- Fix failure in logical replication publisher after a database crash and restart (Vignesh C)
- Avoid memory leak when there are no free dynamic shared memory slots (Thomas Munro)
- Ignore the `CONCURRENTLY` option when performing an index creation, drop, or rebuild on a temporary table (Michael Paquier, Heikki Linnakangas, Andres Freund)

  This avoids strange failures if the temporary table has an `ON COMMIT` action. There is no benefit in using `CONCURRENTLY` for a temporary table anyway, since other sessions cannot access the table, making the extra processing pointless.
- Fix possible failure when resetting expression indexes on temporary tables that are marked `ON COMMIT DELETE ROWS` (Tom Lane)
- Fix possible crash in BRIN index operations with `box`, `range` and `inet` data types (Heikki Linnakangas)
- Fix handling of deleted pages in GIN indexes (Alexander Korotkov)

  Avoid possible deadlocks, incorrect updates of a deleted page's state, and failure to traverse through a recently-deleted page.
- Fix possible crash with a SubPlan (sub-`SELECT`) within a multi-row `VALUES` list (Tom Lane)
- Fix unlikely crash with pass-by-reference aggregate transition states (Andres Freund, Teodor Sigaev)
- Improve error reporting in `to_date()` and `to_timestamp()` (Tom Lane, Álvaro Herrera)

  Reports about incorrect month or day names in input strings could truncate the input in the middle of a multi-byte character, leading to an improperly encoded error message that could cause follow-on failures. Truncate at the next whitespace instead.
- Fix off-by-one result for `EXTRACT(ISOYEAR FROM` *`timestamp`*`)` for BC dates (Tom Lane)
- Avoid stack overflow in `information_schema` views when a self-referential view exists in the system catalogs (Tom Lane)

A self-referential view can't work; it will always result in infinite recursion. We handled that situation correctly when trying to execute the view, but not when inquiring whether it is automatically updatable.

- Improve performance of hash joins with very large inner relations (Thomas Munro)
- Fix edge-case crashes and misestimations in selectivity calculations for the `<@` and `@>` range operators (Michael Paquier, Andrey Borodin, Tom Lane)
- Improve error reporting for attempts to use automatic updating of views with conditional `INSTEAD` rules (Dean Rasheed)

  This has never been supported, but previously the error was thrown only at execution time, so that it could be masked by planner errors.
- Prevent a composite type from being included in itself indirectly via a range type (Tom Lane, Julien Rouhaud)
- Fix error reporting for index expressions of prohibited types (Amit Langote)
- Fix dumping of views that contain only a `VALUES` list to handle cases where a view output column has been renamed (Tom Lane)
- Allow libpq to parse all GSS-related connection parameters even when the GSSAPI code hasn't been compiled in (Tom Lane)

  This makes the behavior similar to our SSL support, where it was long ago deemed to be a good idea to always accept all the related parameters, even if some are ignored or restricted due to lack of the feature in a particular build.
- Fix incorrect handling of `%b` and `%B` format codes in ecpg's `PGTYPEStimestamp_fmt_asc()` function (Tomas Vondra)

  Due to an off-by-one error, these codes would print the wrong month name, or possibly crash.
- Fix parallel pg_dump/pg_restore to more gracefully handle failure to create worker processes (Tom Lane)
- Prevent possible crash or lockup when attempting to terminate a parallel pg_dump/pg_restore run via a signal (Tom Lane)
- In pg_upgrade, look inside arrays and ranges while searching for non-upgradable data types in tables (Tom Lane)
- Apply more thorough syntax checking to createuser's `--connection-limit` option (Álvaro Herrera)
- In `contrib/dict_int`, reject `maxlen` settings less than one (Tomas Vondra)

  This prevents a possible crash with silly settings for that parameter.
- Disallow NULL category values in `contrib/tablefunc`'s `crosstab()` function (Joe Conway)

  This case never worked usefully, and it would crash on some platforms.
- Mark some timeout and statistics-tracking GUC variables as `PGDLLIMPORT`, to allow extensions to access them on Windows (Pascal Legrand)

  This applies to `lock_timeout`, `statement_timeout`, `track_activities`, `track_counts`, and `track_functions`.
- Fix race condition that led to delayed delivery of interprocess signals on Windows (Amit Kapila)

  This caused visible timing oddities in `NOTIFY`, and perhaps other misbehavior.

- On Windows, retry a few times after an ERROR_ACCESS_DENIED file access failure (Alexander Lakhin, Tom Lane)

  This helps cope with cases where a file open attempt fails because the targeted file is flagged for deletion but not yet actually gone. pg_ctl, for example, frequently failed with such an error when probing to see if the postmaster had shut down yet.

E.5. Release 9.5.20

Release date: 2019-11-14

This release contains a variety of fixes from 9.5.19. For information about new features in the 9.5 major release, see Section E.25.

E.5.1. Migration to Version 9.5.20

A dump/restore is not required for those running 9.5.X.

However, if you use the contrib/intarray extension with a GiST index, and you rely on indexed searches for the <@ operator, see the entry below about that.

Also, if you are upgrading from a version earlier than 9.5.13, see Section E.12.

E.5.2. Changes

- Disallow changing a multiply-inherited column's type if not all parent tables were changed (Tom Lane)

  Previously, this was allowed, whereupon queries on the now-out-of-sync parent would fail.

- Prevent VACUUM from trying to freeze an old multixact ID involving a still-running transaction (Nathan Bossart, Jeremy Schneider)

  This case would lead to VACUUM failing until the old transaction terminates.

- Ensure that offset expressions in WINDOW clauses are processed when a query's expressions are manipulated (Andrew Gierth)

  This oversight could result in assorted failures when the offsets are nontrivial expressions. One example is that a function parameter reference in such an expression would fail if the function was inlined.

- Fix handling of whole-row variables in WITH CHECK OPTION expressions and row-level-security policy expressions (Andres Freund)

  Previously, such usage might result in bogus errors about row type mismatches.

- Avoid postmaster failure if a parallel query requests a background worker when no postmaster child process array slots remain free (Tom Lane)

- Prevent possible double-free if a BEFORE UPDATE trigger returns the old tuple as-is, and it is not the last such trigger (Thomas Munro)

- Provide a relevant error context line when an error occurs while setting GUC parameters during parallel worker startup (Thomas Munro)
- In serializable mode, ensure that row-level predicate locks are acquired on the correct version of the row (Thomas Munro, Heikki Linnakangas)

  If the visible version of the row is HOT-updated, the lock might be taken on its now-dead predecessor, resulting in subtle failures to guarantee serialization.
- Ensure that `fsync()` is applied only to files that are opened read/write (Andres Freund, Michael Paquier)

  Some code paths tried to do this after opening a file read-only, but on some platforms that causes "bad file descriptor" or similar errors.
- Allow encoding conversion to succeed on longer strings than before (Álvaro Herrera, Tom Lane)

  Previously, there was a hard limit of 0.25GB on the input string, but now it will work as long as the converted output is not over 1GB.
- Allow `repalloc()` to give back space when a large chunk is reduced in size (Tom Lane)
- Ensure that temporary WAL and history files are removed at the end of archive recovery (Sawada Masahiko)
- Avoid failure in archive recovery if `recovery_min_apply_delay` is enabled (Fujii Masao)

  `recovery_min_apply_delay` is not typically used in this configuration, but it should work.
- Avoid unwanted delay during shutdown of a logical replication walsender (Craig Ringer, Álvaro Herrera)
- Correctly time-stamp replication messages for logical decoding (Jeff Janes)

  This oversight resulted, for example, in `pg_stat_subscription.last_msg_send_time` usually reading as NULL.
- In logical decoding, ensure that sub-transactions are correctly accounted for when reconstructing a snapshot (Masahiko Sawada)

  This error leads to assertion failures; it's unclear whether any bad effects exist in production builds.
- Fix race condition during backend exit, when the backend process has previously waited for synchronous replication to occur (Dongming Liu)
- Fix `ALTER SYSTEM` to cope with duplicate entries in `postgresql.auto.conf` (Ian Barwick)

  `ALTER SYSTEM` itself will not generate such a state, but external tools that modify `postgresql.auto.conf` could do so. Duplicate entries for the target variable will now be removed, and then the new setting (if any) will be appended at the end.
- Reject include directives with empty file names in configuration files, and report include-file recursion more clearly (Ian Barwick, Tom Lane)
- Avoid logging complaints about abandoned connections when using PAM authentication (Tom Lane)

  libpq-based clients will typically make two connection attempts when a password is required, since they don't prompt their user for a password until their first connection attempt fails. Therefore the server is coded not to generate useless log spam when a client closes the connection upon being asked for a password. However, the PAM authentication code hadn't gotten that memo, and would generate several messages about a phantom authentication failure.

- Fix some cases where an incomplete date specification is not detected in `time with time zone` input (Alexander Lakhin)

  If a time zone with a time-varying UTC offset is specified, then a date must be as well, so that the offset can be resolved. Depending on the syntax used, this check was not enforced in some cases, allowing bogus output to be produced.

- Fix misbehavior of `bitshiftright()` (Tom Lane)

  The bitstring right shift operator failed to zero out padding space that exists in the last byte of the result when the bitstring length is not a multiple of 8. While invisible to most operations, any nonzero bits there would result in unexpected comparison behavior, since bitstring comparisons don't bother to ignore the extra bits, expecting them to always be zero.

  If you have inconsistent data as a result of saving the output of `bitshiftright()` in a table, it's possible to fix it with something like

  ```
  UPDATE mytab SET bitcol = ~(~bitcol) WHERE bitcol != ~(~bitcol);
  ```

- Fix detection of edge-case integer overflow in interval multiplication (Yuya Watari)

- Fix incorrect compression logic for GIN posting lists (Heikki Linnakangas)

  A GIN posting list item can require 7 bytes if the distance between adjacent indexed TIDs exceeds 16TB. One step in the logic was out of sync with that, and might try to write the value into a 6-byte buffer. In principle this could cause a stack overrun, but on most architectures it's likely that the next byte would be unused alignment padding, making the bug harmless. In any case the bug would be very difficult to hit.

- Fix handling of infinity, NaN, and NULL values in KNN-GiST (Alexander Korotkov)

  The query's output order could be wrong (different from a plain sort's result) if some distances computed for non-null column values are infinity or NaN.

- Fix handling of searches for NULL in KNN-SP-GiST (Nikita Glukhov)

- On Windows, recognize additional spellings of the "Norwegian (Bokmål)" locale name (Tom Lane)

- Avoid compile failure if an ECPG client includes `ecpglib.h` while having `ENABLE_NLS` defined (Tom Lane)

  This risk was created by a misplaced declaration: `ecpg_gettext()` should not be visible to client code.

- In psql, resynchronize internal state about the server after an unexpected connection loss and successful reconnection (Peter Billen, Tom Lane)

  Ordinarily this is unnecessary since the state would be the same anyway. But it can matter in corner cases, such as where the connection might lead to one of several servers. This change causes psql to re-issue any interactive messages that it would have issued at startup, for example about whether SSL is in use.

- Avoid platform-specific null pointer dereference in psql (Quentin Rameau)

- Fix pg_dump's handling of circular dependencies in views (Tom Lane)

  In some cases a view may depend on an object that pg_dump needs to dump later than the view; the most common example is that a query using `GROUP BY` on a primary-key column may be semantically invalid without the primary key. This is now handled by emitting a dummy `CREATE VIEW` command that just establishes the view's column names and types, and then later emitting `CREATE OR REPLACE`

`VIEW` with the full view definition. Previously, the dummy definition was actually a `CREATE TABLE` command, and this was automagically converted to a view by a later `CREATE RULE` command. The new approach has been used successfully in PostgreSQL version 10 and later. We are back-patching it into older releases now because of reports that the previous method causes bogus error messages about the view's replica identity status. This change also avoids problems when trying to use the `--clean` option during a restore involving such a view.

- In pg_dump, ensure stable output order for similarly-named triggers and row-level-security policy objects (Benjie Gillam)

  Previously, if two triggers on different tables had the same names, they would be sorted in OID-based order, which is less desirable than sorting them by table name. Likewise for RLS policies.

- Fix pg_dump to work again with pre-8.3 source servers (Tom Lane)

  A previous fix caused pg_dump to always try to query `pg_opfamily`, but that catalog doesn't exist before version 8.3.

- In pg_restore, treat `-f -` as meaning "output to stdout" (Álvaro Herrera)

  This synchronizes pg_restore's behavior with some other applications, and in particular makes pre-v12 branches act similarly to version 12's pg_restore, simplifying creation of dump/restore scripts that work across multiple PostgreSQL versions. Before this change, pg_restore interpreted such a switch as meaning "output to a file named -", but few people would want that.

- Improve pg_upgrade's checks for the use of a data type that has changed representation, such as `line` (Tomas Vondra)

  The previous coding could be fooled by cases where the data type of interest underlies a stored column of a domain or composite type.

- Detect file read errors during pg_basebackup (Jeevan Chalke)

- In pg_rewind with an online source cluster, disable timeouts, much as pg_dump does (Alexander Kukushkin)

- Fix failure in pg_waldump with the `-s` option, when a continuation WAL record ends exactly at a page boundary (Andrey Lepikhov)

- In pg_waldump, include the `newitemoff` field in btree page split records (Peter Geoghegan)

- In pg_waldump with the `--bkp-details` option, avoid emitting extra newlines for WAL records involving full-page writes (Andres Freund)

- Fix small memory leak in pg_waldump (Andres Freund)

- Fix vacuumdb with a high `--jobs` option to handle running out of file descriptors better (Michael Paquier)

- Fix `contrib/intarray`'s GiST opclasses to not fail for empty arrays with `<@` (Tom Lane)

  A clause like *`array_column <@ constant_array`* is considered indexable, but the index search may not find empty array values; of course, such entries should trivially match the search.

  The only practical back-patchable fix for this requires making `<@` index searches scan the whole index, which is what this patch does. This is unfortunate: it means that the query performance is likely worse than a plain sequential scan would be.

Applications whose performance is adversely impacted by this change have a couple of options. They could switch to a GIN index, which doesn't have this bug, or they could replace *array_column* <@ *constant_array* with *array_column* <@ *constant_array* AND *array_column* && *constant_array*. That will provide about the same performance as before, and it will find all non-empty subsets of the given constant array, which is all that could reliably be expected of the query before.

- Allow `configure --with-python` to succeed when only `python3` or only `python2` can be found (Peter Eisentraut, Tom Lane)

  Search for `python`, then `python3`, then `python2`, so that configure can succeed in the increasingly-more-common situation where there is no executable named simply `python`. It's still possible to override this choice by setting the `PYTHON` environment variable.

- Fix configure's test for presence of libperl so that it works on recent Red Hat releases (Tom Lane)

  Previously, it could fail if the user sets `CFLAGS` to `-O0`.

- Ensure correct code generation for spinlocks on PowerPC (Noah Misch)

  The previous spinlock coding allowed the compiler to select register zero for use with an assembly instruction that does not accept that register, causing a build failure. We have seen only one long-ago report that matches this bug, but it could cause problems for people trying to build modified PostgreSQL code or use atypical compiler options.

- On PowerPC, avoid depending on the xlc compiler's `__fetch_and_add()` function (Noah Misch)

  xlc 13 and newer interpret this function in a way incompatible with our usage, resulting in an unusable build of PostgreSQL. Fix by using custom assembly code instead.

- On AIX, don't use the compiler option `-qsrcmsg` (Noah Misch)

  This avoids an internal compiler error with xlc v16.1.0, with little consequence other than changing the format of compiler error messages.

- Fix MSVC build process to cope with spaces in the file path of OpenSSL (Andrew Dunstan)

- Update time zone data files to tzdata release 2019c for DST law changes in Fiji and Norfolk Island, plus historical corrections for Alberta, Austria, Belgium, British Columbia, Cambodia, Hong Kong, Indiana (Perry County), Kaliningrad, Kentucky, Michigan, Norfolk Island, South Korea, and Turkey.

E.6. Release 9.5.19

Release date: 2019-08-08

This release contains a variety of fixes from 9.5.18. For information about new features in the 9.5 major release, see Section E.25.

E.6.1. Migration to Version 9.5.19

A dump/restore is not required for those running 9.5.X.

However, if you are upgrading from a version earlier than 9.5.13, see Section E.12.

E.6.2. Changes

- Require schema qualification to cast to a temporary type when using functional cast syntax (Noah Misch)

  We have long required invocations of temporary functions to explicitly specify the temporary schema, that is `pg_temp.`*`func_name`*`(`*`args`*`)`. Require this as well for casting to temporary types using functional notation, for example `pg_temp.`*`type_name`*`(`*`arg`*`)`. Otherwise it's possible to capture a function call using a temporary object, allowing privilege escalation in much the same ways that we blocked in CVE-2007-2138. (CVE-2019-10208)

- Fix failure of `ALTER TABLE ... ALTER COLUMN TYPE` when altering multiple columns' types in one command (Tom Lane)

  This fixes a regression introduced in the most recent minor releases: indexes using the altered columns were not processed correctly, leading to strange failures during `ALTER TABLE`.

- Avoid using unnecessary sort steps for some queries with `GROUPING SETS` (Andrew Gierth, Richard Guo)

- Fix mishandling of multi-column foreign keys when rebuilding a foreign key constraint (Tom Lane)

  `ALTER TABLE` could make an incorrect decision about whether revalidation of a foreign key is necessary, if not all columns of the key are of the same type. It seems likely that the error would always have been in the conservative direction, that is revalidating unnecessarily.

- Prevent incorrect canonicalization of date ranges with `infinity` endpoints (Laurenz Albe)

  It's incorrect to try to convert an open range to a closed one or vice versa by incrementing or decrementing the endpoint value, if the endpoint is infinite; so leave the range alone in such cases.

- Fix loss of fractional digits when converting very large `money` values to `numeric` (Tom Lane)

- Fix spinlock assembly code for MIPS CPUs so that it works on MIPS r6 (YunQiang Su)

- Make libpq ignore carriage return (`\r`) in connection service files (Tom Lane, Michael Paquier)

  In some corner cases, service files containing Windows-style newlines could be mis-parsed, resulting in connection failures.

- Fix pg_dump to ensure that custom operator classes are dumped in the right order (Tom Lane)

  If a user-defined opclass is the subtype opclass of a user-defined range type, related objects were dumped in the wrong order, producing an unrestorable dump. (The underlying failure to handle opclass dependencies might manifest in other cases too, but this is the only known case.)

- Fix `contrib/passwordcheck` to coexist with other users of `check_password_hook` (Michael Paquier)

- Fix `contrib/sepgsql` tests to work under recent SELinux releases (Mike Palmiotto)

- Reduce stderr output from pg_upgrade's test script (Tom Lane)

- Support building Postgres with Microsoft Visual Studio 2019 (Haribabu Kommi)

- In Visual Studio builds, honor `WindowsSDKVersion` environment variable, if that's set (Peifeng Qiu)

  This fixes build failures in some configurations.

- Support OpenSSL 1.1.0 and newer in Visual Studio builds (Juan José Santamaría Flecha, Michael Paquier)
- Allow make options to be passed down to gmake when non-GNU make is invoked at the top level (Thomas Munro)
- Avoid choosing `localtime` or `posixrules` as `TimeZone` during initdb (Tom Lane)

  In some cases initdb would choose one of these artificial zone names over the "real" zone name. Prefer any other match to the C library's timezone behavior over these two.
- Adjust `pg_timezone_names` view to show the `Factory` time zone if and only if it has a short abbreviation (Tom Lane)

  Historically, IANA set up this artificial zone with an "abbreviation" like `Local time zone must be set--see zic manual page`. Modern versions of the tzdb database show `-00` instead, but some platforms alter the data to show one or another of the historical phrases. Show this zone only if it uses the modern abbreviation.
- Sync our copy of the timezone library with IANA tzcode release 2019b (Tom Lane)

  This adds support for zic's new `-b slim` option to reduce the size of the installed zone files. We are not currently using that, but may enable it in future.
- Update time zone data files to tzdata release 2019b for DST law changes in Brazil, plus historical corrections for Hong Kong, Italy, and Palestine.

E.7. Release 9.5.18

Release date: 2019-06-20

This release contains a variety of fixes from 9.5.17. For information about new features in the 9.5 major release, see Section E.25.

E.7.1. Migration to Version 9.5.18

A dump/restore is not required for those running 9.5.X.

However, if you are upgrading from a version earlier than 9.5.13, see Section E.12.

E.7.2. Changes

- Fix failure of `ALTER TABLE ... ALTER COLUMN TYPE` when the table has a partial exclusion constraint (Tom Lane)
- Fix failure of `COMMENT` command for comments on domain constraints (Daniel Gustafsson, Michael Paquier)
- Fix incorrect printing of queries with duplicate join names (Philip Dubé)

  This oversight caused a dump/restore failure for views containing such queries.

- Fix misoptimization of `{1,1}` quantifiers in regular expressions (Tom Lane)

  Such quantifiers were treated as no-ops and optimized away; but the documentation specifies that they impose greediness, or non-greediness in the case of the non-greedy variant `{1,1}?`, on the subexpression they're attached to, and this did not happen. The misbehavior occurred only if the subexpression contained capturing parentheses or a back-reference.

- Avoid possible failures while initializing a new process's `pg_stat_activity` data (Tom Lane)

  Certain operations that could fail, such as converting strings extracted from an SSL certificate into the database encoding, were being performed inside a critical section. Failure there would result in database-wide lockup due to violating the access protocol for shared `pg_stat_activity` data.

- Fix race condition in check to see whether a pre-existing shared memory segment is still in use by a conflicting postmaster (Tom Lane)

- Avoid attempting to do database accesses for parameter checking in processes that are not connected to a specific database (Vignesh C, Andres Freund)

  This error could result in failures like "cannot read pg_class without having selected a database".

- Avoid possible hang in libpq if using SSL and OpenSSL's pending-data buffer contains an exact multiple of 256 bytes (David Binderman)

- Improve initdb's handling of multiple equivalent names for the system time zone (Tom Lane, Andrew Gierth)

  Make initdb examine the `/etc/localtime` symbolic link, if that exists, to break ties between equivalent names for the system time zone. This makes initdb more likely to select the time zone name that the user would expect when multiple identical time zones exist. It will not change the behavior if `/etc/localtime` is not a symlink to a zone data file, nor if the time zone is determined from the `TZ` environment variable.

  Separately, prefer `UTC` over other spellings of that time zone, when neither `TZ` nor `/etc/localtime` provide a hint. This fixes an annoyance introduced by tzdata 2019a's change to make the `UCT` and `UTC` zone names equivalent: initdb was then preferring `UCT`, which almost nobody wants.

- Fix misleading error reports from reindexdb (Julien Rouhaud)

- Ensure that vacuumdb returns correct status if an error occurs while using parallel jobs (Julien Rouhaud)

- In `contrib/postgres_fdw`, account for possible data modifications by local `BEFORE ROW UPDATE` triggers (Shohei Mochizuki)

  If a trigger modified a column that was otherwise not changed by the `UPDATE`, the new value was not transmitted to the remote server.

- On Windows, avoid failure when the database encoding is set to SQL_ASCII and we attempt to log a non-ASCII string (Noah Misch)

  The code had been assuming that such strings must be in UTF-8, and would throw an error if they didn't appear to be validly encoded. Now, just transmit the untranslated bytes to the log.

- Make PL/pgSQL's header files C++-safe (George Tarasov)

E.8. Release 9.5.17

Release date: 2019-05-09

This release contains a variety of fixes from 9.5.16. For information about new features in the 9.5 major release, see Section E.25.

E.8.1. Migration to Version 9.5.17

A dump/restore is not required for those running 9.5.X.

However, if you are upgrading from a version earlier than 9.5.13, see Section E.12.

E.8.2. Changes

- Prevent row-level security policies from being bypassed via selectivity estimators (Dean Rasheed)

  Some of the planner's selectivity estimators apply user-defined operators to values found in `pg_statistic` (e.g., most-common values). A leaky operator therefore can disclose some of the entries in a data column, even if the calling user lacks permission to read that column. In CVE-2017-7484 we added restrictions to forestall that, but we failed to consider the effects of row-level security. A user who has SQL permission to read a column, but who is forbidden to see certain rows due to RLS policy, might still learn something about those rows' contents via a leaky operator. This patch further tightens the rules, allowing leaky operators to be applied to statistics data only when there is no relevant RLS policy. (CVE-2019-10130)

- Fix behavior for an `UPDATE` or `DELETE` on an inheritance tree or partitioned table in which every table can be excluded (Amit Langote, Tom Lane)

  In such cases, the query did not report the correct set of output columns when a `RETURNING` clause was present, and if there were any statement-level triggers that should be fired, it didn't fire them.

- Fix handling of explicit `DEFAULT` items in an `INSERT ... VALUES` command with multiple `VALUES` rows, if the target relation is an updatable view (Amit Langote, Dean Rasheed)

  When the updatable view has no default for the column but its underlying table has one, a single-row `INSERT ... VALUES` will use the underlying table's default. In the multi-row case, however, NULL was always used. Correct it to act like the single-row case.

- Fix `CREATE VIEW` to allow zero-column views (Ashutosh Sharma)

  We should allow this for consistency with allowing zero-column tables. Since a table can be converted to a view, zero-column views could be created even with the restriction in place, leading to dump/reload failures.

- Add missing support for `CREATE TABLE IF NOT EXISTS ... AS EXECUTE ...` (Andreas Karlsson)

  The combination of `IF NOT EXISTS` and `EXECUTE` should work, but the grammar omitted it.

- Ensure that sub-`SELECT`s appearing in row-level-security policy expressions are executed with the correct user's permissions (Dean Rasheed)

Previously, if the table having the RLS policy was accessed via a view, such checks might be executed as the user calling the view, not as the view owner as they should be.

- Accept XML documents as valid values of type `xml` when `xmloption` is set to `content`, as required by SQL:2006 and later (Chapman Flack)

  Previously PostgreSQL followed the SQL:2003 definition, which doesn't allow this. But that creates a serious problem for dump/restore: there is no setting of `xmloption` that will accept all valid XML data. Hence, switch to the 2006 definition.

  pg_dump is also modified to emit `SET xmloption = content` while restoring data, ensuring that dump/restore works even if the prevailing setting is `document`.

- Improve server's startup-time checks for whether a pre-existing shared memory segment is still in use (Noah Misch)

  The postmaster is now more likely to detect that there are still active processes from a previous postmaster incarnation, even if the `postmaster.pid` file has been removed.

- Fix incompatibility of GIN-index WAL records (Alexander Korotkov)

  A fix applied in February's minor releases was not sufficiently careful about backwards compatibility, leading to problems if a standby server of that vintage reads GIN page-deletion WAL records generated by a primary server of a previous minor release.

- Tolerate `EINVAL` and `ENOSYS` error results, where appropriate, for `fsync` and `sync_file_range` calls (Thomas Munro, James Sewell)

  The previous change to panic on file synchronization failures turns out to have been excessively paranoid for certain cases where a failure is predictable and essentially means "operation not supported".

- Fix "failed to build any *N*-way joins" planner failures with lateral references leading out of `FULL` outer joins (Tom Lane)

- Check the appropriate user's permissions when enforcing rules about letting a leaky operator see `pg_statistic` data (Dean Rasheed)

  When an underlying table is being accessed via a view, consider the privileges of the view owner while deciding whether leaky operators may be applied to the table's statistics data, rather than the privileges of the user making the query. This makes the planner's rules about what data is visible match up with the executor's, avoiding unnecessarily-poor plans.

- Avoid O(N^2) performance issue when rolling back a transaction that created many tables (Tomas Vondra)

- Fix race conditions in management of dynamic shared memory (Thomas Munro)

  These could lead to "dsa_area could not attach to segment" or "cannot unpin a segment that is not pinned" errors.

- Fix race condition in which a hot-standby postmaster could fail to shut down after receiving a smart-shutdown request (Tom Lane)

- Fix possible crash when `pg_identify_object_as_address()` is given invalid input (Álvaro Herrera)

- Tighten validation of encoded SCRAM-SHA-256 and MD5 passwords (Jonathan Katz)

A password string that had the right initial characters could be mistaken for one that is correctly hashed into SCRAM-SHA-256 or MD5 format. The password would be accepted but would be unusable later.

- Fix handling of `lc_time` settings that imply an encoding different from the database's encoding (Juan José Santamaría Flecha, Tom Lane)

  Localized month or day names that include non-ASCII characters previously caused unexpected errors or wrong output in such locales.

- Fix incorrect `operator_precedence_warning` checks involving unary minus operators (Rikard Falkeborn)

- Disallow `NaN` as a value for floating-point server parameters (Tom Lane)

- Rearrange `REINDEX` processing to avoid assertion failures when reindexing individual indexes of `pg_class` (Andres Freund, Tom Lane)

- Fix planner assertion failure for parameterized dummy paths (Tom Lane)

- Insert correct test function in the result of `SnapBuildInitialSnapshot()` (Antonin Houska)

  No core code cares about this, but some extensions do.

- Fix intermittent "could not reattach to shared memory" session startup failures on Windows (Noah Misch)

  A previously unrecognized source of these failures is creation of thread stacks for a process's default thread pool. Arrange for such stacks to be allocated in a different memory region.

- Fix error detection in directory scanning on Windows (Konstantin Knizhnik)

  Errors, such as lack of permissions to read the directory, were not detected or reported correctly; instead the code silently acted as though the directory were empty.

- Fix grammar problems in ecpg (Tom Lane)

  A missing semicolon led to mistranslation of `SET` *`variable`* `= DEFAULT` (but not `SET` *`variable`* `TO DEFAULT`) in ecpg programs, producing syntactically invalid output that the server would reject. Additionally, in a `DROP TYPE` or `DROP DOMAIN` command that listed multiple type names, only the first type name was actually processed.

- Sync ecpg's syntax for `CREATE TABLE AS` with the server's (Daisuke Higuchi)

- Fix possible buffer overruns in ecpg's processing of include filenames (Liu Huailing, Fei Wu)

- Avoid crash in `contrib/vacuumlo` if an `lo_unlink()` call failed (Tom Lane)

- Sync our copy of the timezone library with IANA tzcode release 2019a (Tom Lane)

  This corrects a small bug in zic that caused it to output an incorrect year-2440 transition in the `Africa/Casablanca` zone, and adds support for zic's new `-r` option.

- Update time zone data files to tzdata release 2019a for DST law changes in Palestine and Metlakatla, plus historical corrections for Israel.

  `Etc/UCT` is now a backward-compatibility link to `Etc/UTC`, instead of being a separate zone that generates the abbreviation `UCT`, which nowadays is typically a typo. PostgreSQL will still accept `UCT` as an input zone abbreviation, but it won't output it.

E.9. Release 9.5.16

Release date: 2019-02-14

This release contains a variety of fixes from 9.5.15. For information about new features in the 9.5 major release, see Section E.25.

E.9.1. Migration to Version 9.5.16

A dump/restore is not required for those running 9.5.X.

However, if you are upgrading from a version earlier than 9.5.13, see Section E.12.

E.9.2. Changes

- By default, panic instead of retrying after `fsync()` failure, to avoid possible data corruption (Craig Ringer, Thomas Munro)

  Some popular operating systems discard kernel data buffers when unable to write them out, reporting this as `fsync()` failure. If we reissue the `fsync()` request it will succeed, but in fact the data has been lost, so continuing risks database corruption. By raising a panic condition instead, we can replay from WAL, which may contain the only remaining copy of the data in such a situation. While this is surely ugly and inefficient, there are few alternatives, and fortunately the case happens very rarely.

  A new server parameter data_sync_retry has been added to control this; if you are certain that your kernel does not discard dirty data buffers in such scenarios, you can set `data_sync_retry` to `on` to restore the old behavior.

- Include each major release branch's release notes in the documentation for only that branch, rather than that branch and all later ones (Tom Lane)

  The duplication induced by the previous policy was getting out of hand. Our plan is to provide a full archive of release notes on the project's web site, but not duplicate it within each release.

- Avoid possible deadlock when acquiring multiple buffer locks (Nishant Fnu)

- Avoid deadlock between hot-standby queries and replay of GIN index page deletion (Alexander Korotkov)

- Fix possible crashes in logical replication when index expressions or predicates are in use (Peter Eisentraut)

- Avoid useless and expensive logical decoding of TOAST data during a table rewrite (Tomas Vondra)

- Fix logic for stopping a subset of WAL senders when synchronous replication is enabled (Paul Guo, Michael Paquier)

- Avoid possibly writing an incorrect replica identity field in a tuple deletion WAL record (Stas Kelvich)

- Make the archiver prioritize WAL history files over WAL data files while choosing which file to archive next (David Steele)

- Fix possible crash in `UPDATE` with a multiple `SET` clause using a sub-`SELECT` as source (Tom Lane)

- Avoid crash if libxml2 returns a null error message (Sergio Conde Gómez)
- Fix spurious grouping-related parser errors caused by inconsistent handling of collation assignment (Andrew Gierth)

  In some cases, expressions that should be considered to match were not seen as matching, if they included operations on collatable data types.
- Check whether the comparison function underlying `LEAST()` or `GREATEST()` is leakproof, rather than just assuming it is (Tom Lane)

  Actual information leaks from btree comparison functions are typically hard to provoke, but in principle they could happen.
- Fix incorrect planning of queries in which a lateral reference must be evaluated at a foreign table scan (Tom Lane)
- Fix corner-case underestimation of the cost of a merge join (Tom Lane)

  The planner could prefer a merge join when the outer key range is much smaller than the inner key range, even if there are so many duplicate keys on the inner side that this is a poor choice.
- Avoid O(N^2) planning time growth when a query contains many thousand indexable clauses (Tom Lane)
- Improve `ANALYZE`'s handling of concurrently-updated rows (Jeff Janes, Tom Lane)

  Previously, rows deleted by an in-progress transaction were omitted from `ANALYZE`'s sample, but this has been found to lead to more inconsistency than including them would do. In effect, the sample now corresponds to an MVCC snapshot as of `ANALYZE`'s start time.
- Make `TRUNCATE` ignore inheritance child tables that are temporary tables of other sessions (Amit Langote, Michael Paquier)

  This brings `TRUNCATE` into line with the behavior of other commands. Previously, such cases usually ended in failure.
- Fix `TRUNCATE` to update the statistics counters for the right table (Tom Lane)

  If the truncated table had a TOAST table, that table's counters were reset instead.
- Allow `UNLISTEN` in hot-standby mode (Shay Rojansky)

  This is necessarily a no-op, because `LISTEN` isn't allowed in hot-standby mode; but allowing the dummy operation simplifies session-state-reset logic in clients.
- Fix missing role dependencies in some schema and data type permissions lists (Tom Lane)

  In some cases it was possible to drop a role to which permissions had been granted. This caused no immediate problem, but a subsequent dump/reload or upgrade would fail, with symptoms involving attempts to grant privileges to all-numeric role names.
- Ensure relation caches are updated properly after renaming constraints (Amit Langote)
- Make autovacuum more aggressive about removing leftover temporary tables, and also remove leftover temporary tables during `DISCARD TEMP` (Álvaro Herrera)

  This helps ensure that remnants from a crashed session are cleaned up more promptly.
- Prevent empty GIN index pages from being reclaimed too quickly, causing failures of concurrent searches (Andrey Borodin, Alexander Korotkov)

- Fix edge-case failures in float-to-integer coercions (Andrew Gierth, Tom Lane)

  Values very slightly above the maximum valid integer value might not be rejected, and then would overflow, producing the minimum valid integer instead. Also, values that should round to the minimum or maximum integer value might be incorrectly rejected.

- Disallow setting `client_min_messages` higher than `ERROR` (Jonah Harris, Tom Lane)

  Previously, it was possible to set this variable to `FATAL` or `PANIC`, which had the effect of suppressing transmission of ordinary error messages to the client. However, that's contrary to guarantees that are given in the PostgreSQL wire protocol specification, and it caused some clients to become very confused. In released branches, fix this by silently treating such settings as meaning `ERROR` instead. Version 12 and later will reject those alternatives altogether.

- Fix ecpglib to use `uselocale()` or `_configthreadlocale()` in preference to `setlocale()` (Michael Meskes, Tom Lane)

  Since `setlocale()` is not thread-local, and might not even be thread-safe, the previous coding caused problems in multi-threaded ecpg applications.

- Fix incorrect results for numeric data passed through an ecpg SQLDA (SQL Descriptor Area) (Daisuke Higuchi)

  Values with leading zeroes were not copied correctly.

- Fix psql's `\g` *`target`* meta-command to work with `COPY TO STDOUT` (Daniel Vérité)

  Previously, the *`target`* option was ignored, so that the copy data always went to the current query output target.

- Make psql's LaTeX output formats render special characters properly (Tom Lane)

  Backslash and some other ASCII punctuation characters were not rendered correctly, leading to document syntax errors or wrong characters in the output.

- Fix pg_dump's handling of materialized views with indirect dependencies on primary keys (Tom Lane)

  This led to mis-labeling of such views' dump archive entries, causing harmless warnings about "archive items not in correct section order"; less harmlessly, selective-restore options depending on those labels, such as `--section`, might misbehave.

- Avoid null-pointer-dereference crash on some platforms when pg_dump or pg_restore tries to report an error (Tom Lane)

- Fix `contrib/hstore` to calculate correct hash values for empty `hstore` values that were created in version 8.4 or before (Andrew Gierth)

  The previous coding did not give the same result as for an empty `hstore` value created by a newer version, thus potentially causing wrong results in hash joins or hash aggregation. It is advisable to reindex any hash indexes built on `hstore` columns, if the table might contain data that was originally stored as far back as 8.4 and was never dumped/reloaded since then.

- Avoid crashes and excessive runtime with large inputs to `contrib/intarray`'s `gist__int_ops` index support (Andrew Gierth)

- Adjust configure's selection of threading-related compiler flags and libraries to match what later PostgreSQL releases do (Tom Lane)

The coding previously used in the 9.4 and 9.5 branches fails outright on some newer platforms, so sync it with what 9.6 and later have been doing.

- Support new Makefile variables `PG_CFLAGS`, `PG_CXXFLAGS`, and `PG_LDFLAGS` in pgxs builds (Christoph Berg)

  This simplifies customization of extension build processes.

- Fix Perl-coded build scripts to not assume "." is in the search path, since recent Perl versions don't include that (Andrew Dunstan)

- Fix server command-line option parsing problems on OpenBSD (Tom Lane)

- Update time zone data files to tzdata release 2018i for DST law changes in Kazakhstan, Metlakatla, and Sao Tome and Principe. Kazakhstan's Qyzylorda zone is split in two, creating a new zone Asia/Qostanay, as some areas did not change UTC offset. Historical corrections for Hong Kong and numerous Pacific islands.

E.10. Release 9.5.15

Release date: 2018-11-08

This release contains a variety of fixes from 9.5.14. For information about new features in the 9.5 major release, see Section E.25.

E.10.1. Migration to Version 9.5.15

A dump/restore is not required for those running 9.5.X.

However, if you are upgrading from a version earlier than 9.5.13, see Section E.12.

E.10.2. Changes

- Fix corner-case failures in `has_`*`foo`*`_privilege()` family of functions (Tom Lane)

  Return NULL rather than throwing an error when an invalid object OID is provided. Some of these functions got that right already, but not all. `has_column_privilege()` was additionally capable of crashing on some platforms.

- Avoid O(N^2) slowdown in regular expression match/split functions on long strings (Andrew Gierth)

- Fix parsing of standard multi-character operators that are immediately followed by a comment or + or − (Andrew Gierth)

  This oversight could lead to parse errors, or to incorrect assignment of precedence.

- Avoid O(N^3) slowdown in lexer for long strings of + or − characters (Andrew Gierth)

- Fix mis-execution of SubPlans when the outer query is being scanned backwards (Andrew Gierth)

- Fix failure of `UPDATE/DELETE ... WHERE CURRENT OF ...` after rewinding the referenced cursor (Tom Lane)

  A cursor that scans multiple relations (particularly an inheritance tree) could produce wrong behavior if rewound to an earlier relation.

- Fix `EvalPlanQual` to handle conditionally-executed InitPlans properly (Andrew Gierth, Tom Lane)

  This resulted in hard-to-reproduce crashes or wrong answers in concurrent updates, if they contained code such as an uncorrelated sub-`SELECT` inside a `CASE` construct.

- Fix character-class checks to not fail on Windows for Unicode characters above U+FFFF (Tom Lane, Kenji Uno)

  This bug affected full-text-search operations, as well as `contrib/ltree` and `contrib/pg_trgm`.

- Ensure that sequences owned by a foreign table are processed by `ALTER OWNER` on the table (Peter Eisentraut)

  The ownership change should propagate to such sequences as well, but this was missed for foreign tables.

- Ensure that the server will process already-received `NOTIFY` and `SIGTERM` interrupts before waiting for client input (Jeff Janes, Tom Lane)

- Fix over-allocation of space for `array_out()`'s result string (Keiichi Hirobe)

- Fix memory leak in repeated SP-GiST index scans (Tom Lane)

  This is only known to amount to anything significant in cases where an exclusion constraint using SP-GiST receives many new index entries in a single command.

- Ensure that `ApplyLogicalMappingFile()` closes the mapping file when done with it (Tomas Vondra)

  Previously, the file descriptor was leaked, eventually resulting in failures during logical decoding.

- Fix logical decoding to handle cases where a mapped catalog table is repeatedly rewritten, e.g., by `VACUUM FULL` (Andres Freund)

- Prevent starting the server with `wal_level` set to too low a value to support an existing replication slot (Andres Freund)

- Avoid crash if a utility command causes infinite recursion (Tom Lane)

- When initializing a hot standby, cope with duplicate XIDs caused by two-phase transactions on the master (Michael Paquier, Konstantin Knizhnik)

- Fix event triggers to handle nested `ALTER TABLE` commands (Michael Paquier, Álvaro Herrera)

- Propagate parent process's transaction and statement start timestamps to parallel workers (Konstantin Knizhnik)

  This prevents misbehavior of functions such as `transaction_timestamp()` when executed in a worker.

- Fix WAL file recycling logic to work correctly on standby servers (Michael Paquier)

  Depending on the setting of `archive_mode`, a standby might fail to remove some WAL files that could be removed.

- Fix handling of commit-timestamp tracking during recovery (Masahiko Sawada, Michael Paquier)

If commit timestamp tracking has been turned on or off, recovery might fail due to trying to fetch the commit timestamp for a transaction that did not record it.

- Randomize the `random()` seed in bootstrap and standalone backends, and in initdb (Noah Misch)

  The main practical effect of this change is that it avoids a scenario where initdb might mistakenly conclude that POSIX shared memory is not available, due to name collisions caused by always using the same random seed.

- Allow DSM allocation to be interrupted (Chris Travers)
- Properly handle turning `full_page_writes` on dynamically (Kyotaro Horiguchi)
- Avoid possible buffer overrun when replaying GIN page recompression from WAL (Alexander Korotkov, Sivasubramanian Ramasubramanian)
- Fix missed fsync of a replication slot's directory (Konstantin Knizhnik, Michael Paquier)
- Fix unexpected timeouts when using `wal_sender_timeout` on a slow server (Noah Misch)
- Ensure that hot standby processes use the correct WAL consistency point (Alexander Kukushkin, Michael Paquier)

  This prevents possible misbehavior just after a standby server has reached a consistent database state during WAL replay.

- Ensure background workers are stopped properly when the postmaster receives a fast-shutdown request before completing database startup (Alexander Kukushkin)
- Don't run atexit callbacks when servicing `SIGQUIT` (Heikki Linnakangas)
- Don't record foreign-server user mappings as members of extensions (Tom Lane)

  If `CREATE USER MAPPING` is executed in an extension script, an extension dependency was created for the user mapping, which is unexpected. Roles can't be extension members, so user mappings shouldn't be either.

- Make syslogger more robust against failures in opening CSV log files (Tom Lane)
- Fix psql, as well as documentation examples, to call `PQconsumeInput()` before each `PQnotifies()` call (Tom Lane)

  This fixes cases in which psql would not report receipt of a `NOTIFY` message until after the next command.

- Fix possible inconsistency in pg_dump's sorting of dissimilar object names (Jacob Champion)
- Ensure that pg_restore will schema-qualify the table name when emitting `DISABLE`/`ENABLE TRIGGER` commands (Tom Lane)

  This avoids failures due to the new policy of running restores with restrictive search path.

- Fix pg_upgrade to handle event triggers in extensions correctly (Haribabu Kommi)

  pg_upgrade failed to preserve an event trigger's extension-membership status.

- Fix pg_upgrade's cluster state check to work correctly on a standby server (Bruce Momjian)
- Enforce type `cube`'s dimension limit in all `contrib/cube` functions (Andrey Borodin)

  Previously, some cube-related functions could construct values that would be rejected by `cube_in()`, leading to dump/reload failures.

- Fix `contrib/unaccent`'s `unaccent()` function to use the `unaccent` text search dictionary that is in the same schema as the function (Tom Lane)

  Previously it tried to look up the dictionary using the search path, which could fail if the search path has a restrictive value.

- Fix build problems on macOS 10.14 (Mojave) (Tom Lane)

  Adjust configure to add an `-isysroot` switch to `CPPFLAGS`; without this, PL/Perl and PL/Tcl fail to configure or build on macOS 10.14. The specific sysroot used can be overridden at configure time or build time by setting the `PG_SYSROOT` variable in the arguments of configure or make.

  It is now recommended that Perl-related extensions write `$(perl_includespec)` rather than `-I$(perl_archlibexp)/CORE` in their compiler flags. The latter continues to work on most platforms, but not recent macOS.

  Also, it should no longer be necessary to specify `--with-tclconfig` manually to get PL/Tcl to build on recent macOS releases.

- Fix MSVC build and regression-test scripts to work on recent Perl versions (Andrew Dunstan)

  Perl no longer includes the current directory in its search path by default; work around that.

- On Windows, allow the regression tests to be run by an Administrator account (Andrew Dunstan)

  To do this safely, pg_regress now gives up any such privileges at startup.

- Support building on Windows with Visual Studio 2015 or Visual Studio 2017 (Michael Paquier, Haribabu Kommi)

- Allow btree comparison functions to return `INT_MIN` (Tom Lane)

  Up to now, we've forbidden datatype-specific comparison functions from returning `INT_MIN`, which allows callers to invert the sort order just by negating the comparison result. However, this was never safe for comparison functions that directly return the result of `memcmp()`, `strcmp()`, etc, as POSIX doesn't place any such restriction on those functions. At least some recent versions of `memcmp()` can return `INT_MIN`, causing incorrect sort ordering. Hence, we've removed this restriction. Callers must now use the `INVERT_COMPARE_RESULT()` macro if they wish to invert the sort order.

- Fix recursion hazard in shared-invalidation message processing (Tom Lane)

  This error could, for example, result in failure to access a system catalog or index that had just been processed by `VACUUM FULL`.

  This change adds a new result code for `LockAcquire`, which might possibly affect external callers of that function, though only very unusual usage patterns would have an issue with it. The API of `LockAcquireExtended` is also changed.

- Save and restore SPI's global variables during `SPI_connect()` and `SPI_finish()` (Chapman Flack, Tom Lane)

  This prevents possible interference when one SPI-using function calls another.

- Provide `ALLOCSET_DEFAULT_SIZES` and sibling macros in back branches (Tom Lane)

  These macros have existed since 9.6, but there were requests to add them to older branches to allow extensions to rely on them without branch-specific coding.

- Avoid using potentially-under-aligned page buffers (Tom Lane)

Invent new union types `PGAlignedBlock` and `PGAlignedXLogBlock`, and use these in place of plain char arrays, ensuring that the compiler can't place the buffer at a misaligned start address. This fixes potential core dumps on alignment-picky platforms, and may improve performance even on platforms that allow misalignment.

- Make `src/port/snprintf.c` follow the C99 standard's definition of `snprintf()`'s result value (Tom Lane)

  On platforms where this code is used (mostly Windows), its pre-C99 behavior could lead to failure to detect buffer overrun, if the calling code assumed C99 semantics.

- When building on i386 with the clang compiler, require `-msse2` to be used (Andres Freund)

  This avoids problems with missed floating point overflow checks.

- Fix configure's detection of the result type of `strerror_r()` (Tom Lane)

  The previous coding got the wrong answer when building with icc on Linux (and perhaps in other cases), leading to libpq not returning useful error messages for system-reported errors.

- Update time zone data files to tzdata release 2018g for DST law changes in Chile, Fiji, Morocco, and Russia (Volgograd), plus historical corrections for China, Hawaii, Japan, Macau, and North Korea.

E.11. Release 9.5.14

Release date: 2018-08-09

This release contains a variety of fixes from 9.5.13. For information about new features in the 9.5 major release, see Section E.25.

E.11.1. Migration to Version 9.5.14

A dump/restore is not required for those running 9.5.X.

However, if you are upgrading from a version earlier than 9.5.13, see Section E.12.

E.11.2. Changes

- Fix failure to reset libpq's state fully between connection attempts (Tom Lane)

  An unprivileged user of `dblink` or `postgres_fdw` could bypass the checks intended to prevent use of server-side credentials, such as a `~/.pgpass` file owned by the operating-system user running the server. Servers allowing peer authentication on local connections are particularly vulnerable. Other attacks such as SQL injection into a `postgres_fdw` session are also possible. Attacking `postgres_fdw` in this way requires the ability to create a foreign server object with selected connection parameters, but any user with access to `dblink` could exploit the problem. In general, an attacker with the ability to select the connection parameters for a libpq-using application could cause mischief, though other plausible attack scenarios are harder to think of. Our thanks to Andrew Krasichkov for reporting this issue. (CVE-2018-10915)

- Fix `INSERT ... ON CONFLICT UPDATE` through a view that isn't just `SELECT * FROM ...` (Dean Rasheed, Amit Langote)

  Erroneous expansion of an updatable view could lead to crashes or "attribute ... has the wrong type" errors, if the view's `SELECT` list doesn't match one-to-one with the underlying table's columns. Furthermore, this bug could be leveraged to allow updates of columns that an attacking user lacks `UPDATE` privilege for, if that user has `INSERT` and `UPDATE` privileges for some other column(s) of the table. Any user could also use it for disclosure of server memory. (CVE-2018-10925)

- Ensure that updates to the `relfrozenxid` and `relminmxid` values for "nailed" system catalogs are processed in a timely fashion (Andres Freund)

  Overoptimistic caching rules could prevent these updates from being seen by other sessions, leading to spurious errors and/or data corruption. The problem was significantly worse for shared catalogs, such as `pg_authid`, because the stale cache data could persist into new sessions as well as existing ones.

- Fix case where a freshly-promoted standby crashes before having completed its first post-recovery checkpoint (Michael Paquier, Kyotaro Horiguchi, Pavan Deolasee, Álvaro Herrera)

  This led to a situation where the server did not think it had reached a consistent database state during subsequent WAL replay, preventing restart.

- Avoid emitting a bogus WAL record when recycling an all-zero btree page (Amit Kapila)

  This mistake has been seen to cause assertion failures, and potentially it could result in unnecessary query cancellations on hot standby servers.

- During WAL replay, guard against corrupted record lengths exceeding 1GB (Michael Paquier)

  Treat such a case as corrupt data. Previously, the code would try to allocate space and get a hard error, making recovery impossible.

- When ending recovery, delay writing the timeline history file as long as possible (Heikki Linnakangas)

  This avoids some situations where a failure during recovery cleanup (such as a problem with a two-phase state file) led to inconsistent timeline state on-disk.

- Improve performance of WAL replay for transactions that drop many relations (Fujii Masao)

  This change reduces the number of times that shared buffers are scanned, so that it is of most benefit when that setting is large.

- Improve performance of lock releasing in standby server WAL replay (Thomas Munro)

- Make logical WAL senders report streaming state correctly (Simon Riggs, Sawada Masahiko)

  The code previously mis-detected whether or not it had caught up with the upstream server.

- Fix bugs in snapshot handling during logical decoding, allowing wrong decoding results in rare cases (Arseny Sher, Álvaro Herrera)

- Ensure a table's cached index list is correctly rebuilt after an index creation fails partway through (Peter Geoghegan)

  Previously, the failed index's OID could remain in the list, causing problems later in the same session.

- Fix mishandling of empty uncompressed posting list pages in GIN indexes (Sivasubramanian Ramasubramanian, Alexander Korotkov)

  This could result in an assertion failure after pg_upgrade of a pre-9.4 GIN index (9.4 and later will not create such pages).

- Ensure that `VACUUM` will respond to signals within btree page deletion loops (Andres Freund)

  Corrupted btree indexes could result in an infinite loop here, and that previously wasn't interruptible without forcing a crash.

- Fix misoptimization of equivalence classes involving composite-type columns (Tom Lane)

  This resulted in failure to recognize that an index on a composite column could provide the sort order needed for a mergejoin on that column.

- Fix SQL-standard `FETCH FIRST` syntax to allow parameters (`$n`), as the standard expects (Andrew Gierth)

- Fix failure to schema-qualify some object names in `getObjectDescription` output (Kyotaro Horiguchi, Tom Lane)

  Names of collations, conversions, and text search objects were not schema-qualified when they should be.

- Widen `COPY FROM`'s current-line-number counter from 32 to 64 bits (David Rowley)

  This avoids two problems with input exceeding 4G lines: `COPY FROM WITH HEADER` would drop a line every 4G lines, not only the first line, and error reports could show a wrong line number.

- Add a string freeing function to ecpg's `pgtypes` library, so that cross-module memory management problems can be avoided on Windows (Takayuki Tsunakawa)

  On Windows, crashes can ensue if the `free` call for a given chunk of memory is not made from the same DLL that `malloc`'ed the memory. The `pgtypes` library sometimes returns strings that it expects the caller to free, making it impossible to follow this rule. Add a `PGTYPESchar_free()` function that just wraps `free`, allowing applications to follow this rule.

- Fix ecpg's support for `long long` variables on Windows, as well as other platforms that declare `strtoll`/`strtoull` nonstandardly or not at all (Dang Minh Huong, Tom Lane)

- Fix misidentification of SQL statement type in PL/pgSQL, when a rule change causes a change in the semantics of a statement intra-session (Tom Lane)

  This error led to assertion failures, or in rare cases, failure to enforce the `INTO STRICT` option as expected.

- Fix password prompting in client programs so that echo is properly disabled on Windows when `stdin` is not the terminal (Matthew Stickney)

- Further fix mis-quoting of values for list-valued GUC variables in dumps (Tom Lane)

  The previous fix for quoting of `search_path` and other list-valued variables in pg_dump output turned out to misbehave for empty-string list elements, and it risked truncation of long file paths.

- Fix pg_dump's failure to dump `REPLICA IDENTITY` properties for constraint indexes (Tom Lane)

  Manually created unique indexes were properly marked, but not those created by declaring `UNIQUE` or `PRIMARY KEY` constraints.

- Make pg_upgrade check that the old server was shut down cleanly (Bruce Momjian)

  The previous check could be fooled by an immediate-mode shutdown.

- Fix `contrib/hstore_plperl` to look through Perl scalar references, and to not crash if it doesn't find a hash reference where it expects one (Tom Lane)

- Fix crash in `contrib/ltree`'s `lca()` function when the input array is empty (Pierre Ducroquet)
- Fix various error-handling code paths in which an incorrect error code might be reported (Michael Paquier, Tom Lane, Magnus Hagander)
- Rearrange makefiles to ensure that programs link to freshly-built libraries (such as `libpq.so`) rather than ones that might exist in the system library directories (Tom Lane)

  This avoids problems when building on platforms that supply old copies of PostgreSQL libraries.
- Update time zone data files to tzdata release 2018e for DST law changes in North Korea, plus historical corrections for Czechoslovakia.

  This update includes a redefinition of "daylight savings" in Ireland, as well as for some past years in Namibia and Czechoslovakia. In those jurisdictions, legally standard time is observed in summer, and daylight savings time in winter, so that the daylight savings offset is one hour behind standard time not one hour ahead. This does not affect either the actual UTC offset or the timezone abbreviations in use; the only known effect is that the `is_dst` column in the `pg_timezone_names` view will now be true in winter and false in summer in these cases.

E.12. Release 9.5.13

Release date: 2018-05-10

This release contains a variety of fixes from 9.5.12. For information about new features in the 9.5 major release, see Section E.25.

E.12.1. Migration to Version 9.5.13

A dump/restore is not required for those running 9.5.X.

However, if the function marking mistakes mentioned in the first changelog entry below affect you, you will want to take steps to correct your database catalogs.

Also, if you are upgrading from a version earlier than 9.5.12, see Section E.13.

E.12.2. Changes

- Fix incorrect volatility markings on a few built-in functions (Thomas Munro, Tom Lane)

  The functions `query_to_xml`, `cursor_to_xml`, `cursor_to_xmlschema`, `query_to_xmlschema`, and `query_to_xml_and_xmlschema` should be marked volatile because they execute user-supplied queries that might contain volatile operations. They were not, leading to a risk of incorrect query optimization. This has been repaired for new installations by correcting the initial catalog data, but existing installations will continue to contain the incorrect markings. Practical use of these functions seems to pose little hazard, but in case of trouble, it can be fixed by manually updating these functions' `pg_proc` entries, for example `ALTER FUNCTION pg_catalog.query_to_xml(text, boolean,`

`boolean, text) VOLATILE`. (Note that that will need to be done in each database of the installation.) Another option is to pg_upgrade the database to a version containing the corrected initial data.

- Avoid re-using TOAST value OIDs that match dead-but-not-yet-vacuumed TOAST entries (Pavan Deolasee)

  Once the OID counter has wrapped around, it's possible to assign a TOAST value whose OID matches a previously deleted entry in the same TOAST table. If that entry were not yet vacuumed away, this resulted in "unexpected chunk number 0 (expected 1) for toast value *nnnnn*" errors, which would persist until the dead entry was removed by `VACUUM`. Fix by not selecting such OIDs when creating a new TOAST entry.

- Change `ANALYZE`'s algorithm for updating `pg_class.reltuples` (David Gould)

  Previously, pages not actually scanned by `ANALYZE` were assumed to retain their old tuple density. In a large table where `ANALYZE` samples only a small fraction of the pages, this meant that the overall tuple density estimate could not change very much, so that `reltuples` would change nearly proportionally to changes in the table's physical size (`relpages`) regardless of what was actually happening in the table. This has been observed to result in `reltuples` becoming so much larger than reality as to effectively shut off autovacuuming. To fix, assume that `ANALYZE`'s sample is a statistically unbiased sample of the table (as it should be), and just extrapolate the density observed within those pages to the whole table.

- Avoid deadlocks in concurrent `CREATE INDEX CONCURRENTLY` commands that are run under `SERIALIZABLE` or `REPEATABLE READ` transaction isolation (Tom Lane)

- Fix possible slow execution of `REFRESH MATERIALIZED VIEW CONCURRENTLY` (Thomas Munro)

- Fix `UPDATE/DELETE ... WHERE CURRENT OF` to not fail when the referenced cursor uses an index-only-scan plan (Yugo Nagata, Tom Lane)

- Fix incorrect planning of join clauses pushed into parameterized paths (Andrew Gierth, Tom Lane)

  This error could result in misclassifying a condition as a "join filter" for an outer join when it should be a plain "filter" condition, leading to incorrect join output.

- Fix possibly incorrect generation of an index-only-scan plan when the same table column appears in multiple index columns, and only some of those index columns use operator classes that can return the column value (Kyotaro Horiguchi)

- Fix misoptimization of `CHECK` constraints having provably-NULL subclauses of top-level `AND`/`OR` conditions (Tom Lane, Dean Rasheed)

  This could, for example, allow constraint exclusion to exclude a child table that should not be excluded from a query.

- Fix executor crash due to double free in some `GROUPING SET` usages (Peter Geoghegan)

- Avoid crash if a table rewrite event trigger is added concurrently with a command that could call such a trigger (Álvaro Herrera, Andrew Gierth, Tom Lane)

- Avoid failure if a query-cancel or session-termination interrupt occurs while committing a prepared transaction (Stas Kelvich)

- Fix query-lifespan memory leakage in repeatedly executed hash joins (Tom Lane)

- Fix overly strict sanity check in `heap_prepare_freeze_tuple` (Álvaro Herrera)

This could result in incorrect "cannot freeze committed xmax" failures in databases that have been pg_upgrade'd from 9.2 or earlier.

- Prevent dangling-pointer dereference when a C-coded before-update row trigger returns the "old" tuple (Rushabh Lathia)
- Reduce locking during autovacuum worker scheduling (Jeff Janes)

  The previous behavior caused drastic loss of potential worker concurrency in databases with many tables.
- Ensure client hostname is copied while copying `pg_stat_activity` data to local memory (Edmund Horner)

  Previously the supposedly-local snapshot contained a pointer into shared memory, allowing the client hostname column to change unexpectedly if any existing session disconnected.
- Fix incorrect processing of multiple compound affixes in `ispell` dictionaries (Arthur Zakirov)
- Fix collation-aware searches (that is, indexscans using inequality operators) in SP-GiST indexes on text columns (Tom Lane)

  Such searches would return the wrong set of rows in most non-C locales.
- Count the number of index tuples correctly during initial build of an SP-GiST index (Tomas Vondra)

  Previously, the tuple count was reported to be the same as that of the underlying table, which is wrong if the index is partial.
- Count the number of index tuples correctly during vacuuming of a GiST index (Andrey Borodin)

  Previously it reported the estimated number of heap tuples, which might be inaccurate, and is certainly wrong if the index is partial.
- Fix a corner case where a streaming standby gets stuck at a WAL continuation record (Kyotaro Horiguchi)
- In logical decoding, avoid possible double processing of WAL data when a walsender restarts (Craig Ringer)
- Allow `scalarltsel` and `scalargtsel` to be used on non-core datatypes (Tomas Vondra)
- Reduce libpq's memory consumption when a server error is reported after a large amount of query output has been collected (Tom Lane)

  Discard the previous output before, not after, processing the error message. On some platforms, notably Linux, this can make a difference in the application's subsequent memory footprint.
- Fix double-free crashes in ecpg (Patrick Krecker, Jeevan Ladhe)
- Fix ecpg to handle `long long int` variables correctly in MSVC builds (Michael Meskes, Andrew Gierth)
- Fix mis-quoting of values for list-valued GUC variables in dumps (Michael Paquier, Tom Lane)

  The `local_preload_libraries`, `session_preload_libraries`, `shared_preload_libraries`, and `temp_tablespaces` variables were not correctly quoted in pg_dump output. This would cause problems if settings for these variables appeared in `CREATE FUNCTION ... SET` or `ALTER DATABASE/ROLE ... SET` clauses.
- Fix pg_recvlogical to not fail against pre-v10 PostgreSQL servers (Michael Paquier)

A previous fix caused pg_recvlogical to issue a command regardless of server version, but it should only be issued to v10 and later servers.

- Ensure that pg_rewind deletes files on the target server if they are deleted from the source server during the run (Takayuki Tsunakawa)

  Failure to do this could result in data inconsistency on the target, particularly if the file in question is a WAL segment.

- Fix pg_rewind to handle tables in non-default tablespaces correctly (Takayuki Tsunakawa)

- Fix overflow handling in PL/pgSQL integer `FOR` loops (Tom Lane)

  The previous coding failed to detect overflow of the loop variable on some non-gcc compilers, leading to an infinite loop.

- Adjust PL/Python regression tests to pass under Python 3.7 (Peter Eisentraut)

- Support testing PL/Python and related modules when building with Python 3 and MSVC (Andrew Dunstan)

- Support building with Microsoft Visual Studio 2015 (Michael Paquier)

  Various fixes needed for VS2015 compatibility were previously back-patched into the 9.5 branch, but this one was missed.

- Rename internal `b64_encode` and `b64_decode` functions to avoid conflict with Solaris 11.4 built-in functions (Rainer Orth)

- Sync our copy of the timezone library with IANA tzcode release 2018e (Tom Lane)

  This fixes the zic timezone data compiler to cope with negative daylight-savings offsets. While the PostgreSQL project will not immediately ship such timezone data, zic might be used with timezone data obtained directly from IANA, so it seems prudent to update zic now.

- Update time zone data files to tzdata release 2018d for DST law changes in Palestine and Antarctica (Casey Station), plus historical corrections for Portugal and its colonies, as well as Enderbury, Jamaica, Turks & Caicos Islands, and Uruguay.

E.13. Release 9.5.12

Release date: 2018-03-01

This release contains a variety of fixes from 9.5.11. For information about new features in the 9.5 major release, see Section E.25.

E.13.1. Migration to Version 9.5.12

A dump/restore is not required for those running 9.5.X.

However, if you run an installation in which not all users are mutually trusting, or if you maintain an application or extension that is intended for use in arbitrary situations, it is strongly recommended that you read the documentation changes described in the first changelog entry below, and take suitable steps to ensure that your installation or code is secure.

Also, the changes described in the second changelog entry below may cause functions used in index expressions or materialized views to fail during auto-analyze, or when reloading from a dump. After upgrading, monitor the server logs for such problems, and fix affected functions.

Also, if you are upgrading from a version earlier than 9.5.10, see Section E.15.

E.13.2. Changes

- Document how to configure installations and applications to guard against search-path-dependent trojan-horse attacks from other users (Noah Misch)

  Using a `search_path` setting that includes any schemas writable by a hostile user enables that user to capture control of queries and then run arbitrary SQL code with the permissions of the attacked user. While it is possible to write queries that are proof against such hijacking, it is notationally tedious, and it's very easy to overlook holes. Therefore, we now recommend configurations in which no untrusted schemas appear in one's search path. Relevant documentation appears in Section 5.8.6 (for database administrators and users), Section 31.1 (for application authors), Section 35.15.5 (for extension authors), and CREATE FUNCTION (for authors of `SECURITY DEFINER` functions). (CVE-2018-1058)

- Avoid use of insecure `search_path` settings in pg_dump and other client programs (Noah Misch, Tom Lane)

  pg_dump, pg_upgrade, vacuumdb and other PostgreSQL-provided applications were themselves vulnerable to the type of hijacking described in the previous changelog entry; since these applications are commonly run by superusers, they present particularly attractive targets. To make them secure whether or not the installation as a whole has been secured, modify them to include only the `pg_catalog` schema in their `search_path` settings. Autovacuum worker processes now do the same, as well.

  In cases where user-provided functions are indirectly executed by these programs — for example, user-provided functions in index expressions — the tighter `search_path` may result in errors, which will need to be corrected by adjusting those user-provided functions to not assume anything about what search path they are invoked under. That has always been good practice, but now it will be necessary for correct behavior. (CVE-2018-1058)

- Fix misbehavior of concurrent-update rechecks with CTE references appearing in subplans (Tom Lane)

  If a CTE (`WITH` clause reference) is used in an InitPlan or SubPlan, and the query requires a recheck due to trying to update or lock a concurrently-updated row, incorrect results could be obtained.

- Fix planner failures with overlapping mergejoin clauses in an outer join (Tom Lane)

  These mistakes led to "left and right pathkeys do not match in mergejoin" or "outer pathkeys do not match mergeclauses" planner errors in corner cases.

- Repair pg_upgrade's failure to preserve `relfrozenxid` for materialized views (Tom Lane, Andres Freund)

  This oversight could lead to data corruption in materialized views after an upgrade, manifesting as "could not access status of transaction" or "found xmin from before relfrozenxid" errors. The problem would be more likely to occur in seldom-refreshed materialized views, or ones that were maintained only with `REFRESH MATERIALIZED VIEW CONCURRENTLY`.

If such corruption is observed, it can be repaired by refreshing the materialized view (without CONCURRENTLY).

- Fix incorrect reporting of PL/Python function names in error CONTEXT stacks (Tom Lane)

  An error occurring within a nested PL/Python function call (that is, one reached via a SPI query from another PL/Python function) would result in a stack trace showing the inner function's name twice, rather than the expected results. Also, an error in a nested PL/Python DO block could result in a null pointer dereference crash on some platforms.

- Allow contrib/auto_explain's log_min_duration setting to range up to INT_MAX, or about 24 days instead of 35 minutes (Tom Lane)

E.14. Release 9.5.11

Release date: 2018-02-08

This release contains a variety of fixes from 9.5.10. For information about new features in the 9.5 major release, see Section E.25.

E.14.1. Migration to Version 9.5.11

A dump/restore is not required for those running 9.5.X.

However, if you are upgrading from a version earlier than 9.5.10, see Section E.15.

E.14.2. Changes

- Ensure that all temporary files made by pg_upgrade are non-world-readable (Tom Lane, Noah Misch)

  pg_upgrade normally restricts its temporary files to be readable and writable only by the calling user. But the temporary file containing pg_dumpall -g output would be group- or world-readable, or even writable, if the user's umask setting allows. In typical usage on multi-user machines, the umask and/or the working directory's permissions would be tight enough to prevent problems; but there may be people using pg_upgrade in scenarios where this oversight would permit disclosure of database passwords to unfriendly eyes. (CVE-2018-1053)

- Fix vacuuming of tuples that were updated while key-share locked (Andres Freund, Álvaro Herrera)

  In some cases VACUUM would fail to remove such tuples even though they are now dead, leading to assorted data corruption scenarios.

- Fix inadequate buffer locking in some LSN fetches (Jacob Champion, Asim Praveen, Ashwin Agrawal)

  These errors could result in misbehavior under concurrent load. The potential consequences have not been characterized fully.

- Fix incorrect query results from cases involving flattening of subqueries whose outputs are used in GROUPING SETS (Heikki Linnakangas)

- Avoid unnecessary failure in a query on an inheritance tree that occurs concurrently with some child table being removed from the tree by `ALTER TABLE NO INHERIT` (Tom Lane)
- Fix spurious deadlock failures when multiple sessions are running `CREATE INDEX CONCURRENTLY` (Jeff Janes)
- Fix failures when an inheritance tree contains foreign child tables (Etsuro Fujita)

  A mix of regular and foreign tables in an inheritance tree resulted in creation of incorrect plans for `UPDATE` and `DELETE` queries. This led to visible failures in some cases, notably when there are row-level triggers on a foreign child table.
- Repair failure with correlated sub-`SELECT` inside `VALUES` inside a `LATERAL` subquery (Tom Lane)
- Fix "could not devise a query plan for the given query" planner failure for some cases involving nested `UNION ALL` inside a lateral subquery (Tom Lane)
- Fix logical decoding to correctly clean up disk files for crashed transactions (Atsushi Torikoshi)

  Logical decoding may spill WAL records to disk for transactions generating many WAL records. Normally these files are cleaned up after the transaction's commit or abort record arrives; but if no such record is ever seen, the removal code misbehaved.
- Fix walsender timeout failure and failure to respond to interrupts when processing a large transaction (Petr Jelinek)
- Fix `has_sequence_privilege()` to support `WITH GRANT OPTION` tests, as other privilege-testing functions do (Joe Conway)
- In databases using UTF8 encoding, ignore any XML declaration that asserts a different encoding (Pavel Stehule, Noah Misch)

  We always store XML strings in the database encoding, so allowing libxml to act on a declaration of another encoding gave wrong results. In encodings other than UTF8, we don't promise to support non-ASCII XML data anyway, so retain the previous behavior for bug compatibility. This change affects only `xpath()` and related functions; other XML code paths already acted this way.
- Provide for forward compatibility with future minor protocol versions (Robert Haas, Badrul Chowdhury)

  Up to now, PostgreSQL servers simply rejected requests to use protocol versions newer than 3.0, so that there was no functional difference between the major and minor parts of the protocol version number. Allow clients to request versions 3.x without failing, sending back a message showing that the server only understands 3.0. This makes no difference at the moment, but back-patching this change should allow speedier introduction of future minor protocol upgrades.
- Cope with failure to start a parallel worker process (Amit Kapila, Robert Haas)

  Parallel query previously tended to hang indefinitely if a worker could not be started, as the result of `fork()` failure or other low-probability problems.
- Avoid unsafe alignment assumptions when working with `__int128` (Tom Lane)

  Typically, compilers assume that `__int128` variables are aligned on 16-byte boundaries, but our memory allocation infrastructure isn't prepared to guarantee that, and increasing the setting of MAXALIGN seems infeasible for multiple reasons. Adjust the code to allow use of `__int128` only when we can tell the compiler to assume lesser alignment. The only known symptom of this problem so far is crashes in some parallel aggregation queries.

- Prevent stack-overflow crashes when planning extremely deeply nested set operations (`UNION`/`INTERSECT`/`EXCEPT`) (Tom Lane)
- Fix null-pointer crashes for some types of LDAP URLs appearing in `pg_hba.conf` (Thomas Munro)
- Fix sample `INSTR()` functions in the PL/pgSQL documentation (Yugo Nagata, Tom Lane)

  These functions are stated to be Oracle® compatible, but they weren't exactly. In particular, there was a discrepancy in the interpretation of a negative third parameter: Oracle thinks that a negative value indicates the last place where the target substring can begin, whereas our functions took it as the last place where the target can end. Also, Oracle throws an error for a zero or negative fourth parameter, whereas our functions returned zero.

  The sample code has been adjusted to match Oracle's behavior more precisely. Users who have copied this code into their applications may wish to update their copies.
- Fix pg_dump to make ACL (permissions), comment, and security label entries reliably identifiable in archive output formats (Tom Lane)

  The "tag" portion of an ACL archive entry was usually just the name of the associated object. Make it start with the object type instead, bringing ACLs into line with the convention already used for comment and security label archive entries. Also, fix the comment and security label entries for the whole database, if present, to make their tags start with `DATABASE` so that they also follow this convention. This prevents false matches in code that tries to identify large-object-related entries by seeing if the tag starts with `LARGE OBJECT`. That could have resulted in misclassifying entries as data rather than schema, with undesirable results in a schema-only or data-only dump.

  Note that this change has user-visible results in the output of `pg_restore --list`.
- Rename pg_rewind's `copy_file_range` function to avoid conflict with new Linux system call of that name (Andres Freund)

  This change prevents build failures with newer glibc versions.
- In ecpg, detect indicator arrays that do not have the correct length and report an error (David Rader)
- Avoid triggering a libc assertion in `contrib/hstore`, due to use of `memcpy()` with equal source and destination pointers (Tomas Vondra)
- Provide modern examples of how to auto-start Postgres on macOS (Tom Lane)

  The scripts in `contrib/start-scripts/osx` use infrastructure that's been deprecated for over a decade, and which no longer works at all in macOS releases of the last couple of years. Add a new subdirectory `contrib/start-scripts/macos` containing scripts that use the newer launchd infrastructure.
- Fix incorrect selection of configuration-specific libraries for OpenSSL on Windows (Andrew Dunstan)
- Support linking to MinGW-built versions of libperl (Noah Misch)

  This allows building PL/Perl with some common Perl distributions for Windows.
- Fix MSVC build to test whether 32-bit libperl needs `-D_USE_32BIT_TIME_T` (Noah Misch)

  Available Perl distributions are inconsistent about what they expect, and lack any reliable means of reporting it, so resort to a build-time test on what the library being used actually does.
- On Windows, install the crash dump handler earlier in postmaster startup (Takayuki Tsunakawa)

This may allow collection of a core dump for some early-startup failures that did not produce a dump before.

- On Windows, avoid encoding-conversion-related crashes when emitting messages very early in postmaster startup (Takayuki Tsunakawa)
- Use our existing Motorola 68K spinlock code on OpenBSD as well as NetBSD (David Carlier)
- Add support for spinlocks on Motorola 88K (David Carlier)
- Update time zone data files to tzdata release 2018c for DST law changes in Brazil, Sao Tome and Principe, plus historical corrections for Bolivia, Japan, and South Sudan. The `US/Pacific-New` zone has been removed (it was only an alias for `America/Los_Angeles` anyway).

E.15. Release 9.5.10

Release date: 2017-11-09

This release contains a variety of fixes from 9.5.9. For information about new features in the 9.5 major release, see Section E.25.

E.15.1. Migration to Version 9.5.10

A dump/restore is not required for those running 9.5.X.

However, if you use BRIN indexes, see the fourth changelog entry below.

Also, if you are upgrading from a version earlier than 9.5.8, see Section E.17.

E.15.2. Changes

- Ensure that `INSERT ... ON CONFLICT DO UPDATE` checks table permissions and RLS policies in all cases (Dean Rasheed)

  The update path of `INSERT ... ON CONFLICT DO UPDATE` requires `SELECT` permission on the columns of the arbiter index, but it failed to check for that in the case of an arbiter specified by constraint name. In addition, for a table with row level security enabled, it failed to check updated rows against the table's `SELECT` policies (regardless of how the arbiter index was specified). (CVE-2017-15099)

- Fix crash due to rowtype mismatch in `json{b}_populate_recordset()` (Michael Paquier, Tom Lane)

  These functions used the result rowtype specified in the `FROM ... AS` clause without checking that it matched the actual rowtype of the supplied tuple value. If it didn't, that would usually result in a crash, though disclosure of server memory contents seems possible as well. (CVE-2017-15098)

- Fix sample server-start scripts to become `$PGUSER` before opening `$PGLOG` (Noah Misch)

Previously, the postmaster log file was opened while still running as root. The database owner could therefore mount an attack against another system user by making `$PGLOG` be a symbolic link to some other file, which would then become corrupted by appending log messages.

By default, these scripts are not installed anywhere. Users who have made use of them will need to manually recopy them, or apply the same changes to their modified versions. If the existing `$PGLOG` file is root-owned, it will need to be removed or renamed out of the way before restarting the server with the corrected script. (CVE-2017-12172)

- Fix BRIN index summarization to handle concurrent table extension correctly (Álvaro Herrera)

  Previously, a race condition allowed some table rows to be omitted from the index. It may be necessary to reindex existing BRIN indexes to recover from past occurrences of this problem.

- Fix possible failures during concurrent updates of a BRIN index (Tom Lane)

  These race conditions could result in errors like "invalid index offnum" or "inconsistent range map".

- Fix crash when logical decoding is invoked from a SPI-using function, in particular any function written in a PL language (Tom Lane)

- Fix `json_build_array()`, `json_build_object()`, and their `jsonb` equivalents to handle explicit `VARIADIC` arguments correctly (Michael Paquier)

- Properly reject attempts to convert infinite float values to type `numeric` (Tom Lane, KaiGai Kohei)

  Previously the behavior was platform-dependent.

- Fix corner-case crashes when columns have been added to the end of a view (Tom Lane)

- Record proper dependencies when a view or rule contains `FieldSelect` or `FieldStore` expression nodes (Tom Lane)

  Lack of these dependencies could allow a column or data type `DROP` to go through when it ought to fail, thereby causing later uses of the view or rule to get errors. This patch does not do anything to protect existing views/rules, only ones created in the future.

- Correctly detect hashability of range data types (Tom Lane)

  The planner mistakenly assumed that any range type could be hashed for use in hash joins or hash aggregation, but actually it must check whether the range's subtype has hash support. This does not affect any of the built-in range types, since they're all hashable anyway.

- Correctly ignore `RelabelType` expression nodes when determining relation distinctness (David Rowley)

  This allows the intended optimization to occur when a subquery has a result column of type `varchar`.

- Fix low-probability loss of `NOTIFY` messages due to XID wraparound (Marko Tiikkaja, Tom Lane)

  If a session executed no queries, but merely listened for notifications, for more than 2 billion transactions, it started to miss some notifications from concurrently-committing transactions.

- Avoid SIGBUS crash on Linux when a DSM memory request exceeds the space available in tmpfs (Thomas Munro)

- Prevent low-probability crash in processing of nested trigger firings (Tom Lane)

- Allow `COPY`'s `FREEZE` option to work when the transaction isolation level is `REPEATABLE READ` or higher (Noah Misch)

This case was unintentionally broken by a previous bug fix.

- Correctly restore the umask setting when file creation fails in `COPY` or `lo_export()` (Peter Eisentraut)
- Give a better error message for duplicate column names in `ANALYZE` (Nathan Bossart)
- Fix mis-parsing of the last line in a non-newline-terminated `pg_hba.conf` file (Tom Lane)
- Fix pg_basebackup's matching of tablespace paths to canonicalize both paths before comparing (Michael Paquier)

  This is particularly helpful on Windows.
- Fix libpq to not require user's home directory to exist (Tom Lane)

  In v10, failure to find the home directory while trying to read `~/.pgpass` was treated as a hard error, but it should just cause that file to not be found. Both v10 and previous release branches made the same mistake when reading `~/.pg_service.conf`, though this was less obvious since that file is not sought unless a service name is specified.
- Fix libpq to guard against integer overflow in the row count of a `PGresult` (Michael Paquier)
- Fix ecpg's handling of out-of-scope cursor declarations with pointer or array variables (Michael Meskes)
- In ecpglib, correctly handle backslashes in string literals depending on whether `standard_conforming_strings` is set (Tsunakawa Takayuki)
- Make ecpglib's Informix-compatibility mode ignore fractional digits in integer input strings, as expected (Gao Zengqi, Michael Meskes)
- Fix missing temp-install prerequisites for `check`-like Make targets (Noah Misch)

  Some non-default test procedures that are meant to work like `make check` failed to ensure that the temporary installation was up to date.
- Sync our copy of the timezone library with IANA release tzcode2017c (Tom Lane)

  This fixes various issues; the only one likely to be user-visible is that the default DST rules for a POSIX-style zone name, if no `posixrules` file exists in the timezone data directory, now match current US law rather than what it was a dozen years ago.
- Update time zone data files to tzdata release 2017c for DST law changes in Fiji, Namibia, Northern Cyprus, Sudan, Tonga, and Turks & Caicos Islands, plus historical corrections for Alaska, Apia, Burma, Calcutta, Detroit, Ireland, Namibia, and Pago Pago.

E.16. Release 9.5.9

Release date: 2017-08-31

This release contains a small number of fixes from 9.5.8. For information about new features in the 9.5 major release, see Section E.25.

E.16.1. Migration to Version 9.5.9

A dump/restore is not required for those running 9.5.X.

However, if you are upgrading from a version earlier than 9.5.8, see Section E.17.

E.16.2. Changes

- Show foreign tables in `information_schema.table_privileges` view (Peter Eisentraut)

  All other relevant `information_schema` views include foreign tables, but this one ignored them.

  Since this view definition is installed by initdb, merely upgrading will not fix the problem. If you need to fix this in an existing installation, you can, as a superuser, do this in psql:

```
SET search_path TO information_schema;
CREATE OR REPLACE VIEW table_privileges AS
    SELECT CAST(u_grantor.rolname AS sql_identifier) AS grantor,
           CAST(grantee.rolname AS sql_identifier) AS grantee,
           CAST(current_database() AS sql_identifier) AS table_catalog,
           CAST(nc.nspname AS sql_identifier) AS table_schema,
           CAST(c.relname AS sql_identifier) AS table_name,
           CAST(c.prtype AS character_data) AS privilege_type,
           CAST(
             CASE WHEN
                  -- object owner always has grant options
                  pg_has_role(grantee.oid, c.relowner, 'USAGE')
                  OR c.grantable
                  THEN 'YES' ELSE 'NO' END AS yes_or_no) AS is_grantable,
           CAST(CASE WHEN c.prtype = 'SELECT' THEN 'YES' ELSE 'NO' END AS yes_or_no) AS wi

    FROM (
            SELECT oid, relname, relnamespace, relkind, relowner, (aclexplode(coalesce(rel
         ) AS c (oid, relname, relnamespace, relkind, relowner, grantor, grantee, prtype,
         pg_namespace nc,
         pg_authid u_grantor,
         (
           SELECT oid, rolname FROM pg_authid
           UNION ALL
           SELECT 0::oid, 'PUBLIC'
         ) AS grantee (oid, rolname)

    WHERE c.relnamespace = nc.oid
          AND c.relkind IN ('r', 'v', 'f')
          AND c.grantee = grantee.oid
          AND c.grantor = u_grantor.oid
          AND c.prtype IN ('INSERT', 'SELECT', 'UPDATE', 'DELETE', 'TRUNCATE', 'REFERENCES
          AND (pg_has_role(u_grantor.oid, 'USAGE')
               OR pg_has_role(grantee.oid, 'USAGE')
               OR grantee.rolname = 'PUBLIC');
```

  This must be repeated in each database to be fixed, including `template0`.

- Clean up handling of a fatal exit (e.g., due to receipt of SIGTERM) that occurs while trying to execute a `ROLLBACK` of a failed transaction (Tom Lane)

  This situation could result in an assertion failure. In production builds, the exit would still occur, but it would log an unexpected message about "cannot drop active portal".

- Remove assertion that could trigger during a fatal exit (Tom Lane)

- Correctly identify columns that are of a range type or domain type over a composite type or domain type being searched for (Tom Lane)

  Certain `ALTER` commands that change the definition of a composite type or domain type are supposed to fail if there are any stored values of that type in the database, because they lack the infrastructure needed to update or check such values. Previously, these checks could miss relevant values that are wrapped inside range types or sub-domains, possibly allowing the database to become inconsistent.

- Fix crash in pg_restore when using parallel mode and using a list file to select a subset of items to restore (Fabrízio de Royes Mello)

- Change ecpg's parser to allow `RETURNING` clauses without attached C variables (Michael Meskes)

  This allows ecpg programs to contain SQL constructs that use `RETURNING` internally (for example, inside a CTE) rather than using it to define values to be returned to the client.

- Improve selection of compiler flags for PL/Perl on Windows (Tom Lane)

  This fix avoids possible crashes of PL/Perl due to inconsistent assumptions about the width of `time_t` values. A side-effect that may be visible to extension developers is that `_USE_32BIT_TIME_T` is no longer defined globally in PostgreSQL Windows builds. This is not expected to cause problems, because type `time_t` is not used in any PostgreSQL API definitions.

- Fix `make check` to behave correctly when invoked via a non-GNU make program (Thomas Munro)

E.17. Release 9.5.8

Release date: 2017-08-10

This release contains a variety of fixes from 9.5.7. For information about new features in the 9.5 major release, see Section E.25.

E.17.1. Migration to Version 9.5.8

A dump/restore is not required for those running 9.5.X.

However, if you use foreign data servers that make use of user passwords for authentication, see the first changelog entry below.

Also, if you are upgrading from a version earlier than 9.5.7, see Section E.18.

E.17.2. Changes

- Further restrict visibility of `pg_user_mappings.umoptions`, to protect passwords stored as user mapping options (Noah Misch)

  The fix for CVE-2017-7486 was incorrect: it allowed a user to see the options in her own user mapping, even if she did not have `USAGE` permission on the associated foreign server. Such options might include a password that had been provided by the server owner rather than the user herself. Since `information_schema.user_mapping_options` does not show the options in such cases, `pg_user_mappings` should not either. (CVE-2017-7547)

  By itself, this patch will only fix the behavior in newly initdb'd databases. If you wish to apply this change in an existing database, you will need to do the following:

  1. Restart the postmaster after adding `allow_system_table_mods = true` to `postgresql.conf`. (In versions supporting `ALTER SYSTEM`, you can use that to make the configuration change, but you'll still need a restart.)

  2. In *each* database of the cluster, run the following commands as superuser:

```
SET search_path = pg_catalog;
CREATE OR REPLACE VIEW pg_user_mappings AS
    SELECT
        U.oid       AS umid,
        S.oid       AS srvid,
        S.srvname   AS srvname,
        U.umuser    AS umuser,
        CASE WHEN U.umuser = 0 THEN
            'public'
        ELSE
            A.rolname
        END AS usename,
        CASE WHEN (U.umuser <> 0 AND A.rolname = current_user
                    AND (pg_has_role(S.srvowner, 'USAGE')
                         OR has_server_privilege(S.oid, 'USAGE')))
                   OR (U.umuser = 0 AND pg_has_role(S.srvowner, 'USAGE'))
                   OR (SELECT rolsuper FROM pg_authid WHERE rolname = current_user)
                   THEN U.umoptions
               ELSE NULL END AS umoptions
    FROM pg_user_mapping U
         LEFT JOIN pg_authid A ON (A.oid = U.umuser) JOIN
        pg_foreign_server S ON (U.umserver = S.oid);
```

  3. Do not forget to include the `template0` and `template1` databases, or the vulnerability will still exist in databases you create later. To fix `template0`, you'll need to temporarily make it accept connections. In PostgreSQL 9.5 and later, you can use

```
ALTER DATABASE template0 WITH ALLOW_CONNECTIONS true;
```

  and then after fixing `template0`, undo that with

```
ALTER DATABASE template0 WITH ALLOW_CONNECTIONS false;
```

  In prior versions, instead use

```
UPDATE pg_database SET datallowconn = true WHERE datname = 'template0';
```

```
UPDATE pg_database SET datallowconn = false WHERE datname = 'template0';
```

4. Finally, remove the `allow_system_table_mods` configuration setting, and again restart the postmaster.

- Disallow empty passwords in all password-based authentication methods (Heikki Linnakangas)

  libpq ignores empty password specifications, and does not transmit them to the server. So, if a user's password has been set to the empty string, it's impossible to log in with that password via psql or other libpq-based clients. An administrator might therefore believe that setting the password to empty is equivalent to disabling password login. However, with a modified or non-libpq-based client, logging in could be possible, depending on which authentication method is configured. In particular the most common method, `md5`, accepted empty passwords. Change the server to reject empty passwords in all cases. (CVE-2017-7546)

- Make `lo_put()` check for `UPDATE` privilege on the target large object (Tom Lane, Michael Paquier)

  `lo_put()` should surely require the same permissions as `lowrite()`, but the check was missing, allowing any user to change the data in a large object. (CVE-2017-7548)

- Correct the documentation about the process for upgrading standby servers with pg_upgrade (Bruce Momjian)

  The previous documentation instructed users to start/stop the primary server after running pg_upgrade but before syncing the standby servers. This sequence is unsafe.

- Fix concurrent locking of tuple update chains (Álvaro Herrera)

  If several sessions concurrently lock a tuple update chain with nonconflicting lock modes using an old snapshot, and they all succeed, it was possible for some of them to nonetheless fail (and conclude there is no live tuple version) due to a race condition. This had consequences such as foreign-key checks failing to see a tuple that definitely exists but is being updated concurrently.

- Fix potential data corruption when freezing a tuple whose XMAX is a multixact with exactly one still-interesting member (Teodor Sigaev)

- Avoid integer overflow and ensuing crash when sorting more than one billion tuples in-memory (Sergey Koposov)

- On Windows, retry process creation if we fail to reserve the address range for our shared memory in the new process (Tom Lane, Amit Kapila)

  This is expected to fix infrequent child-process-launch failures that are probably due to interference from antivirus products.

- Fix low-probability corruption of shared predicate-lock hash table in Windows builds (Thomas Munro, Tom Lane)

- Avoid logging clean closure of an SSL connection as though it were a connection reset (Michael Paquier)

- Prevent sending SSL session tickets to clients (Tom Lane)

  This fix prevents reconnection failures with ticket-aware client-side SSL code.

- Fix code for setting tcp_keepalives_idle on Solaris (Tom Lane)

- Fix statistics collector to honor inquiry messages issued just after a postmaster shutdown and immediate restart (Tom Lane)

  Statistics inquiries issued within half a second of the previous postmaster shutdown were effectively ignored.

- Ensure that the statistics collector's receive buffer size is at least 100KB (Tom Lane)

  This reduces the risk of dropped statistics data on older platforms whose default receive buffer size is less than that.

- Fix possible creation of an invalid WAL segment when a standby is promoted just after it processes an `XLOG_SWITCH` WAL record (Andres Freund)

- Fix walsender to exit promptly when client requests shutdown (Tom Lane)

- Fix SIGHUP and SIGUSR1 handling in walsender processes (Petr Jelinek, Andres Freund)

- Prevent walsender-triggered panics during shutdown checkpoints (Andres Freund, Michael Paquier)

- Fix unnecessarily slow restarts of walreceiver processes due to race condition in postmaster (Tom Lane)

- Fix leakage of small subtransactions spilled to disk during logical decoding (Andres Freund)

  This resulted in temporary files consuming excessive disk space.

- Reduce the work needed to build snapshots during creation of logical-decoding slots (Andres Freund, Petr Jelinek)

  The previous algorithm was infeasibly expensive on a server with a lot of open transactions.

- Fix race condition that could indefinitely delay creation of logical-decoding slots (Andres Freund, Petr Jelinek)

- Reduce overhead in processing syscache invalidation events (Tom Lane)

  This is particularly helpful for logical decoding, which triggers frequent cache invalidation.

- Fix cases where an `INSERT` or `UPDATE` assigns to more than one element of a column that is of domain-over-array type (Tom Lane)

- Allow window functions to be used in sub-`SELECT`s that are within the arguments of an aggregate function (Tom Lane)

- Move autogenerated array types out of the way during `ALTER ... RENAME` (Vik Fearing)

  Previously, we would rename a conflicting autogenerated array type out of the way during `CREATE`; this fix extends that behavior to renaming operations.

- Fix dangling pointer in `ALTER TABLE` when there is a comment on a constraint belonging to the table (David Rowley)

  Re-applying the comment to the reconstructed constraint could fail with a weird error message, or even crash.

- Ensure that `ALTER USER ... SET` accepts all the syntax variants that `ALTER ROLE ... SET` does (Peter Eisentraut)

- Properly update dependency info when changing a datatype I/O function's argument or return type from `opaque` to the correct type (Heikki Linnakangas)

  `CREATE TYPE` updates I/O functions declared in this long-obsolete style, but it forgot to record a dependency on the type, allowing a subsequent `DROP TYPE` to leave broken function definitions behind.

- Reduce memory usage when `ANALYZE` processes a `tsvector` column (Heikki Linnakangas)
- Fix unnecessary precision loss and sloppy rounding when multiplying or dividing `money` values by integers or floats (Tom Lane)
- Tighten checks for whitespace in functions that parse identifiers, such as `regprocedurein()` (Tom Lane)

  Depending on the prevailing locale, these functions could misinterpret fragments of multibyte characters as whitespace.
- Use relevant `#define` symbols from Perl while compiling PL/Perl (Ashutosh Sharma, Tom Lane)

  This avoids portability problems, typically manifesting as a "handshake" mismatch during library load, when working with recent Perl versions.
- In libpq, reset GSS/SASL and SSPI authentication state properly after a failed connection attempt (Michael Paquier)

  Failure to do this meant that when falling back from SSL to non-SSL connections, a GSS/SASL failure in the SSL attempt would always cause the non-SSL attempt to fail. SSPI did not fail, but it leaked memory.
- In psql, fix failure when `COPY FROM STDIN` is ended with a keyboard EOF signal and then another `COPY FROM STDIN` is attempted (Thomas Munro)

  This misbehavior was observed on BSD-derived platforms (including macOS), but not on most others.
- Fix pg_dump and pg_restore to emit `REFRESH MATERIALIZED VIEW` commands last (Tom Lane)

  This prevents errors during dump/restore when a materialized view refers to tables owned by a different user.
- Improve pg_dump/pg_restore's reporting of error conditions originating in zlib (Vladimir Kunschikov, Álvaro Herrera)
- Fix pg_dump with the `--clean` option to drop event triggers as expected (Tom Lane)

  It also now correctly assigns ownership of event triggers; before, they were restored as being owned by the superuser running the restore script.
- Fix pg_dump to not emit invalid SQL for an empty operator class (Daniel Gustafsson)
- Fix pg_dump output to stdout on Windows (Kuntal Ghosh)

  A compressed plain-text dump written to stdout would contain corrupt data due to failure to put the file descriptor into binary mode.
- Fix `pg_get_ruledef()` to print correct output for the `ON SELECT` rule of a view whose columns have been renamed (Tom Lane)

  In some corner cases, pg_dump relies on `pg_get_ruledef()` to dump views, so that this error could result in dump/reload failures.
- Fix dumping of outer joins with empty constraints, such as the result of a `NATURAL LEFT JOIN` with no common columns (Tom Lane)
- Fix dumping of function expressions in the `FROM` clause in cases where the expression does not deparse into something that looks like a function call (Tom Lane)
- Fix pg_basebackup output to stdout on Windows (Haribabu Kommi)

A backup written to stdout would contain corrupt data due to failure to put the file descriptor into binary mode.

- Fix pg_rewind to correctly handle files exceeding 2GB (Kuntal Ghosh, Michael Paquier)

  Ordinarily such files won't appear in PostgreSQL data directories, but they could be present in some cases.

- Fix pg_upgrade to ensure that the ending WAL record does not have wal_level = `minimum` (Bruce Momjian)

  This condition could prevent upgraded standby servers from reconnecting.

- Fix pg_xlogdump's computation of WAL record length (Andres Freund)

- In `postgres_fdw`, re-establish connections to remote servers after `ALTER SERVER` or `ALTER USER MAPPING` commands (Kyotaro Horiguchi)

  This ensures that option changes affecting connection parameters will be applied promptly.

- In `postgres_fdw`, allow cancellation of remote transaction control commands (Robert Haas, Rafia Sabih)

  This change allows us to quickly escape a wait for an unresponsive remote server in many more cases than previously.

- Increase `MAX_SYSCACHE_CALLBACKS` to provide more room for extensions (Tom Lane)

- Always use `-fPIC`, not `-fpic`, when building shared libraries with gcc (Tom Lane)

  This supports larger extension libraries on platforms where it makes a difference.

- Fix unescaped-braces issue in our build scripts for Microsoft MSVC, to avoid a warning or error from recent Perl versions (Andrew Dunstan)

- In MSVC builds, handle the case where the openssl library is not within a `VC` subdirectory (Andrew Dunstan)

- In MSVC builds, add proper include path for libxml2 header files (Andrew Dunstan)

  This fixes a former need to move things around in standard Windows installations of libxml2.

- In MSVC builds, recognize a Tcl library that is named `tcl86.lib` (Noah Misch)

- In MSVC builds, honor `PROVE_FLAGS` settings on `vcregress.pl`'s command line (Andrew Dunstan)

E.18. Release 9.5.7

Release date: 2017-05-11

This release contains a variety of fixes from 9.5.6. For information about new features in the 9.5 major release, see Section E.25.

E.18.1. Migration to Version 9.5.7

A dump/restore is not required for those running 9.5.X.

However, if you use foreign data servers that make use of user passwords for authentication, see the first changelog entry below.

Also, if you are using third-party replication tools that depend on "logical decoding", see the fourth changelog entry below.

Also, if you are upgrading from a version earlier than 9.5.6, see Section E.19.

E.18.2. Changes

- Restrict visibility of `pg_user_mappings.umoptions`, to protect passwords stored as user mapping options (Michael Paquier, Feike Steenbergen)

  The previous coding allowed the owner of a foreign server object, or anyone he has granted server `USAGE` permission to, to see the options for all user mappings associated with that server. This might well include passwords for other users. Adjust the view definition to match the behavior of `information_schema.user_mapping_options`, namely that these options are visible to the user being mapped, or if the mapping is for `PUBLIC` and the current user is the server owner, or if the current user is a superuser. (CVE-2017-7486)

  By itself, this patch will only fix the behavior in newly initdb'd databases. If you wish to apply this change in an existing database, follow the corrected procedure shown in the changelog entry for CVE-2017-7547, in Section E.17.

- Prevent exposure of statistical information via leaky operators (Peter Eisentraut)

  Some selectivity estimation functions in the planner will apply user-defined operators to values obtained from `pg_statistic`, such as most common values and histogram entries. This occurs before table permissions are checked, so a nefarious user could exploit the behavior to obtain these values for table columns he does not have permission to read. To fix, fall back to a default estimate if the operator's implementation function is not certified leak-proof and the calling user does not have permission to read the table column whose statistics are needed. At least one of these criteria is satisfied in most cases in practice. (CVE-2017-7484)

- Restore libpq's recognition of the `PGREQUIRESSL` environment variable (Daniel Gustafsson)

  Processing of this environment variable was unintentionally dropped in PostgreSQL 9.3, but its documentation remained. This creates a security hazard, since users might be relying on the environment variable to force SSL-encrypted connections, but that would no longer be guaranteed. Restore handling of the variable, but give it lower priority than `PGSSLMODE`, to avoid breaking configurations that work correctly with post-9.3 code. (CVE-2017-7485)

- Fix possibly-invalid initial snapshot during logical decoding (Petr Jelinek, Andres Freund)

  The initial snapshot created for a logical decoding replication slot was potentially incorrect. This could cause third-party tools that use logical decoding to copy incomplete/inconsistent initial data. This was more likely to happen if the source server was busy at the time of slot creation, or if another logical slot already existed.

  If you are using a replication tool that depends on logical decoding, and it should have copied a nonempty data set at the start of replication, it is advisable to recreate the replica after installing this update, or to verify its contents against the source server.

- Fix possible corruption of "init forks" of unlogged indexes (Robert Haas, Michael Paquier)

  This could result in an unlogged index being set to an invalid state after a crash and restart. Such a problem would persist until the index was dropped and rebuilt.

- Fix incorrect reconstruction of `pg_subtrans` entries when a standby server replays a prepared but uncommitted two-phase transaction (Tom Lane)

  In most cases this turned out to have no visible ill effects, but in corner cases it could result in circular references in `pg_subtrans`, potentially causing infinite loops in queries that examine rows modified by the two-phase transaction.

- Avoid possible crash in walsender due to failure to initialize a string buffer (Stas Kelvich, Fujii Masao)

- Fix possible crash when rescanning a nearest-neighbor index-only scan on a GiST index (Tom Lane)

- Fix postmaster's handling of `fork()` failure for a background worker process (Tom Lane)

  Previously, the postmaster updated portions of its state as though the process had been launched successfully, resulting in subsequent confusion.

- Fix crash or wrong answers when a `GROUPING SETS` column's data type is hashable but not sortable (Pavan Deolasee)

- Avoid applying "physical targetlist" optimization to custom scans (Dmitry Ivanov, Tom Lane)

  This optimization supposed that retrieving all columns of a tuple is inexpensive, which is true for ordinary Postgres tuples; but it might not be the case for a custom scan provider.

- Use the correct sub-expression when applying a `FOR ALL` row-level-security policy (Stephen Frost)

  In some cases the `WITH CHECK` restriction would be applied when the `USING` restriction is more appropriate.

- Ensure parsing of queries in extension scripts sees the results of immediately-preceding DDL (Julien Rouhaud, Tom Lane)

  Due to lack of a cache flush step between commands in an extension script file, non-utility queries might not see the effects of an immediately preceding catalog change, such as `ALTER TABLE ... RENAME`.

- Skip tablespace privilege checks when `ALTER TABLE ... ALTER COLUMN TYPE` rebuilds an existing index (Noah Misch)

  The command failed if the calling user did not currently have `CREATE` privilege for the tablespace containing the index. That behavior seems unhelpful, so skip the check, allowing the index to be rebuilt where it is.

- Fix `ALTER TABLE ... VALIDATE CONSTRAINT` to not recurse to child tables when the constraint is marked `NO INHERIT` (Amit Langote)

  This fix prevents unwanted "constraint does not exist" failures when no matching constraint is present in the child tables.

- Avoid dangling pointer in `COPY ... TO` when row-level security is active for the source table (Tom Lane)

  Usually this had no ill effects, but sometimes it would cause unexpected errors or crashes.

- Avoid accessing an already-closed relcache entry in `CLUSTER` and `VACUUM FULL` (Tom Lane)

With some bad luck, this could lead to indexes on the target relation getting rebuilt with the wrong persistence setting.

- Fix `VACUUM` to account properly for pages that could not be scanned due to conflicting page pins (Andrew Gierth)

  This tended to lead to underestimation of the number of tuples in the table. In the worst case of a small heavily-contended table, `VACUUM` could incorrectly report that the table contained no tuples, leading to very bad planning choices.

- Ensure that bulk-tuple-transfer loops within a hash join are interruptible by query cancel requests (Tom Lane, Thomas Munro)

- Fix integer-overflow problems in `interval` comparison (Kyotaro Horiguchi, Tom Lane)

  The comparison operators for type `interval` could yield wrong answers for intervals larger than about 296000 years. Indexes on columns containing such large values should be reindexed, since they may be corrupt.

- Fix `cursor_to_xml()` to produce valid output with *`tableforest`* = false (Thomas Munro, Peter Eisentraut)

  Previously it failed to produce a wrapping <table> element.

- Fix roundoff problems in `float8_timestamptz()` and `make_interval()` (Tom Lane)

  These functions truncated, rather than rounded, when converting a floating-point value to integer microseconds; that could cause unexpectedly off-by-one results.

- Fix `pg_get_object_address()` to handle members of operator families correctly (Álvaro Herrera)

- Improve performance of `pg_timezone_names` view (Tom Lane, David Rowley)

- Reduce memory management overhead for contexts containing many large blocks (Tom Lane)

- Fix sloppy handling of corner-case errors from `lseek()` and `close()` (Tom Lane)

  Neither of these system calls are likely to fail in typical situations, but if they did, `fd.c` could get quite confused.

- Fix incorrect check for whether postmaster is running as a Windows service (Michael Paquier)

  This could result in attempting to write to the event log when that isn't accessible, so that no logging happens at all.

- Fix ecpg to support `COMMIT PREPARED` and `ROLLBACK PREPARED` (Masahiko Sawada)

- Fix a double-free error when processing dollar-quoted string literals in ecpg (Michael Meskes)

- In pg_dump, fix incorrect schema and owner marking for comments and security labels of some types of database objects (Giuseppe Broccolo, Tom Lane)

  In simple cases this caused no ill effects; but for example, a schema-selective restore might omit comments it should include, because they were not marked as belonging to the schema of their associated object.

- Avoid emitting an invalid list file in `pg_restore -l` when SQL object names contain newlines (Tom Lane)

  Replace newlines by spaces, which is sufficient to make the output valid for `pg_restore -L`'s purposes.

- Fix pg_upgrade to transfer comments and security labels attached to "large objects" (blobs) (Stephen Frost)

  Previously, blobs were correctly transferred to the new database, but any comments or security labels attached to them were lost.

- Improve error handling in `contrib/adminpack`'s `pg_file_write()` function (Noah Misch)

  Notably, it failed to detect errors reported by `fclose()`.

- In `contrib/dblink`, avoid leaking the previous unnamed connection when establishing a new unnamed connection (Joe Conway)

- Fix `contrib/pg_trgm`'s extraction of trigrams from regular expressions (Tom Lane)

  In some cases it would produce a broken data structure that could never match anything, leading to GIN or GiST indexscans that use a trigram index not finding any matches to the regular expression.

- In `contrib/postgres_fdw`, transmit query cancellation requests to the remote server (Michael Paquier, Etsuro Fujita)

  Previously, a local query cancellation request did not cause an already-sent remote query to terminate early. This is a back-patch of work originally done for 9.6.

- Support Tcl 8.6 in MSVC builds (Álvaro Herrera)

- Sync our copy of the timezone library with IANA release tzcode2017b (Tom Lane)

  This fixes a bug affecting some DST transitions in January 2038.

- Update time zone data files to tzdata release 2017b for DST law changes in Chile, Haiti, and Mongolia, plus historical corrections for Ecuador, Kazakhstan, Liberia, and Spain. Switch to numeric abbreviations for numerous time zones in South America, the Pacific and Indian oceans, and some Asian and Middle Eastern countries.

  The IANA time zone database previously provided textual abbreviations for all time zones, sometimes making up abbreviations that have little or no currency among the local population. They are in process of reversing that policy in favor of using numeric UTC offsets in zones where there is no evidence of real-world use of an English abbreviation. At least for the time being, PostgreSQL will continue to accept such removed abbreviations for timestamp input. But they will not be shown in the `pg_timezone_names` view nor used for output.

- Use correct daylight-savings rules for POSIX-style time zone names in MSVC builds (David Rowley)

  The Microsoft MSVC build scripts neglected to install the `posixrules` file in the timezone directory tree. This resulted in the timezone code falling back to its built-in rule about what DST behavior to assume for a POSIX-style time zone name. For historical reasons that still corresponds to the DST rules the USA was using before 2007 (i.e., change on first Sunday in April and last Sunday in October). With this fix, a POSIX-style zone name will use the current and historical DST transition dates of the `US/Eastern` zone. If you don't want that, remove the `posixrules` file, or replace it with a copy of some other zone file (see Section 8.5.3). Note that due to caching, you may need to restart the server to get such changes to take effect.

E.19. Release 9.5.6

Release date: 2017-02-09

This release contains a variety of fixes from 9.5.5. For information about new features in the 9.5 major release, see Section E.25.

E.19.1. Migration to Version 9.5.6

A dump/restore is not required for those running 9.5.X.

However, if your installation has been affected by the bug described in the first changelog entry below, then after updating you may need to take action to repair corrupted indexes.

Also, if you are upgrading from a version earlier than 9.5.5, see Section E.20.

E.19.2. Changes

- Fix a race condition that could cause indexes built with `CREATE INDEX CONCURRENTLY` to be corrupt (Pavan Deolasee, Tom Lane)

  If `CREATE INDEX CONCURRENTLY` was used to build an index that depends on a column not previously indexed, then rows updated by transactions that ran concurrently with the `CREATE INDEX` command could have received incorrect index entries. If you suspect this may have happened, the most reliable solution is to rebuild affected indexes after installing this update.

- Ensure that the special snapshot used for catalog scans is not invalidated by premature data pruning (Tom Lane)

  Backends failed to account for this snapshot when advertising their oldest xmin, potentially allowing concurrent vacuuming operations to remove data that was still needed. This led to transient failures along the lines of "cache lookup failed for relation 1255".

- Fix incorrect WAL logging for BRIN indexes (Kuntal Ghosh)

  The WAL record emitted for a BRIN "revmap" page when moving an index tuple to a different page was incorrect. Replay would make the related portion of the index useless, forcing it to be recomputed.

- Unconditionally WAL-log creation of the "init fork" for an unlogged table (Michael Paquier)

  Previously, this was skipped when wal_level = `minimal`, but actually it's necessary even in that case to ensure that the unlogged table is properly reset to empty after a crash.

- Reduce interlocking on standby servers during the replay of btree index vacuuming operations (Simon Riggs)

  This change avoids substantial replication delays that sometimes occurred while replaying such operations.

- If the stats collector dies during hot standby, restart it (Takayuki Tsunakawa)

- Ensure that hot standby feedback works correctly when it's enabled at standby server start (Ants Aasma, Craig Ringer)

- Check for interrupts while hot standby is waiting for a conflicting query (Simon Riggs)
- Avoid constantly respawning the autovacuum launcher in a corner case (Amit Khandekar)

  This fix avoids problems when autovacuum is nominally off and there are some tables that require freezing, but all such tables are already being processed by autovacuum workers.
- Fix check for when an extension member object can be dropped (Tom Lane)

  Extension upgrade scripts should be able to drop member objects, but this was disallowed for serial-column sequences, and possibly other cases.
- Make sure `ALTER TABLE` preserves index tablespace assignments when rebuilding indexes (Tom Lane, Michael Paquier)

  Previously, non-default settings of default_tablespace could result in broken indexes.
- Fix incorrect updating of trigger function properties when changing a foreign-key constraint's deferrability properties with `ALTER TABLE ... ALTER CONSTRAINT` (Tom Lane)

  This led to odd failures during subsequent exercise of the foreign key, as the triggers were fired at the wrong times.
- Prevent dropping a foreign-key constraint if there are pending trigger events for the referenced relation (Tom Lane)

  This avoids "could not find trigger *`NNN`*" or "relation *`NNN`* has no triggers" errors.
- Fix `ALTER TABLE ... SET DATA TYPE ... USING` when child table has different column ordering than the parent (Álvaro Herrera)

  Failure to adjust the column numbering in the `USING` expression led to errors, typically "attribute *`N`* has wrong type".
- Fix processing of OID column when a table with OIDs is associated to a parent with OIDs via `ALTER TABLE ... INHERIT` (Amit Langote)

  The OID column should be treated the same as regular user columns in this case, but it wasn't, leading to odd behavior in later inheritance changes.
- Fix `CREATE OR REPLACE VIEW` to update the view query before attempting to apply the new view options (Dean Rasheed)

  Previously the command would fail if the new options were inconsistent with the old view definition.
- Report correct object identity during `ALTER TEXT SEARCH CONFIGURATION` (Artur Zakirov)

  The wrong catalog OID was reported to extensions such as logical decoding.
- Fix commit timestamp mechanism to not fail when queried about the special XIDs `FrozenTransactionId` and `BootstrapTransactionId` (Craig Ringer)
- Check for serializability conflicts before reporting constraint-violation failures (Thomas Munro)

  When using serializable transaction isolation, it is desirable that any error due to concurrent transactions should manifest as a serialization failure, thereby cueing the application that a retry might succeed. Unfortunately, this does not reliably happen for duplicate-key failures caused by concurrent insertions. This change ensures that such an error will be reported as a serialization error if the application explicitly checked for the presence of a conflicting key (and did not find it) earlier in the transaction.
- Fix incorrect use of view reloptions as regular table reloptions (Tom Lane)

The symptom was spurious "ON CONFLICT is not supported on table ... used as a catalog table" errors when the target of `INSERT ... ON CONFLICT` is a view with cascade option.

- Fix incorrect "target lists can have at most *N* entries" complaint when using `ON CONFLICT` with wide tables (Tom Lane)
- Prevent multicolumn expansion of *foo*`.*` in an `UPDATE` source expression (Tom Lane)

  This led to "UPDATE target count mismatch --- internal error". Now the syntax is understood as a whole-row variable, as it would be in other contexts.
- Ensure that column typmods are determined accurately for multi-row `VALUES` constructs (Tom Lane)

  This fixes problems occurring when the first value in a column has a determinable typmod (e.g., length for a `varchar` value) but later values don't share the same limit.
- Throw error for an unfinished Unicode surrogate pair at the end of a Unicode string (Tom Lane)

  Normally, a Unicode surrogate leading character must be followed by a Unicode surrogate trailing character, but the check for this was missed if the leading character was the last character in a Unicode string literal (`U&'...'`) or Unicode identifier (`U&"..."`).
- Ensure that a purely negative text search query, such as `!foo`, matches empty `tsvector`s (Tom Dunstan)

  Such matches were found by GIN index searches, but not by sequential scans or GiST index searches.
- Prevent crash when `ts_rewrite()` replaces a non-top-level subtree with an empty query (Artur Zakirov)
- Fix performance problems in `ts_rewrite()` (Tom Lane)
- Fix `ts_rewrite()`'s handling of nested NOT operators (Tom Lane)
- Improve speed of user-defined aggregates that use `array_append()` as transition function (Tom Lane)
- Fix `array_fill()` to handle empty arrays properly (Tom Lane)
- Fix possible crash in `array_position()` or `array_positions()` when processing arrays of records (Junseok Yang)
- Fix one-byte buffer overrun in `quote_literal_cstr()` (Heikki Linnakangas)

  The overrun occurred only if the input consisted entirely of single quotes and/or backslashes.
- Prevent multiple calls of `pg_start_backup()` and `pg_stop_backup()` from running concurrently (Michael Paquier)

  This avoids an assertion failure, and possibly worse things, if someone tries to run these functions in parallel.
- Disable transform that attempted to remove no-op `AT TIME ZONE` conversions (Tom Lane)

  This resulted in wrong answers when the simplified expression was used in an index condition.
- Avoid discarding `interval`-to-`interval` casts that aren't really no-ops (Tom Lane)

  In some cases, a cast that should result in zeroing out low-order `interval` fields was mistakenly deemed to be a no-op and discarded. An example is that casting from `INTERVAL MONTH` to `INTERVAL YEAR` failed to clear the months field.
- Fix bugs in transmitting GUC parameter values to parallel workers (Michael Paquier, Tom Lane)

- Ensure that cached plans are invalidated by changes in foreign-table options (Amit Langote, Etsuro Fujita, Ashutosh Bapat)
- Fix pg_dump to dump user-defined casts and transforms that use built-in functions (Stephen Frost)
- Fix pg_restore with `--create --if-exists` to behave more sanely if an archive contains unrecognized `DROP` commands (Tom Lane)

  This doesn't fix any live bug, but it may improve the behavior in future if pg_restore is used with an archive generated by a later pg_dump version.
- Fix pg_basebackup's rate limiting in the presence of slow I/O (Antonin Houska)

  If disk I/O was transiently much slower than the specified rate limit, the calculation overflowed, effectively disabling the rate limit for the rest of the run.
- Fix pg_basebackup's handling of symlinked `pg_stat_tmp` and `pg_replslot` subdirectories (Magnus Hagander, Michael Paquier)
- Fix possible pg_basebackup failure on standby server when including WAL files (Amit Kapila, Robert Haas)
- Fix possible mishandling of expanded arrays in domain check constraints and `CASE` execution (Tom Lane)

  It was possible for a PL/pgSQL function invoked in these contexts to modify or even delete an array value that needs to be preserved for additional operations.
- Fix nested uses of PL/pgSQL functions in contexts such as domain check constraints evaluated during assignment to a PL/pgSQL variable (Tom Lane)
- Ensure that the Python exception objects we create for PL/Python are properly reference-counted (Rafa de la Torre, Tom Lane)

  This avoids failures if the objects are used after a Python garbage collection cycle has occurred.
- Fix PL/Tcl to support triggers on tables that have `.tupno` as a column name (Tom Lane)

  This matches the (previously undocumented) behavior of PL/Tcl's `spi_exec` and `spi_execp` commands, namely that a magic `.tupno` column is inserted only if there isn't a real column named that.
- Allow DOS-style line endings in `~/.pgpass` files, even on Unix (Vik Fearing)

  This change simplifies use of the same password file across Unix and Windows machines.
- Fix one-byte buffer overrun if ecpg is given a file name that ends with a dot (Takayuki Tsunakawa)
- Fix psql's tab completion for `ALTER DEFAULT PRIVILEGES` (Gilles Darold, Stephen Frost)
- In psql, treat an empty or all-blank setting of the `PAGER` environment variable as meaning "no pager" (Tom Lane)

  Previously, such a setting caused output intended for the pager to vanish entirely.
- Improve `contrib/dblink`'s reporting of low-level libpq errors, such as out-of-memory (Joe Conway)
- Teach `contrib/dblink` to ignore irrelevant server options when it uses a `contrib/postgres_fdw` foreign server as the source of connection options (Corey Huinker)

  Previously, if the foreign server object had options that were not also libpq connection options, an error occurred.

- Fix portability problems in `contrib/pageinspect`'s functions for GIN indexes (Peter Eisentraut, Tom Lane)
- On Windows, ensure that environment variable changes are propagated to DLLs built with debug options (Christian Ullrich)
- Sync our copy of the timezone library with IANA release tzcode2016j (Tom Lane)

  This fixes various issues, most notably that timezone data installation failed if the target directory didn't support hard links.
- Update time zone data files to tzdata release 2016j for DST law changes in northern Cyprus (adding a new zone Asia/Famagusta), Russia (adding a new zone Europe/Saratov), Tonga, and Antarctica/Casey. Historical corrections for Italy, Kazakhstan, Malta, and Palestine. Switch to preferring numeric zone abbreviations for Tonga.

E.20. Release 9.5.5

Release date: 2016-10-27

This release contains a variety of fixes from 9.5.4. For information about new features in the 9.5 major release, see Section E.25.

E.20.1. Migration to Version 9.5.5

A dump/restore is not required for those running 9.5.X.

However, if your installation has been affected by the bug described in the first changelog entry below, then after updating you may need to take action to repair corrupted free space maps.

Also, if you are upgrading from a version earlier than 9.5.2, see Section E.23.

E.20.2. Changes

- Fix WAL-logging of truncation of relation free space maps and visibility maps (Pavan Deolasee, Heikki Linnakangas)

  It was possible for these files to not be correctly restored during crash recovery, or to be written incorrectly on a standby server. Bogus entries in a free space map could lead to attempts to access pages that have been truncated away from the relation itself, typically producing errors like "could not read block *xxx*: read only 0 of 8192 bytes". Checksum failures in the visibility map are also possible, if checksumming is enabled.

  Procedures for determining whether there is a problem and repairing it if so are discussed at https://wiki.postgresql.org/wiki/Free_Space_Map_Problems.
- Fix incorrect creation of GIN index WAL records on big-endian machines (Tom Lane)

  The typical symptom was "unexpected GIN leaf action" errors during WAL replay.

- Fix `SELECT FOR UPDATE/SHARE` to correctly lock tuples that have been updated by a subsequently-aborted transaction (Álvaro Herrera)

  In 9.5 and later, the `SELECT` would sometimes fail to return such tuples at all. A failure has not been proven to occur in earlier releases, but might be possible with concurrent updates.

- Fix EvalPlanQual rechecks involving CTE scans (Tom Lane)

  The recheck would always see the CTE as returning no rows, typically leading to failure to update rows that were recently updated.

- Fix deletion of speculatively inserted TOAST tuples when backing out of `INSERT ... ON CONFLICT` (Oskari Saarenmaa)

  In the race condition where two transactions try to insert conflicting tuples at about the same time, the loser would fail with an "attempted to delete invisible tuple" error if its insertion included any TOAST'ed fields.

- Don't throw serialization errors for self-conflicting insertions in `INSERT ... ON CONFLICT` (Thomas Munro, Peter Geoghegan)

- Fix improper repetition of previous results from hashed aggregation in a subquery (Andrew Gierth)

  The test to see if we can reuse a previously-computed hash table of the aggregate state values neglected the possibility of an outer query reference appearing in an aggregate argument expression. A change in the value of such a reference should lead to recalculating the hash table, but did not.

- Fix query-lifespan memory leak in a bulk `UPDATE` on a table with a `PRIMARY KEY` or `REPLICA IDENTITY` index (Tom Lane)

- Fix `COPY` with a column name list from a table that has row-level security enabled (Adam Brightwell)

- Fix `EXPLAIN` to emit valid XML when track_io_timing is on (Markus Winand)

  Previously the XML output-format option produced syntactically invalid tags such as `<I/O-Read-Time>`. That is now rendered as `<I-O-Read-Time>`.

- Suppress printing of zeroes for unmeasured times in `EXPLAIN` (Maksim Milyutin)

  Certain option combinations resulted in printing zero values for times that actually aren't ever measured in that combination. Our general policy in `EXPLAIN` is not to print such fields at all, so do that consistently in all cases.

- Fix statistics update for `TRUNCATE` in a prepared transaction (Stas Kelvich)

- Fix timeout length when `VACUUM` is waiting for exclusive table lock so that it can truncate the table (Simon Riggs)

  The timeout was meant to be 50 milliseconds, but it was actually only 50 microseconds, causing `VACUUM` to give up on truncation much more easily than intended. Set it to the intended value.

- Fix bugs in merging inherited `CHECK` constraints while creating or altering a table (Tom Lane, Amit Langote)

  Allow identical `CHECK` constraints to be added to a parent and child table in either order. Prevent merging of a valid constraint from the parent table with a `NOT VALID` constraint on the child. Likewise, prevent merging of a `NO INHERIT` child constraint with an inherited constraint.

- Show a sensible value in `pg_settings.unit` for `min_wal_size` and `max_wal_size` (Tom Lane)

- Remove artificial restrictions on the values accepted by `numeric_in()` and `numeric_recv()` (Tom Lane)

  We allow numeric values up to the limit of the storage format (more than `1e100000`), so it seems fairly pointless that `numeric_in()` rejected scientific-notation exponents above 1000. Likewise, it was silly for `numeric_recv()` to reject more than 1000 digits in an input value.

- Avoid very-low-probability data corruption due to testing tuple visibility without holding buffer lock (Thomas Munro, Peter Geoghegan, Tom Lane)

- Preserve commit timestamps across server restart (Julien Rouhaud, Craig Ringer)

  With track_commit_timestamp turned on, old commit timestamps became inaccessible after a clean server restart.

- Fix logical WAL decoding to work properly when a subtransaction's WAL output is large enough to spill to disk (Andres Freund)

- Fix possible sorting error when aborting use of abbreviated keys (Peter Geoghegan)

  In the worst case, this could result in a corrupt btree index, which would need to be rebuilt using `REINDEX`. However, the situation is believed to be rare.

- Fix file descriptor leakage when truncating a temporary relation of more than 1GB (Andres Freund)

- Disallow starting a standalone backend with `standby_mode` turned on (Michael Paquier)

  This can't do anything useful, since there will be no WAL receiver process to fetch more WAL data; and it could result in misbehavior in code that wasn't designed with this situation in mind.

- Properly initialize replication slot state when recycling a previously-used slot (Michael Paquier)

  This failure to reset all of the fields of the slot could prevent `VACUUM` from removing dead tuples.

- Round shared-memory allocation request to a multiple of the actual huge page size when attempting to use huge pages on Linux (Tom Lane)

  This avoids possible failures during `munmap()` on systems with atypical default huge page sizes. Except in crash-recovery cases, there were no ill effects other than a log message.

- Use a more random value for the dynamic shared memory control segment's ID (Robert Haas, Tom Lane)

  Previously, the same value would be chosen every time, because it was derived from `random()` but `srandom()` had not yet been called. While relatively harmless, this was not the intended behavior.

- On Windows, retry creation of the dynamic shared memory control segment after an access-denied error (Kyotaro Horiguchi, Amit Kapila)

  Windows sometimes returns `ERROR_ACCESS_DENIED` rather than `ERROR_ALREADY_EXISTS` when there is an existing segment. This led to postmaster startup failure due to believing that the former was an unrecoverable error.

- Fix PL/pgSQL to not misbehave with parameters and local variables of type `int2vector` or `oidvector` (Tom Lane)

- Don't try to share SSL contexts across multiple connections in libpq (Heikki Linnakangas)

  This led to assorted corner-case bugs, particularly when trying to use different SSL parameters for different connections.

- Avoid corner-case memory leak in libpq (Tom Lane)

  The reported problem involved leaking an error report during `PQreset()`, but there might be related cases.

- Make ecpg's `--help` and `--version` options work consistently with our other executables (Haribabu Kommi)

- Fix pgbench's calculation of average latency (Fabien Coelho)

  The calculation was incorrect when there were `\sleep` commands in the script, or when the test duration was specified in number of transactions rather than total time.

- In pg_upgrade, check library loadability in name order (Tom Lane)

  This is a workaround to deal with cross-extension dependencies from language transform modules to their base language and data type modules.

- In pg_dump, never dump range constructor functions (Tom Lane)

  This oversight led to pg_upgrade failures with extensions containing range types, due to duplicate creation of the constructor functions.

- In pg_dump with `-C`, suppress `TABLESPACE` clause of `CREATE DATABASE` if `--no-tablespaces` is specified (Tom Lane)

- Make pg_receivexlog work correctly with `--synchronous` without slots (Gabriele Bartolini)

- Disallow specifying both `--source-server` and `--source-target` options to pg_rewind (Michael Banck)

- Make pg_rewind turn off `synchronous_commit` in its session on the source server (Michael Banck, Michael Paquier)

  This allows pg_rewind to work even when the source server is using synchronous replication that is not working for some reason.

- In pg_xlogdump, retry opening new WAL segments when using `--follow` option (Magnus Hagander)

  This allows for a possible delay in the server's creation of the next segment.

- Fix pg_xlogdump to cope with a WAL file that begins with a continuation record spanning more than one page (Pavan Deolasee)

- Fix `contrib/pg_buffercache` to work when `shared_buffers` exceeds 256GB (KaiGai Kohei)

- Fix `contrib/intarray/bench/bench.pl` to print the results of the `EXPLAIN` it does when given the `-e` option (Daniel Gustafsson)

- Support OpenSSL 1.1.0 (Heikki Linnakangas)

- Install TAP test infrastructure so that it's available for extension testing (Craig Ringer)

  When PostgreSQL has been configured with `--enable-tap-tests`, "make install" will now install the Perl support files for TAP testing where PGXS can find them. This allows non-core extensions to use `$(prove_check)` without extra tests.

- In MSVC builds, include pg_recvlogical in a client-only installation (MauMau)

- Update Windows time zone mapping to recognize some time zone names added in recent Windows versions (Michael Paquier)

- Prevent failure of obsolete dynamic time zone abbreviations (Tom Lane)

If a dynamic time zone abbreviation does not match any entry in the referenced time zone, treat it as equivalent to the time zone name. This avoids unexpected failures when IANA removes abbreviations from their time zone database, as they did in tzdata release 2016f and seem likely to do again in the future. The consequences were not limited to not recognizing the individual abbreviation; any mismatch caused the `pg_timezone_abbrevs` view to fail altogether.

- Update time zone data files to tzdata release 2016h for DST law changes in Palestine and Turkey, plus historical corrections for Turkey and some regions of Russia. Switch to numeric abbreviations for some time zones in Antarctica, the former Soviet Union, and Sri Lanka.

  The IANA time zone database previously provided textual abbreviations for all time zones, sometimes making up abbreviations that have little or no currency among the local population. They are in process of reversing that policy in favor of using numeric UTC offsets in zones where there is no evidence of real-world use of an English abbreviation. At least for the time being, PostgreSQL will continue to accept such removed abbreviations for timestamp input. But they will not be shown in the `pg_timezone_names` view nor used for output.

  In this update, `AMT` is no longer shown as being in use to mean Armenia Time. Therefore, we have changed the `Default` abbreviation set to interpret it as Amazon Time, thus UTC-4 not UTC+4.

E.21. Release 9.5.4

Release date: 2016-08-11

This release contains a variety of fixes from 9.5.3. For information about new features in the 9.5 major release, see Section E.25.

E.21.1. Migration to Version 9.5.4

A dump/restore is not required for those running 9.5.X.

However, if you are upgrading from a version earlier than 9.5.2, see Section E.23.

E.21.2. Changes

- Fix possible mis-evaluation of nested `CASE-WHEN` expressions (Heikki Linnakangas, Michael Paquier, Tom Lane)

  A `CASE` expression appearing within the test value subexpression of another `CASE` could become confused about whether its own test value was null or not. Also, inlining of a SQL function implementing the equality operator used by a `CASE` expression could result in passing the wrong test value to functions called within a `CASE` expression in the SQL function's body. If the test values were of different data types, a crash might result; moreover such situations could be abused to allow disclosure of portions of server memory. (CVE-2016-5423)

- Fix client programs' handling of special characters in database and role names (Noah Misch, Nathan Bossart, Michael Paquier)

Numerous places in vacuumdb and other client programs could become confused by database and role names containing double quotes or backslashes. Tighten up quoting rules to make that safe. Also, ensure that when a conninfo string is used as a database name parameter to these programs, it is correctly treated as such throughout.

Fix handling of paired double quotes in psql's `\connect` and `\password` commands to match the documentation.

Introduce a new `-reuse-previous` option in psql's `\connect` command to allow explicit control of whether to re-use connection parameters from a previous connection. (Without this, the choice is based on whether the database name looks like a conninfo string, as before.) This allows secure handling of database names containing special characters in pg_dumpall scripts.

pg_dumpall now refuses to deal with database and role names containing carriage returns or newlines, as it seems impractical to quote those characters safely on Windows. In future we may reject such names on the server side, but that step has not been taken yet.

These are considered security fixes because crafted object names containing special characters could have been used to execute commands with superuser privileges the next time a superuser executes pg_dumpall or other routine maintenance operations. (CVE-2016-5424)

- Fix corner-case misbehaviors for `IS NULL`/`IS NOT NULL` applied to nested composite values (Andrew Gierth, Tom Lane)

  The SQL standard specifies that `IS NULL` should return TRUE for a row of all null values (thus `ROW(NULL,NULL) IS NULL` yields TRUE), but this is not meant to apply recursively (thus `ROW(NULL, ROW(NULL,NULL)) IS NULL` yields FALSE). The core executor got this right, but certain planner optimizations treated the test as recursive (thus producing TRUE in both cases), and `contrib/postgres_fdw` could produce remote queries that misbehaved similarly.

- Fix "unrecognized node type" error for `INSERT ... ON CONFLICT` within a recursive CTE (a `WITH` item) (Peter Geoghegan)

- Fix `INSERT ... ON CONFLICT` to successfully match index expressions or index predicates that are simplified during the planner's expression preprocessing phase (Tom Lane)

- Correctly handle violations of exclusion constraints that apply to the target table of an `INSERT ... ON CONFLICT` command, but are not one of the selected arbiter indexes (Tom Lane)

  Such a case should raise a normal constraint-violation error, but it got into an infinite loop instead.

- Fix `INSERT ... ON CONFLICT` to not fail if the target table has a unique index on OID (Tom Lane)

- Make the `inet` and `cidr` data types properly reject IPv6 addresses with too many colon-separated fields (Tom Lane)

- Prevent crash in `close_ps()` (the `point ## lseg` operator) for NaN input coordinates (Tom Lane)

  Make it return NULL instead of crashing.

- Avoid possible crash in `pg_get_expr()` when inconsistent values are passed to it (Michael Paquier, Thomas Munro)

- Fix several one-byte buffer over-reads in `to_number()` (Peter Eisentraut)

  In several cases the `to_number()` function would read one more character than it should from the input string. There is a small chance of a crash, if the input happens to be adjacent to the end of memory.

- Do not run the planner on the query contained in `CREATE MATERIALIZED VIEW` or `CREATE TABLE AS` when `WITH NO DATA` is specified (Michael Paquier, Tom Lane)

  This avoids some unnecessary failure conditions, for example if a stable function invoked by the materialized view depends on a table that doesn't exist yet.

- Avoid unsafe intermediate state during expensive paths through `heap_update()` (Masahiko Sawada, Andres Freund)

  Previously, these cases locked the target tuple (by setting its XMAX) but did not WAL-log that action, thus risking data integrity problems if the page were spilled to disk and then a database crash occurred before the tuple update could be completed.

- Fix hint bit update during WAL replay of row locking operations (Andres Freund)

  The only known consequence of this problem is that row locks held by a prepared, but uncommitted, transaction might fail to be enforced after a crash and restart.

- Avoid unnecessary "could not serialize access" errors when acquiring `FOR KEY SHARE` row locks in serializable mode (Álvaro Herrera)

- Make sure "expanded" datums returned by a plan node are read-only (Tom Lane)

  This avoids failures in some cases where the result of a lower plan node is referenced in multiple places in upper nodes. So far as core PostgreSQL is concerned, only array values returned by PL/pgSQL functions are at risk; but extensions might use expanded datums for other things.

- Avoid crash in `postgres -C` when the specified variable has a null string value (Michael Paquier)

- Prevent unintended waits for the receiver in WAL sender processes (Kyotaro Horiguchi)

- Fix possible loss of large subtransactions in logical decoding (Petru-Florin Mihancea)

- Fix failure of logical decoding when a subtransaction contains no actual changes (Marko Tiikkaja, Andrew Gierth)

- Ensure that backends see up-to-date statistics for shared catalogs (Tom Lane)

  The statistics collector failed to update the statistics file for shared catalogs after a request from a regular backend. This problem was partially masked because the autovacuum launcher regularly makes requests that did cause such updates; however, it became obvious with autovacuum disabled.

- Avoid redundant writes of the statistics files when multiple backends request updates close together (Tom Lane, Tomas Vondra)

- Avoid consuming a transaction ID during `VACUUM` (Alexander Korotkov)

  Some cases in `VACUUM` unnecessarily caused an XID to be assigned to the current transaction. Normally this is negligible, but if one is up against the XID wraparound limit, consuming more XIDs during anti-wraparound vacuums is a very bad thing.

- Prevent possible failure when vacuuming multixact IDs in an installation that has been pg_upgrade'd from pre-9.3 (Andrew Gierth, Álvaro Herrera)

  The usual symptom of this bug is errors like "MultiXactId *`NNN`* has not been created yet -- apparent wraparound".

- When a manual `ANALYZE` specifies a column list, don't reset the table's `changes_since_analyze` counter (Tom Lane)

If we're only analyzing some columns, we should not prevent routine auto-analyze from happening for the other columns.

- Fix `ANALYZE`'s overestimation of `n_distinct` for a unique or nearly-unique column with many null entries (Tom Lane)

  The nulls could get counted as though they were themselves distinct values, leading to serious planner misestimates in some types of queries.

- Prevent autovacuum from starting multiple workers for the same shared catalog (Álvaro Herrera)

  Normally this isn't much of a problem because the vacuum doesn't take long anyway; but in the case of a severely bloated catalog, it could result in all but one worker uselessly waiting instead of doing useful work on other tables.

- Fix bug in b-tree mark/restore processing (Kevin Grittner)

  This error could lead to incorrect join results or assertion failures in a merge join whose inner source node is a b-tree indexscan.

- Avoid duplicate buffer lock release when abandoning a b-tree index page deletion attempt (Tom Lane)

  This mistake prevented `VACUUM` from completing in some cases involving corrupt b-tree indexes.

- Fix building of large (bigger than `shared_buffers`) hash indexes (Tom Lane)

  The code path used for large indexes contained a bug causing incorrect hash values to be inserted into the index, so that subsequent index searches always failed, except for tuples inserted into the index after the initial build.

- Prevent infinite loop in GiST index build for geometric columns containing NaN component values (Tom Lane)

- Fix possible crash during a nearest-neighbor (`ORDER BY` distance) indexscan on a `contrib/btree_gist` index on an `interval` column (Peter Geoghegan)

- Fix "PANIC: failed to add BRIN tuple" error when attempting to update a BRIN index entry (Álvaro Herrera)

- Fix possible crash during background worker shutdown (Dmitry Ivanov)

- Fix PL/pgSQL's handling of the `INTO` clause within `IMPORT FOREIGN SCHEMA` commands (Tom Lane)

- Fix `contrib/btree_gin` to handle the smallest possible `bigint` value correctly (Peter Eisentraut)

- Teach libpq to correctly decode server version from future servers (Peter Eisentraut)

  It's planned to switch to two-part instead of three-part server version numbers for releases after 9.6. Make sure that `PQserverVersion()` returns the correct value for such cases.

- Fix ecpg's code for `unsigned long long` array elements (Michael Meskes)

- In pg_dump with both `-c` and `-C` options, avoid emitting an unwanted `CREATE SCHEMA public` command (David Johnston, Tom Lane)

- Improve handling of SIGTERM/control-C in parallel pg_dump and pg_restore (Tom Lane)

  Make sure that the worker processes will exit promptly, and also arrange to send query-cancel requests to the connected backends, in case they are doing something long-running such as a `CREATE INDEX`.

- Fix error reporting in parallel pg_dump and pg_restore (Tom Lane)

Previously, errors reported by pg_dump or pg_restore worker processes might never make it to the user's console, because the messages went through the master process, and there were various deadlock scenarios that would prevent the master process from passing on the messages. Instead, just print everything to `stderr`. In some cases this will result in duplicate messages (for instance, if all the workers report a server shutdown), but that seems better than no message.

- Ensure that parallel pg_dump or pg_restore on Windows will shut down properly after an error (Kyotaro Horiguchi)

  Previously, it would report the error, but then just sit until manually stopped by the user.

- Make parallel pg_dump fail cleanly when run against a standby server (Magnus Hagander)

  This usage is not supported unless `--no-synchronized-snapshots` is specified, but the error was not handled very well.

- Make pg_dump behave better when built without zlib support (Kyotaro Horiguchi)

  It didn't work right for parallel dumps, and emitted some rather pointless warnings in other cases.

- Make pg_basebackup accept `-Z 0` as specifying no compression (Fujii Masao)

- Fix makefiles' rule for building AIX shared libraries to be safe for parallel make (Noah Misch)

- Fix TAP tests and MSVC scripts to work when build directory's path name contains spaces (Michael Paquier, Kyotaro Horiguchi)

- Be more predictable about reporting "statement timeout" versus "lock timeout" (Tom Lane)

  On heavily loaded machines, the regression tests sometimes failed due to reporting "lock timeout" even though the statement timeout should have occurred first.

- Make regression tests safe for Danish and Welsh locales (Jeff Janes, Tom Lane)

  Change some test data that triggered the unusual sorting rules of these locales.

- Update our copy of the timezone code to match IANA's tzcode release 2016c (Tom Lane)

  This is needed to cope with anticipated future changes in the time zone data files. It also fixes some corner-case bugs in coping with unusual time zones.

- Update time zone data files to tzdata release 2016f for DST law changes in Kemerovo and Novosibirsk, plus historical corrections for Azerbaijan, Belarus, and Morocco.

E.22. Release 9.5.3

Release date: 2016-05-12

This release contains a variety of fixes from 9.5.2. For information about new features in the 9.5 major release, see Section E.25.

E.22.1. Migration to Version 9.5.3

A dump/restore is not required for those running 9.5.X.

However, if you are upgrading from a version earlier than 9.5.2, see Section E.23.

E.22.2. Changes

- Clear the OpenSSL error queue before OpenSSL calls, rather than assuming it's clear already; and make sure we leave it clear afterwards (Peter Geoghegan, Dave Vitek, Peter Eisentraut)

  This change prevents problems when there are multiple connections using OpenSSL within a single process and not all the code involved follows the same rules for when to clear the error queue. Failures have been reported specifically when a client application uses SSL connections in libpq concurrently with SSL connections using the PHP, Python, or Ruby wrappers for OpenSSL. It's possible for similar problems to arise within the server as well, if an extension module establishes an outgoing SSL connection.

- Fix "failed to build any *N*-way joins" planner error with a full join enclosed in the right-hand side of a left join (Tom Lane)

- Fix incorrect handling of equivalence-class tests in multilevel nestloop plans (Tom Lane)

  Given a three-or-more-way equivalence class of variables, such as `X.X = Y.Y = Z.Z`, it was possible for the planner to omit some of the tests needed to enforce that all the variables are actually equal, leading to join rows being output that didn't satisfy the `WHERE` clauses. For various reasons, erroneous plans were seldom selected in practice, so that this bug has gone undetected for a long time.

- Fix corner-case parser failures occurring when operator_precedence_warning is turned on (Tom Lane)

  An example is that `SELECT (ARRAY[])::text[]` gave an error, though it worked without the parentheses.

- Fix query-lifespan memory leak in GIN index scans (Julien Rouhaud)

- Fix query-lifespan memory leak and potential index corruption hazard in GIN index insertion (Tom Lane)

  The memory leak would typically not amount to much in simple queries, but it could be very substantial during a large GIN index build with high `maintenance_work_mem`.

- Fix possible misbehavior of `TH`, `th`, and `Y,YYY` format codes in `to_timestamp()` (Tom Lane)

  These could advance off the end of the input string, causing subsequent format codes to read garbage.

- Fix dumping of rules and views in which the *`array`* argument of a *`value operator`* `ANY (`*`array`*`)` construct is a sub-SELECT (Tom Lane)

- Disallow newlines in `ALTER SYSTEM` parameter values (Tom Lane)

  The configuration-file parser doesn't support embedded newlines in string literals, so we mustn't allow them in values to be inserted by `ALTER SYSTEM`.

- Fix `ALTER TABLE ... REPLICA IDENTITY USING INDEX` to work properly if an index on OID is selected (David Rowley)

- Avoid possible misbehavior after failing to remove a tablespace symlink (Tom Lane)

- Fix crash in logical decoding on alignment-picky platforms (Tom Lane, Andres Freund)

  The failure occurred only with a transaction large enough to spill to disk and a primary-key change within that transaction.

- Avoid repeated requests for feedback from receiver while shutting down walsender (Nick Cleaton)

- Make pg_regress use a startup timeout from the `PGCTLTIMEOUT` environment variable, if that's set (Tom Lane)

  This is for consistency with a behavior recently added to pg_ctl; it eases automated testing on slow machines.

- Fix pg_upgrade to correctly restore extension membership for operator families containing only one operator class (Tom Lane)

  In such a case, the operator family was restored into the new database, but it was no longer marked as part of the extension. This had no immediate ill effects, but would cause later pg_dump runs to emit output that would cause (harmless) errors on restore.

- Fix pg_upgrade to not fail when new-cluster TOAST rules differ from old (Tom Lane)

  pg_upgrade had special-case code to handle the situation where the new PostgreSQL version thinks that a table should have a TOAST table while the old version did not. That code was broken, so remove it, and instead do nothing in such cases; there seems no reason to believe that we can't get along fine without a TOAST table if that was okay according to the old version's rules.

- Fix atomic operations for PPC when using IBM's XLC compiler (Noah Misch)

- Reduce the number of SysV semaphores used by a build configured with `--disable-spinlocks` (Tom Lane)

- Rename internal function `strtoi()` to `strtoint()` to avoid conflict with a NetBSD library function (Thomas Munro)

- Fix reporting of errors from `bind()` and `listen()` system calls on Windows (Tom Lane)

- Reduce verbosity of compiler output when building with Microsoft Visual Studio (Christian Ullrich)

- Support building with Visual Studio 2015 (Michael Paquier, Petr Jelínek)

  Note that builds made with VS2015 will not run on Windows versions before Windows Vista.

- Fix `putenv()` to work properly with Visual Studio 2013 (Michael Paquier)

- Avoid possibly-unsafe use of Windows' `FormatMessage()` function (Christian Ullrich)

  Use the `FORMAT_MESSAGE_IGNORE_INSERTS` flag where appropriate. No live bug is known to exist here, but it seems like a good idea to be careful.

- Update time zone data files to tzdata release 2016d for DST law changes in Russia and Venezuela. There are new zone names `Europe/Kirov` and `Asia/Tomsk` to reflect the fact that these regions now have different time zone histories from adjacent regions.

E.23. Release 9.5.2

Release date: 2016-03-31

This release contains a variety of fixes from 9.5.1. For information about new features in the 9.5 major release, see Section E.25.

E.23.1. Migration to Version 9.5.2

A dump/restore is not required for those running 9.5.X.

However, you may need to REINDEX some indexes after applying the update, as per the first changelog entry below.

E.23.2. Changes

- Disable abbreviated keys for string sorting in non-C locales (Robert Haas)

  PostgreSQL 9.5 introduced logic for speeding up comparisons of string data types by using the standard C library function strxfrm() as a substitute for strcoll(). It now emerges that most versions of glibc (Linux's implementation of the C library) have buggy implementations of strxfrm() that, in some locales, can produce string comparison results that do not match strcoll(). Until this problem can be better characterized, disable the optimization in all non-C locales. (C locale is safe since it uses neither strcoll() nor strxfrm().)

  Unfortunately, this problem affects not only sorting but also entry ordering in B-tree indexes, which means that B-tree indexes on text, varchar, or char columns may now be corrupt if they sort according to an affected locale and were built or modified under PostgreSQL 9.5.0 or 9.5.1. Users should REINDEX indexes that might be affected.

  It is not possible at this time to give an exhaustive list of known-affected locales. C locale is known safe, and there is no evidence of trouble in English-based locales such as en_US, but some other popular locales such as de_DE are affected in most glibc versions.

- Maintain row-security status properly in cached plans (Stephen Frost)

  In a session that performs queries as more than one role, the plan cache might incorrectly re-use a plan that was generated for another role ID, thus possibly applying the wrong set of policies when row-level security (RLS) is in use. (CVE-2016-2193)

- Add must-be-superuser checks to some new contrib/pageinspect functions (Andreas Seltenreich)

  Most functions in the pageinspect extension that inspect bytea values disallow calls by non-superusers, but brin_page_type() and brin_metapage_info() failed to do so. Passing contrived bytea values to them might crash the server or disclose a few bytes of server memory. Add the missing permissions checks to prevent misuse. (CVE-2016-3065)

- Fix incorrect handling of indexed ROW() comparisons (Simon Riggs)

  Flaws in a minor optimization introduced in 9.5 caused incorrect results if the ROW() comparison matches the index ordering partially but not exactly (for example, differing column order, or the index contains both ASC and DESC columns). Pending a better solution, the optimization has been removed.

- Fix incorrect handling of NULL index entries in indexed ROW() comparisons (Tom Lane)

  An index search using a row comparison such as ROW(a, b) > ROW('x', 'y') would stop upon reaching a NULL entry in the b column, ignoring the fact that there might be non-NULL b values associated with later values of a.

- Avoid unlikely data-loss scenarios due to renaming files without adequate fsync() calls before and after (Michael Paquier, Tomas Vondra, Andres Freund)

- Fix incorrect behavior when rechecking a just-modified row in a query that does `SELECT FOR UPDATE/SHARE` and contains some relations that need not be locked (Tom Lane)

  Rows from non-locked relations were incorrectly treated as containing all NULLs during the recheck, which could result in incorrectly deciding that the updated row no longer passes the `WHERE` condition, or in incorrectly outputting NULLs.

- Fix bug in `json_to_record()` when a field of its input object contains a sub-object with a field name matching one of the requested output column names (Tom Lane)

- Fix nonsense result from two-argument form of `jsonb_object()` when called with empty arrays (Michael Paquier, Andrew Dunstan)

- Fix misbehavior in `jsonb_set()` when converting a path array element into an integer for use as an array subscript (Michael Paquier)

- Fix misformatting of negative time zone offsets by `to_char()`'s `OF` format code (Thomas Munro, Tom Lane)

- Fix possible incorrect logging of waits done by `INSERT ... ON CONFLICT` (Peter Geoghegan)

  Log messages would sometimes claim that the wait was due to an exclusion constraint although no such constraint was responsible.

- Ignore recovery_min_apply_delay parameter until recovery has reached a consistent state (Michael Paquier)

  Previously, standby servers would delay application of WAL records in response to `recovery_min_apply_delay` even while replaying the initial portion of WAL needed to make their database state valid. Since the standby is useless until it's reached a consistent database state, this was deemed unhelpful.

- Correctly handle cases where `pg_subtrans` is close to XID wraparound during server startup (Jeff Janes)

- Fix assorted bugs in logical decoding (Andres Freund)

  Trouble cases included tuples larger than one page when replica identity is `FULL`, `UPDATE`s that change a primary key within a transaction large enough to be spooled to disk, incorrect reports of "subxact logged without previous toplevel record", and incorrect reporting of a transaction's commit time.

- Fix planner error with nested security barrier views when the outer view has a `WHERE` clause containing a correlated subquery (Dean Rasheed)

- Fix memory leak in GIN index searches (Tom Lane)

- Fix corner-case crash due to trying to free `localeconv()` output strings more than once (Tom Lane)

- Fix parsing of affix files for `ispell` dictionaries (Tom Lane)

  The code could go wrong if the affix file contained any characters whose byte length changes during case-folding, for example `I` in Turkish UTF8 locales.

- Avoid use of `sscanf()` to parse `ispell` dictionary files (Artur Zakirov)

  This dodges a portability problem on FreeBSD-derived platforms (including macOS).

- Fix atomic-operations code used on PPC with IBM's xlc compiler (Noah Misch)

  This error led to rare failures of concurrent operations on that platform.

- Avoid a crash on old Windows versions (before 7SP1/2008R2SP1) with an AVX2-capable CPU and a Postgres build done with Visual Studio 2013 (Christian Ullrich)

  This is a workaround for a bug in Visual Studio 2013's runtime library, which Microsoft have stated they will not fix in that version.

- Fix psql's tab completion logic to handle multibyte characters properly (Kyotaro Horiguchi, Robert Haas)

- Fix psql's tab completion for `SECURITY LABEL` (Tom Lane)

  Pressing TAB after `SECURITY LABEL` might cause a crash or offering of inappropriate keywords.

- Make pg_ctl accept a wait timeout from the `PGCTLTIMEOUT` environment variable, if none is specified on the command line (Noah Misch)

  This eases testing of slower buildfarm members by allowing them to globally specify a longer-than-normal timeout for postmaster startup and shutdown.

- Fix incorrect test for Windows service status in pg_ctl (Manuel Mathar)

  The previous set of minor releases attempted to fix pg_ctl to properly determine whether to send log messages to Window's Event Log, but got the test backwards.

- Fix pgbench to correctly handle the combination of `-C` and `-M prepared` options (Tom Lane)

- In pg_upgrade, skip creating a deletion script when the new data directory is inside the old data directory (Bruce Momjian)

  Blind application of the script in such cases would result in loss of the new data directory.

- In PL/Perl, properly translate empty Postgres arrays into empty Perl arrays (Alex Hunsaker)

- Make PL/Python cope with function names that aren't valid Python identifiers (Jim Nasby)

- Fix multiple mistakes in the statistics returned by `contrib/pgstattuple`'s `pgstatindex()` function (Tom Lane)

- Remove dependency on `psed` in MSVC builds, since it's no longer provided by core Perl (Michael Paquier, Andrew Dunstan)

- Update time zone data files to tzdata release 2016c for DST law changes in Azerbaijan, Chile, Haiti, Palestine, and Russia (Altai, Astrakhan, Kirov, Sakhalin, Ulyanovsk regions), plus historical corrections for Lithuania, Moldova, and Russia (Kaliningrad, Samara, Volgograd).

E.24. Release 9.5.1

Release date: 2016-02-11

This release contains a variety of fixes from 9.5.0. For information about new features in the 9.5 major release, see Section E.25.

E.24.1. Migration to Version 9.5.1

A dump/restore is not required for those running 9.5.X.

E.24.2. Changes

- Fix infinite loops and buffer-overrun problems in regular expressions (Tom Lane)

  Very large character ranges in bracket expressions could cause infinite loops in some cases, and memory overwrites in other cases. (CVE-2016-0773)

- Fix an oversight that caused hash joins to miss joining to some tuples of the inner relation in rare cases (Tomas Vondra, Tom Lane)

- Avoid pushdown of `HAVING` clauses when grouping sets are used (Andrew Gierth)

- Fix deparsing of `ON CONFLICT` arbiter `WHERE` clauses (Peter Geoghegan)

- Make `%h` and `%r` escapes in `log_line_prefix` work for messages emitted due to `log_connections` (Tom Lane)

  Previously, `%h`/`%r` started to work just after a new session had emitted the "connection received" log message; now they work for that message too.

- Avoid leaking a token handle during SSPI authentication (Christian Ullrich)

- Fix psql's `\det` command to interpret its pattern argument the same way as other `\d` commands with potentially schema-qualified patterns do (Reece Hart)

- In pg_ctl on Windows, check service status to decide where to send output, rather than checking if standard output is a terminal (Michael Paquier)

- Fix assorted corner-case bugs in pg_dump's processing of extension member objects (Tom Lane)

- Fix improper quoting of domain constraint names in pg_dump (Elvis Pranskevichus)

- Make pg_dump mark a view's triggers as needing to be processed after its rule, to prevent possible failure during parallel pg_restore (Tom Lane)

- Install guards in pgbench against corner-case overflow conditions during evaluation of script-specified division or modulo operators (Fabien Coelho, Michael Paquier)

- Suppress useless warning message when pg_receivexlog connects to a pre-9.4 server (Marco Nenciarini)

- Avoid dump/reload problems when using both plpython2 and plpython3 (Tom Lane)

  In principle, both versions of PL/Python can be used in the same database, though not in the same session (because the two versions of libpython cannot safely be used concurrently). However, pg_restore and pg_upgrade both do things that can fall foul of the same-session restriction. Work around that by changing the timing of the check.

- Fix PL/Python regression tests to pass with Python 3.5 (Peter Eisentraut)

- Prevent certain PL/Java parameters from being set by non-superusers (Noah Misch)

  This change mitigates a PL/Java security bug (CVE-2016-0766), which was fixed in PL/Java by marking these parameters as superuser-only. To fix the security hazard for sites that update PostgreSQL more frequently than PL/Java, make the core code aware of them also.

- Fix ecpg-supplied header files to not contain comments continued from a preprocessor directive line onto the next line (Michael Meskes)

  Such a comment is rejected by ecpg. It's not yet clear whether ecpg itself should be changed.

- Fix `hstore_to_json_loose()`'s test for whether an `hstore` value can be converted to a JSON number (Tom Lane)

  Previously this function could be fooled by non-alphanumeric trailing characters, leading to emitting syntactically-invalid JSON.

- In `contrib/postgres_fdw`, fix bugs triggered by use of `tableoid` in data-modifying commands (Etsuro Fujita, Robert Haas)
- Fix ill-advised restriction of `NAMEDATALEN` to be less than 256 (Robert Haas, Tom Lane)
- Improve reproducibility of build output by ensuring filenames are given to the linker in a fixed order (Christoph Berg)

  This avoids possible bitwise differences in the produced executable files from one build to the next.

- Ensure that `dynloader.h` is included in the installed header files in MSVC builds (Bruce Momjian, Michael Paquier)
- Update time zone data files to tzdata release 2016a for DST law changes in Cayman Islands, Metlakatla, and Trans-Baikal Territory (Zabaykalsky Krai), plus historical corrections for Pakistan.

E.25. Release 9.5

Release date: 2016-01-07

E.25.1. Overview

Major enhancements in PostgreSQL 9.5 include:

- Allow `INSERT`s that would generate constraint conflicts to be turned into `UPDATE`s or ignored
- Add `GROUP BY` analysis features `GROUPING SETS`, `CUBE` and `ROLLUP`
- Add row-level security control
- Create mechanisms for tracking the progress of replication, including methods for identifying the origin of individual changes during logical replication
- Add Block Range Indexes (BRIN)
- Substantial performance improvements for sorting
- Substantial performance improvements for multi-CPU machines

The above items are explained in more detail in the sections below.

E.25.2. Migration to Version 9.5

A dump/restore using pg_dumpall, or use of pg_upgrade, is required for those wishing to migrate data from any previous release.

Version 9.5 contains a number of changes that may affect compatibility with previous releases. Observe the following incompatibilities:

- Adjust operator precedence to match the SQL standard (Tom Lane)

  The precedence of <=, >= and <> has been reduced to match that of <, > and =. The precedence of `IS` tests (e.g., *x* `IS NULL`) has been reduced to be just below these six comparison operators. Also, multi-keyword operators beginning with `NOT` now have the precedence of their base operator (for example, `NOT BETWEEN` now has the same precedence as `BETWEEN`) whereas before they had inconsistent precedence, behaving like `NOT` with respect to their left operand but like their base operator with respect to their right operand. The new configuration parameter operator_precedence_warning can be enabled to warn about queries in which these precedence changes result in different parsing choices.

- Change pg_ctl's default shutdown mode from `smart` to `fast` (Bruce Momjian)

  This means the default behavior will be to forcibly cancel existing database sessions, not simply wait for them to exit.

- Use assignment cast behavior for data type conversions in PL/pgSQL assignments, rather than converting to and from text (Tom Lane)

  This change causes conversions of Booleans to strings to produce `true` or `false`, not `t` or `f`. Other type conversions may succeed in more cases than before; for example, assigning a numeric value `3.9` to an integer variable will now assign 4 rather than failing. If no assignment-grade cast is defined for the particular source and destination types, PL/pgSQL will fall back to its old I/O conversion behavior.

- Allow characters in server command-line options to be escaped with a backslash (Andres Freund)

  Formerly, spaces in the options string always separated options, so there was no way to include a space in an option value. Including a backslash in an option value now requires writing `\\`.

- Change the default value of the GSSAPI `include_realm` parameter to 1, so that by default the realm is not removed from a GSS or SSPI principal name (Stephen Frost)

- Replace configuration parameter `checkpoint_segments` with min_wal_size and max_wal_size (Heikki Linnakangas)

  If you previously adjusted `checkpoint_segments`, the following formula will give you an approximately equivalent setting:

  ```
  max_wal_size = (3 * checkpoint_segments) * 16MB
  ```

  Note that the default setting for `max_wal_size` is much higher than the default `checkpoint_segments` used to be, so adjusting it might no longer be necessary.

- Control the Linux OOM killer via new environment variables `PG_OOM_ADJUST_FILE` and `PG_OOM_ADJUST_VALUE`, instead of compile-time options `LINUX_OOM_SCORE_ADJ` and `LINUX_OOM_ADJ` (Gurjeet Singh)

- Decommission server configuration parameter `ssl_renegotiation_limit`, which was deprecated in earlier releases (Andres Freund)

  While SSL renegotiation is a good idea in theory, it has caused enough bugs to be considered a net negative in practice, and it is due to be removed from future versions of the relevant standards. We have therefore removed support for it from PostgreSQL. The `ssl_renegotiation_limit` parameter still exists, but cannot be set to anything but zero (disabled). It's not documented anymore, either.

- Remove server configuration parameter `autocommit`, which was already deprecated and non-operational (Tom Lane)
- Remove the `pg_authid` catalog's `rolcatupdate` field, as it had no usefulness (Adam Brightwell)
- The `pg_stat_replication` system view's `sent` field is now NULL, not zero, when it has no valid value (Magnus Hagander)
- Allow `json` and `jsonb` array extraction operators to accept negative subscripts, which count from the end of JSON arrays (Peter Geoghegan, Andrew Dunstan)

  Previously, these operators returned `NULL` for negative subscripts.

E.25.3. Changes

Below you will find a detailed account of the changes between PostgreSQL 9.5 and the previous major release.

E.25.3.1. Server

E.25.3.1.1. Indexes

- Add Block Range Indexes (BRIN) (Álvaro Herrera)

  BRIN indexes store only summary data (such as minimum and maximum values) for ranges of heap blocks. They are therefore very compact and cheap to update; but if the data is naturally clustered, they can still provide substantial speedup of searches.
- Allow queries to perform accurate distance filtering of bounding-box-indexed objects (polygons, circles) using GiST indexes (Alexander Korotkov, Heikki Linnakangas)

  Previously, to exploit such an index a subquery had to be used to select a large number of rows ordered by bounding-box distance, and the result then had to be filtered further with a more accurate distance calculation.
- Allow GiST indexes to perform index-only scans (Anastasia Lubennikova, Heikki Linnakangas, Andreas Karlsson)
- Add configuration parameter gin_pending_list_limit to control the size of GIN pending lists (Fujii Masao)

  This value can also be set on a per-index basis as an index storage parameter. Previously the pending-list size was controlled by work_mem, which was awkward because appropriate values for `work_mem` are often much too large for this purpose.
- Issue a warning during the creation of hash indexes because they are not crash-safe (Bruce Momjian)

E.25.3.1.2. General Performance

- Improve the speed of sorting of `varchar`, `text`, and `numeric` fields via "abbreviated" keys (Peter Geoghegan, Andrew Gierth, Robert Haas)

- Extend the infrastructure that allows sorting to be performed by inlined, non-SQL-callable comparison functions to cover `CREATE INDEX`, `REINDEX`, and `CLUSTER` (Peter Geoghegan)
- Improve performance of hash joins (Tomas Vondra, Robert Haas)
- Improve concurrency of shared buffer replacement (Robert Haas, Amit Kapila, Andres Freund)
- Reduce the number of page locks and pins during index scans (Kevin Grittner)

  The primary benefit of this is to allow index vacuums to be blocked less often.
- Make per-backend tracking of buffer pins more memory-efficient (Andres Freund)
- Improve lock scalability (Andres Freund)

  This particularly addresses scalability problems when running on systems with multiple CPU sockets.
- Allow the optimizer to remove unnecessary references to left-joined subqueries (David Rowley)
- Allow pushdown of query restrictions into subqueries with window functions, where appropriate (David Rowley)
- Allow a non-leakproof function to be pushed down into a security barrier view if the function does not receive any view output columns (Dean Rasheed)
- Teach the planner to use statistics obtained from an expression index on a boolean-returning function, when a matching function call appears in `WHERE` (Tom Lane)
- Make `ANALYZE` compute basic statistics (null fraction and average column width) even for columns whose data type lacks an equality function (Oleksandr Shulgin)
- Speed up CRC (cyclic redundancy check) computations and switch to CRC-32C (Abhijit Menon-Sen, Heikki Linnakangas)
- Improve bitmap index scan performance (Teodor Sigaev, Tom Lane)
- Speed up `CREATE INDEX` by avoiding unnecessary memory copies (Robert Haas)
- Increase the number of buffer mapping partitions (Amit Kapila, Andres Freund, Robert Haas)

  This improves performance for highly concurrent workloads.

E.25.3.1.3. Monitoring

- Add per-table autovacuum logging control via new `log_autovacuum_min_duration` storage parameter (Michael Paquier)
- Add new configuration parameter cluster_name (Thomas Munro)

  This string, typically set in `postgresql.conf`, allows clients to identify the cluster. This name also appears in the process title of all server processes, allowing for easier identification of processes belonging to the same cluster.
- Prevent non-superusers from changing log_disconnections on connection startup (Fujii Masao)

E.25.3.1.4. SSL

- Check “Subject Alternative Names” in SSL server certificates, if present (Alexey Klyukin)

When they are present, this replaces checks against the certificate's "Common Name".

- Add system view `pg_stat_ssl` to report SSL connection information (Magnus Hagander)
- Add libpq functions to return SSL information in an implementation-independent way (Heikki Linnakangas)

  While `PQgetssl()` can still be used to call OpenSSL functions, it is now considered deprecated because future versions of libpq might support other SSL implementations. When possible, use the new functions `PQsslAttribute()`, `PQsslAttributeNames()`, and `PQsslInUse()` to obtain SSL information in an SSL-implementation-independent way.
- Make libpq honor any OpenSSL thread callbacks (Jan Urbanski)

  Previously they were overwritten.

E.25.3.1.5. Server Settings

- Replace configuration parameter `checkpoint_segments` with min_wal_size and max_wal_size (Heikki Linnakangas)

  This change allows the allocation of a large number of WAL files without keeping them after they are no longer needed. Therefore the default for `max_wal_size` has been set to `1GB`, much larger than the old default for `checkpoint_segments`. Also note that standby servers perform restartpoints to try to limit their WAL space consumption to `max_wal_size`; previously they did not pay any attention to `checkpoint_segments`.
- Control the Linux OOM killer via new environment variables `PG_OOM_ADJUST_FILE` and `PG_OOM_ADJUST_VALUE` (Gurjeet Singh)

  The previous OOM control infrastructure involved compile-time options `LINUX_OOM_SCORE_ADJ` and `LINUX_OOM_ADJ`, which are no longer supported. The new behavior is available in all builds.
- Allow recording of transaction commit time stamps when configuration parameter track_commit_timestamp is enabled (Álvaro Herrera, Petr Jelínek)

  Time stamp information can be accessed using functions `pg_xact_commit_timestamp()` and `pg_last_committed_xact()`.
- Allow local_preload_libraries to be set by `ALTER ROLE SET` (Peter Eisentraut, Kyotaro Horiguchi)
- Allow autovacuum workers to respond to configuration parameter changes during a run (Michael Paquier)
- Make configuration parameter debug_assertions read-only (Andres Freund)

  This means that assertions can no longer be turned off if they were enabled at compile time, allowing for more efficient code optimization. This change also removes the postgres `-A` option.
- Allow setting effective_io_concurrency on systems where it has no effect (Peter Eisentraut)
- Add system view `pg_file_settings` to show the contents of the server's configuration files (Sawada Masahiko)
- Add `pending_restart` to the system view `pg_settings` to indicate a change has been made but will not take effect until a database restart (Peter Eisentraut)
- Allow `ALTER SYSTEM` values to be reset with `ALTER SYSTEM RESET` (Vik Fearing)

This command removes the specified setting from `postgresql.auto.conf`.

E.25.3.2. Replication and Recovery

- Create mechanisms for tracking the progress of replication, including methods for identifying the origin of individual changes during logical replication (Andres Freund)

  This is helpful when implementing replication solutions.

- Rework truncation of the multixact commit log to be properly WAL-logged (Andres Freund)

  This makes things substantially simpler and more robust.

- Add `recovery.conf` parameter `recovery_target_action` to control post-recovery activity (Petr Jelínek)

  This replaces the old parameter `pause_at_recovery_target`.

- Add new archive_mode value `always` to allow standbys to always archive received WAL files (Fujii Masao)

- Add configuration parameter wal_retrieve_retry_interval to control WAL read retry after failure (Alexey Vasiliev, Michael Paquier)

  This is particularly helpful for warm standbys.

- Allow compression of full-page images stored in WAL (Rahila Syed, Michael Paquier)

  This feature reduces WAL volume, at the cost of more CPU time spent on WAL logging and WAL replay. It is controlled by a new configuration parameter wal_compression, which currently is off by default.

- Archive WAL files with suffix `.partial` during standby promotion (Heikki Linnakangas)

- Add configuration parameter log_replication_commands to log replication commands (Fujii Masao)

  By default, replication commands, e.g., `IDENTIFY_SYSTEM`, are not logged, even when log_statement is set to `all`.

- Report the processes holding replication slots in `pg_replication_slots` (Craig Ringer)

  The new output column is `active_pid`.

- Allow `recovery.conf`'s `primary_conninfo` setting to use connection URIs, e.g., `postgres://` (Alexander Shulgin)

E.25.3.3. Queries

- Allow `INSERT`s that would generate constraint conflicts to be turned into `UPDATE`s or ignored (Peter Geoghegan, Heikki Linnakangas, Andres Freund)

  The syntax is `INSERT ... ON CONFLICT DO NOTHING/UPDATE`. This is the Postgres implementation of the popular `UPSERT` command.

- Add `GROUP BY` analysis features `GROUPING SETS`, `CUBE` and `ROLLUP` (Andrew Gierth, Atri Sharma)

- Allow setting multiple target columns in an `UPDATE` from the result of a single sub-SELECT (Tom Lane)

  This is accomplished using the syntax `UPDATE tab SET (col1, col2, ...) = (SELECT ...)`.

- Add `SELECT` option `SKIP LOCKED` to skip locked rows (Thomas Munro)

  This does not throw an error for locked rows like `NOWAIT` does.

- Add `SELECT` option `TABLESAMPLE` to return a subset of a table (Petr Jelínek)

  This feature supports the SQL-standard table sampling methods. In addition, there are provisions for user-defined table sampling methods.

- Suggest possible matches for mistyped column names (Peter Geoghegan, Robert Haas)

E.25.3.4. Utility Commands

- Add more details about sort ordering in `EXPLAIN` output (Marius Timmer, Lukas Kreft, Arne Scheffer)

  Details include `COLLATE`, `DESC`, `USING`, and `NULLS FIRST/LAST`.

- Make `VACUUM` log the number of pages skipped due to pins (Jim Nasby)
- Make `TRUNCATE` properly update the `pg_stat*` tuple counters (Alexander Shulgin)

E.25.3.4.1. REINDEX

- Allow `REINDEX` to reindex an entire schema using the `SCHEMA` option (Sawada Masahiko)
- Add `VERBOSE` option to `REINDEX` (Sawada Masahiko)
- Prevent `REINDEX DATABASE` and `SCHEMA` from outputting object names, unless `VERBOSE` is used (Simon Riggs)
- Remove obsolete `FORCE` option from `REINDEX` (Fujii Masao)

E.25.3.5. Object Manipulation

- Add row-level security control (Craig Ringer, KaiGai Kohei, Adam Brightwell, Dean Rasheed, Stephen Frost)

  This feature allows row-by-row control over which users can add, modify, or even see rows in a table. This is controlled by new commands `CREATE`/`ALTER`/`DROP POLICY` and `ALTER TABLE ... ENABLE/DISABLE ROW SECURITY`.

- Allow changing of the WAL logging status of a table after creation with `ALTER TABLE ... SET LOGGED / UNLOGGED` (Fabrízio de Royes Mello)
- Add `IF NOT EXISTS` clause to `CREATE TABLE AS`, `CREATE INDEX`, `CREATE SEQUENCE`, and `CREATE MATERIALIZED VIEW` (Fabrízio de Royes Mello)
- Add support for `IF EXISTS` to `ALTER TABLE ... RENAME CONSTRAINT` (Bruce Momjian)

- Allow some DDL commands to accept `CURRENT_USER` or `SESSION_USER`, meaning the current user or session user, in place of a specific user name (Kyotaro Horiguchi, Álvaro Herrera)

  This feature is now supported in ALTER USER, ALTER GROUP, ALTER ROLE, GRANT, and `ALTER *object* OWNER TO` commands.

- Support comments on domain constraints (Álvaro Herrera)
- Reduce lock levels of some create and alter trigger and foreign key commands (Simon Riggs, Andreas Karlsson)
- Allow `LOCK TABLE ... ROW EXCLUSIVE MODE` for those with `INSERT` privileges on the target table (Stephen Frost)

  Previously this command required `UPDATE`, `DELETE`, or `TRUNCATE` privileges.

- Apply table and domain `CHECK` constraints in order by name (Tom Lane)

  The previous ordering was indeterminate.

- Allow `CREATE`/`ALTER DATABASE` to manipulate `datistemplate` and `datallowconn` (Vik Fearing)

  This allows these per-database settings to be changed without manually modifying the `pg_database` system catalog.

E.25.3.5.1. Foreign Tables

- Add support for IMPORT FOREIGN SCHEMA (Ronan Dunklau, Michael Paquier, Tom Lane)

  This command allows automatic creation of local foreign tables that match the structure of existing tables on a remote server.

- Allow `CHECK` constraints to be placed on foreign tables (Shigeru Hanada, Etsuro Fujita)

  Such constraints are assumed to be enforced on the remote server, and are not enforced locally. However, they are assumed to hold for purposes of query optimization, such as constraint exclusion.

- Allow foreign tables to participate in inheritance (Shigeru Hanada, Etsuro Fujita)

  To let this work naturally, foreign tables are now allowed to have check constraints marked as not valid, and to set storage and `OID` characteristics, even though these operations are effectively no-ops for a foreign table.

- Allow foreign data wrappers and custom scans to implement join pushdown (KaiGai Kohei)

E.25.3.5.2. Event Triggers

- Whenever a `ddl_command_end` event trigger is installed, capture details of DDL activity for it to inspect (Álvaro Herrera)

  This information is available through a set-returning function `pg_event_trigger_ddl_commands()`, or by inspection of C data structures if that function doesn't provide enough detail.

- Allow event triggers on table rewrites caused by `ALTER TABLE` (Dimitri Fontaine)

- Add event trigger support for database-level `COMMENT`, `SECURITY LABEL`, and `GRANT`/`REVOKE` (Álvaro Herrera)
- Add columns to the output of `pg_event_trigger_dropped_objects` (Álvaro Herrera)

  This allows simpler processing of delete operations.

E.25.3.6. Data Types

- Allow the `xml` data type to accept empty or all-whitespace content values (Peter Eisentraut)

  This is required by the SQL/XML specification.
- Allow `macaddr` input using the format `xxxx-xxxx-xxxx` (Herwin Weststrate)
- Disallow non-SQL-standard syntax for `interval` with both precision and field specifications (Bruce Momjian)

  Per the standard, such type specifications should be written as, for example, `INTERVAL MINUTE TO SECOND(2)`. PostgreSQL formerly allowed this to be written as `INTERVAL(2) MINUTE TO SECOND`, but it must now be written in the standard way.
- Add selectivity estimators for `inet`/`cidr` operators and improve estimators for text search functions (Emre Hasegeli, Tom Lane)
- Add data types `regrole` and `regnamespace` to simplify entering and pretty-printing the `OID` of a role or namespace (Kyotaro Horiguchi)

E.25.3.6.1. JSON

- Add `jsonb` functions `jsonb_set()` and `jsonb_pretty()` (Dmitry Dolgov, Andrew Dunstan, Petr Jelínek)
- Add `jsonb` generator functions `to_jsonb()`, `jsonb_object()`, `jsonb_build_object()`, `jsonb_build_array()`, `jsonb_agg()`, and `jsonb_object_agg()` (Andrew Dunstan)

  Equivalent functions already existed for type `json`.
- Reduce casting requirements to/from `json` and `jsonb` (Tom Lane)
- Allow `text`, `text` array, and `integer` values to be subtracted from `jsonb` documents (Dmitry Dolgov, Andrew Dunstan)
- Add `jsonb ||` operator (Dmitry Dolgov, Andrew Dunstan)
- Add `json_strip_nulls()` and `jsonb_strip_nulls()` functions to remove JSON null values from documents (Andrew Dunstan)

E.25.3.7. Functions

- Add `generate_series()` for `numeric` values (Plato Malugin)

- Allow `array_agg()` and `ARRAY()` to take arrays as inputs (Ali Akbar, Tom Lane)
- Add functions `array_position()` and `array_positions()` to return subscripts of array values (Pavel Stehule)
- Add a `point`-to-`polygon` distance operator <-> (Alexander Korotkov)
- Allow multibyte characters as escapes in `SIMILAR TO` and `SUBSTRING` (Jeff Davis)

  Previously, only a single-byte character was allowed as an escape.
- Add a `width_bucket()` variant that supports any sortable data type and non-uniform bucket widths (Petr Jelínek)
- Add an optional *`missing_ok`* argument to `pg_read_file()` and related functions (Michael Paquier, Heikki Linnakangas)
- Allow => to specify named parameters in function calls (Pavel Stehule)

  Previously only := could be used. This requires removing the possibility for => to be a user-defined operator. Creation of user-defined => operators has been issuing warnings since PostgreSQL 9.0.
- Add POSIX-compliant rounding for platforms that use PostgreSQL-supplied rounding functions (Pedro Gimeno Fortea)

E.25.3.7.1. System Information Functions and Views

- Add function `pg_get_object_address()` to return `OID`s that uniquely identify an object, and function `pg_identify_object_as_address()` to return object information based on `OID`s (Álvaro Herrera)
- Loosen security checks for viewing queries in `pg_stat_activity`, executing `pg_cancel_backend()`, and executing `pg_terminate_backend()` (Stephen Frost)

  Previously, only the specific role owning the target session could perform these operations; now membership in that role is sufficient.
- Add `pg_stat_get_snapshot_timestamp()` to output the time stamp of the statistics snapshot (Matt Kelly)

  This represents the last time the snapshot file was written to the file system.
- Add `mxid_age()` to compute multi-xid age (Bruce Momjian)

E.25.3.7.2. Aggregates

- Add `min()`/`max()` aggregates for `inet`/`cidr` data types (Haribabu Kommi)
- Use 128-bit integers, where supported, as accumulators for some aggregate functions (Andreas Karlsson)

E.25.3.8. Server-Side Languages

- Improve support for composite types in PL/Python (Ed Behn, Ronan Dunklau)

  This allows PL/Python functions to return arrays of composite types.
- Reduce lossiness of PL/Python floating-point value conversions (Marko Kreen)
- Allow specification of conversion routines between SQL data types and data types of procedural languages (Peter Eisentraut)

  This change adds new commands `CREATE/DROP TRANSFORM`. This also adds optional transformations between the hstore and ltree types to/from PL/Perl and PL/Python.

E.25.3.8.1. PL/pgSQL Server-Side Language

- Improve PL/pgSQL array performance (Tom Lane)
- Add an `ASSERT` statement in PL/pgSQL (Pavel Stehule)
- Allow more PL/pgSQL keywords to be used as identifiers (Tom Lane)

E.25.3.9. Client Applications

- Move pg_archivecleanup, pg_test_fsync, pg_test_timing, and pg_xlogdump from `contrib` to `src/bin` (Peter Eisentraut)

  This should result in these programs being installed by default in most installations.
- Add pg_rewind, which allows re-synchronizing a master server after failback (Heikki Linnakangas)
- Allow pg_receivexlog to manage physical replication slots (Michael Paquier)

  This is controlled via new `--create-slot` and `--drop-slot` options.
- Allow pg_receivexlog to synchronously flush WAL to storage using new `--synchronous` option (Furuya Osamu, Fujii Masao)

  Without this, WAL files are fsync'ed only on close.
- Allow vacuumdb to vacuum in parallel using new `--jobs` option (Dilip Kumar)
- In vacuumdb, do not prompt for the same password repeatedly when multiple connections are necessary (Haribabu Kommi, Michael Paquier)
- Add `--verbose` option to reindexdb (Sawada Masahiko)
- Make pg_basebackup use a tablespace mapping file when using tar format, to support symbolic links and file paths of 100+ characters in length on MS Windows (Amit Kapila)
- Add pg_xlogdump option `--stats` to display summary statistics (Abhijit Menon-Sen)

E.25.3.9.1. psql

- Allow psql to produce AsciiDoc output (Szymon Guz)
- Add an `errors` mode that displays only failed commands to psql's `ECHO` variable (Pavel Stehule)

  This behavior can also be selected with psql's `-b` option.
- Provide separate column, header, and border linestyle control in psql's unicode linestyle (Pavel Stehule)

  Single or double lines are supported; the default is `single`.
- Add new option `%l` in psql's `PROMPT` variables to display the current multiline statement line number (Sawada Masahiko)
- Add `\pset` option `pager_min_lines` to control pager invocation (Andrew Dunstan)
- Improve psql line counting used when deciding to invoke the pager (Andrew Dunstan)
- psql now fails if the file specified by an `--output` or `--log-file` switch cannot be written (Tom Lane, Daniel Vérité)

  Previously, it effectively ignored the switch in such cases.
- Add psql tab completion when setting the search_path variable (Jeff Janes)

  Currently only the first schema can be tab-completed.
- Improve psql's tab completion for triggers and rules (Andreas Karlsson)

E.25.3.9.1.1. Backslash Commands

- Add psql `\?` help sections `variables` and `options` (Pavel Stehule)

  `\? variables` shows psql's special variables and `\? options` shows the command-line options. `\? commands` shows the meta-commands, which is the traditional output and remains the default. These help displays can also be obtained with the command-line option `--help=`*`section`*.
- Show tablespace size in psql's `\db+` (Fabrízio de Royes Mello)
- Show data type owners in psql's `\dT+` (Magnus Hagander)
- Allow psql's `\watch` to output `\timing` information (Fujii Masao)

  Also prevent `--echo-hidden` from echoing `\watch` queries, since that is generally unwanted.
- Make psql's `\sf` and `\ef` commands honor `ECHO_HIDDEN` (Andrew Dunstan)
- Improve psql tab completion for `\set`, `\unset`, and `:variable` names (Pavel Stehule)
- Allow tab completion of role names in psql `\c` commands (Ian Barwick)

E.25.3.9.2. pg_dump

- Allow pg_dump to share a snapshot taken by another session using `--snapshot` (Simon Riggs, Michael Paquier)

The remote snapshot must have been exported by `pg_export_snapshot()` or logical replication slot creation. This can be used to share a consistent snapshot across multiple pg_dump processes.

- Support table sizes exceeding 8GB in tar archive format (Tom Lane)

  The POSIX standard for tar format does not allow elements of a tar archive to exceed 8GB, but most modern implementations of tar support an extension that does allow it. Use the extension format when necessary, rather than failing.

- Make pg_dump always print the server and pg_dump versions (Jing Wang)

  Previously, version information was only printed in `--verbose` mode.

- Remove the long-ignored `-i`/`--ignore-version` option from pg_dump, pg_dumpall, and pg_restore (Fujii Masao)

E.25.3.9.3. pg_ctl

- Support multiple pg_ctl `-o` options, concatenating their values (Bruce Momjian)
- Allow control of pg_ctl's event source logging on MS Windows (MauMau)

  This only controls pg_ctl, not the server, which has separate settings in `postgresql.conf`.

- If the server's listen address is set to a wildcard value (`0.0.0.0` in IPv4 or `::` in IPv6), connect via the loopback address rather than trying to use the wildcard address literally (Kondo Yuta)

  This fix primarily affects Windows, since on other platforms pg_ctl will prefer to use a Unix-domain socket.

E.25.3.9.4. pg_upgrade

- Move pg_upgrade from `contrib` to `src/bin` (Peter Eisentraut)

  In connection with this change, the functionality previously provided by the pg_upgrade_support module has been moved into the core server.

- Support multiple pg_upgrade `-o`/`-O` options, concatenating their values (Bruce Momjian)
- Improve database collation comparisons in pg_upgrade (Heikki Linnakangas)
- Remove support for upgrading from 8.3 clusters (Bruce Momjian)

E.25.3.9.5. pgbench

- Move pgbench from `contrib` to `src/bin` (Peter Eisentraut)
- Fix calculation of TPS number "excluding connections establishing" (Tatsuo Ishii, Fabien Coelho)

  The overhead for connection establishment was miscalculated whenever the number of pgbench threads was less than the number of client connections. Although this is clearly a bug, we won't back-patch it into pre-9.5 branches since it makes TPS numbers not comparable to previous results.

- Allow counting of pgbench transactions that take over a specified amount of time (Fabien Coelho)

This is controlled by a new `--latency-limit` option.

- Allow pgbench to generate Gaussian/exponential distributions using `\setrandom` (Kondo Mitsumasa, Fabien Coelho)
- Allow pgbench's `\set` command to handle arithmetic expressions containing more than one operator, and add `%` (modulo) to the set of operators it supports (Robert Haas, Fabien Coelho)

E.25.3.10. Source Code

- Simplify WAL record format (Heikki Linnakangas)

  This allows external tools to more easily track what blocks are modified.
- Improve the representation of transaction commit and abort WAL records (Andres Freund)
- Add atomic memory operations API (Andres Freund)
- Allow custom path and scan methods (KaiGai Kohei, Tom Lane)

  This allows extensions greater control over the optimizer and executor.
- Allow foreign data wrappers to do post-filter locking (Etsuro Fujita)
- Foreign tables can now take part in `INSERT ... ON CONFLICT DO NOTHING` queries (Peter Geoghegan, Heikki Linnakangas, Andres Freund)

  Foreign data wrappers must be modified to handle this. `INSERT ... ON CONFLICT DO UPDATE` is not supported on foreign tables.
- Improve `hash_create()`'s API for selecting simple-binary-key hash functions (Teodor Sigaev, Tom Lane)
- Improve parallel execution infrastructure (Robert Haas, Amit Kapila, Noah Misch, Rushabh Lathia, Jeevan Chalke)
- Remove Alpha (CPU) and Tru64 (OS) ports (Andres Freund)
- Remove swap-byte-based spinlock implementation for ARMv5 and earlier CPUs (Robert Haas)

  ARMv5's weak memory ordering made this locking implementation unsafe. Spinlock support is still possible on newer gcc implementations with atomics support.
- Generate an error when excessively long (100+ character) file paths are written to tar files (Peter Eisentraut)

  Tar does not support such overly-long paths.
- Change index operator class for columns `pg_seclabel.provider` and `pg_shseclabel.provider` to be `text_pattern_ops` (Tom Lane)

  This avoids possible problems with these indexes when different databases of a cluster have different default collations.
- Change the spinlock primitives to function as compiler barriers (Robert Haas)

E.25.3.10.1. MS Windows

- Allow higher-precision time stamp resolution on Windows 8, Windows Server 2012, and later Windows systems (Craig Ringer)
- Install shared libraries to `bin` in MS Windows (Peter Eisentraut, Michael Paquier)
- Install `src/test/modules` together with `contrib` on MSVC builds (Michael Paquier)
- Allow configure's `--with-extra-version` option to be honored by the MSVC build (Michael Paquier)
- Pass `PGFILEDESC` into MSVC contrib builds (Michael Paquier)
- Add icons to all MSVC-built binaries and version information to all MS Windows binaries (Noah Misch)

  MinGW already had such icons.
- Add optional-argument support to the internal `getopt_long()` implementation (Michael Paquier, Andres Freund)

  This is used by the MSVC build.

E.25.3.11. Additional Modules

- Add statistics for minimum, maximum, mean, and standard deviation times to pg_stat_statements (Mitsumasa Kondo, Andrew Dunstan)
- Add pgcrypto function `pgp_armor_headers()` to extract PGP armor headers (Marko Tiikkaja, Heikki Linnakangas)
- Allow empty replacement strings in unaccent (Mohammad Alhashash)

  This is useful in languages where diacritic signs are represented as separate characters.
- Allow multicharacter source strings in unaccent (Tom Lane)

  This could be useful in languages where diacritic signs are represented as separate characters. It also allows more complex unaccent dictionaries.
- Add `contrib` modules tsm_system_rows and tsm_system_time to allow additional table sampling methods (Petr Jelínek)
- Add GIN index inspection functions to pageinspect (Heikki Linnakangas, Peter Geoghegan, Michael Paquier)
- Add information about buffer pins to pg_buffercache display (Andres Freund)
- Allow pgstattuple to report approximate answers with less overhead using `pgstattuple_approx()` (Abhijit Menon-Sen)
- Move dummy_seclabel, test_shm_mq, test_parser, and worker_spi from `contrib` to `src/test/modules` (Álvaro Herrera)

  These modules are only meant for server testing, so they do not need to be built or installed when packaging PostgreSQL.

E.26. Prior Releases

Release notes for prior release branches can be found at `https://www.postgresql.org/docs/release/`

Appendix F. Additional Supplied Modules

This appendix and the next one contain information regarding the modules that can be found in the `contrib` directory of the PostgreSQL distribution. These include porting tools, analysis utilities, and plug-in features that are not part of the core PostgreSQL system, mainly because they address a limited audience or are too experimental to be part of the main source tree. This does not preclude their usefulness.

This appendix covers extensions and other server plug-in modules found in `contrib`. Appendix G covers utility programs.

When building from the source distribution, these components are not built automatically, unless you build the "world" target (see step 2). You can build and install all of them by running:

```
make
make install
```

in the `contrib` directory of a configured source tree; or to build and install just one selected module, do the same in that module's subdirectory. Many of the modules have regression tests, which can be executed by running:

```
make check
```

before installation or

```
make installcheck
```

once you have a PostgreSQL server running.

If you are using a pre-packaged version of PostgreSQL, these modules are typically made available as a separate subpackage, such as `postgresql-contrib`.

Many modules supply new user-defined functions, operators, or types. To make use of one of these modules, after you have installed the code you need to register the new SQL objects in the database system. In PostgreSQL 9.1 and later, this is done by executing a CREATE EXTENSION command. In a fresh database, you can simply do

```
CREATE EXTENSION module_name;
```

This command must be run by a database superuser. This registers the new SQL objects in the current database only, so you need to run this command in each database that you want the module's facilities to be available in. Alternatively, run it in database `template1` so that the extension will be copied into subsequently-created databases by default.

Many modules allow you to install their objects in a schema of your choice. To do that, add `SCHEMA` *`schema_name`* to the `CREATE EXTENSION` command. By default, the objects will be placed in your current creation target schema, which in turn defaults to `public`.

If your database was brought forward by dump and reload from a pre-9.1 version of PostgreSQL, and you had been using the pre-9.1 version of the module in it, you should instead do

```
CREATE EXTENSION module_name FROM unpackaged;
```

This will update the pre-9.1 objects of the module into a proper *extension* object. Future updates to the module will be managed by ALTER EXTENSION. For more information about extension updates, see Section 35.15.

Note, however, that some of these modules are not "extensions" in this sense, but are loaded into the server in some other way, for instance by way of shared_preload_libraries. See the documentation of each module for details.

F.1. adminpack

`adminpack` provides a number of support functions which pgAdmin and other administration and management tools can use to provide additional functionality, such as remote management of server log files. Use of all these functions is restricted to superusers.

The functions shown in Table F-1 provide write access to files on the machine hosting the server. (See also the functions in Table 9-76, which provide read-only access.) Only files within the database cluster directory can be accessed, but either a relative or absolute path is allowable.

Table F-1. `adminpack` Functions

Name	Return Type	Description
`pg_catalog.pg_file_write(filename text, data text, append boolean)`	`bigint`	Write, or append to, a text file
`pg_catalog.pg_file_rename(oldname text, newname text [, archivename text])`	`boolean`	Rename a file
`pg_catalog.pg_file_unlink(filename text)`	`boolean`	Remove a file
`pg_catalog.pg_logdir_ls()`	`setof record`	List the log files in the `log_directory` directory

`pg_file_write` writes the specified `data` into the file named by `filename`. If `append` is false, the file must not already exist. If `append` is true, the file can already exist, and will be appended to if so. Returns the number of bytes written.

`pg_file_rename` renames a file. If `archivename` is omitted or NULL, it simply renames `oldname` to `newname` (which must not already exist). If `archivename` is provided, it first renames `newname` to `archivename` (which must not already exist), and then renames `oldname` to `newname`. In event of failure of the second rename step, it will try to rename `archivename` back to `newname` before reporting the error. Returns true on success, false if the source file(s) are not present or not writable; other cases throw errors.

`pg_file_unlink` removes the specified file. Returns true on success, false if the specified file is not present or the `unlink()` call fails; other cases throw errors.

`pg_logdir_ls` returns the start timestamps and path names of all the log files in the log_directory directory. The log_filename parameter must have its default setting (`postgresql-%Y-%m-%d_%H%M%S.log`) to use this function.

The functions shown in Table F-2 are deprecated and should not be used in new applications; instead

use those shown in Table 9-67 and Table 9-76. These functions are provided in `adminpack` only for compatibility with old versions of pgAdmin.

Table F-2. Deprecated `adminpack` Functions

Name	Return Type	Description
`pg_catalog.pg_file_read(filename text, offset bigint, nbytes bigint)`	`text`	Alternate name for `pg_read_file()`
`pg_catalog.pg_file_length(filename text)`	`bigint`	Same as `size` column returned by `pg_stat_file()`
`pg_catalog.pg_logfile_rotate()`	`integer`	Alternate name for `pg_rotate_logfile()`, but note that it returns integer 0 or 1 rather than boolean

F.2. auth_delay

`auth_delay` causes the server to pause briefly before reporting authentication failure, to make brute-force attacks on database passwords more difficult. Note that it does nothing to prevent denial-of-service attacks, and may even exacerbate them, since processes that are waiting before reporting authentication failure will still consume connection slots.

In order to function, this module must be loaded via shared_preload_libraries in `postgresql.conf`.

F.2.1. Configuration Parameters

`auth_delay.milliseconds` (`int`)

The number of milliseconds to wait before reporting an authentication failure. The default is 0.

These parameters must be set in `postgresql.conf`. Typical usage might be:

```
# postgresql.conf
shared_preload_libraries = 'auth_delay'

auth_delay.milliseconds = '500'
```

F.2.2. Author

KaiGai Kohei <`kaigai@ak.jp.nec.com`>

F.3. auto_explain

The auto_explain module provides a means for logging execution plans of slow statements automatically, without having to run EXPLAIN by hand. This is especially helpful for tracking down un-optimized queries in large applications.

The module provides no SQL-accessible functions. To use it, simply load it into the server. You can load it into an individual session:

```
LOAD 'auto_explain';
```

(You must be superuser to do that.) More typical usage is to preload it into some or all sessions by including auto_explain in session_preload_libraries or shared_preload_libraries in postgresql.conf. Then you can track unexpectedly slow queries no matter when they happen. Of course there is a price in overhead for that.

F.3.1. Configuration Parameters

There are several configuration parameters that control the behavior of auto_explain. Note that the default behavior is to do nothing, so you must set at least auto_explain.log_min_duration if you want any results.

auto_explain.log_min_duration (integer)

auto_explain.log_min_duration is the minimum statement execution time, in milliseconds, that will cause the statement's plan to be logged. Setting this to zero logs all plans. Minus-one (the default) disables logging of plans. For example, if you set it to 250ms then all statements that run 250ms or longer will be logged. Only superusers can change this setting.

auto_explain.log_analyze (boolean)

auto_explain.log_analyze causes EXPLAIN ANALYZE output, rather than just EXPLAIN output, to be printed when an execution plan is logged. This parameter is off by default. Only superusers can change this setting.

> **Note:** When this parameter is on, per-plan-node timing occurs for all statements executed, whether or not they run long enough to actually get logged. This can have an extremely negative impact on performance. Turning off auto_explain.log_timing ameliorates the performance cost, at the price of obtaining less information.

auto_explain.log_buffers (boolean)

auto_explain.log_buffers controls whether buffer usage statistics are printed when an execution plan is logged; it's equivalent to the BUFFERS option of EXPLAIN. This parameter has no effect unless auto_explain.log_analyze is enabled. This parameter is off by default. Only superusers can change this setting.

auto_explain.log_timing (boolean)

auto_explain.log_timing controls whether per-node timing information is printed when an execution plan is logged; it's equivalent to the TIMING option of EXPLAIN. The overhead of repeatedly

reading the system clock can slow down queries significantly on some systems, so it may be useful to set this parameter to off when only actual row counts, and not exact times, are needed. This parameter has no effect unless `auto_explain.log_analyze` is enabled. This parameter is on by default. Only superusers can change this setting.

`auto_explain.log_triggers` (`boolean`)

`auto_explain.log_triggers` causes trigger execution statistics to be included when an execution plan is logged. This parameter has no effect unless `auto_explain.log_analyze` is enabled. This parameter is off by default. Only superusers can change this setting.

`auto_explain.log_verbose` (`boolean`)

`auto_explain.log_verbose` controls whether verbose details are printed when an execution plan is logged; it's equivalent to the `VERBOSE` option of `EXPLAIN`. This parameter is off by default. Only superusers can change this setting.

`auto_explain.log_format` (`enum`)

`auto_explain.log_format` selects the `EXPLAIN` output format to be used. The allowed values are `text`, `xml`, `json`, and `yaml`. The default is text. Only superusers can change this setting.

`auto_explain.log_nested_statements` (`boolean`)

`auto_explain.log_nested_statements` causes nested statements (statements executed inside a function) to be considered for logging. When it is off, only top-level query plans are logged. This parameter is off by default. Only superusers can change this setting.

In ordinary usage, these parameters are set in `postgresql.conf`, although superusers can alter them on-the-fly within their own sessions. Typical usage might be:

```
# postgresql.conf
session_preload_libraries = 'auto_explain'

auto_explain.log_min_duration = '3s'
```

F.3.2. Example

```
postgres=# LOAD 'auto_explain';
postgres=# SET auto_explain.log_min_duration = 0;
postgres=# SET auto_explain.log_analyze = true;
postgres=# SELECT count(*)
           FROM pg_class, pg_index
           WHERE oid = indrelid AND indisunique;
```

This might produce log output such as:

```
LOG:  duration: 3.651 ms  plan:
  Query Text: SELECT count(*)
              FROM pg_class, pg_index
              WHERE oid = indrelid AND indisunique;
  Aggregate  (cost=16.79..16.80 rows=1 width=0) (actual time=3.626..3.627 rows=1 loops=1)
    ->  Hash Join  (cost=4.17..16.55 rows=92 width=0) (actual time=3.349..3.594 rows=92 loo
          Hash Cond: (pg_class.oid = pg_index.indrelid)
```

```
->  Seq Scan on pg_class  (cost=0.00..9.55 rows=255 width=4) (actual time=0.016..
->  Hash  (cost=3.02..3.02 rows=92 width=4) (actual time=3.238..3.238 rows=92 loo
      Buckets: 1024  Batches: 1  Memory Usage: 4kB
      ->  Seq Scan on pg_index  (cost=0.00..3.02 rows=92 width=4) (actual time=0.
            Filter: indisunique
```

F.3.3. Author

Takahiro Itagaki <itagaki.takahiro@oss.ntt.co.jp>

F.4. btree_gin

btree_gin provides sample GIN operator classes that implement B-tree equivalent behavior for the data types int2, int4, int8, float4, float8, timestamp with time zone, timestamp without time zone, time with time zone, time without time zone, date, interval, oid, money, "char", varchar, text, bytea, bit, varbit, macaddr, inet, and cidr.

In general, these operator classes will not outperform the equivalent standard B-tree index methods, and they lack one major feature of the standard B-tree code: the ability to enforce uniqueness. However, they are useful for GIN testing and as a base for developing other GIN operator classes. Also, for queries that test both a GIN-indexable column and a B-tree-indexable column, it might be more efficient to create a multicolumn GIN index that uses one of these operator classes than to create two separate indexes that would have to be combined via bitmap ANDing.

F.4.1. Example Usage

```
CREATE TABLE test (a int4);
-- create index
CREATE INDEX testidx ON test USING GIN (a);
-- query
SELECT * FROM test WHERE a < 10;
```

F.4.2. Authors

Teodor Sigaev (<teodor@stack.net>) and Oleg Bartunov (<oleg@sai.msu.su>). See http://www.sai.msu.su/~megera/oddmuse/index.cgi/Gin for additional information.

F.5. btree_gist

btree_gist provides GiST index operator classes that implement B-tree equivalent behavior for the data types int2, int4, int8, float4, float8, numeric, timestamp with time zone, timestamp

`without time zone`, `time with time zone`, `time without time zone`, `date`, `interval`, `oid`, `money`, `char`, `varchar`, `text`, `bytea`, `bit`, `varbit`, `macaddr`, `inet`, and `cidr`.

In general, these operator classes will not outperform the equivalent standard B-tree index methods, and they lack one major feature of the standard B-tree code: the ability to enforce uniqueness. However, they provide some other features that are not available with a B-tree index, as described below. Also, these operator classes are useful when a multicolumn GiST index is needed, wherein some of the columns are of data types that are only indexable with GiST but other columns are just simple data types. Lastly, these operator classes are useful for GiST testing and as a base for developing other GiST operator classes.

In addition to the typical B-tree search operators, `btree_gist` also provides index support for <> ("not equals"). This may be useful in combination with an exclusion constraint, as described below.

Also, for data types for which there is a natural distance metric, `btree_gist` defines a distance operator <->, and provides GiST index support for nearest-neighbor searches using this operator. Distance operators are provided for `int2`, `int4`, `int8`, `float4`, `float8`, `timestamp with time zone`, `timestamp without time zone`, `time without time zone`, `date`, `interval`, `oid`, and `money`.

F.5.1. Example Usage

Simple example using `btree_gist` instead of `btree`:

```
CREATE TABLE test (a int4);
-- create index
CREATE INDEX testidx ON test USING GIST (a);
-- query
SELECT * FROM test WHERE a < 10;
-- nearest-neighbor search: find the ten entries closest to "42"
SELECT *, a <-> 42 AS dist FROM test ORDER BY a <-> 42 LIMIT 10;
```

Use an exclusion constraint to enforce the rule that a cage at a zoo can contain only one kind of animal:

```
=> CREATE TABLE zoo (
  cage   INTEGER,
  animal TEXT,
  EXCLUDE USING GIST (cage WITH =, animal WITH <>)
);

=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'zebra');
INSERT 0 1
=> INSERT INTO zoo VALUES(123, 'lion');
ERROR:  conflicting key value violates exclusion constraint "zoo_cage_animal_excl"
DETAIL:  Key (cage, animal)=(123, lion) conflicts with existing key (cage, animal)=(123, ze
=> INSERT INTO zoo VALUES(124, 'lion');
INSERT 0 1
```

F.5.2. Authors

Teodor Sigaev (<teodor@stack.net>), Oleg Bartunov (<oleg@sai.msu.su>), and Janko Richter (<jankorichter@yahoo.de>). See http://www.sai.msu.su/~megera/postgres/gist/ for additional information.

F.6. chkpass

This module implements a data type `chkpass` that is designed for storing encrypted passwords. Each password is automatically converted to encrypted form upon entry, and is always stored encrypted. To compare, simply compare against a clear text password and the comparison function will encrypt it before comparing.

There are provisions in the code to report an error if the password is determined to be easily crackable. However, this is currently just a stub that does nothing.

If you precede an input string with a colon, it is assumed to be an already-encrypted password, and is stored without further encryption. This allows entry of previously-encrypted passwords.

On output, a colon is prepended. This makes it possible to dump and reload passwords without re-encrypting them. If you want the encrypted password without the colon then use the `raw()` function. This allows you to use the type with things like Apache's `Auth_PostgreSQL` module.

The encryption uses the standard Unix function `crypt()`, and so it suffers from all the usual limitations of that function; notably that only the first eight characters of a password are considered.

Note that the `chkpass` data type is not indexable.

Sample usage:

```
test=# create table test (p chkpass);
CREATE TABLE
test=# insert into test values ('hello');
INSERT 0 1
test=# select * from test;
       p
----------------
 :dVGkpXdOrE3ko
(1 row)

test=# select raw(p) from test;
      raw
---------------
 dVGkpXdOrE3ko
(1 row)

test=# select p = 'hello' from test;
 ?column?
----------
 t
(1 row)
```

```
test=# select p = 'goodbye' from test;
 ?column?
----------
 f
(1 row)
```

F.6.1. Author

D'Arcy J.M. Cain (<darcy@druid.net>)

F.7. citext

The citext module provides a case-insensitive character string type, citext. Essentially, it internally calls lower when comparing values. Otherwise, it behaves almost exactly like text.

F.7.1. Rationale

The standard approach to doing case-insensitive matches in PostgreSQL has been to use the lower function when comparing values, for example

```
SELECT * FROM tab WHERE lower(col) = LOWER(?);
```

This works reasonably well, but has a number of drawbacks:

- It makes your SQL statements verbose, and you always have to remember to use lower on both the column and the query value.
- It won't use an index, unless you create a functional index using lower.
- If you declare a column as UNIQUE or PRIMARY KEY, the implicitly generated index is case-sensitive. So it's useless for case-insensitive searches, and it won't enforce uniqueness case-insensitively.

The citext data type allows you to eliminate calls to lower in SQL queries, and allows a primary key to be case-insensitive. citext is locale-aware, just like text, which means that the matching of upper case and lower case characters is dependent on the rules of the database's LC_CTYPE setting. Again, this behavior is identical to the use of lower in queries. But because it's done transparently by the data type, you don't have to remember to do anything special in your queries.

F.7.2. How to Use It

Here's a simple example of usage:

```
CREATE TABLE users (
    nick CITEXT PRIMARY KEY,
    pass TEXT   NOT NULL
```

```
);

INSERT INTO users VALUES ( 'larry',  md5(random()::text) );
INSERT INTO users VALUES ( 'Tom',    md5(random()::text) );
INSERT INTO users VALUES ( 'Damian', md5(random()::text) );
INSERT INTO users VALUES ( 'NEAL',   md5(random()::text) );
INSERT INTO users VALUES ( 'Bjørn',  md5(random()::text) );

SELECT * FROM users WHERE nick = 'Larry';
```

The `SELECT` statement will return one tuple, even though the `nick` column was set to `larry` and the query was for `Larry`.

F.7.3. String Comparison Behavior

`citext` performs comparisons by converting each string to lower case (as though `lower` were called) and then comparing the results normally. Thus, for example, two strings are considered equal if `lower` would produce identical results for them.

In order to emulate a case-insensitive collation as closely as possible, there are `citext`-specific versions of a number of string-processing operators and functions. So, for example, the regular expression operators `~` and `~*` exhibit the same behavior when applied to `citext`: they both match case-insensitively. The same is true for `!~` and `!~*`, as well as for the `LIKE` operators `~~` and `~~*`, and `!~~` and `!~~*`. If you'd like to match case-sensitively, you can cast the operator's arguments to `text`.

Similarly, all of the following functions perform matching case-insensitively if their arguments are `citext`:

- `regexp_matches()`
- `regexp_replace()`
- `regexp_split_to_array()`
- `regexp_split_to_table()`
- `replace()`
- `split_part()`
- `strpos()`
- `translate()`

For the regexp functions, if you want to match case-sensitively, you can specify the "c" flag to force a case-sensitive match. Otherwise, you must cast to `text` before using one of these functions if you want case-sensitive behavior.

F.7.4. Limitations

- `citext`'s case-folding behavior depends on the `LC_CTYPE` setting of your database. How it compares values is therefore determined when the database is created. It is not truly case-insensitive in the terms

defined by the Unicode standard. Effectively, what this means is that, as long as you're happy with your collation, you should be happy with `citext`'s comparisons. But if you have data in different languages stored in your database, users of one language may find their query results are not as expected if the collation is for another language.

- As of PostgreSQL 9.1, you can attach a `COLLATE` specification to `citext` columns or data values. Currently, `citext` operators will honor a non-default `COLLATE` specification while comparing case-folded strings, but the initial folding to lower case is always done according to the database's `LC_CTYPE` setting (that is, as though `COLLATE "default"` were given). This may be changed in a future release so that both steps follow the input `COLLATE` specification.
- `citext` is not as efficient as `text` because the operator functions and the B-tree comparison functions must make copies of the data and convert it to lower case for comparisons. It is, however, slightly more efficient than using `lower` to get case-insensitive matching.
- `citext` doesn't help much if you need data to compare case-sensitively in some contexts and case-insensitively in other contexts. The standard answer is to use the `text` type and manually use the `lower` function when you need to compare case-insensitively; this works all right if case-insensitive comparison is needed only infrequently. If you need case-insensitive behavior most of the time and case-sensitive infrequently, consider storing the data as `citext` and explicitly casting the column to `text` when you want case-sensitive comparison. In either situation, you will need two indexes if you want both types of searches to be fast.
- The schema containing the `citext` operators must be in the current `search_path` (typically `public`); if it is not, the normal case-sensitive `text` operators will be invoked instead.

F.7.5. Author

David E. Wheeler <david@kineticode.com>

Inspired by the original `citext` module by Donald Fraser.

F.8. cube

This module implements a data type `cube` for representing multidimensional cubes.

F.8.1. Syntax

Table F-3 shows the valid external representations for the `cube` type. *x*, *y*, etc. denote floating-point numbers.

Table F-3. Cube External Representations

x	A one-dimensional point (or, zero-length one-dimensional interval)
(*x*)	Same as above

x1,x2,...,xn	A point in n-dimensional space, represented internally as a zero-volume cube
(x1,x2,...,xn)	Same as above
(x),(y)	A one-dimensional interval starting at x and ending at y or vice versa; the order does not matter
[(x),(y)]	Same as above
(x1,...,xn),(y1,...,yn)	An n-dimensional cube represented by a pair of its diagonally opposite corners
[(x1,...,xn),(y1,...,yn)]	Same as above

It does not matter which order the opposite corners of a cube are entered in. The cube functions automatically swap values if needed to create a uniform "lower left — upper right" internal representation.

White space is ignored, so [(x),(y)] is the same as [(x), (y)].

F.8.2. Precision

Values are stored internally as 64-bit floating point numbers. This means that numbers with more than about 16 significant digits will be truncated.

F.8.3. Usage

The cube module includes a GiST index operator class for cube values. The operators supported by the GiST operator class are shown in Table F-4.

Table F-4. Cube GiST Operators

Operator	Description
a = b	The cubes a and b are identical.
a && b	The cubes a and b overlap.
a @> b	The cube a contains the cube b.
a <@ b	The cube a is contained in the cube b.

(Before PostgreSQL 8.2, the containment operators @> and <@ were respectively called @ and ~. These names are still available, but are deprecated and will eventually be retired. Notice that the old names are reversed from the convention formerly followed by the core geometric data types!)

The standard B-tree operators are also provided, for example

Operator	Description
[a, b] < [c, d]	Less than
[a, b] > [c, d]	Greater than

These operators do not make a lot of sense for any practical purpose but sorting. These operators first compare (a) to (c), and if these are equal, compare (b) to (d). That results in reasonably good sorting in most cases, which is useful if you want to use ORDER BY with this type.

Table F-5 shows the available functions.

Table F-5. Cube Functions

`cube(float8) returns cube`	Makes a one dimensional cube with both coordinates the same. `cube(1) == '(1)'`
`cube(float8, float8) returns cube`	Makes a one dimensional cube. `cube(1,2) == '(1),(2)'`
`cube(float8[]) returns cube`	Makes a zero-volume cube using the coordinates defined by the array. `cube(ARRAY[1,2]) == '(1,2)'`
`cube(float8[], float8[]) returns cube`	Makes a cube with upper right and lower left coordinates as defined by the two arrays, which must be of the same length. `cube('{1,2}'::float[], '{3,4}'::float[]) == '(1,2),(3,4)'`
`cube(cube, float8) returns cube`	Makes a new cube by adding a dimension on to an existing cube with the same values for both parts of the new coordinate. This is useful for building cubes piece by piece from calculated values. `cube('(1)',2) == '(1,2),(1,2)'`
`cube(cube, float8, float8) returns cube`	Makes a new cube by adding a dimension on to an existing cube. This is useful for building cubes piece by piece from calculated values. `cube('(1,2)',3,4) == '(1,3),(2,4)'`
`cube_dim(cube) returns int`	Returns the number of dimensions of the cube
`cube_ll_coord(cube, int) returns double`	Returns the n'th coordinate value for the lower left corner of a cube
`cube_ur_coord(cube, int) returns double`	Returns the n'th coordinate value for the upper right corner of a cube
`cube_is_point(cube) returns bool`	Returns true if a cube is a point, that is, the two defining corners are the same.
`cube_distance(cube, cube) returns double`	Returns the distance between two cubes. If both cubes are points, this is the normal distance function.

`cube_subset(cube, int[]) returns cube`	Makes a new cube from an existing cube, using a list of dimension indexes from an array. Can be used to find both the LL and UR coordinates of a single dimension, e.g., `cube_subset(cube('(1,3,5),(6,7,8)'), ARRAY[2]) = '(3),(7)'`. Or can be used to drop dimensions, or reorder them as desired, e.g., `cube_subset(cube('(1,3,5),(6,7,8)'), ARRAY[3,2,1,1]) = '(5, 3, 1, 1),(8, 7, 6, 6)'`.
`cube_union(cube, cube) returns cube`	Produces the union of two cubes
`cube_inter(cube, cube) returns cube`	Produces the intersection of two cubes
`cube_enlarge(cube c, double r, int n) returns cube`	Increases the size of a cube by a specified radius in at least n dimensions. If the radius is negative the cube is shrunk instead. This is useful for creating bounding boxes around a point for searching for nearby points. All defined dimensions are changed by the radius r. LL coordinates are decreased by r and UR coordinates are increased by r. If a LL coordinate is increased to larger than the corresponding UR coordinate (this can only happen when r < 0) than both coordinates are set to their average. If n is greater than the number of defined dimensions and the cube is being increased (r >= 0) then 0 is used as the base for the extra coordinates.

F.8.4. Defaults

I believe this union:

```
select cube_union('(0,5,2),(2,3,1)', '0');
cube_union
-------------------
(0, 0, 0),(2, 5, 2)
(1 row)
```

does not contradict common sense, neither does the intersection

```
select cube_inter('(0,-1),(1,1)', '(-2),(2)');
cube_inter
-------------
(0, 0),(1, 0)
(1 row)
```

In all binary operations on differently-dimensioned cubes, I assume the lower-dimensional one to be a Cartesian projection, i. e., having zeroes in place of coordinates omitted in the string representation. The above examples are equivalent to:

```
cube_union('(0,5,2),(2,3,1)','(0,0,0),(0,0,0)');
cube_inter('(0,-1),(1,1)','(-2,0),(2,0)');
```

The following containment predicate uses the point syntax, while in fact the second argument is internally represented by a box. This syntax makes it unnecessary to define a separate point type and functions for (box,point) predicates.

```
select cube_contains('(0,0),(1,1)', '0.5,0.5');
cube_contains
--------------
t
(1 row)
```

F.8.5. Notes

For examples of usage, see the regression test `sql/cube.sql`.

To make it harder for people to break things, there is a limit of 100 on the number of dimensions of cubes. This is set in `cubedata.h` if you need something bigger.

F.8.6. Credits

Original author: Gene Selkov, Jr. <`selkovjr@mcs.anl.gov`>, Mathematics and Computer Science Division, Argonne National Laboratory.

My thanks are primarily to Prof. Joe Hellerstein (https://dsf.berkeley.edu/jmh/) for elucidating the gist of the GiST (http://gist.cs.berkeley.edu/), and to his former student Andy Dong for his example written for Illustra. I am also grateful to all Postgres developers, present and past, for enabling myself to create my own world and live undisturbed in it. And I would like to acknowledge my gratitude to Argonne Lab and to the U.S. Department of Energy for the years of faithful support of my database research.

Minor updates to this package were made by Bruno Wolff III <`bruno@wolff.to`> in August/September of 2002. These include changing the precision from single precision to double precision and adding some new functions.

Additional updates were made by Joshua Reich <`josh@root.net`> in July 2006. These include `cube(float8[], float8[])` and cleaning up the code to use the V1 call protocol instead of the deprecated V0 protocol.

F.9. dblink

`dblink` is a module that supports connections to other PostgreSQL databases from within a database session.

See also postgres_fdw, which provides roughly the same functionality using a more modern and standards-compliant infrastructure.

dblink_connect

Name

dblink_connect — opens a persistent connection to a remote database

Synopsis

```
dblink_connect(text connstr) returns text
dblink_connect(text connname, text connstr) returns text
```

Description

dblink_connect() establishes a connection to a remote PostgreSQL database. The server and database to be contacted are identified through a standard libpq connection string. Optionally, a name can be assigned to the connection. Multiple named connections can be open at once, but only one unnamed connection is permitted at a time. The connection will persist until closed or until the database session is ended.

The connection string may also be the name of an existing foreign server. It is recommended to use the foreign-data wrapper dblink_fdw when defining the foreign server. See the example below, as well as CREATE SERVER and CREATE USER MAPPING.

Arguments

connname

The name to use for this connection; if omitted, an unnamed connection is opened, replacing any existing unnamed connection.

connstr

libpq-style connection info string, for example hostaddr=127.0.0.1 port=5432 dbname=mydb user=postgres password=mypasswd options=-csearch_path=. For details see Section 31.1.1. Alternatively, the name of a foreign server.

Return Value

Returns status, which is always OK (since any error causes the function to throw an error instead of returning).

Notes

If untrusted users have access to a database that has not adopted a secure schema usage pattern, begin each session by removing publicly-writable schemas from `search_path`. One could, for example, add `options=-csearch_path=` to `connstr`. This consideration is not specific to `dblink`; it applies to every interface for executing arbitrary SQL commands.

Only superusers may use `dblink_connect` to create non-password-authenticated connections. If non-superusers need this capability, use `dblink_connect_u` instead.

It is unwise to choose connection names that contain equal signs, as this opens a risk of confusion with connection info strings in other `dblink` functions.

Examples

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_connect('myconn', 'dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

-- FOREIGN DATA WRAPPER functionality
-- Note: local connection must require password authentication for this to work properly
--       Otherwise, you will receive the following error from dblink_connect():
--       ------------------------------------------------------------------------
--       ERROR:  password is required
--       DETAIL:  Non-superuser cannot connect if the server does not request a password.
--       HINT:  Target server's authentication method must be changed.

CREATE SERVER fdtest FOREIGN DATA WRAPPER dblink_fdw OPTIONS (hostaddr '127.0.0.1', dbname

CREATE USER dblink_regression_test WITH PASSWORD 'secret';
CREATE USER MAPPING FOR dblink_regression_test SERVER fdtest OPTIONS (user 'dblink_regressi
GRANT USAGE ON FOREIGN SERVER fdtest TO dblink_regression_test;
GRANT SELECT ON TABLE foo TO dblink_regression_test;

\set ORIGINAL_USER :USER
\c - dblink_regression_test
SELECT dblink_connect('myconn', 'fdtest');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('myconn', 'SELECT * FROM foo') AS t(a int, b text, c text[]);
 a  | b |       c
```

```
----+---+---------------
  0 | a | {a0,b0,c0}
  1 | b | {a1,b1,c1}
  2 | c | {a2,b2,c2}
  3 | d | {a3,b3,c3}
  4 | e | {a4,b4,c4}
  5 | f | {a5,b5,c5}
  6 | g | {a6,b6,c6}
  7 | h | {a7,b7,c7}
  8 | i | {a8,b8,c8}
  9 | j | {a9,b9,c9}
 10 | k | {a10,b10,c10}
(11 rows)

\c - :ORIGINAL_USER
REVOKE USAGE ON FOREIGN SERVER fdtest FROM dblink_regression_test;
REVOKE SELECT ON TABLE foo FROM dblink_regression_test;
DROP USER MAPPING FOR dblink_regression_test SERVER fdtest;
DROP USER dblink_regression_test;
DROP SERVER fdtest;
```

dblink_connect_u

Name

dblink_connect_u — opens a persistent connection to a remote database, insecurely

Synopsis

```
dblink_connect_u(text connstr) returns text
dblink_connect_u(text connname, text connstr) returns text
```

Description

dblink_connect_u() is identical to dblink_connect(), except that it will allow non-superusers to connect using any authentication method.

If the remote server selects an authentication method that does not involve a password, then impersonation and subsequent escalation of privileges can occur, because the session will appear to have originated from the user as which the local PostgreSQL server runs. Also, even if the remote server does demand a password, it is possible for the password to be supplied from the server environment, such as a ~/.pgpass file belonging to the server's user. This opens not only a risk of impersonation, but the possibility of exposing a password to an untrustworthy remote server. Therefore, dblink_connect_u() is initially installed with all privileges revoked from PUBLIC, making it un-callable except by superusers. In some situations it may be appropriate to grant EXECUTE permission for dblink_connect_u() to specific users who are considered trustworthy, but this should be done with care. It is also recommended that any ~/.pgpass file belonging to the server's user *not* contain any records specifying a wildcard host name.

For further details see dblink_connect().

dblink_disconnect

Name

dblink_disconnect — closes a persistent connection to a remote database

Synopsis

```
dblink_disconnect() returns text
dblink_disconnect(text connname) returns text
```

Description

dblink_disconnect() closes a connection previously opened by dblink_connect(). The form with no arguments closes an unnamed connection.

Arguments

connname

The name of a named connection to be closed.

Return Value

Returns status, which is always OK (since any error causes the function to throw an error instead of returning).

Examples

```
SELECT dblink_disconnect();
 dblink_disconnect
-------------------
 OK
(1 row)

SELECT dblink_disconnect('myconn');
 dblink_disconnect
-------------------
 OK
(1 row)
```

dblink

Name

dblink — executes a query in a remote database

Synopsis

```
dblink(text connname, text sql [, bool fail_on_error]) returns setof record
dblink(text connstr, text sql [, bool fail_on_error]) returns setof record
dblink(text sql [, bool fail_on_error]) returns setof record
```

Description

dblink executes a query (usually a SELECT, but it can be any SQL statement that returns rows) in a remote database.

When two text arguments are given, the first one is first looked up as a persistent connection's name; if found, the command is executed on that connection. If not found, the first argument is treated as a connection info string as for dblink_connect, and the indicated connection is made just for the duration of this command.

Arguments

connname

Name of the connection to use; omit this parameter to use the unnamed connection.

connstr

A connection info string, as previously described for dblink_connect.

sql

The SQL query that you wish to execute in the remote database, for example select * from foo.

fail_on_error

If true (the default when omitted) then an error thrown on the remote side of the connection causes an error to also be thrown locally. If false, the remote error is locally reported as a NOTICE, and the function returns no rows.

Return Value

The function returns the row(s) produced by the query. Since dblink can be used with any query, it is declared to return record, rather than specifying any particular set of columns. This means that you must

specify the expected set of columns in the calling query — otherwise PostgreSQL would not know what to expect. Here is an example:

```
SELECT *
    FROM dblink('dbname=mydb options=-csearch_path=',
                'select proname, prosrc from pg_proc')
      AS t1(proname name, prosrc text)
    WHERE proname LIKE 'bytea%';
```

The "alias" part of the `FROM` clause must specify the column names and types that the function will return. (Specifying column names in an alias is actually standard SQL syntax, but specifying column types is a PostgreSQL extension.) This allows the system to understand what `*` should expand to, and what `proname` in the `WHERE` clause refers to, in advance of trying to execute the function. At run time, an error will be thrown if the actual query result from the remote database does not have the same number of columns shown in the `FROM` clause. The column names need not match, however, and `dblink` does not insist on exact type matches either. It will succeed so long as the returned data strings are valid input for the column type declared in the `FROM` clause.

Notes

A convenient way to use `dblink` with predetermined queries is to create a view. This allows the column type information to be buried in the view, instead of having to spell it out in every query. For example,

```
CREATE VIEW myremote_pg_proc AS
  SELECT *
    FROM dblink('dbname=postgres options=-csearch_path=',
                'select proname, prosrc from pg_proc')
    AS t1(proname name, prosrc text);

SELECT * FROM myremote_pg_proc WHERE proname LIKE 'bytea%';
```

Examples

```
SELECT * FROM dblink('dbname=postgres options=-csearch_path=',
                     'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
```

```
 byteanlike | byteanlike
 byteain    | byteain
 byteaout   | byteaout
(12 rows)

SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteain    | byteain
 byteaout   | byteaout
(12 rows)

SELECT dblink_connect('myconn', 'dbname=regression options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT * FROM dblink('myconn', 'select proname, prosrc from pg_proc')
  AS t1(proname name, prosrc text) WHERE proname LIKE 'bytea%';
  proname   |   prosrc
------------+------------
 bytearecv  | bytearecv
 byteasend  | byteasend
 byteale    | byteale
 byteagt    | byteagt
 byteage    | byteage
 byteane    | byteane
 byteacmp   | byteacmp
 bytealike  | bytealike
 byteanlike | byteanlike
 byteacat   | byteacat
 byteaeq    | byteaeq
 bytealt    | bytealt
 byteain    | byteain
 byteaout   | byteaout
```

```
(14 rows)
```

dblink_exec

Name

dblink_exec — executes a command in a remote database

Synopsis

```
dblink_exec(text connname, text sql [, bool fail_on_error]) returns text
dblink_exec(text connstr, text sql [, bool fail_on_error]) returns text
dblink_exec(text sql [, bool fail_on_error]) returns text
```

Description

dblink_exec executes a command (that is, any SQL statement that doesn't return rows) in a remote database.

When two text arguments are given, the first one is first looked up as a persistent connection's name; if found, the command is executed on that connection. If not found, the first argument is treated as a connection info string as for dblink_connect, and the indicated connection is made just for the duration of this command.

Arguments

connname

Name of the connection to use; omit this parameter to use the unnamed connection.

connstr

A connection info string, as previously described for dblink_connect.

sql

The SQL command that you wish to execute in the remote database, for example insert into foo values(0, 'a', '{"a0","b0","c0"}').

fail_on_error

If true (the default when omitted) then an error thrown on the remote side of the connection causes an error to also be thrown locally. If false, the remote error is locally reported as a NOTICE, and the function's return value is set to ERROR.

Return Value

Returns status, either the command's status string or ERROR.

Examples

```
SELECT dblink_connect('dbname=dblink_test_standby');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_exec('insert into foo values(21, "z", "{"a0","b0","c0"}");');
   dblink_exec
-----------------
 INSERT 943366 1
(1 row)

SELECT dblink_connect('myconn', 'dbname=regression');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_exec('myconn', 'insert into foo values(21, "z", "{"a0","b0","c0"}");');
   dblink_exec
------------------
 INSERT 6432584 1
(1 row)

SELECT dblink_exec('myconn', 'insert into pg_class values ("foo")',false);
NOTICE:  sql error
DETAIL:  ERROR:  null value in column "relnamespace" violates not-null constraint

 dblink_exec
-------------
 ERROR
(1 row)
```

dblink_open

Name

dblink_open — opens a cursor in a remote database

Synopsis

```
dblink_open(text cursorname, text sql [, bool fail_on_error]) returns text
dblink_open(text connname, text cursorname, text sql [, bool fail_on_error]) returns text
```

Description

dblink_open() opens a cursor in a remote database. The cursor can subsequently be manipulated with dblink_fetch() and dblink_close().

Arguments

connname

Name of the connection to use; omit this parameter to use the unnamed connection.

cursorname

The name to assign to this cursor.

sql

The SELECT statement that you wish to execute in the remote database, for example select * from pg_class.

fail_on_error

If true (the default when omitted) then an error thrown on the remote side of the connection causes an error to also be thrown locally. If false, the remote error is locally reported as a NOTICE, and the function's return value is set to ERROR.

Return Value

Returns status, either OK or ERROR.

Notes

Since a cursor can only persist within a transaction, dblink_open starts an explicit transaction block (BEGIN) on the remote side, if the remote side was not already within a transaction. This transaction will be closed again when the matching dblink_close is executed. Note that if you use dblink_exec

to change data between `dblink_open` and `dblink_close`, and then an error occurs or you use `dblink_disconnect` before `dblink_close`, your change *will be lost* because the transaction will be aborted.

Examples

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
 OK
(1 row)
```

dblink_fetch

Name

dblink_fetch — returns rows from an open cursor in a remote database

Synopsis

```
dblink_fetch(text cursorname, int howmany [, bool fail_on_error]) returns setof record
dblink_fetch(text connname, text cursorname, int howmany [, bool fail_on_error]) returns se
```

Description

dblink_fetch fetches rows from a cursor previously established by dblink_open.

Arguments

connname

Name of the connection to use; omit this parameter to use the unnamed connection.

cursorname

The name of the cursor to fetch from.

howmany

The maximum number of rows to retrieve. The next howmany rows are fetched, starting at the current cursor position, moving forward. Once the cursor has reached its end, no more rows are produced.

fail_on_error

If true (the default when omitted) then an error thrown on the remote side of the connection causes an error to also be thrown locally. If false, the remote error is locally reported as a NOTICE, and the function returns no rows.

Return Value

The function returns the row(s) fetched from the cursor. To use this function, you will need to specify the expected set of columns, as previously discussed for dblink.

Notes

On a mismatch between the number of return columns specified in the FROM clause, and the actual number of columns returned by the remote cursor, an error will be thrown. In this event, the remote cursor is still

advanced by as many rows as it would have been if the error had not occurred. The same is true for any other error occurring in the local query after the remote `FETCH` has been done.

Examples

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc where proname like "bytea%"');
 dblink_open
-------------
 OK
(1 row)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname |  source
----------+----------
 byteacat | byteacat
 byteacmp | byteacmp
 byteaeq  | byteaeq
 byteage  | byteage
 byteagt  | byteagt
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname  |  source
-----------+-----------
 byteain   | byteain
 byteale   | byteale
 bytealike | bytealike
 bytealt   | bytealt
 byteane   | byteane
(5 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
  funcname  |   source
------------+------------
 byteanlike | byteanlike
 byteaout   | byteaout
(2 rows)

SELECT * FROM dblink_fetch('foo', 5) AS (funcname name, source text);
 funcname | source
----------+--------
(0 rows)
```

dblink_close

Name

dblink_close — closes a cursor in a remote database

Synopsis

```
dblink_close(text cursorname [, bool fail_on_error]) returns text
dblink_close(text connname, text cursorname [, bool fail_on_error]) returns text
```

Description

dblink_close closes a cursor previously opened with dblink_open.

Arguments

connname

Name of the connection to use; omit this parameter to use the unnamed connection.

cursorname

The name of the cursor to close.

fail_on_error

If true (the default when omitted) then an error thrown on the remote side of the connection causes an error to also be thrown locally. If false, the remote error is locally reported as a NOTICE, and the function's return value is set to ERROR.

Return Value

Returns status, either OK or ERROR.

Notes

If dblink_open started an explicit transaction block, and this is the last remaining open cursor in this connection, dblink_close will issue the matching COMMIT.

Examples

```
SELECT dblink_connect('dbname=postgres options=-csearch_path=');
 dblink_connect
```

```
----------------
 OK
(1 row)

SELECT dblink_open('foo', 'select proname, prosrc from pg_proc');
 dblink_open
-------------
 OK
(1 row)

SELECT dblink_close('foo');
 dblink_close
--------------
 OK
(1 row)
```

dblink_get_connections

Name

`dblink_get_connections` — returns the names of all open named dblink connections

Synopsis

```
dblink_get_connections() returns text[]
```

Description

`dblink_get_connections` returns an array of the names of all open named `dblink` connections.

Return Value

Returns a text array of connection names, or NULL if none.

Examples

```
SELECT dblink_get_connections();
```

dblink_error_message

Name

dblink_error_message — gets last error message on the named connection

Synopsis

```
dblink_error_message(text connname) returns text
```

Description

dblink_error_message fetches the most recent remote error message for a given connection.

Arguments

connname

Name of the connection to use.

Return Value

Returns last error message, or OK if there has been no error in this connection.

Notes

When asynchronous queries are initiated by dblink_send_query, the error message associated with the connection might not get updated until the server's response message is consumed. This typically means that dblink_is_busy or dblink_get_result should be called prior to dblink_error_message, so that any error generated by the asynchronous query will be visible.

Examples

```
SELECT dblink_error_message('dtest1');
```

dblink_send_query

Name

`dblink_send_query` — sends an async query to a remote database

Synopsis

```
dblink_send_query(text connname, text sql) returns int
```

Description

`dblink_send_query` sends a query to be executed asynchronously, that is, without immediately waiting for the result. There must not be an async query already in progress on the connection.

After successfully dispatching an async query, completion status can be checked with `dblink_is_busy`, and the results are ultimately collected with `dblink_get_result`. It is also possible to attempt to cancel an active async query using `dblink_cancel_query`.

Arguments

`connname`

Name of the connection to use.

`sql`

The SQL statement that you wish to execute in the remote database, for example `select * from pg_class`.

Return Value

Returns 1 if the query was successfully dispatched, 0 otherwise.

Examples

```
SELECT dblink_send_query('dtest1', 'SELECT * FROM foo WHERE f1 < 3');
```

dblink_is_busy

Name

`dblink_is_busy` — checks if connection is busy with an async query

Synopsis

```
dblink_is_busy(text connname) returns int
```

Description

`dblink_is_busy` tests whether an async query is in progress.

Arguments

`connname`

Name of the connection to check.

Return Value

Returns 1 if connection is busy, 0 if it is not busy. If this function returns 0, it is guaranteed that `dblink_get_result` will not block.

Examples

```
SELECT dblink_is_busy('dtest1');
```

dblink_get_notify

Name

dblink_get_notify — retrieve async notifications on a connection

Synopsis

```
dblink_get_notify() returns setof (notify_name text, be_pid int, extra text)
dblink_get_notify(text connname) returns setof (notify_name text, be_pid int, extra text)
```

Description

dblink_get_notify retrieves notifications on either the unnamed connection, or on a named connection if specified. To receive notifications via dblink, LISTEN must first be issued, using dblink_exec. For details see LISTEN and NOTIFY.

Arguments

connname

The name of a named connection to get notifications on.

Return Value

Returns setof (notify_name text, be_pid int, extra text), or an empty set if none.

Examples

```
SELECT dblink_exec('LISTEN virtual');
 dblink_exec
-------------
 LISTEN
(1 row)

SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-------------+--------+-------
(0 rows)

NOTIFY virtual;
NOTIFY
```

```
SELECT * FROM dblink_get_notify();
 notify_name | be_pid | extra
-------------+--------+-------
 virtual     |   1229 |
(1 row)
```

dblink_get_result

Name

dblink_get_result — gets an async query result

Synopsis

```
dblink_get_result(text connname [, bool fail_on_error]) returns setof record
```

Description

dblink_get_result collects the results of an asynchronous query previously sent with dblink_send_query. If the query is not already completed, dblink_get_result will wait until it is.

Arguments

connname

Name of the connection to use.

fail_on_error

If true (the default when omitted) then an error thrown on the remote side of the connection causes an error to also be thrown locally. If false, the remote error is locally reported as a NOTICE, and the function returns no rows.

Return Value

For an async query (that is, a SQL statement returning rows), the function returns the row(s) produced by the query. To use this function, you will need to specify the expected set of columns, as previously discussed for dblink.

For an async command (that is, a SQL statement not returning rows), the function returns a single row with a single text column containing the command's status string. It is still necessary to specify that the result will have a single text column in the calling FROM clause.

Notes

This function *must* be called if dblink_send_query returned 1. It must be called once for each query sent, and one additional time to obtain an empty set result, before the connection can be used again.

When using dblink_send_query and dblink_get_result, dblink fetches the entire remote query result before returning any of it to the local query processor. If the query returns a large number of rows, this can result in transient memory bloat in the local session. It may be better to open such a query as a

cursor with `dblink_open` and then fetch a manageable number of rows at a time. Alternatively, use plain `dblink()`, which avoids memory bloat by spooling large result sets to disk.

Examples

```
contrib_regression=# SELECT dblink_connect('dtest1', 'dbname=contrib_regression');
 dblink_connect
----------------
 OK
(1 row)

contrib_regression=# SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 < 3') AS t1;
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 te
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 te
 f1 | f2 | f3
----+----+----
(0 rows)

contrib_regression=# SELECT * FROM
contrib_regression-# dblink_send_query('dtest1', 'select * from foo where f1 < 3; select *
 t1
----
  1
(1 row)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 te
 f1 | f2 |     f3
----+----+------------
  0 | a  | {a0,b0,c0}
  1 | b  | {a1,b1,c1}
  2 | c  | {a2,b2,c2}
(3 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 te
 f1 | f2 |      f3
----+----+---------------
  7 | h  | {a7,b7,c7}
  8 | i  | {a8,b8,c8}
```

```
  9 | j  | {a9,b9,c9}
 10 | k  | {a10,b10,c10}
(4 rows)

contrib_regression=# SELECT * FROM dblink_get_result('dtest1') AS t1(f1 int, f2 text, f3 te
 f1 | f2 | f3
----+----+----
(0 rows)
```

dblink_cancel_query

Name

dblink_cancel_query — cancels any active query on the named connection

Synopsis

```
dblink_cancel_query(text connname) returns text
```

Description

dblink_cancel_query attempts to cancel any query that is in progress on the named connection. Note that this is not certain to succeed (since, for example, the remote query might already have finished). A cancel request simply improves the odds that the query will fail soon. You must still complete the normal query protocol, for example by calling dblink_get_result.

Arguments

connname

Name of the connection to use.

Return Value

Returns OK if the cancel request has been sent, or the text of an error message on failure.

Examples

```
SELECT dblink_cancel_query('dtest1');
```

dblink_get_pkey

Name

dblink_get_pkey — returns the positions and field names of a relation's primary key fields

Synopsis

```
dblink_get_pkey(text relname) returns setof dblink_pkey_results
```

Description

dblink_get_pkey provides information about the primary key of a relation in the local database. This is sometimes useful in generating queries to be sent to remote databases.

Arguments

relname

Name of a local relation, for example foo or myschema.mytab. Include double quotes if the name is mixed-case or contains special characters, for example "FooBar"; without quotes, the string will be folded to lower case.

Return Value

Returns one row for each primary key field, or no rows if the relation has no primary key. The result row type is defined as

```
CREATE TYPE dblink_pkey_results AS (position int, colname text);
```

The position column simply runs from 1 to *N*; it is the number of the field within the primary key, not the number within the table's columns.

Examples

```
CREATE TABLE foobar (
    f1 int,
    f2 int,
    f3 int,
    PRIMARY KEY (f1, f2, f3)
);
CREATE TABLE
```

```
SELECT * FROM dblink_get_pkey('foobar');
 position | colname
----------+---------
        1 | f1
        2 | f2
        3 | f3
(3 rows)
```

dblink_build_sql_insert

Name

dblink_build_sql_insert — builds an INSERT statement using a local tuple, replacing the primary key field values with alternative supplied values

Synopsis

```
dblink_build_sql_insert(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) returns text
```

Description

dblink_build_sql_insert can be useful in doing selective replication of a local table to a remote database. It selects a row from the local table based on primary key, and then builds a SQL INSERT command that will duplicate that row, but with the primary key values replaced by the values in the last argument. (To make an exact copy of the row, just specify the same values for the last two arguments.)

Arguments

relname

Name of a local relation, for example foo or myschema.mytab. Include double quotes if the name is mixed-case or contains special characters, for example "FooBar"; without quotes, the string will be folded to lower case.

primary_key_attnums

Attribute numbers (1-based) of the primary key fields, for example 1 2.

num_primary_key_atts

The number of primary key fields.

src_pk_att_vals_array

Values of the primary key fields to be used to look up the local tuple. Each field is represented in text form. An error is thrown if there is no local row with these primary key values.

tgt_pk_att_vals_array

Values of the primary key fields to be placed in the resulting INSERT command. Each field is represented in text form.

Return Value

Returns the requested SQL statement as text.

Notes

As of PostgreSQL 9.0, the attribute numbers in `primary_key_attnums` are interpreted as logical column numbers, corresponding to the column's position in `SELECT * FROM relname`. Previous versions interpreted the numbers as physical column positions. There is a difference if any column(s) to the left of the indicated column have been dropped during the lifetime of the table.

Examples

```
SELECT dblink_build_sql_insert('foo', '1 2', 2, '{"1", "a"}', '{"1", "b"a"}');
             dblink_build_sql_insert
------------------------------------------------
 INSERT INTO foo(f1,f2,f3) VALUES('1','b"a','1')
(1 row)
```

dblink_build_sql_delete

Name

dblink_build_sql_delete — builds a DELETE statement using supplied values for primary key field values

Synopsis

```
dblink_build_sql_delete(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] tgt_pk_att_vals_array) returns text
```

Description

dblink_build_sql_delete can be useful in doing selective replication of a local table to a remote database. It builds a SQL DELETE command that will delete the row with the given primary key values.

Arguments

relname

Name of a local relation, for example foo or myschema.mytab. Include double quotes if the name is mixed-case or contains special characters, for example "FooBar"; without quotes, the string will be folded to lower case.

primary_key_attnums

Attribute numbers (1-based) of the primary key fields, for example 1 2.

num_primary_key_atts

The number of primary key fields.

tgt_pk_att_vals_array

Values of the primary key fields to be used in the resulting DELETE command. Each field is represented in text form.

Return Value

Returns the requested SQL statement as text.

Notes

As of PostgreSQL 9.0, the attribute numbers in `primary_key_attnums` are interpreted as logical column numbers, corresponding to the column's position in `SELECT * FROM relname`. Previous versions interpreted the numbers as physical column positions. There is a difference if any column(s) to the left of the indicated column have been dropped during the lifetime of the table.

Examples

```
SELECT dblink_build_sql_delete('"MyFoo"', '1 2', 2, '{"1", "b"}');
           dblink_build_sql_delete
---------------------------------------------
 DELETE FROM "MyFoo" WHERE f1='1' AND f2='b'
(1 row)
```

dblink_build_sql_update

Name

dblink_build_sql_update — builds an UPDATE statement using a local tuple, replacing the primary key field values with alternative supplied values

Synopsis

```
dblink_build_sql_update(text relname,
                        int2vector primary_key_attnums,
                        integer num_primary_key_atts,
                        text[] src_pk_att_vals_array,
                        text[] tgt_pk_att_vals_array) returns text
```

Description

dblink_build_sql_update can be useful in doing selective replication of a local table to a remote database. It selects a row from the local table based on primary key, and then builds a SQL UPDATE command that will duplicate that row, but with the primary key values replaced by the values in the last argument. (To make an exact copy of the row, just specify the same values for the last two arguments.) The UPDATE command always assigns all fields of the row — the main difference between this and dblink_build_sql_insert is that it's assumed that the target row already exists in the remote table.

Arguments

relname

Name of a local relation, for example foo or myschema.mytab. Include double quotes if the name is mixed-case or contains special characters, for example "FooBar"; without quotes, the string will be folded to lower case.

primary_key_attnums

Attribute numbers (1-based) of the primary key fields, for example 1 2.

num_primary_key_atts

The number of primary key fields.

src_pk_att_vals_array

Values of the primary key fields to be used to look up the local tuple. Each field is represented in text form. An error is thrown if there is no local row with these primary key values.

tgt_pk_att_vals_array

Values of the primary key fields to be placed in the resulting UPDATE command. Each field is represented in text form.

Return Value

Returns the requested SQL statement as text.

Notes

As of PostgreSQL 9.0, the attribute numbers in `primary_key_attnums` are interpreted as logical column numbers, corresponding to the column's position in `SELECT * FROM relname`. Previous versions interpreted the numbers as physical column positions. There is a difference if any column(s) to the left of the indicated column have been dropped during the lifetime of the table.

Examples

```
SELECT dblink_build_sql_update('foo', '1 2', 2, '{"1", "a"}', '{"1", "b"}');
                   dblink_build_sql_update
-------------------------------------------------------------
 UPDATE foo SET f1='1',f2='b',f3='1' WHERE f1='1' AND f2='b'
(1 row)
```

F.10. dict_int

`dict_int` is an example of an add-on dictionary template for full-text search. The motivation for this example dictionary is to control the indexing of integers (signed and unsigned), allowing such numbers to be indexed while preventing excessive growth in the number of unique words, which greatly affects the performance of searching.

F.10.1. Configuration

The dictionary accepts two options:

- The `maxlen` parameter specifies the maximum number of digits allowed in an integer word. The default value is 6.
- The `rejectlong` parameter specifies whether an overlength integer should be truncated or ignored. If `rejectlong` is `false` (the default), the dictionary returns the first `maxlen` digits of the integer. If `rejectlong` is `true`, the dictionary treats an overlength integer as a stop word, so that it will not be indexed. Note that this also means that such an integer cannot be searched for.

F.10.2. Usage

Installing the `dict_int` extension creates a text search template `intdict_template` and a dictionary `intdict` based on it, with the default parameters. You can alter the parameters, for example

```
mydb# ALTER TEXT SEARCH DICTIONARY intdict (MAXLEN = 4, REJECTLONG = true);
ALTER TEXT SEARCH DICTIONARY
```

or create new dictionaries based on the template.

To test the dictionary, you can try

```
mydb# select ts_lexize('intdict', '12345678');
 ts_lexize
-----------
 {123456}
```

but real-world usage will involve including it in a text search configuration as described in Chapter 12. That might look like this:

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR int, uint WITH intdict;
```

F.11. dict_xsyn

dict_xsyn (Extended Synonym Dictionary) is an example of an add-on dictionary template for full-text search. This dictionary type replaces words with groups of their synonyms, and so makes it possible to search for a word using any of its synonyms.

F.11.1. Configuration

A dict_xsyn dictionary accepts the following options:

- matchorig controls whether the original word is accepted by the dictionary. Default is true.
- matchsynonyms controls whether the synonyms are accepted by the dictionary. Default is false.
- keeporig controls whether the original word is included in the dictionary's output. Default is true.
- keepsynonyms controls whether the synonyms are included in the dictionary's output. Default is true.
- rules is the base name of the file containing the list of synonyms. This file must be stored in $SHAREDIR/tsearch_data/ (where $SHAREDIR means the PostgreSQL installation's shared-data directory). Its name must end in .rules (which is not to be included in the rules parameter).

The rules file has the following format:

- Each line represents a group of synonyms for a single word, which is given first on the line. Synonyms are separated by whitespace, thus:

  ```
  word syn1 syn2 syn3
  ```

- The sharp (#) sign is a comment delimiter. It may appear at any position in a line. The rest of the line will be skipped.

Look at xsyn_sample.rules, which is installed in $SHAREDIR/tsearch_data/, for an example.

F.11.2. Usage

Installing the dict_xsyn extension creates a text search template xsyn_template and a dictionary xsyn based on it, with default parameters. You can alter the parameters, for example

```
mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false);
ALTER TEXT SEARCH DICTIONARY
```

or create new dictionaries based on the template.

To test the dictionary, you can try

```
mydb=# SELECT ts_lexize('xsyn', 'word');
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true);
ALTER TEXT SEARCH DICTIONARY
```

```
mydb=# SELECT ts_lexize('xsyn', 'word');
      ts_lexize
-----------------------
 {word,syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=false, MATCHSYNONYMS=tr
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'syn1');
      ts_lexize
-----------------------
 {syn1,syn2,syn3}

mydb# ALTER TEXT SEARCH DICTIONARY xsyn (RULES='my_rules', KEEPORIG=true, MATCHORIG=false,
ALTER TEXT SEARCH DICTIONARY

mydb=# SELECT ts_lexize('xsyn', 'syn1');
      ts_lexize
-----------------------
 {word}
```

Real-world usage will involve including it in a text search configuration as described in Chapter 12. That might look like this:

```
ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR word, asciiword WITH xsyn, english_stem;
```

F.12. earthdistance

The `earthdistance` module provides two different approaches to calculating great circle distances on the surface of the Earth. The one described first depends on the `cube` module. The second one is based on the built-in `point` data type, using longitude and latitude for the coordinates.

In this module, the Earth is assumed to be perfectly spherical. (If that's too inaccurate for you, you might want to look at the PostGIS[1] project.)

The `cube` module must be installed before `earthdistance` can be installed.

> **Caution**
>
> It is strongly recommended that `earthdistance` and `cube` be installed in the same schema, and that that schema be one for which CREATE privilege has not been and will not be granted to any untrusted users. Otherwise there are installation-time security hazards if `earthdistance`'s schema contains objects defined by a hostile user. Furthermore, when using `earthdistance`'s functions after installation, the entire search path should contain only trusted schemas.

1. https://postgis.net/

F.12.1. Cube-based Earth Distances

Data is stored in cubes that are points (both corners are the same) using 3 coordinates representing the x, y, and z distance from the center of the Earth. A domain `earth` over `cube` is provided, which includes constraint checks that the value meets these restrictions and is reasonably close to the actual surface of the Earth.

The radius of the Earth is obtained from the `earth()` function. It is given in meters. But by changing this one function you can change the module to use some other units, or to use a different value of the radius that you feel is more appropriate.

This package has applications to astronomical databases as well. Astronomers will probably want to change `earth()` to return a radius of `180/pi()` so that distances are in degrees.

Functions are provided to support input in latitude and longitude (in degrees), to support output of latitude and longitude, to calculate the great circle distance between two points and to easily specify a bounding box usable for index searches.

The provided functions are shown in Table F-6.

Table F-6. Cube-based Earthdistance Functions

Function	Returns	Description
`earth()`	`float8`	Returns the assumed radius of the Earth.
`sec_to_gc(float8)`	`float8`	Converts the normal straight line (secant) distance between two points on the surface of the Earth to the great circle distance between them.
`gc_to_sec(float8)`	`float8`	Converts the great circle distance between two points on the surface of the Earth to the normal straight line (secant) distance between them.
`ll_to_earth(float8, float8)`	`earth`	Returns the location of a point on the surface of the Earth given its latitude (argument 1) and longitude (argument 2) in degrees.
`latitude(earth)`	`float8`	Returns the latitude in degrees of a point on the surface of the Earth.
`longitude(earth)`	`float8`	Returns the longitude in degrees of a point on the surface of the Earth.
`earth_distance(earth, earth)`	`float8`	Returns the great circle distance between two points on the surface of the Earth.

Function	Returns	Description
earth_box(earth, float8)	cube	Returns a box suitable for an indexed search using the cube @> operator for points within a given great circle distance of a location. Some points in this box are further than the specified great circle distance from the location, so a second check using earth_distance should be included in the query.

F.12.2. Point-based Earth Distances

The second part of the module relies on representing Earth locations as values of type point, in which the first component is taken to represent longitude in degrees, and the second component is taken to represent latitude in degrees. Points are taken as (longitude, latitude) and not vice versa because longitude is closer to the intuitive idea of x-axis and latitude to y-axis.

A single operator is provided, shown in Table F-7.

Table F-7. Point-based Earthdistance Operators

Operator	Returns	Description
point <@> point	float8	Gives the distance in statute miles between two points on the Earth's surface.

Note that unlike the cube-based part of the module, units are hardwired here: changing the earth() function will not affect the results of this operator.

One disadvantage of the longitude/latitude representation is that you need to be careful about the edge conditions near the poles and near +/- 180 degrees of longitude. The cube-based representation avoids these discontinuities.

F.13. file_fdw

The file_fdw module provides the foreign-data wrapper file_fdw, which can be used to access data files in the server's file system. Data files must be in a format that can be read by COPY FROM; see COPY for details. Access to such data files is currently read-only.

A foreign table created using this wrapper can have the following options:

filename

Specifies the file to be read. Required. Relative paths are relative to the data directory.

`format`

Specifies the file's format, the same as `COPY`'s `FORMAT` option.

`header`

Specifies whether the file has a header line, the same as `COPY`'s `HEADER` option.

`delimiter`

Specifies the file's delimiter character, the same as `COPY`'s `DELIMITER` option.

`quote`

Specifies the file's quote character, the same as `COPY`'s `QUOTE` option.

`escape`

Specifies the file's escape character, the same as `COPY`'s `ESCAPE` option.

`null`

Specifies the file's null string, the same as `COPY`'s `NULL` option.

`encoding`

Specifies the file's encoding, the same as `COPY`'s `ENCODING` option.

Note that while `COPY` allows options such as OIDS and HEADER to be specified without a corresponding value, the foreign data wrapper syntax requires a value to be present in all cases. To activate `COPY` options normally supplied without a value, you can instead pass the value TRUE.

A column of a foreign table created using this wrapper can have the following options:

`force_not_null`

This is a Boolean option. If true, it specifies that values of the column should not be matched against the null string (that is, the file-level `null` option). This has the same effect as listing the column in `COPY`'s `FORCE_NOT_NULL` option.

`force_null`

This is a Boolean option. If true, it specifies that values of the column which match the null string are returned as `NULL` even if the value is quoted. Without this option, only unquoted values matching the null string are returned as `NULL`. This has the same effect as listing the column in `COPY`'s `FORCE_NULL` option.

`COPY`'s `OIDS` and `FORCE_QUOTE` options are currently not supported by `file_fdw`.

These options can only be specified for a foreign table or its columns, not in the options of the `file_fdw` foreign-data wrapper, nor in the options of a server or user mapping using the wrapper.

Changing table-level options requires superuser privileges, for security reasons: only a superuser should be able to determine which file is read. In principle non-superusers could be allowed to change the other options, but that's not supported at present.

For a foreign table using `file_fdw`, `EXPLAIN` shows the name of the file to be read. Unless `COSTS OFF` is specified, the file size (in bytes) is shown as well.

Example F-1. Create a Foreign Table for PostgreSQL CSV Logs

One of the obvious uses for `file_fdw` is to make the PostgreSQL activity log available as a table for querying. To do this, first you must be logging to a CSV file, which here we will call `pglog.csv`. First, install `file_fdw` as an extension:

```
CREATE EXTENSION file_fdw;
```

Then create a foreign server:

```
CREATE SERVER pglog FOREIGN DATA WRAPPER file_fdw;
```

Now you are ready to create the foreign data table. Using the `CREATE FOREIGN TABLE` command, you will need to define the columns for the table, the CSV file name, and its format:

```
CREATE FOREIGN TABLE pglog (
  log_time timestamp(3) with time zone,
  user_name text,
  database_name text,
  process_id integer,
  connection_from text,
  session_id text,
  session_line_num bigint,
  command_tag text,
  session_start_time timestamp with time zone,
  virtual_transaction_id text,
  transaction_id bigint,
  error_severity text,
  sql_state_code text,
  message text,
  detail text,
  hint text,
  internal_query text,
  internal_query_pos integer,
  context text,
  query text,
  query_pos integer,
  location text,
  application_name text
) SERVER pglog
OPTIONS ( filename '/home/josh/9.1/data/pg_log/pglog.csv', format 'csv' );
```

That's it — now you can query your log directly. In production, of course, you would need to define some way to deal with log rotation.

F.14. fuzzystrmatch

The `fuzzystrmatch` module provides several functions to determine similarities and distance between strings.

Caution

At present, the `soundex`, `metaphone`, `dmetaphone`, and `dmetaphone_alt` functions do not work well with multibyte encodings (such as UTF-8).

F.14.1. Soundex

The Soundex system is a method of matching similar-sounding names by converting them to the same code. It was initially used by the United States Census in 1880, 1900, and 1910. Note that Soundex is not very useful for non-English names.

The `fuzzystrmatch` module provides two functions for working with Soundex codes:

```
soundex(text) returns text
difference(text, text) returns int
```

The `soundex` function converts a string to its Soundex code. The `difference` function converts two strings to their Soundex codes and then reports the number of matching code positions. Since Soundex codes have four characters, the result ranges from zero to four, with zero being no match and four being an exact match. (Thus, the function is misnamed — `similarity` would have been a better name.)

Here are some usage examples:

```
SELECT soundex('hello world!');

SELECT soundex('Anne'), soundex('Ann'), difference('Anne', 'Ann');
SELECT soundex('Anne'), soundex('Andrew'), difference('Anne', 'Andrew');
SELECT soundex('Anne'), soundex('Margaret'), difference('Anne', 'Margaret');

CREATE TABLE s (nm text);

INSERT INTO s VALUES ('john');
INSERT INTO s VALUES ('joan');
INSERT INTO s VALUES ('wobbly');
INSERT INTO s VALUES ('jack');

SELECT * FROM s WHERE soundex(nm) = soundex('john');

SELECT * FROM s WHERE difference(s.nm, 'john') > 2;
```

F.14.2. Levenshtein

This function calculates the Levenshtein distance between two strings:

```
levenshtein(text source, text target, int ins_cost, int del_cost, int sub_cost) returns int
levenshtein(text source, text target) returns int
levenshtein_less_equal(text source, text target, int ins_cost, int del_cost, int sub_cost,
levenshtein_less_equal(text source, text target, int max_d) returns int
```

Both `source` and `target` can be any non-null string, with a maximum of 255 characters. The cost parameters specify how much to charge for a character insertion, deletion, or substitution, respectively. You can omit the cost parameters, as in the second version of the function; in that case they all default to 1.

`levenshtein_less_equal` is an accelerated version of the Levenshtein function for use when only small distances are of interest. If the actual distance is less than or equal to `max_d`, then `levenshtein_less_equal` returns the correct distance; otherwise it returns some value greater than `max_d`. If `max_d` is negative then the behavior is the same as `levenshtein`.

Examples:

```
test=# SELECT levenshtein('GUMBO', 'GAMBOL');
 levenshtein
-------------
           2
(1 row)

test=# SELECT levenshtein('GUMBO', 'GAMBOL', 2,1,1);
 levenshtein
-------------
           3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive',2);
 levenshtein_less_equal
------------------------
                      3
(1 row)

test=# SELECT levenshtein_less_equal('extensive', 'exhaustive',4);
 levenshtein_less_equal
------------------------
                      4
(1 row)
```

F.14.3. Metaphone

Metaphone, like Soundex, is based on the idea of constructing a representative code for an input string. Two strings are then deemed similar if they have the same codes.

This function calculates the metaphone code of an input string:

```
metaphone(text source, int max_output_length) returns text
```

`source` has to be a non-null string with a maximum of 255 characters. `max_output_length` sets the maximum length of the output metaphone code; if longer, the output is truncated to this length.

Example:

```
test=# SELECT metaphone('GUMBO', 4);
 metaphone
```

```
-----------
 KM
(1 row)
```

F.14.4. Double Metaphone

The Double Metaphone system computes two "sounds like" strings for a given input string — a "primary" and an "alternate". In most cases they are the same, but for non-English names especially they can be a bit different, depending on pronunciation. These functions compute the primary and alternate codes:

```
dmetaphone(text source) returns text
dmetaphone_alt(text source) returns text
```

There is no length limit on the input strings.

Example:

```
test=# select dmetaphone('gumbo');
 dmetaphone
------------
 KMP
(1 row)
```

F.15. hstore

This module implements the `hstore` data type for storing sets of key/value pairs within a single PostgreSQL value. This can be useful in various scenarios, such as rows with many attributes that are rarely examined, or semi-structured data. Keys and values are simply text strings.

F.15.1. `hstore` External Representation

The text representation of an `hstore`, used for input and output, includes zero or more *`key => value`* pairs separated by commas. Some examples:

```
k => v
foo => bar, baz => whatever
"1-a" => "anything at all"
```

The order of the pairs is not significant (and may not be reproduced on output). Whitespace between pairs or around the => sign is ignored. Double-quote keys and values that include whitespace, commas, =s or >s. To include a double quote or a backslash in a key or value, escape it with a backslash.

Each key in an `hstore` is unique. If you declare an `hstore` with duplicate keys, only one will be stored in the `hstore` and there is no guarantee as to which will be kept:

```
SELECT 'a=>1,a=>2'::hstore;
  hstore
```

```
----------
 "a"=>"1"
```

A value (but not a key) can be an SQL `NULL`. For example:

```
key => NULL
```

The `NULL` keyword is case-insensitive. Double-quote the `NULL` to treat it as the ordinary string "NULL".

> **Note:** Keep in mind that the `hstore` text format, when used for input, applies *before* any required quoting or escaping. If you are passing an `hstore` literal via a parameter, then no additional processing is needed. But if you're passing it as a quoted literal constant, then any single-quote characters and (depending on the setting of the `standard_conforming_strings` configuration parameter) backslash characters need to be escaped correctly. See Section 4.1.2.1 for more on the handling of string constants.

On output, double quotes always surround keys and values, even when it's not strictly necessary.

F.15.2. `hstore` Operators and Functions

The operators provided by the `hstore` module are shown in Table F-8, the functions in Table F-9.

Table F-8. `hstore` Operators

Operator	Description	Example	Result
`hstore -> text`	get value for key (`NULL` if not present)	`'a=>x, b=>y'::hstore -> 'a'`	`x`
`hstore -> text[]`	get values for keys (`NULL` if not present)	`'a=>x, b=>y, c=>z'::hstore -> ARRAY['c','a']`	`{"z","x"}`
`hstore || hstore`	concatenate `hstore`s	`'a=>b, c=>d'::hstore || 'c=>x, d=>q'::hstore`	`"a"=>"b", "c"=>"x", "d"=>"q"`
`hstore ? text`	does `hstore` contain key?	`'a=>1'::hstore ? 'a'`	`t`
`hstore ?& text[]`	does `hstore` contain all specified keys?	`'a=>1,b=>2'::hstore ?& ARRAY['a','b']`	`t`
`hstore ?| text[]`	does `hstore` contain any of the specified keys?	`'a=>1,b=>2'::hstore ?| ARRAY['b','c']`	`t`

Operator	Description	Example	Result
`hstore @> hstore`	does left operand contain right?	`'a=>b, b=>1, c=>NULL'::hstore @> 'b=>1'`	`t`
`hstore <@ hstore`	is left operand contained in right?	`'a=>c'::hstore <@ 'a=>b, b=>1, c=>NULL'`	`f`
`hstore - text`	delete key from left operand	`'a=>1, b=>2, c=>3'::hstore - 'b'::text`	`"a"=>"1", "c"=>"3"`
`hstore - text[]`	delete keys from left operand	`'a=>1, b=>2, c=>3'::hstore - ARRAY['a','b']`	`"c"=>"3"`
`hstore - hstore`	delete matching pairs from left operand	`'a=>1, b=>2, c=>3'::hstore - 'a=>4, b=>2'::hstore`	`"a"=>"1", "c"=>"3"`
`record #= hstore`	replace fields in `record` with matching values from `hstore`	see Examples section	
`%% hstore`	convert `hstore` to array of alternating keys and values	`%% 'a=>foo, b=>bar'::hstore`	`{a,foo,b,bar}`
`%# hstore`	convert `hstore` to two-dimensional key/value array	`%# 'a=>foo, b=>bar'::hstore`	`{{a,foo},{b,bar}}`

Note: Prior to PostgreSQL 8.2, the containment operators `@>` and `<@` were called `@` and `~`, respectively. These names are still available, but are deprecated and will eventually be removed. Notice that the old names are reversed from the convention formerly followed by the core geometric data types!

Table F-9. `hstore` Functions

Function	Return Type	Description	Example	Result
`hstore(record)`	`hstore`	construct an `hstore` from a record or row	`hstore(ROW(1,2))`	`f1=>1,f2=>2`

Function	Return Type	Description	Example	Result
`hstore(text[])`	`hstore`	construct an `hstore` from an array, which may be either a key/value array, or a two-dimensional array	`hstore(ARRAY['a','1','b','2']) || hstore(ARRAY[['c','3'],['d','4']])`	`"a"=>"1", "b"=>"2", c=>3, d=>4`
`hstore(text[], text[])`	`hstore`	construct an `hstore` from separate key and value arrays	`hstore(ARRAY['a','b'], ARRAY['1','2'])`	`"a"=>"1","b"=>"2"`
`hstore(text, text)`	`hstore`	make single-item `hstore`	`hstore('a', 'b')`	`"a"=>"b"`
`akeys(hstore)`	`text[]`	get `hstore`'s keys as an array	`akeys('a=>1,b=>2')`	`{a,b}`
`skeys(hstore)`	`setof text`	get `hstore`'s keys as a set	`skeys('a=>1,b=>2')`	`a` `b`
`avals(hstore)`	`text[]`	get `hstore`'s values as an array	`avals('a=>1,b=>2')`	`{1,2}`
`svals(hstore)`	`setof text`	get `hstore`'s values as a set	`svals('a=>1,b=>2')`	`1` `2`
`hstore_to_array(hstore)`	`text[]`	get `hstore`'s keys and values as an array of alternating keys and values	`hstore_to_array('a=>1,b=>2')`	`{a,1,b,2}`
`hstore_to_matrix(hstore)`	`text[]`	get `hstore`'s keys and values as a two-dimensional array	`hstore_to_matrix('a=>1,b=>2')`	`{{a,1},{b,2}}`
`hstore_to_json(hstore)`	`json`	get `hstore` as a `json` value, converting all non-null values to JSON strings	`hstore_to_json('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')`	`{"a key": "1", "b": "t", "c": null, "d": "12345", "e": "012345", "f": "1.234", "g": "2.345e+4"}`

Function	Return Type	Description	Example	Result
`hstore_to_jsonb(hstore)`	`jsonb`	get `hstore` as a `jsonb` value, converting all non-null values to JSON strings	`hstore_to_jsonb('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')`	`{"a key": "1", "b": "t", "c": null, "d": "12345", "e": "012345", "f": "1.234", "g": "2.345e+4"}`
`hstore_to_json_loose(hstore)`	`json`	get `hstore` as a `json` value, but attempt to distinguish numerical and Boolean values so they are unquoted in the JSON	`hstore_to_json_loose('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')`	`{"a key": 1, "b": true, "c": null, "d": 12345, "e": "012345", "f": 1.234, "g": 2.345e+4}`
`hstore_to_jsonb_loose(hstore)`	`jsonb`	get `hstore` as a `jsonb` value, but attempt to distinguish numerical and Boolean values so they are unquoted in the JSON	`hstore_to_jsonb_loose('"a key"=>1, b=>t, c=>null, d=>12345, e=>012345, f=>1.234, g=>2.345e+4')`	`{"a key": 1, "b": true, "c": null, "d": 12345, "e": "012345", "f": 1.234, "g": 2.345e+4}`
`slice(hstore, text[])`	`hstore`	extract a subset of an `hstore`	`slice('a=>1,b=>2,c=>3'::hstore, ARRAY['b','c','x'])`	`"b"=>"2", "c"=>"3"`

`each(hstore)`	`setof(key text, value text)`	get `hstore`'s keys and values as a set	`select * from each('a=>1,b=>2')`	`key | value`
				`----+-------`
				`a | 1`
				`b | 2`

`exist(hstore,text)`	`boolean`	does `hstore` contain key?	`exist('a=>1','a')`	`t`
`defined(hstore,text)`	`boolean`	does `hstore` contain non-`NULL` value for key?	`defined('a=>NULL','a')`	`f`
`delete(hstore,text)`	`hstore`	delete pair with matching key	`delete('a=>1,b=>2','b')`	`"a"=>"1"`
`delete(hstore,text[])`	`hstore`	delete pairs with matching keys	`delete('a=>1,b=>2,c=>3',ARRAY['a','b'])`	`"c"=>"3"`
`delete(hstore,hstore)`	`hstore`	delete pairs matching those in the second argument	`delete('a=>1,b=>2','a=>4,b=>2'::hstore)`	`"a"=>"1"`

Function	Return Type	Description	Example	Result
`populate_record(record,hstore)`	`record`	replace fields in `record` with matching values from `hstore`	see Examples section	

Note: The function `hstore_to_json` is used when an `hstore` value is cast to `json`. Likewise, `hstore_to_jsonb` is used when an `hstore` value is cast to `jsonb`.

Note: The function `populate_record` is actually declared with `anyelement`, not `record`, as its first argument, but it will reject non-record types with a run-time error.

F.15.3. Indexes

`hstore` has GiST and GIN index support for the `@>`, `?`, `?&` and `?|` operators. For example:

```
CREATE INDEX hidx ON testhstore USING GIST (h);

CREATE INDEX hidx ON testhstore USING GIN (h);
```

`hstore` also supports `btree` or `hash` indexes for the `=` operator. This allows `hstore` columns to be declared `UNIQUE`, or to be used in `GROUP BY`, `ORDER BY` or `DISTINCT` expressions. The sort ordering for `hstore` values is not particularly useful, but these indexes may be useful for equivalence lookups. Create indexes for `=` comparisons as follows:

```
CREATE INDEX hidx ON testhstore USING BTREE (h);

CREATE INDEX hidx ON testhstore USING HASH (h);
```

F.15.4. Examples

Add a key, or update an existing key with a new value:

```
UPDATE tab SET h = h || hstore('c', '3');
```

Delete a key:

```
UPDATE tab SET h = delete(h, 'k1');
```

Convert a `record` to an `hstore`:

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT hstore(t) FROM test AS t;
                  hstore
-------------------------------------------
 "col1"=>"123", "col2"=>"foo", "col3"=>"bar"
(1 row)
```

Convert an `hstore` to a predefined `record` type:

```
CREATE TABLE test (col1 integer, col2 text, col3 text);

SELECT * FROM populate_record(null::test,
                              '"col1"=>"456", "col2"=>"zzz"');
 col1 | col2 | col3
------+------+------
  456 | zzz  |
(1 row)
```

Modify an existing record using the values from an `hstore`:

```
CREATE TABLE test (col1 integer, col2 text, col3 text);
INSERT INTO test VALUES (123, 'foo', 'bar');

SELECT (r).* FROM (SELECT t #= '"col3"=>"baz"' AS r FROM test t) s;
 col1 | col2 | col3
------+------+------
  123 | foo  | baz
(1 row)
```

F.15.5. Statistics

The `hstore` type, because of its intrinsic liberality, could contain a lot of different keys. Checking for valid keys is the task of the application. The following examples demonstrate several techniques for checking keys and obtaining statistics.

Simple example:

```
SELECT * FROM each('aaa=>bq, b=>NULL, ""=>1');
```

Using a table:

```
SELECT (each(h)).key, (each(h)).value INTO stat FROM testhstore;
```

Online statistics:

```
SELECT key, count(*) FROM
  (SELECT (each(h)).key FROM testhstore) AS stat
  GROUP BY key
  ORDER BY count DESC, key;
    key    | count
-----------+-------
 line      |   883
 query     |   207
 pos       |   203
 node      |   202
 space     |   197
 status    |   195
 public    |   194
 title     |   190
 org       |   189
.................
```

F.15.6. Compatibility

As of PostgreSQL 9.0, `hstore` uses a different internal representation than previous versions. This presents no obstacle for dump/restore upgrades since the text representation (used in the dump) is unchanged.

In the event of a binary upgrade, upward compatibility is maintained by having the new code recognize old-format data. This will entail a slight performance penalty when processing data that has not yet been modified by the new code. It is possible to force an upgrade of all values in a table column by doing an `UPDATE` statement as follows:

```
UPDATE tablename SET hstorecol = hstorecol || '';
```

Another way to do it is:

```
ALTER TABLE tablename ALTER hstorecol TYPE hstore USING hstorecol || '';
```

The `ALTER TABLE` method requires an exclusive lock on the table, but does not result in bloating the table with old row versions.

F.15.7. Transforms

Additional extensions are available that implement transforms for the `hstore` type for the languages PL/Perl and PL/Python. The extensions for PL/Perl are called `hstore_plperl` and `hstore_plperlu`, for trusted and untrusted PL/Perl. If you install these transforms and specify them when creating a function, `hstore` values are mapped to Perl hashes. The extensions for PL/Python are called

hstore_plpythonu, hstore_plpython2u, and hstore_plpython3u (see Section 43.1 for the PL/Python naming convention). If you use them, hstore values are mapped to Python dictionaries.

Caution

It is strongly recommended that the transform extensions be installed in the same schema as hstore. Otherwise there are installation-time security hazards if a transform extension's schema contains objects defined by a hostile user.

F.15.8. Authors

Oleg Bartunov <oleg@sai.msu.su>, Moscow, Moscow University, Russia

Teodor Sigaev <teodor@sigaev.ru>, Moscow, Delta-Soft Ltd., Russia

Additional enhancements by Andrew Gierth <andrew@tao11.riddles.org.uk>, United Kingdom

F.16. intagg

The intagg module provides an integer aggregator and an enumerator. intagg is now obsolete, because there are built-in functions that provide a superset of its capabilities. However, the module is still provided as a compatibility wrapper around the built-in functions.

F.16.1. Functions

The aggregator is an aggregate function int_array_aggregate(integer) that produces an integer array containing exactly the integers it is fed. This is a wrapper around array_agg, which does the same thing for any array type.

The enumerator is a function int_array_enum(integer[]) that returns setof integer. It is essentially the reverse operation of the aggregator: given an array of integers, expand it into a set of rows. This is a wrapper around unnest, which does the same thing for any array type.

F.16.2. Sample Uses

Many database systems have the notion of a one to many table. Such a table usually sits between two indexed tables, for example:

```
CREATE TABLE left (id INT PRIMARY KEY, ...);
CREATE TABLE right (id INT PRIMARY KEY, ...);
CREATE TABLE one_to_many(left INT REFERENCES left, right INT REFERENCES right);
```

It is typically used like this:

```
SELECT right.* from right JOIN one_to_many ON (right.id = one_to_many.right)
  WHERE one_to_many.left = item;
```

This will return all the items in the right hand table for an entry in the left hand table. This is a very common construct in SQL.

Now, this methodology can be cumbersome with a very large number of entries in the `one_to_many` table. Often, a join like this would result in an index scan and a fetch for each right hand entry in the table for a particular left hand entry. If you have a very dynamic system, there is not much you can do. However, if you have some data which is fairly static, you can create a summary table with the aggregator.

```
CREATE TABLE summary AS
  SELECT left, int_array_aggregate(right) AS right
  FROM one_to_many
  GROUP BY left;
```

This will create a table with one row per left item, and an array of right items. Now this is pretty useless without some way of using the array; that's why there is an array enumerator. You can do

```
SELECT left, int_array_enum(right) FROM summary WHERE left = item;
```

The above query using `int_array_enum` produces the same results as

```
SELECT left, right FROM one_to_many WHERE left = item;
```

The difference is that the query against the summary table has to get only one row from the table, whereas the direct query against `one_to_many` must index scan and fetch a row for each entry.

On one system, an `EXPLAIN` showed a query with a cost of 8488 was reduced to a cost of 329. The original query was a join involving the `one_to_many` table, which was replaced by:

```
SELECT right, count(right) FROM
  ( SELECT left, int_array_enum(right) AS right
    FROM summary JOIN (SELECT left FROM left_table WHERE left = item) AS lefts
         ON (summary.left = lefts.left)
  ) AS list
  GROUP BY right
  ORDER BY count DESC;
```

F.17. intarray

The `intarray` module provides a number of useful functions and operators for manipulating null-free arrays of integers. There is also support for indexed searches using some of the operators.

All of these operations will throw an error if a supplied array contains any NULL elements.

Many of these operations are only sensible for one-dimensional arrays. Although they will accept input arrays of more dimensions, the data is treated as though it were a linear array in storage order.

F.17.1. `intarray` Functions and Operators

The functions provided by the `intarray` module are shown in Table F-10, the operators in Table F-11.

Table F-10. `intarray` Functions

Function	Return Type	Description	Example	Result
`icount(int[])`	`int`	number of elements in array	`icount('{1,2,3}'::int[])`	`3`
`sort(int[], text dir)`	`int[]`	sort array — `dir` must be `asc` or `desc`	`sort('{1,2,3}'::int[], 'desc')`	`{3,2,1}`
`sort(int[])`	`int[]`	sort in ascending order	`sort(array[11,77,44])`	`{11,44,77}`
`sort_asc(int[])`	`int[]`	sort in ascending order		
`sort_desc(int[])`	`int[]`	sort in descending order		
`uniq(int[])`	`int[]`	remove adjacent duplicates	`uniq(sort('{1,2,3,11,22,33}'::int[]))`	`{1,2,3,11,22,33}`
`idx(int[], int item)`	`int`	index of first element matching `item` (0 if none)	`idx(array[11,22,33,22,11], 22)`	`2`
`subarray(int[], int start, int len)`	`int[]`	portion of array starting at position `start`, `len` elements	`subarray('{1,2,3,2,1}'::int[], 2, 3)`	`{2,3,2}`
`subarray(int[], int start)`	`int[]`	portion of array starting at position `start`	`subarray('{1,2,3,2,1}'::int[], 2)`	`{2,3,2,1}`
`intset(int)`	`int[]`	make single-element array	`intset(42)`	`{42}`

Table F-11. `intarray` Operators

Operator	Returns	Description
`int[] && int[]`	`boolean`	overlap — `true` if arrays have at least one common element
`int[] @> int[]`	`boolean`	contains — `true` if left array contains right array
`int[] <@ int[]`	`boolean`	contained — `true` if left array is contained in right array
`# int[]`	`int`	number of elements in array

Operator	Returns	Description
int[] # int	int	index (same as idx function)
int[] + int	int[]	push element onto array (add it to end of array)
int[] + int[]	int[]	array concatenation (right array added to the end of left one)
int[] - int	int[]	remove entries matching right argument from array
int[] - int[]	int[]	remove elements of right array from left
int[] | int	int[]	union of arguments
int[] | int[]	int[]	union of arrays
int[] & int[]	int[]	intersection of arrays
int[] @@ query_int	boolean	true if array satisfies query (see below)
query_int ~~ int[]	boolean	true if array satisfies query (commutator of @@)

(Before PostgreSQL 8.2, the containment operators @> and <@ were respectively called @ and ~. These names are still available, but are deprecated and will eventually be retired. Notice that the old names are reversed from the convention formerly followed by the core geometric data types!)

The operators &&, @> and <@ are equivalent to PostgreSQL's built-in operators of the same names, except that they work only on integer arrays that do not contain nulls, while the built-in operators work for any array type. This restriction makes them faster than the built-in operators in many cases.

The @@ and ~~ operators test whether an array satisfies a *query*, which is expressed as a value of a specialized data type query_int. A *query* consists of integer values that are checked against the elements of the array, possibly combined using the operators & (AND), | (OR), and ! (NOT). Parentheses can be used as needed. For example, the query 1&(2|3) matches arrays that contain 1 and also contain either 2 or 3.

F.17.2. Index Support

intarray provides index support for the &&, @>, <@, and @@ operators, as well as regular array equality.

Two GiST index operator classes are provided: gist__int_ops (used by default) is suitable for small- to medium-size data sets, while gist__intbig_ops uses a larger signature and is more suitable for indexing large data sets (i.e., columns containing a large number of distinct array values). The implementation uses an RD-tree data structure with built-in lossy compression.

There is also a non-default GIN operator class gin__int_ops supporting the same operators.

The choice between GiST and GIN indexing depends on the relative performance characteristics of GiST and GIN, which are discussed elsewhere. As a rule of thumb, a GIN index is faster to search than a GiST index, but slower to build or update; so GIN is better suited for static data and GiST for often-updated data.

F.17.3. Example

```
-- a message can be in one or more "sections"
CREATE TABLE message (mid INT PRIMARY KEY, sections INT[], ...);

-- create specialized index
CREATE INDEX message_rdtree_idx ON message USING GIST (sections gist__int_ops);

-- select messages in section 1 OR 2 - OVERLAP operator
SELECT message.mid FROM message WHERE message.sections && '{1,2}';

-- select messages in sections 1 AND 2 - CONTAINS operator
SELECT message.mid FROM message WHERE message.sections @> '{1,2}';

-- the same, using QUERY operator
SELECT message.mid FROM message WHERE message.sections @@ '1&2'::query_int;
```

F.17.4. Benchmark

The source directory `contrib/intarray/bench` contains a benchmark test suite. To run:

```
cd .../bench
createdb TEST
psql TEST < ../_int.sql
./create_test.pl | psql TEST
./bench.pl
```

The `bench.pl` script has numerous options, which are displayed when it is run without any arguments.

F.17.5. Authors

All work was done by Teodor Sigaev (<`teodor@sigaev.ru`>) and Oleg Bartunov (<`oleg@sai.msu.su`>). See http://www.sai.msu.su/~megera/postgres/gist/ for additional information. Andrey Oktyabrski did a great work on adding new functions and operations.

F.18. isn

The `isn` module provides data types for the following international product numbering standards: EAN13, UPC, ISBN (books), ISMN (music), and ISSN (serials). Numbers are validated on input according to a hard-coded list of prefixes; this list of prefixes is also used to hyphenate numbers on output. Since new prefixes are assigned from time to time, the list of prefixes may be out of date. It is hoped that a future version of this module will obtain the prefix list from one or more tables that can be easily updated by users as needed; however, at present, the list can only be updated by modifying the source code and recompiling. Alternatively, prefix validation and hyphenation support may be dropped from a future version of this module.

F.18.1. Data Types

Table F-12 shows the data types provided by the `isn` module.

Table F-12. `isn` Data Types

Data Type	Description
`EAN13`	European Article Numbers, always displayed in the EAN13 display format
`ISBN13`	International Standard Book Numbers to be displayed in the new EAN13 display format
`ISMN13`	International Standard Music Numbers to be displayed in the new EAN13 display format
`ISSN13`	International Standard Serial Numbers to be displayed in the new EAN13 display format
`ISBN`	International Standard Book Numbers to be displayed in the old short display format
`ISMN`	International Standard Music Numbers to be displayed in the old short display format
`ISSN`	International Standard Serial Numbers to be displayed in the old short display format
`UPC`	Universal Product Codes

Some notes:

1. ISBN13, ISMN13, ISSN13 numbers are all EAN13 numbers.
2. EAN13 numbers aren't always ISBN13, ISMN13 or ISSN13 (some are).
3. Some ISBN13 numbers can be displayed as ISBN.
4. Some ISMN13 numbers can be displayed as ISMN.
5. Some ISSN13 numbers can be displayed as ISSN.
6. UPC numbers are a subset of the EAN13 numbers (they are basically EAN13 without the first `0` digit).
7. All UPC, ISBN, ISMN and ISSN numbers can be represented as EAN13 numbers.

Internally, all these types use the same representation (a 64-bit integer), and all are interchangeable. Multiple types are provided to control display formatting and to permit tighter validity checking of input that is supposed to denote one particular type of number.

The `ISBN`, `ISMN`, and `ISSN` types will display the short version of the number (ISxN 10) whenever it's possible, and will show ISxN 13 format for numbers that do not fit in the short version. The `EAN13`, `ISBN13`, `ISMN13` and `ISSN13` types will always display the long version of the ISxN (EAN13).

F.18.2. Casts

The `isn` module provides the following pairs of type casts:

- ISBN13 <=> EAN13
- ISMN13 <=> EAN13
- ISSN13 <=> EAN13
- ISBN <=> EAN13
- ISMN <=> EAN13
- ISSN <=> EAN13
- UPC <=> EAN13
- ISBN <=> ISBN13
- ISMN <=> ISMN13
- ISSN <=> ISSN13

When casting from `EAN13` to another type, there is a run-time check that the value is within the domain of the other type, and an error is thrown if not. The other casts are simply relabelings that will always succeed.

F.18.3. Functions and Operators

The `isn` module provides the standard comparison operators, plus B-tree and hash indexing support for all these data types. In addition there are several specialized functions; shown in Table F-13. In this table, `isn` means any one of the module's data types.

Table F-13. `isn` Functions

Function	Returns	Description
`isn_weak(boolean)`	`boolean`	Sets the weak input mode (returns new setting)
`isn_weak()`	`boolean`	Gets the current status of the weak mode
`make_valid(isn)`	`isn`	Validates an invalid number (clears the invalid flag)
`is_valid(isn)`	`boolean`	Checks for the presence of the invalid flag

Weak mode is used to be able to insert invalid data into a table. Invalid means the check digit is wrong, not that there are missing numbers.

Why would you want to use the weak mode? Well, it could be that you have a huge collection of ISBN numbers, and that there are so many of them that for weird reasons some have the wrong check digit (perhaps the numbers were scanned from a printed list and the OCR got the numbers wrong, perhaps the numbers were manually captured... who knows). Anyway, the point is you might want to clean the mess

up, but you still want to be able to have all the numbers in your database and maybe use an external tool to locate the invalid numbers in the database so you can verify the information and validate it more easily; so for example you'd want to select all the invalid numbers in the table.

When you insert invalid numbers in a table using the weak mode, the number will be inserted with the corrected check digit, but it will be displayed with an exclamation mark (`!`) at the end, for example `0-11-000322-5!`. This invalid marker can be checked with the `is_valid` function and cleared with the `make_valid` function.

You can also force the insertion of invalid numbers even when not in the weak mode, by appending the `!` character at the end of the number.

Another special feature is that during input, you can write `?` in place of the check digit, and the correct check digit will be inserted automatically.

F.18.4. Examples

```
--Using the types directly:
SELECT isbn('978-0-393-04002-9');
SELECT isbn13('0901690546');
SELECT issn('1436-4522');

--Casting types:
-- note that you can only cast from ean13 to another type when the
-- number would be valid in the realm of the target type;
-- thus, the following will NOT work: select isbn(ean13('0220356483481'));
-- but these will:
SELECT upc(ean13('0220356483481'));
SELECT ean13(upc('220356483481'));

--Create a table with a single column to hold ISBN numbers:
CREATE TABLE test (id isbn);
INSERT INTO test VALUES('9780393040029');

--Automatically calculate check digits (observe the '?'):
INSERT INTO test VALUES('220500896?');
INSERT INTO test VALUES('978055215372?');

SELECT issn('3251231?');
SELECT ismn('979047213542?');

--Using the weak mode:
SELECT isn_weak(true);
INSERT INTO test VALUES('978-0-11-000533-4');
INSERT INTO test VALUES('9780141219307');
INSERT INTO test VALUES('2-205-00876-X');
SELECT isn_weak(false);

SELECT id FROM test WHERE NOT is_valid(id);
UPDATE test SET id = make_valid(id) WHERE id = '2-205-00876-X!';

SELECT * FROM test;
```

```
SELECT isbn13(id) FROM test;
```

F.18.5. Bibliography

The information to implement this module was collected from several sites, including:

- https://www.isbn-international.org/
- https://www.issn.org/
- https://www.ismn-international.org/
- https://www.wikipedia.org/

The prefixes used for hyphenation were also compiled from:

- http://www.gs1.org/productssolutions/idkeys/support/prefix_list.html
- http://en.wikipedia.org/wiki/List_of_ISBN_identifier_groups
- https://www.isbn-international.org/content/isbn-users-manual
- http://en.wikipedia.org/wiki/International_Standard_Music_Number
- http://www.ismn-international.org/ranges.html

Care was taken during the creation of the algorithms and they were meticulously verified against the suggested algorithms in the official ISBN, ISMN, ISSN User Manuals.

F.18.6. Author

Germán Méndez Bravo (Kronuz), 2004 - 2006

This module was inspired by Garrett A. Wollman's `isbn_issn` code.

F.19. lo

The `lo` module provides support for managing Large Objects (also called LOs or BLOBs). This includes a data type `lo` and a trigger `lo_manage`.

F.19.1. Rationale

One of the problems with the JDBC driver (and this affects the ODBC driver also), is that the specification assumes that references to BLOBs (Binary Large OBjects) are stored within a table, and if that entry is changed, the associated BLOB is deleted from the database.

As PostgreSQL stands, this doesn't occur. Large objects are treated as objects in their own right; a table entry can reference a large object by OID, but there can be multiple table entries referencing the same

large object OID, so the system doesn't delete the large object just because you change or remove one such entry.

Now this is fine for PostgreSQL-specific applications, but standard code using JDBC or ODBC won't delete the objects, resulting in orphan objects — objects that are not referenced by anything, and simply occupy disk space.

The `lo` module allows fixing this by attaching a trigger to tables that contain LO reference columns. The trigger essentially just does a `lo_unlink` whenever you delete or modify a value referencing a large object. When you use this trigger, you are assuming that there is only one database reference to any large object that is referenced in a trigger-controlled column!

The module also provides a data type `lo`, which is really just a domain of the `oid` type. This is useful for differentiating database columns that hold large object references from those that are OIDs of other things. You don't have to use the `lo` type to use the trigger, but it may be convenient to use it to keep track of which columns in your database represent large objects that you are managing with the trigger. It is also rumored that the ODBC driver gets confused if you don't use `lo` for BLOB columns.

F.19.2. How to Use It

Here's a simple example of usage:

```
CREATE TABLE image (title TEXT, raster lo);

CREATE TRIGGER t_raster BEFORE UPDATE OR DELETE ON image
    FOR EACH ROW EXECUTE PROCEDURE lo_manage(raster);
```

For each column that will contain unique references to large objects, create a `BEFORE UPDATE OR DELETE` trigger, and give the column name as the sole trigger argument. You can also restrict the trigger to only execute on updates to the column by using `BEFORE UPDATE OF` *`column_name`*. If you need multiple `lo` columns in the same table, create a separate trigger for each one, remembering to give a different name to each trigger on the same table.

F.19.3. Limitations

- Dropping a table will still orphan any objects it contains, as the trigger is not executed. You can avoid this by preceding the `DROP TABLE` with `DELETE FROM` *`table`*.

  `TRUNCATE` has the same hazard.

  If you already have, or suspect you have, orphaned large objects, see the vacuumlo module to help you clean them up. It's a good idea to run vacuumlo occasionally as a back-stop to the `lo_manage` trigger.

- Some frontends may create their own tables, and will not create the associated trigger(s). Also, users may not remember (or know) to create the triggers.

F.19.4. Author

Peter Mount <peter@retep.org.uk>

F.20. ltree

This module implements a data type `ltree` for representing labels of data stored in a hierarchical tree-like structure. Extensive facilities for searching through label trees are provided.

F.20.1. Definitions

A *label* is a sequence of alphanumeric characters and underscores (for example, in C locale the characters `A-Za-z0-9_` are allowed). Labels must be less than 256 characters long.

Examples: `42`, `Personal_Services`

A *label path* is a sequence of zero or more labels separated by dots, for example `L1.L2.L3`, representing a path from the root of a hierarchical tree to a particular node. The length of a label path cannot exceed 65535 labels.

Example: `Top.Countries.Europe.Russia`

The `ltree` module provides several data types:

- `ltree` stores a label path.
- `lquery` represents a regular-expression-like pattern for matching `ltree` values. A simple word matches that label within a path. A star symbol (`*`) matches zero or more labels. For example:

  ```
  foo         Match the exact label path foo
  *.foo.*     Match any label path containing the label foo
  *.foo       Match any label path whose last label is foo
  ```

  Star symbols can also be quantified to restrict how many labels they can match:

  ```
  *{n}        Match exactly n labels
  *{n,}       Match at least n labels
  *{n,m}      Match at least n but not more than m labels
  *{,m}       Match at most m labels -- same as  *{0,m}
  ```

  There are several modifiers that can be put at the end of a non-star label in `lquery` to make it match more than just the exact match:

  ```
  @           Match case-insensitively, for example a@ matches A
  *           Match any label with this prefix, for example foo* matches foobar
  %           Match initial underscore-separated words
  ```

  The behavior of `%` is a bit complicated. It tries to match words rather than the entire label. For example `foo_bar%` matches `foo_bar_baz` but not `foo_barbaz`. If combined with `*`, prefix matching applies to each word separately, for example `foo_bar%*` matches `foo1_bar2_baz` but not `foo1_br2_baz`.

  Also, you can write several possibly-modified labels separated with `|` (OR) to match any of those labels, and you can put `!` (NOT) at the start to match any label that doesn't match any of the alternatives.

Here's an annotated example of `lquery`:

```
Top.*{0,2}.sport*@.!football|tennis.Russ*|Spain
a.  b.     c.      d.               e.
```

This query will match any label path that:

a. begins with the label `Top`

b. and next has zero to two labels before

c. a label beginning with the case-insensitive prefix `sport`

d. then a label not matching `football` nor `tennis`

e. and then ends with a label beginning with `Russ` or exactly matching `Spain`.

- `ltxtquery` represents a full-text-search-like pattern for matching `ltree` values. An `ltxtquery` value contains words, possibly with the modifiers `@`, `*`, `%` at the end; the modifiers have the same meanings as in `lquery`. Words can be combined with `&` (AND), `|` (OR), `!` (NOT), and parentheses. The key difference from `lquery` is that `ltxtquery` matches words without regard to their position in the label path.

  Here's an example `ltxtquery`:

  ```
  Europe & Russia*@ & !Transportation
  ```

  This will match paths that contain the label `Europe` and any label beginning with `Russia` (case-insensitive), but not paths containing the label `Transportation`. The location of these words within the path is not important. Also, when `%` is used, the word can be matched to any underscore-separated word within a label, regardless of position.

Note: `ltxtquery` allows whitespace between symbols, but `ltree` and `lquery` do not.

F.20.2. Operators and Functions

Type `ltree` has the usual comparison operators =, <>, <, >, <=, >=. Comparison sorts in the order of a tree traversal, with the children of a node sorted by label text. In addition, the specialized operators shown in Table F-14 are available.

Table F-14. `ltree` Operators

Operator	Returns	Description
`ltree @> ltree`	`boolean`	is left argument an ancestor of right (or equal)?
`ltree <@ ltree`	`boolean`	is left argument a descendant of right (or equal)?
`ltree ~ lquery`	`boolean`	does `ltree` match `lquery`?
`lquery ~ ltree`	`boolean`	does `ltree` match `lquery`?
`ltree ? lquery[]`	`boolean`	does `ltree` match any `lquery` in array?

Operator	Returns	Description
lquery[] ? ltree	boolean	does ltree match any lquery in array?
ltree @ ltxtquery	boolean	does ltree match ltxtquery?
ltxtquery @ ltree	boolean	does ltree match ltxtquery?
ltree || ltree	ltree	concatenate ltree paths
ltree || text	ltree	convert text to ltree and concatenate
text || ltree	ltree	convert text to ltree and concatenate
ltree[] @> ltree	boolean	does array contain an ancestor of ltree?
ltree <@ ltree[]	boolean	does array contain an ancestor of ltree?
ltree[] <@ ltree	boolean	does array contain a descendant of ltree?
ltree @> ltree[]	boolean	does array contain a descendant of ltree?
ltree[] ~ lquery	boolean	does array contain any path matching lquery?
lquery ~ ltree[]	boolean	does array contain any path matching lquery?
ltree[] ? lquery[]	boolean	does ltree array contain any path matching any lquery?
lquery[] ? ltree[]	boolean	does ltree array contain any path matching any lquery?
ltree[] @ ltxtquery	boolean	does array contain any path matching ltxtquery?
ltxtquery @ ltree[]	boolean	does array contain any path matching ltxtquery?
ltree[] ?@> ltree	ltree	first array entry that is an ancestor of ltree; NULL if none
ltree[] ?<@ ltree	ltree	first array entry that is a descendant of ltree; NULL if none
ltree[] ?~ lquery	ltree	first array entry that matches lquery; NULL if none
ltree[] ?@ ltxtquery	ltree	first array entry that matches ltxtquery; NULL if none

The operators <@, @>, @ and ~ have analogues ^<@, ^@>, ^@, ^~, which are the same except they do not use indexes. These are useful only for testing purposes.

The available functions are shown in Table F-15.

Table F-15. `ltree` Functions

Function	Return Type	Description	Example	Result
`subltree(ltree, int start, int end)`	`ltree`	subpath of `ltree` from position `start` to position `end`-1 (counting from 0)	`subltree('Top.Child1.Child2',1,2)`	`Child1.Child2`
`subpath(ltree, int offset, int len)`	`ltree`	subpath of `ltree` starting at position `offset`, length `len`. If `offset` is negative, subpath starts that far from the end of the path. If `len` is negative, leaves that many labels off the end of the path.	`subpath('Top.Child1.Child2',0,2)`	`Top.Child1`
`subpath(ltree, int offset)`	`ltree`	subpath of `ltree` starting at position `offset`, extending to end of path. If `offset` is negative, subpath starts that far from the end of the path.	`subpath('Top.Child1.Child2',1)`	`Child1.Child2`
`nlevel(ltree)`	`integer`	number of labels in path	`nlevel('Top.Child1.Child2')`	3
`index(ltree a, ltree b)`	`integer`	position of first occurrence of `b` in `a`; -1 if not found	`index('0.1.2.3.5.4.5.6.8.5.6.8','5.6')`	6
`index(ltree a, ltree b, int offset)`	`integer`	position of first occurrence of `b` in `a`, searching starting at `offset`; negative `offset` means start `-offset` labels from the end of the path	`index('0.1.2.3.5.4.5.6.8.5.6.8','5.6',-4)`	9
`text2ltree(text)`	`ltree`	cast `text` to `ltree`		

Function	Return Type	Description	Example	Result
`ltree2text(ltree)`	`text`	cast `ltree` to `text`		
`lca(ltree, ltree, ...)`	`ltree`	longest common ancestor of paths (up to 8 arguments supported)	`lca('1.2.3','1.2.4.5.6')`	`1.2`
`lca(ltree[])`	`ltree`	longest common ancestor of paths in array	`lca(array['1.2.3'::ltree,'1.2.3.4'])`	`1.2`

F.20.3. Indexes

`ltree` supports several types of indexes that can speed up the indicated operators:

- B-tree index over `ltree`: <, <=, =, >=, >
- GiST index over `ltree`: <, <=, =, >=, >, `@>`, `<@`, `@`, `~`, `?`

  Example of creating such an index:

  ```
  CREATE INDEX path_gist_idx ON test USING GIST (path);
  ```

- GiST index over `ltree[]`: `ltree[] <@ ltree`, `ltree @> ltree[]`, `@`, `~`, `?`

  Example of creating such an index:

  ```
  CREATE INDEX path_gist_idx ON test USING GIST (array_path);
  ```

  Note: This index type is lossy.

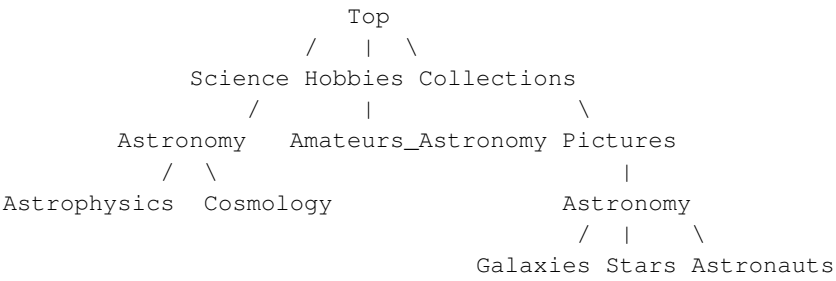
F.20.4. Example

This example uses the following data (also available in file `contrib/ltree/ltreetest.sql` in the source distribution):

```
CREATE TABLE test (path ltree);
INSERT INTO test VALUES ('Top');
INSERT INTO test VALUES ('Top.Science');
INSERT INTO test VALUES ('Top.Science.Astronomy');
INSERT INTO test VALUES ('Top.Science.Astronomy.Astrophysics');
INSERT INTO test VALUES ('Top.Science.Astronomy.Cosmology');
INSERT INTO test VALUES ('Top.Hobbies');
INSERT INTO test VALUES ('Top.Hobbies.Amateurs_Astronomy');
INSERT INTO test VALUES ('Top.Collections');
INSERT INTO test VALUES ('Top.Collections.Pictures');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Stars');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Galaxies');
INSERT INTO test VALUES ('Top.Collections.Pictures.Astronomy.Astronauts');
CREATE INDEX path_gist_idx ON test USING GIST (path);
```

```
CREATE INDEX path_idx ON test USING BTREE (path);
```

Now, we have a table `test` populated with data describing the hierarchy shown below:



We can do inheritance:

```
ltreetest=> SELECT path FROM test WHERE path <@ 'Top.Science';
                path
-----------------------------------
 Top.Science
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(4 rows)
```

Here are some examples of path matching:

```
ltreetest=> SELECT path FROM test WHERE path ~ '*.Astronomy.*';
                     path
-----------------------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Collections.Pictures.Astronomy
 Top.Collections.Pictures.Astronomy.Stars
 Top.Collections.Pictures.Astronomy.Galaxies
 Top.Collections.Pictures.Astronomy.Astronauts
(7 rows)

ltreetest=> SELECT path FROM test WHERE path ~ '*.!pictures@.*.Astronomy.*';
                path
-----------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)
```

Here are some examples of full text search:

```
ltreetest=> SELECT path FROM test WHERE path @ 'Astro*% & !pictures@';
                path
```

```
-----------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
 Top.Hobbies.Amateurs_Astronomy
(4 rows)

ltreetest=> SELECT path FROM test WHERE path @ 'Astro* & !pictures@';
               path
-----------------------------------
 Top.Science.Astronomy
 Top.Science.Astronomy.Astrophysics
 Top.Science.Astronomy.Cosmology
(3 rows)
```

Path construction using functions:

```
ltreetest=> SELECT subpath(path,0,2)||'Space'||subpath(path,2) FROM test WHERE path <@ 'Top
                ?column?
----------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

We could simplify this by creating a SQL function that inserts a label at a specified position in a path:

```
CREATE FUNCTION ins_label(ltree, int, text) RETURNS ltree
    AS 'select subpath($1,0,$2) || $3 || subpath($1,$2);'
    LANGUAGE SQL IMMUTABLE;

ltreetest=> SELECT ins_label(path,2,'Space') FROM test WHERE path <@ 'Top.Science.Astronomy
                ins_label
----------------------------------------
 Top.Science.Space.Astronomy
 Top.Science.Space.Astronomy.Astrophysics
 Top.Science.Space.Astronomy.Cosmology
(3 rows)
```

F.20.5. Transforms

Additional extensions are available that implement transforms for the `ltree` type for PL/Python. The extensions are called `ltree_plpythonu`, `ltree_plpython2u`, and `ltree_plpython3u` (see Section 43.1 for the PL/Python naming convention). If you install these transforms and specify them when creating a function, `ltree` values are mapped to Python lists. (The reverse is currently not supported, however.)

Caution

It is strongly recommended that the transform extensions be installed in the same schema as `ltree`. Otherwise there are installation-time security hazards if a transform extension's schema contains objects defined by a hostile user.

F.20.6. Authors

All work was done by Teodor Sigaev (<teodor@stack.net>) and Oleg Bartunov (<oleg@sai.msu.su>). See http://www.sai.msu.su/~megera/postgres/gist/ for additional information. Authors would like to thank Eugeny Rodichev for helpful discussions. Comments and bug reports are welcome.

F.21. pageinspect

The `pageinspect` module provides functions that allow you to inspect the contents of database pages at a low level, which is useful for debugging purposes. All of these functions may be used only by superusers.

F.21.1. Functions

`get_raw_page(relname text, fork text, blkno int) returns bytea`

`get_raw_page` reads the specified block of the named relation and returns a copy as a `bytea` value. This allows a single time-consistent copy of the block to be obtained. *`fork`* should be `'main'` for the main data fork, `'fsm'` for the free space map, `'vm'` for the visibility map, or `'init'` for the initialization fork.

`get_raw_page(relname text, blkno int) returns bytea`

A shorthand version of `get_raw_page`, for reading from the main fork. Equivalent to `get_raw_page(relname, 'main', blkno)`

`page_header(page bytea) returns record`

`page_header` shows fields that are common to all PostgreSQL heap and index pages.

A page image obtained with `get_raw_page` should be passed as argument. For example:

```
test=# SELECT * FROM page_header(get_raw_page('pg_class', 0));
    lsn    | checksum | flags  | lower | upper | special | pagesize | version | prune_xi
-----------+----------+--------+-------+-------+---------+----------+---------+---------
 0/24A1B50 |        1 |      1 |   232 |   368 |    8192 |     8192 |       4 |
```

The returned columns correspond to the fields in the `PageHeaderData` struct. See `src/include/storage/bufpage.h` for details.

`heap_page_items(page bytea) returns setof record`

`heap_page_items` shows all line pointers on a heap page. For those line pointers that are in use, tuple headers are also shown. All tuples are shown, whether or not the tuples were visible to an MVCC snapshot at the time the raw page was copied.

A heap page image obtained with `get_raw_page` should be passed as argument. For example:

```
test=# SELECT * FROM heap_page_items(get_raw_page('pg_class', 0));
```

See `src/include/storage/itemid.h` and `src/include/access/htup_details.h` for explanations of the fields returned.

`bt_metap(relname text) returns record`

`bt_metap` returns information about a B-tree index's metapage. For example:

```
test=# SELECT * FROM bt_metap('pg_cast_oid_index');
-[ RECORD 1 ]-----
magic     | 340322
version   | 2
root      | 1
level     | 0
fastroot  | 1
fastlevel | 0
```

`bt_page_stats(relname text, blkno int) returns record`

`bt_page_stats` returns summary information about single pages of B-tree indexes. For example:

```
test=# SELECT * FROM bt_page_stats('pg_cast_oid_index', 1);
-[ RECORD 1 ]-+-----
blkno         | 1
type          | 1
live_items    | 256
dead_items    | 0
avg_item_size | 12
page_size     | 8192
free_size     | 4056
btpo_prev     | 0
btpo_next     | 0
btpo          | 0
btpo_flags    | 3
```

`bt_page_items(relname text, blkno int) returns setof record`

`bt_page_items` returns detailed information about all of the items on a B-tree index page. For example:

```
test=# SELECT * FROM bt_page_items('pg_cast_oid_index', 1);
 itemoffset |  ctid   | itemlen | nulls | vars |    data
------------+---------+---------+-------+------+-------------
          1 | (0,1)   |      12 | f     | f    | 23 27 00 00
          2 | (0,2)   |      12 | f     | f    | 24 27 00 00
          3 | (0,3)   |      12 | f     | f    | 25 27 00 00
          4 | (0,4)   |      12 | f     | f    | 26 27 00 00
          5 | (0,5)   |      12 | f     | f    | 27 27 00 00
          6 | (0,6)   |      12 | f     | f    | 28 27 00 00
          7 | (0,7)   |      12 | f     | f    | 29 27 00 00
```

```
          8 | (0,8)   |      12 | f     | f    | 2a 27 00 00
```

In a B-tree leaf page, `ctid` points to a heap tuple. In an internal page, the block number part of `ctid` points to another page in the index itself, while the offset part (the second number) is ignored and is usually 1.

Note that the first item on any non-rightmost page (any page with a non-zero value in the `btpo_next` field) is the page's "high key", meaning its `data` serves as an upper bound on all items appearing on the page, while its `ctid` field is meaningless. Also, on non-leaf pages, the first real data item (the first item that is not a high key) is a "minus infinity" item, with no actual value in its `data` field. Such an item does have a valid downlink in its `ctid` field, however.

`brin_page_type(page bytea) returns text`

`brin_page_type` returns the page type of the given BRIN index page, or throws an error if the page is not a valid BRIN page. For example:

```
test=# SELECT brin_page_type(get_raw_page('brinidx', 0));
 brin_page_type
----------------
 meta
```

`brin_metapage_info(page bytea) returns record`

`brin_metapage_info` returns assorted information about a BRIN index metapage. For example:

```
test=# SELECT * FROM brin_metapage_info(get_raw_page('brinidx', 0));
   magic    | version | pagesperrange | lastrevmappage
------------+---------+---------------+---------------
 0xA8109CFA |       1 |             4 |              2
```

`brin_revmap_data(page bytea) returns setof tid`

`brin_revmap_data` returns the list of tuple identifiers in a BRIN index range map page. For example:

```
test=# SELECT * FROM brin_revmap_data(get_raw_page('brinidx', 2)) limit 5;
  pages
---------
 (6,137)
 (6,138)
 (6,139)
 (6,140)
 (6,141)
```

`brin_page_items(page bytea, index oid) returns setof record`

`brin_page_items` returns the data stored in the BRIN data page. For example:

```
test=# SELECT * FROM brin_page_items(get_raw_page('brinidx', 5),
                                     'brinidx')
       ORDER BY blknum, attnum LIMIT 6;
 itemoffset | blknum | attnum | allnulls | hasnulls | placeholder |    value
------------+--------+--------+----------+----------+-------------+--------------
        137 |      0 |      1 | t        | f        | f           |
        137 |      0 |      2 | f        | f        | f           | {1 .. 88}
        138 |      4 |      1 | t        | f        | f           |
        138 |      4 |      2 | f        | f        | f           | {89 .. 176}
        139 |      8 |      1 | t        | f        | f           |
```

```
        139 |      8 |      2 | f        | f        | f           | {177 .. 264}
```

The returned columns correspond to the fields in the `BrinMemTuple` and `BrinValues` structs. See `src/include/access/brin_tuple.h` for details.

`gin_metapage_info(page bytea) returns record`

`gin_metapage_info` returns information about a GIN index metapage. For example:

```
test=# SELECT * FROM gin_metapage_info(get_raw_page('gin_index', 0));
-[ RECORD 1 ]----+-----------
pending_head     | 4294967295
pending_tail     | 4294967295
tail_free_size   | 0
n_pending_pages  | 0
n_pending_tuples | 0
n_total_pages    | 7
n_entry_pages    | 6
n_data_pages     | 0
n_entries        | 693
version          | 2
```

`gin_page_opaque_info(page bytea) returns record`

`gin_page_opaque_info` returns information about a GIN index opaque area, like the page type. For example:

```
test=# SELECT * FROM gin_page_opaque_info(get_raw_page('gin_index', 2));
 rightlink | maxoff |         flags
-----------+--------+-----------------------
         5 |      0 | {data,leaf,compressed}
(1 row)
```

`gin_leafpage_items(page bytea) returns setof record`

`gin_leafpage_items` returns information about the data stored in a GIN leaf page. For example:

```
test=# SELECT first_tid, nbytes, tids[0:5] as some_tids
        FROM gin_leafpage_items(get_raw_page('gin_test_idx', 2));
 first_tid | nbytes |                        some_tids
-----------+--------+-----------------------------------------------------------
 (8,41)    |    244 | {"(8,41)","(8,43)","(8,44)","(8,45)","(8,46)"}
 (10,45)   |    248 | {"(10,45)","(10,46)","(10,47)","(10,48)","(10,49)"}
 (12,52)   |    248 | {"(12,52)","(12,53)","(12,54)","(12,55)","(12,56)"}
 (14,59)   |    320 | {"(14,59)","(14,60)","(14,61)","(14,62)","(14,63)"}
 (167,16)  |    376 | {"(167,16)","(167,17)","(167,18)","(167,19)","(167,20)"}
 (170,30)  |    376 | {"(170,30)","(170,31)","(170,32)","(170,33)","(170,34)"}
 (173,44)  |    197 | {"(173,44)","(173,45)","(173,46)","(173,47)","(173,48)"}
(7 rows)
```

`fsm_page_contents(page bytea) returns text`

`fsm_page_contents` shows the internal node structure of a FSM page. The output is a multiline string, with one line per node in the binary tree within the page. Only those nodes that are not zero are printed. The so-called "next" pointer, which points to the next slot to be returned from the page, is also printed.

See `src/backend/storage/freespace/README` for more information on the structure of an FSM page.

F.22. passwordcheck

The `passwordcheck` module checks users' passwords whenever they are set with CREATE ROLE or ALTER ROLE. If a password is considered too weak, it will be rejected and the command will terminate with an error.

To enable this module, add `'$libdir/passwordcheck'` to shared_preload_libraries in `postgresql.conf`, then restart the server.

You can adapt this module to your needs by changing the source code. For example, you can use CrackLib[2] to check passwords — this only requires uncommenting two lines in the `Makefile` and rebuilding the module. (We cannot include CrackLib by default for license reasons.) Without CrackLib, the module enforces a few simple rules for password strength, which you can modify or extend as you see fit.

> **Caution**
>
> To prevent unencrypted passwords from being sent across the network, written to the server log or otherwise stolen by a database administrator, PostgreSQL allows the user to supply pre-encrypted passwords. Many client programs make use of this functionality and encrypt the password before sending it to the server.
>
> This limits the usefulness of the `passwordcheck` module, because in that case it can only try to guess the password. For this reason, `passwordcheck` is not recommended if your security requirements are high. It is more secure to use an external authentication method such as GSSAPI (see Chapter 19) than to rely on passwords within the database.
>
> Alternatively, you could modify `passwordcheck` to reject pre-encrypted passwords, but forcing users to set their passwords in clear text carries its own security risks.

F.23. pg_buffercache

The `pg_buffercache` module provides a means for examining what's happening in the shared buffer cache in real time.

The module provides a C function `pg_buffercache_pages` that returns a set of records, plus a view `pg_buffercache` that wraps the function for convenient use.

By default public access is revoked from both of these, just in case there are security issues lurking.

F.23.1. The `pg_buffercache` View

The definitions of the columns exposed by the view are shown in Table F-16.

Table F-16. `pg_buffercache` Columns

Name	Type	References	Description

2. http://sourceforge.net/projects/cracklib/

Name	Type	References	Description
bufferid	integer		ID, in the range 1..shared_buffers
relfilenode	oid	pg_class.relfilenode	Filenode number of the relation
reltablespace	oid	pg_tablespace.oid	Tablespace OID of the relation
reldatabase	oid	pg_database.oid	Database OID of the relation
relforknumber	smallint		Fork number within the relation; see include/common/relpath.h
relblocknumber	bigint		Page number within the relation
isdirty	boolean		Is the page dirty?
usagecount	smallint		Clock-sweep access count
pinning_backends	integer		Number of backends pinning this buffer

There is one row for each buffer in the shared cache. Unused buffers are shown with all fields null except bufferid. Shared system catalogs are shown as belonging to database zero.

Because the cache is shared by all the databases, there will normally be pages from relations not belonging to the current database. This means that there may not be matching join rows in pg_class for some rows, or that there could even be incorrect joins. If you are trying to join against pg_class, it's a good idea to restrict the join to rows having reldatabase equal to the current database's OID or zero.

When the pg_buffercache view is accessed, internal buffer manager locks are taken for long enough to copy all the buffer state data that the view will display. This ensures that the view produces a consistent set of results, while not blocking normal buffer activity longer than necessary. Nonetheless there could be some impact on database performance if this view is read often.

F.23.2. Sample Output

```
regression=# SELECT n.nspname, c.relname, count(*) AS buffers
             FROM pg_buffercache b JOIN pg_class c
             ON b.relfilenode = pg_relation_filenode(c.oid) AND
                b.reldatabase IN (0, (SELECT oid FROM pg_database
                                      WHERE datname = current_database()))
             JOIN pg_namespace n ON n.oid = c.relnamespace
             GROUP BY n.nspname, c.relname
             ORDER BY 3 DESC
             LIMIT 10;

  nspname   |        relname         | buffers
```

```
------------+------------------------+---------
 public     | delete_test_table      |     593
 public     | delete_test_table_pkey |     494
 pg_catalog | pg_attribute           |     472
 public     | quad_poly_tbl          |     353
 public     | tenk2                  |     349
 public     | tenk1                  |     349
 public     | gin_test_idx           |     306
 pg_catalog | pg_largeobject         |     206
 public     | gin_test_tbl           |     188
 public     | spgist_text_tbl        |     182
(10 rows)
```

F.23.3. Authors

Mark Kirkwood <markir@paradise.net.nz>

Design suggestions: Neil Conway <neilc@samurai.com>

Debugging advice: Tom Lane <tgl@sss.pgh.pa.us>

F.24. pgcrypto

The pgcrypto module provides cryptographic functions for PostgreSQL.

F.24.1. General Hashing Functions

F.24.1.1. digest()

```
digest(data text, type text) returns bytea
digest(data bytea, type text) returns bytea
```

Computes a binary hash of the given data. type is the algorithm to use. Standard algorithms are md5, sha1, sha224, sha256, sha384 and sha512. If pgcrypto was built with OpenSSL, more algorithms are available, as detailed in Table F-20.

If you want the digest as a hexadecimal string, use encode() on the result. For example:

```
CREATE OR REPLACE FUNCTION sha1(bytea) returns text AS $$
    SELECT encode(digest($1, 'sha1'), 'hex')
$$ LANGUAGE SQL STRICT IMMUTABLE;
```

F.24.1.2. `hmac()`

```
hmac(data text, key text, type text) returns bytea
hmac(data bytea, key bytea, type text) returns bytea
```

Calculates hashed MAC for `data` with key `key`. `type` is the same as in `digest()`.

This is similar to `digest()` but the hash can only be recalculated knowing the key. This prevents the scenario of someone altering data and also changing the hash to match.

If the key is larger than the hash block size it will first be hashed and the result will be used as key.

F.24.2. Password Hashing Functions

The functions `crypt()` and `gen_salt()` are specifically designed for hashing passwords. `crypt()` does the hashing and `gen_salt()` prepares algorithm parameters for it.

The algorithms in `crypt()` differ from the usual MD5 or SHA1 hashing algorithms in the following respects:

1. They are slow. As the amount of data is so small, this is the only way to make brute-forcing passwords hard.
2. They use a random value, called the *salt*, so that users having the same password will have different encrypted passwords. This is also an additional defense against reversing the algorithm.
3. They include the algorithm type in the result, so passwords hashed with different algorithms can co-exist.
4. Some of them are adaptive — that means when computers get faster, you can tune the algorithm to be slower, without introducing incompatibility with existing passwords.

Table F-17 lists the algorithms supported by the `crypt()` function.

Table F-17. Supported Algorithms for `crypt()`

Algorithm	Max Password Length	Adaptive?	Salt Bits	Output Length	Description
`bf`	72	yes	128	60	Blowfish-based, variant 2a
`md5`	unlimited	no	48	34	MD5-based crypt
`xdes`	8	yes	24	20	Extended DES
`des`	8	no	12	13	Original UNIX crypt

F.24.2.1. `crypt()`

```
crypt(password text, salt text) returns text
```

Calculates a crypt(3)-style hash of `password`. When storing a new password, you need to use `gen_salt()` to generate a new `salt` value. To check a password, pass the stored hash value as `salt`, and test whether the result matches the stored value.

Example of setting a new password:

```
UPDATE ... SET pswhash = crypt('new password', gen_salt('md5'));
```

Example of authentication:

```
SELECT (pswhash = crypt('entered password', pswhash)) AS pswmatch FROM ... ;
```

This returns `true` if the entered password is correct.

F.24.2.2. `gen_salt()`

```
gen_salt(type text [, iter_count integer ]) returns text
```

Generates a new random salt string for use in `crypt()`. The salt string also tells `crypt()` which algorithm to use.

The `type` parameter specifies the hashing algorithm. The accepted types are: `des`, `xdes`, `md5` and `bf`.

The `iter_count` parameter lets the user specify the iteration count, for algorithms that have one. The higher the count, the more time it takes to hash the password and therefore the more time to break it. Although with too high a count the time to calculate a hash may be several years — which is somewhat impractical. If the `iter_count` parameter is omitted, the default iteration count is used. Allowed values for `iter_count` depend on the algorithm and are shown in Table F-18.

Table F-18. Iteration Counts for `crypt()`

Algorithm	Default	Min	Max
`xdes`	725	1	16777215
`bf`	6	4	31

For `xdes` there is an additional limitation that the iteration count must be an odd number.

To pick an appropriate iteration count, consider that the original DES crypt was designed to have the speed of 4 hashes per second on the hardware of that time. Slower than 4 hashes per second would probably dampen usability. Faster than 100 hashes per second is probably too fast.

Table F-19 gives an overview of the relative slowness of different hashing algorithms. The table shows how much time it would take to try all combinations of characters in an 8-character password, assuming that the password contains either only lower case letters, or upper- and lower-case letters and numbers. In the `crypt-bf` entries, the number after a slash is the `iter_count` parameter of `gen_salt`.

Table F-19. Hash Algorithm Speeds

Algorithm	Hashes/sec	For `[a-z]`	For `[A-Za-z0-9]`	Duration relative to `md5 hash`
`crypt-bf/8`	1792	4 years	3927 years	100k
`crypt-bf/7`	3648	2 years	1929 years	50k
`crypt-bf/6`	7168	1 year	982 years	25k
`crypt-bf/5`	13504	188 days	521 years	12.5k
`crypt-md5`	171584	15 days	41 years	1k
`crypt-des`	23221568	157.5 minutes	108 days	7
`sha1`	37774272	90 minutes	68 days	4
`md5` (hash)	150085504	22.5 minutes	17 days	1

Notes:

- The machine used is an Intel Mobile Core i3.
- `crypt-des` and `crypt-md5` algorithm numbers are taken from John the Ripper v1.6.38 `-test` output.
- `md5 hash` numbers are from mdcrack 1.2.
- `sha1` numbers are from lcrack-20031130-beta.
- `crypt-bf` numbers are taken using a simple program that loops over 1000 8-character passwords. That way I can show the speed with different numbers of iterations. For reference: `john -test` shows 213 loops/sec for `crypt-bf/5`. (The very small difference in results is in accordance with the fact that the `crypt-bf` implementation in `pgcrypto` is the same one used in John the Ripper.)

Note that "try all combinations" is not a realistic exercise. Usually password cracking is done with the help of dictionaries, which contain both regular words and various mutations of them. So, even somewhat word-like passwords could be cracked much faster than the above numbers suggest, while a 6-character non-word-like password may escape cracking. Or not.

F.24.3. PGP Encryption Functions

The functions here implement the encryption part of the OpenPGP (RFC 4880) standard. Supported are both symmetric-key and public-key encryption.

An encrypted PGP message consists of 2 parts, or *packets*:

- Packet containing a session key — either symmetric-key or public-key encrypted.
- Packet containing data encrypted with the session key.

When encrypting with a symmetric key (i.e., a password):

1. The given password is hashed using a String2Key (S2K) algorithm. This is rather similar to `crypt()` algorithms — purposefully slow and with random salt — but it produces a full-length binary key.

2. If a separate session key is requested, a new random key will be generated. Otherwise the S2K key will be used directly as the session key.
3. If the S2K key is to be used directly, then only S2K settings will be put into the session key packet. Otherwise the session key will be encrypted with the S2K key and put into the session key packet.

When encrypting with a public key:

1. A new random session key is generated.
2. It is encrypted using the public key and put into the session key packet.

In either case the data to be encrypted is processed as follows:

1. Optional data-manipulation: compression, conversion to UTF-8, and/or conversion of line-endings.
2. The data is prefixed with a block of random bytes. This is equivalent to using a random IV.
3. An SHA1 hash of the random prefix and data is appended.
4. All this is encrypted with the session key and placed in the data packet.

F.24.3.1. `pgp_sym_encrypt()`

```
pgp_sym_encrypt(data text, psw text [, options text ]) returns bytea
pgp_sym_encrypt_bytea(data bytea, psw text [, options text ]) returns bytea
```

Encrypt `data` with a symmetric PGP key `psw`. The `options` parameter can contain option settings, as described below.

F.24.3.2. `pgp_sym_decrypt()`

```
pgp_sym_decrypt(msg bytea, psw text [, options text ]) returns text
pgp_sym_decrypt_bytea(msg bytea, psw text [, options text ]) returns bytea
```

Decrypt a symmetric-key-encrypted PGP message.

Decrypting `bytea` data with `pgp_sym_decrypt` is disallowed. This is to avoid outputting invalid character data. Decrypting originally textual data with `pgp_sym_decrypt_bytea` is fine.

The `options` parameter can contain option settings, as described below.

F.24.3.3. `pgp_pub_encrypt()`

```
pgp_pub_encrypt(data text, key bytea [, options text ]) returns bytea
pgp_pub_encrypt_bytea(data bytea, key bytea [, options text ]) returns bytea
```

Encrypt `data` with a public PGP key `key`. Giving this function a secret key will produce an error.

The `options` parameter can contain option settings, as described below.

F.24.3.4. pgp_pub_decrypt()

```
pgp_pub_decrypt(msg bytea, key bytea [, psw text [, options text ]]) returns text
pgp_pub_decrypt_bytea(msg bytea, key bytea [, psw text [, options text ]]) returns bytea
```

Decrypt a public-key-encrypted message. `key` must be the secret key corresponding to the public key that was used to encrypt. If the secret key is password-protected, you must give the password in `psw`. If there is no password, but you want to specify options, you need to give an empty password.

Decrypting `bytea` data with `pgp_pub_decrypt` is disallowed. This is to avoid outputting invalid character data. Decrypting originally textual data with `pgp_pub_decrypt_bytea` is fine.

The `options` parameter can contain option settings, as described below.

F.24.3.5. pgp_key_id()

```
pgp_key_id(bytea) returns text
```

`pgp_key_id` extracts the key ID of a PGP public or secret key. Or it gives the key ID that was used for encrypting the data, if given an encrypted message.

It can return 2 special key IDs:

- `SYMKEY`

  The message is encrypted with a symmetric key.

- `ANYKEY`

  The message is public-key encrypted, but the key ID has been removed. That means you will need to try all your secret keys on it to see which one decrypts it. `pgcrypto` itself does not produce such messages.

Note that different keys may have the same ID. This is rare but a normal event. The client application should then try to decrypt with each one, to see which fits — like handling `ANYKEY`.

F.24.3.6. armor(), dearmor()

```
armor(data bytea [ , keys text[], values text[] ]) returns text
dearmor(data text) returns bytea
```

These functions wrap/unwrap binary data into PGP ASCII-armor format, which is basically Base64 with CRC and additional formatting.

If the `keys` and `values` arrays are specified, an *armor header* is added to the armored format for each key/value pair. Both arrays must be single-dimensional, and they must be of the same length. The keys and values cannot contain any non-ASCII characters.

F.24.3.7. pgp_armor_headers

```
pgp_armor_headers(data text, key out text, value out text) returns setof record
```

`pgp_armor_headers()` extracts the armor headers from `data`. The return value is a set of rows with two columns, key and value. If the keys or values contain any non-ASCII characters, they are treated as UTF-8.

F.24.3.8. Options for PGP Functions

Options are named to be similar to GnuPG. An option's value should be given after an equal sign; separate options from each other with commas. For example:

```
pgp_sym_encrypt(data, psw, 'compress-algo=1, cipher-algo=aes256')
```

All of the options except `convert-crlf` apply only to encrypt functions. Decrypt functions get the parameters from the PGP data.

The most interesting options are probably `compress-algo` and `unicode-mode`. The rest should have reasonable defaults.

F.24.3.8.1. cipher-algo

Which cipher algorithm to use.

Values: bf, aes128, aes192, aes256 (OpenSSL-only: `3des`, `cast5`)
Default: aes128
Applies to: pgp_sym_encrypt, pgp_pub_encrypt

F.24.3.8.2. compress-algo

Which compression algorithm to use. Only available if PostgreSQL was built with zlib.

Values:
 0 - no compression
 1 - ZIP compression
 2 - ZLIB compression (= ZIP plus meta-data and block CRCs)
Default: 0
Applies to: pgp_sym_encrypt, pgp_pub_encrypt

F.24.3.8.3. compress-level

How much to compress. Higher levels compress smaller but are slower. 0 disables compression.

Values: 0, 1-9
Default: 6
Applies to: pgp_sym_encrypt, pgp_pub_encrypt

F.24.3.8.4. convert-crlf

Whether to convert `\n` into `\r\n` when encrypting and `\r\n` to `\n` when decrypting. RFC 4880 specifies that text data should be stored using `\r\n` line-feeds. Use this to get fully RFC-compliant behavior.

Values: 0, 1
Default: 0
Applies to: pgp_sym_encrypt, pgp_pub_encrypt, pgp_sym_decrypt, pgp_pub_decrypt

F.24.3.8.5. disable-mdc

Do not protect data with SHA-1. The only good reason to use this option is to achieve compatibility with ancient PGP products, predating the addition of SHA-1 protected packets to RFC 4880. Recent gnupg.org and pgp.com software supports it fine.

Values: 0, 1
Default: 0
Applies to: pgp_sym_encrypt, pgp_pub_encrypt

F.24.3.8.6. sess-key

Use separate session key. Public-key encryption always uses a separate session key; this option is for symmetric-key encryption, which by default uses the S2K key directly.

Values: 0, 1
Default: 0
Applies to: pgp_sym_encrypt

F.24.3.8.7. s2k-mode

Which S2K algorithm to use.

Values:
 0 - Without salt. Dangerous!
 1 - With salt but with fixed iteration count.
 3 - Variable iteration count.
Default: 3
Applies to: pgp_sym_encrypt

F.24.3.8.8. s2k-digest-algo

Which digest algorithm to use in S2K calculation.

Values: md5, sha1
Default: sha1
Applies to: pgp_sym_encrypt

F.24.3.8.9. s2k-cipher-algo

Which cipher to use for encrypting separate session key.

Values: bf, aes, aes128, aes192, aes256
Default: use cipher-algo
Applies to: pgp_sym_encrypt

F.24.3.8.10. unicode-mode

Whether to convert textual data from database internal encoding to UTF-8 and back. If your database already is UTF-8, no conversion will be done, but the message will be tagged as UTF-8. Without this option it will not be.

Values: 0, 1
Default: 0
Applies to: pgp_sym_encrypt, pgp_pub_encrypt

F.24.3.9. Generating PGP Keys with GnuPG

To generate a new key:

```
gpg --gen-key
```

The preferred key type is "DSA and Elgamal".

For RSA encryption you must create either DSA or RSA sign-only key as master and then add an RSA encryption subkey with `gpg --edit-key`.

To list keys:

```
gpg --list-secret-keys
```

To export a public key in ASCII-armor format:

```
gpg -a --export KEYID > public.key
```

To export a secret key in ASCII-armor format:

```
gpg -a --export-secret-keys KEYID > secret.key
```

You need to use `dearmor()` on these keys before giving them to the PGP functions. Or if you can handle binary data, you can drop `-a` from the command.

For more details see `man gpg`, The GNU Privacy Handbook[3] and other documentation on http://www.gnupg.org.

F.24.3.10. Limitations of PGP Code

- No support for signing. That also means that it is not checked whether the encryption subkey belongs to the master key.
- No support for encryption key as master key. As such practice is generally discouraged, this should not be a problem.
- No support for several subkeys. This may seem like a problem, as this is common practice. On the other hand, you should not use your regular GPG/PGP keys with `pgcrypto`, but create new ones, as the usage scenario is rather different.

F.24.4. Raw Encryption Functions

These functions only run a cipher over data; they don't have any advanced features of PGP encryption. Therefore they have some major problems:

1. They use user key directly as cipher key.
2. They don't provide any integrity checking, to see if the encrypted data was modified.
3. They expect that users manage all encryption parameters themselves, even IV.
4. They don't handle text.

So, with the introduction of PGP encryption, usage of raw encryption functions is discouraged.

```
encrypt(data bytea, key bytea, type text) returns bytea
decrypt(data bytea, key bytea, type text) returns bytea

encrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
decrypt_iv(data bytea, key bytea, iv bytea, type text) returns bytea
```

Encrypt/decrypt data using the cipher method specified by `type`. The syntax of the `type` string is:

```
algorithm [ - mode ] [ /pad: padding ]
```

where *`algorithm`* is one of:

- `bf` — Blowfish
- `aes` — AES (Rijndael-128, -192 or -256)

and *`mode`* is one of:

3. http://www.gnupg.org/gph/en/manual.html

- `cbc` — next block depends on previous (default)
- `ecb` — each block is encrypted separately (for testing only)

and *`padding`* is one of:

- `pkcs` — data may be any length (default)
- `none` — data must be multiple of cipher block size

So, for example, these are equivalent:

```
encrypt(data, 'fooz', 'bf')
encrypt(data, 'fooz', 'bf-cbc/pad:pkcs')
```

In `encrypt_iv` and `decrypt_iv`, the `iv` parameter is the initial value for the CBC mode; it is ignored for ECB. It is clipped or padded with zeroes if not exactly block size. It defaults to all zeroes in the functions without this parameter.

F.24.5. Random-Data Functions

```
gen_random_bytes(count integer) returns bytea
```

Returns `count` cryptographically strong random bytes. At most 1024 bytes can be extracted at a time. This is to avoid draining the randomness generator pool.

```
gen_random_uuid() returns uuid
```

Returns a version 4 (random) UUID.

F.24.6. Notes

F.24.6.1. Configuration

`pgcrypto` configures itself according to the findings of the main PostgreSQL `configure` script. The options that affect it are `--with-zlib` and `--with-openssl`.

When compiled with zlib, PGP encryption functions are able to compress data before encrypting.

When compiled with OpenSSL, there will be more algorithms available. Also public-key encryption functions will be faster as OpenSSL has more optimized BIGNUM functions.

Table F-20. Summary of Functionality with and without OpenSSL

Functionality	Built-in	With OpenSSL
MD5	yes	yes
SHA1	yes	yes

Functionality	Built-in	With OpenSSL
SHA224/256/384/512	yes	yes (Note 1)
Other digest algorithms	no	yes (Note 2)
Blowfish	yes	yes
AES	yes	yes (Note 3)
DES/3DES/CAST5	no	yes
Raw encryption	yes	yes
PGP Symmetric encryption	yes	yes
PGP Public-Key encryption	yes	yes

Notes:

1. SHA2 algorithms were added to OpenSSL in version 0.9.8. For older versions, `pgcrypto` will use built-in code.
2. Any digest algorithm OpenSSL supports is automatically picked up. This is not possible with ciphers, which need to be supported explicitly.
3. AES is included in OpenSSL since version 0.9.7. For older versions, `pgcrypto` will use built-in code.

F.24.6.2. NULL Handling

As is standard in SQL, all functions return NULL, if any of the arguments are NULL. This may create security risks on careless usage.

F.24.6.3. Security Limitations

All `pgcrypto` functions run inside the database server. That means that all the data and passwords move between `pgcrypto` and client applications in clear text. Thus you must:

1. Connect locally or use SSL connections.
2. Trust both system and database administrator.

If you cannot, then better do crypto inside client application.

The implementation does not resist side-channel attacks[4]. For example, the time required for a `pgcrypto` decryption function to complete varies among ciphertexts of a given size.

F.24.6.4. Useful Reading

- http://www.gnupg.org/gph/en/manual.html

  The GNU Privacy Handbook.

4. http://en.wikipedia.org/wiki/Side-channel_attack

- https://www.openwall.com/crypt/

  Describes the crypt-blowfish algorithm.
- https://www.iusmentis.com/security/passphrasefaq/

  How to choose a good password.
- http://world.std.com/~reinhold/diceware.html

  Interesting idea for picking passwords.
- http://www.interhack.net/people/cmcurtin/snake-oil-faq.html

  Describes good and bad cryptography.

F.24.6.5. Technical References

- https://tools.ietf.org/html/rfc4880

  OpenPGP message format.
- https://tools.ietf.org/html/rfc1321

  The MD5 Message-Digest Algorithm.
- https://tools.ietf.org/html/rfc2104

  HMAC: Keyed-Hashing for Message Authentication.
- http://www.usenix.org/events/usenix99/provos.html

  Comparison of crypt-des, crypt-md5 and bcrypt algorithms.
- http://en.wikipedia.org/wiki/Fortuna_(PRNG)

  Description of Fortuna CSPRNG.
- https://jlcooke.ca/random/

  Jean-Luc Cooke Fortuna-based `/dev/random` driver for Linux.

F.24.7. Author

Marko Kreen <`markokr@gmail.com`>

`pgcrypto` uses code from the following sources:

Algorithm	Author	Source origin
DES crypt	David Burren and others	FreeBSD libcrypt
MD5 crypt	Poul-Henning Kamp	FreeBSD libcrypt
Blowfish crypt	Solar Designer	www.openwall.com
Blowfish cipher	Simon Tatham	PuTTY
Rijndael cipher	Brian Gladman	OpenBSD sys/crypto
MD5 hash and SHA1	WIDE Project	KAME kame/sys/crypto

Algorithm	Author	Source origin
SHA256/384/512	Aaron D. Gifford	OpenBSD sys/crypto
BIGNUM math	Michael J. Fromberger	dartmouth.edu/~sting/sw/imath

F.25. pg_freespacemap

The `pg_freespacemap` module provides a means for examining the free space map (FSM). It provides a function called `pg_freespace`, or two overloaded functions, to be precise. The functions show the value recorded in the free space map for a given page, or for all pages in the relation.

By default public access is revoked from the functions, just in case there are security issues lurking.

F.25.1. Functions

`pg_freespace(rel regclass IN, blkno bigint IN) returns int2`

Returns the amount of free space on the page of the relation, specified by `blkno`, according to the FSM.

`pg_freespace(rel regclass IN, blkno OUT bigint, avail OUT int2)`

Displays the amount of free space on each page of the relation, according to the FSM. A set of `(blkno bigint, avail int2)` tuples is returned, one tuple for each page in the relation.

The values stored in the free space map are not exact. They're rounded to precision of 1/256th of `BLCKSZ` (32 bytes with default `BLCKSZ`), and they're not kept fully up-to-date as tuples are inserted and updated.

For indexes, what is tracked is entirely-unused pages, rather than free space within pages. Therefore, the values are not meaningful, just whether a page is full or empty.

> **Note:** The interface was changed in version 8.4, to reflect the new FSM implementation introduced in the same version.

F.25.2. Sample Output

```
postgres=# SELECT * FROM pg_freespace('foo');
 blkno | avail
-------+-------
     0 |     0
     1 |     0
     2 |     0
     3 |    32
     4 |   704
     5 |   704
     6 |   704
```

```
     7 |  1216
     8 |   704
     9 |   704
    10 |   704
    11 |   704
    12 |   704
    13 |   704
    14 |   704
    15 |   704
    16 |   704
    17 |   704
    18 |   704
    19 |  3648
(20 rows)

postgres=# SELECT * FROM pg_freespace('foo', 7);
 pg_freespace
--------------
         1216
(1 row)
```

F.25.3. Author

Original version by Mark Kirkwood <markir@paradise.net.nz>. Rewritten in version 8.4 to suit new FSM implementation by Heikki Linnakangas <heikki@enterprisedb.com>

F.26. pg_prewarm

The pg_prewarm module provides a convenient way to load relation data into either the operating system buffer cache or the PostgreSQL buffer cache.

F.26.1. Functions

```
pg_prewarm(regclass, mode text default 'buffer', fork text default 'main',
           first_block int8 default null,
           last_block int8 default null) RETURNS int8
```

The first argument is the relation to be prewarmed. The second argument is the prewarming method to be used, as further discussed below; the third is the relation fork to be prewarmed, usually main. The fourth argument is the first block number to prewarm (NULL is accepted as a synonym for zero). The fifth argument is the last block number to prewarm (NULL means prewarm through the last block in the relation). The return value is the number of blocks prewarmed.

There are three available prewarming methods. prefetch issues asynchronous prefetch requests to the operating system, if this is supported, or throws an error otherwise. read reads the requested range of

blocks; unlike `prefetch`, this is synchronous and supported on all platforms and builds, but may be slower. `buffer` reads the requested range of blocks into the database buffer cache.

Note that with any of these methods, attempting to prewarm more blocks than can be cached — by the OS when using `prefetch` or `read`, or by PostgreSQL when using `buffer` — will likely result in lower-numbered blocks being evicted as higher numbered blocks are read in. Prewarmed data also enjoys no special protection from cache evictions, so it is possible that other system activity may evict the newly prewarmed blocks shortly after they are read; conversely, prewarming may also evict other data from cache. For these reasons, prewarming is typically most useful at startup, when caches are largely empty.

F.26.2. Author

Robert Haas <rhaas@postgresql.org>

F.27. pgrowlocks

The `pgrowlocks` module provides a function to show row locking information for a specified table.

F.27.1. Overview

```
pgrowlocks(text) returns setof record
```

The parameter is the name of a table. The result is a set of records, with one row for each locked row within the table. The output columns are shown in Table F-21.

Table F-21. `pgrowlocks` Output Columns

Name	Type	Description
`locked_row`	`tid`	Tuple ID (TID) of locked row
`locker`	`xid`	Transaction ID of locker, or multixact ID if multitransaction
`multi`	`boolean`	True if locker is a multitransaction
`xids`	`xid[]`	Transaction IDs of lockers (more than one if multitransaction)
`modes`	`text[]`	Lock mode of lockers (more than one if multitransaction), an array of `Key Share`, `Share`, `For No Key Update`, `No Key Update`, `For Update`, `Update`.
`pids`	`integer[]`	Process IDs of locking backends (more than one if multitransaction)

pgrowlocks takes AccessShareLock for the target table and reads each row one by one to collect the row locking information. This is not very speedy for a large table. Note that:

1. If the table as a whole is exclusive-locked by someone else, pgrowlocks will be blocked.
2. pgrowlocks is not guaranteed to produce a self-consistent snapshot. It is possible that a new row lock is taken, or an old lock is freed, during its execution.

pgrowlocks does not show the contents of locked rows. If you want to take a look at the row contents at the same time, you could do something like this:

```
SELECT * FROM accounts AS a, pgrowlocks('accounts') AS p
  WHERE p.locked_row = a.ctid;
```

Be aware however that such a query will be very inefficient.

F.27.2. Sample Output

```
=# SELECT * FROM pgrowlocks('t1');
 locked_row | locker | multi | xids  |     modes      |  pids
------------+--------+-------+-------+----------------+--------
 (0,1)      |    609 | f     | {609} | {"For Share"}  | {3161}
 (0,2)      |    609 | f     | {609} | {"For Share"}  | {3161}
 (0,3)      |    607 | f     | {607} | {"For Update"} | {3107}
 (0,4)      |    607 | f     | {607} | {"For Update"} | {3107}
(4 rows)
```

F.27.3. Author

Tatsuo Ishii

F.28. pg_stat_statements

The pg_stat_statements module provides a means for tracking execution statistics of all SQL statements executed by a server.

The module must be loaded by adding pg_stat_statements to shared_preload_libraries in postgresql.conf, because it requires additional shared memory. This means that a server restart is needed to add or remove the module.

When pg_stat_statements is loaded, it tracks statistics across all databases of the server. To access and manipulate these statistics, the module provides a view, pg_stat_statements, and the utility functions pg_stat_statements_reset and pg_stat_statements. These are not available globally but can be enabled for a specific database with CREATE EXTENSION pg_stat_statements.

F.28.1. The pg_stat_statements View

The statistics gathered by the module are made available via a view named pg_stat_statements. This view contains one row for each distinct database ID, user ID and query ID (up to the maximum number of distinct statements that the module can track). The columns of the view are shown in Table F-22.

Table F-22. pg_stat_statements Columns

Name	Type	References	Description
userid	oid	pg_authid.oid	OID of user who executed the statement
dbid	oid	pg_database.oid	OID of database in which the statement was executed
queryid	bigint		Internal hash code, computed from the statement's parse tree
query	text		Text of a representative statement
calls	bigint		Number of times executed
total_time	double precision		Total time spent in the statement, in milliseconds
min_time	double precision		Minimum time spent in the statement, in milliseconds
max_time	double precision		Maximum time spent in the statement, in milliseconds
mean_time	double precision		Mean time spent in the statement, in milliseconds
stddev_time	double precision		Population standard deviation of time spent in the statement, in milliseconds
rows	bigint		Total number of rows retrieved or affected by the statement
shared_blks_hit	bigint		Total number of shared block cache hits by the statement

Name	Type	References	Description
shared_blks_read	bigint		Total number of shared blocks read by the statement
shared_blks_dirtied	bigint		Total number of shared blocks dirtied by the statement
shared_blks_written	bigint		Total number of shared blocks written by the statement
local_blks_hit	bigint		Total number of local block cache hits by the statement
local_blks_read	bigint		Total number of local blocks read by the statement
local_blks_dirtied	bigint		Total number of local blocks dirtied by the statement
local_blks_written	bigint		Total number of local blocks written by the statement
temp_blks_read	bigint		Total number of temp blocks read by the statement
temp_blks_written	bigint		Total number of temp blocks written by the statement
blk_read_time	double precision		Total time the statement spent reading blocks, in milliseconds (if track_io_timing is enabled, otherwise zero)
blk_write_time	double precision		Total time the statement spent writing blocks, in milliseconds (if track_io_timing is enabled, otherwise zero)

For security reasons, non-superusers are not allowed to see the SQL text or queryid of queries executed by other users. They can see the statistics, however, if the view has been installed in their database.

Plannable queries (that is, SELECT, INSERT, UPDATE, and DELETE) are combined into a single pg_stat_statements entry whenever they have identical query structures according to an internal

hash calculation. Typically, two queries will be considered the same for this purpose if they are semantically equivalent except for the values of literal constants appearing in the query. Utility commands (that is, all other commands) are compared strictly on the basis of their textual query strings, however.

When a constant's value has been ignored for purposes of matching the query to other queries, the constant is replaced by `?` in the `pg_stat_statements` display. The rest of the query text is that of the first query that had the particular `queryid` hash value associated with the `pg_stat_statements` entry.

In some cases, queries with visibly different texts might get merged into a single `pg_stat_statements` entry. Normally this will happen only for semantically equivalent queries, but there is a small chance of hash collisions causing unrelated queries to be merged into one entry. (This cannot happen for queries belonging to different users or databases, however.)

Since the `queryid` hash value is computed on the post-parse-analysis representation of the queries, the opposite is also possible: queries with identical texts might appear as separate entries, if they have different meanings as a result of factors such as different `search_path` settings.

Consumers of `pg_stat_statements` may wish to use `queryid` (perhaps in combination with `dbid` and `userid`) as a more stable and reliable identifier for each entry than its query text. However, it is important to understand that there are only limited guarantees around the stability of the `queryid` hash value. Since the identifier is derived from the post-parse-analysis tree, its value is a function of, among other things, the internal object identifiers appearing in this representation. This has some counterintuitive implications. For example, `pg_stat_statements` will consider two apparently-identical queries to be distinct, if they reference a table that was dropped and recreated between the executions of the two queries. The hashing process is also sensitive to differences in machine architecture and other facets of the platform. Furthermore, it is not safe to assume that `queryid` will be stable across major versions of PostgreSQL.

As a rule of thumb, `queryid` values can be assumed to be stable and comparable only so long as the underlying server version and catalog metadata details stay exactly the same. Two servers participating in replication based on physical WAL replay can be expected to have identical `queryid` values for the same query. However, logical replication schemes do not promise to keep replicas identical in all relevant details, so `queryid` will not be a useful identifier for accumulating costs across a set of logical replicas. If in doubt, direct testing is recommended.

The representative query texts are kept in an external disk file, and do not consume shared memory. Therefore, even very lengthy query texts can be stored successfully. However, if many long query texts are accumulated, the external file might grow unmanageably large. As a recovery method if that happens, `pg_stat_statements` may choose to discard the query texts, whereupon all existing entries in the `pg_stat_statements` view will show null `query` fields, though the statistics associated with each `queryid` are preserved. If this happens, consider reducing `pg_stat_statements.max` to prevent recurrences.

F.28.2. Functions

`pg_stat_statements_reset() returns void`

`pg_stat_statements_reset` discards all statistics gathered so far by `pg_stat_statements`. By default, this function can only be executed by superusers.

`pg_stat_statements(showtext boolean) returns setof record`

The `pg_stat_statements` view is defined in terms of a function also named `pg_stat_statements`. It is possible for clients to call the `pg_stat_statements` function directly, and by specifying `showtext := false` have query text be omitted (that is, the `OUT` argument that corresponds to the view's `query` column will return nulls). This feature is intended to support external tools that might wish to avoid the overhead of repeatedly retrieving query texts of indeterminate length. Such tools can instead cache the first query text observed for each entry themselves, since that is all `pg_stat_statements` itself does, and then retrieve query texts only as needed. Since the server stores query texts in a file, this approach may reduce physical I/O for repeated examination of the `pg_stat_statements` data.

F.28.3. Configuration Parameters

`pg_stat_statements.max` (`integer`)

`pg_stat_statements.max` is the maximum number of statements tracked by the module (i.e., the maximum number of rows in the `pg_stat_statements` view). If more distinct statements than that are observed, information about the least-executed statements is discarded. The default value is 5000. This parameter can only be set at server start.

`pg_stat_statements.track` (`enum`)

`pg_stat_statements.track` controls which statements are counted by the module. Specify `top` to track top-level statements (those issued directly by clients), `all` to also track nested statements (such as statements invoked within functions), or `none` to disable statement statistics collection. The default value is `top`. Only superusers can change this setting.

`pg_stat_statements.track_utility` (`boolean`)

`pg_stat_statements.track_utility` controls whether utility commands are tracked by the module. Utility commands are all those other than `SELECT`, `INSERT`, `UPDATE` and `DELETE`. The default value is `on`. Only superusers can change this setting.

`pg_stat_statements.save` (`boolean`)

`pg_stat_statements.save` specifies whether to save statement statistics across server shutdowns. If it is `off` then statistics are not saved at shutdown nor reloaded at server start. The default value is `on`. This parameter can only be set in the `postgresql.conf` file or on the server command line.

The module requires additional shared memory proportional to `pg_stat_statements.max`. Note that this memory is consumed whenever the module is loaded, even if `pg_stat_statements.track` is set to `none`.

These parameters must be set in `postgresql.conf`. Typical usage might be:

```
# postgresql.conf
shared_preload_libraries = 'pg_stat_statements'

pg_stat_statements.max = 10000
pg_stat_statements.track = all
```

F.28.4. Sample Output

```
bench=# SELECT pg_stat_statements_reset();

$ pgbench -i bench
$ pgbench -c10 -t300 bench

bench=# \x
bench=# SELECT query, calls, total_time, rows, 100.0 * shared_blks_hit /
               nullif(shared_blks_hit + shared_blks_read, 0) AS hit_percent
          FROM pg_stat_statements ORDER BY total_time DESC LIMIT 5;
-[ RECORD 1 ]---------------------------------------------------------------------
query       | UPDATE pgbench_branches SET bbalance = bbalance + ? WHERE bid = ?;
calls       | 3000
total_time  | 9609.00100000002
rows        | 2836
hit_percent | 99.9778970000200936
-[ RECORD 2 ]---------------------------------------------------------------------
query       | UPDATE pgbench_tellers SET tbalance = tbalance + ? WHERE tid = ?;
calls       | 3000
total_time  | 8015.156
rows        | 2990
hit_percent | 99.9731126579631345
-[ RECORD 3 ]---------------------------------------------------------------------
query       | copy pgbench_accounts from stdin
calls       | 1
total_time  | 310.624
rows        | 100000
hit_percent | 0.30395136778115501520
-[ RECORD 4 ]---------------------------------------------------------------------
query       | UPDATE pgbench_accounts SET abalance = abalance + ? WHERE aid = ?;
calls       | 3000
total_time  | 271.741999999997
rows        | 3000
hit_percent | 93.7968855088209426
-[ RECORD 5 ]---------------------------------------------------------------------
query       | alter table pgbench_accounts add primary key (aid)
calls       | 1
total_time  | 81.42
rows        | 0
hit_percent | 34.4947735191637631
```

F.28.5. Authors

Takahiro Itagaki <itagaki.takahiro@oss.ntt.co.jp>. Query normalization added by Peter Geoghegan <peter@2ndquadrant.com>.

F.29. pgstattuple

The pgstattuple module provides various functions to obtain tuple-level statistics.

F.29.1. Functions

pgstattuple(regclass) returns record

pgstattuple returns a relation's physical length, percentage of "dead" tuples, and other info. This may help users to determine whether vacuum is necessary or not. The argument is the target relation's name (optionally schema-qualified) or OID. For example:

```
test=> SELECT * FROM pgstattuple('pg_catalog.pg_proc');
-[ RECORD 1 ]------+-------
table_len          | 458752
tuple_count        | 1470
tuple_len          | 438896
tuple_percent      | 95.67
dead_tuple_count   | 11
dead_tuple_len     | 3157
dead_tuple_percent | 0.69
free_space         | 8932
free_percent       | 1.95
```

The output columns are described in Table F-23.

Table F-23. pgstattuple Output Columns

Column	Type	Description
table_len	bigint	Physical relation length in bytes
tuple_count	bigint	Number of live tuples
tuple_len	bigint	Total length of live tuples in bytes
tuple_percent	float8	Percentage of live tuples
dead_tuple_count	bigint	Number of dead tuples
dead_tuple_len	bigint	Total length of dead tuples in bytes
dead_tuple_percent	float8	Percentage of dead tuples
free_space	bigint	Total free space in bytes
free_percent	float8	Percentage of free space

Note: The table_len will always be greater than the sum of the tuple_len, dead_tuple_len and free_space. The difference is accounted for by fixed page overhead, the per-page table of pointers to tuples, and padding to ensure that tuples are correctly aligned.

pgstattuple acquires only a read lock on the relation. So the results do not reflect an instantaneous

snapshot; concurrent updates will affect them.

`pgstattuple` judges a tuple is "dead" if `HeapTupleSatisfiesDirty` returns false.

`pgstattuple(text) returns record`

This is the same as `pgstattuple(regclass)`, except that the target relation is specified as TEXT. This function is kept because of backward-compatibility so far, and will be deprecated in some future release.

`pgstatindex(regclass) returns record`

`pgstatindex` returns a record showing information about a B-tree index. For example:

```
test=> SELECT * FROM pgstatindex('pg_cast_oid_index');
-[ RECORD 1 ]------+------
version            | 2
tree_level         | 0
index_size         | 16384
root_block_no      | 1
internal_pages     | 0
leaf_pages         | 1
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 54.27
leaf_fragmentation | 0
```

The output columns are:

Column	Type	Description
`version`	`integer`	B-tree version number
`tree_level`	`integer`	Tree level of the root page
`index_size`	`bigint`	Total index size in bytes
`root_block_no`	`bigint`	Location of root page (zero if none)
`internal_pages`	`bigint`	Number of "internal" (upper-level) pages
`leaf_pages`	`bigint`	Number of leaf pages
`empty_pages`	`bigint`	Number of empty pages
`deleted_pages`	`bigint`	Number of deleted pages
`avg_leaf_density`	`float8`	Average density of leaf pages
`leaf_fragmentation`	`float8`	Leaf page fragmentation

The reported `index_size` will normally correspond to one more page than is accounted for by `internal_pages + leaf_pages + empty_pages + deleted_pages`, because it also includes the index's metapage.

As with `pgstattuple`, the results are accumulated page-by-page, and should not be expected to represent an instantaneous snapshot of the whole index.

`pgstatindex(text) returns record`

This is the same as `pgstatindex(regclass)`, except that the target index is specified as TEXT. This function is kept because of backward-compatibility so far, and will be deprecated in some future release.

`pgstatginindex(regclass) returns record`

`pgstatginindex` returns a record showing information about a GIN index. For example:

```
test=> SELECT * FROM pgstatginindex('test_gin_index');
-[ RECORD 1 ]--+--
version        | 1
pending_pages  | 0
pending_tuples | 0
```

The output columns are:

Column	Type	Description
`version`	`integer`	GIN version number
`pending_pages`	`integer`	Number of pages in the pending list
`pending_tuples`	`bigint`	Number of tuples in the pending list

`pg_relpages(regclass) returns bigint`

`pg_relpages` returns the number of pages in the relation.

`pg_relpages(text) returns bigint`

This is the same as `pg_relpages(regclass)`, except that the target relation is specified as TEXT. This function is kept because of backward-compatibility so far, and will be deprecated in some future release.

`pgstattuple_approx(regclass) returns record`

`pgstattuple_approx` is a faster alternative to `pgstattuple` that returns approximate results. The argument is the target relation's name or OID. For example:

```
test=> SELECT * FROM pgstattuple_approx('pg_catalog.pg_proc'::regclass);
-[ RECORD 1 ]--------+-------
table_len            | 573440
scanned_percent      | 2
approx_tuple_count   | 2740
approx_tuple_len     | 561210
approx_tuple_percent | 97.87
dead_tuple_count     | 0
dead_tuple_len       | 0
dead_tuple_percent   | 0
approx_free_space    | 11996
approx_free_percent  | 2.09
```

The output columns are described in Table F-24.

Whereas pgstattuple always performs a full-table scan and returns an exact count of live and dead tuples (and their sizes) and free space, pgstattuple_approx tries to avoid the full-table scan and returns exact dead tuple statistics along with an approximation of the number and size of live tuples and free space.

It does this by skipping pages that have only visible tuples according to the visibility map (if a page has the corresponding VM bit set, then it is assumed to contain no dead tuples). For such pages, it derives the free space value from the free space map, and assumes that the rest of the space on the page is taken up by live tuples.

For pages that cannot be skipped, it scans each tuple, recording its presence and size in the appropriate counters, and adding up the free space on the page. At the end, it estimates the total number of live tuples based on the number of pages and tuples scanned (in the same way that VACUUM estimates pg_class.reltuples).

Table F-24. pgstattuple_approx Output Columns

Column	Type	Description
table_len	bigint	Physical relation length in bytes (exact)
scanned_percent	float8	Percentage of table scanned
approx_tuple_count	bigint	Number of live tuples (estimated)
approx_tuple_len	bigint	Total length of live tuples in bytes (estimated)
approx_tuple_percent	float8	Percentage of live tuples
dead_tuple_count	bigint	Number of dead tuples (exact)
dead_tuple_len	bigint	Total length of dead tuples in bytes (exact)
dead_tuple_percent	float8	Percentage of dead tuples
approx_free_space	bigint	Total free space in bytes (estimated)
approx_free_percent	float8	Percentage of free space

In the above output, the free space figures may not match the pgstattuple output exactly, because the free space map gives us an exact figure, but is not guaranteed to be accurate to the byte.

F.29.2. Authors

Tatsuo Ishii, Satoshi Nagayasu and Abhijit Menon-Sen

F.30. pg_trgm

The pg_trgm module provides functions and operators for determining the similarity of alphanumeric

text based on trigram matching, as well as index operator classes that support fast searching for similar strings.

F.30.1. Trigram (or Trigraph) Concepts

A trigram is a group of three consecutive characters taken from a string. We can measure the similarity of two strings by counting the number of trigrams they share. This simple idea turns out to be very effective for measuring the similarity of words in many natural languages.

> **Note:** `pg_trgm` ignores non-word characters (non-alphanumerics) when extracting trigrams from a string. Each word is considered to have two spaces prefixed and one space suffixed when determining the set of trigrams contained in the string. For example, the set of trigrams in the string "`cat`" is "` c`", "` ca`", "`cat`", and "`at `". The set of trigrams in the string "`foo|bar`" is "` f`", "` fo`", "`foo`", "`oo `", "` b`", "` ba`", "`bar`", and "`ar `".

F.30.2. Functions and Operators

The functions provided by the `pg_trgm` module are shown in Table F-25, the operators in Table F-26.

Table F-25. `pg_trgm` Functions

Function	Returns	Description
`similarity(text, text)`	`real`	Returns a number that indicates how similar the two arguments are. The range of the result is zero (indicating that the two strings are completely dissimilar) to one (indicating that the two strings are identical).
`show_trgm(text)`	`text[]`	Returns an array of all the trigrams in the given string. (In practice this is seldom useful except for debugging.)
`show_limit()`	`real`	Returns the current similarity threshold used by the `%` operator. This sets the minimum similarity between two words for them to be considered similar enough to be misspellings of each other, for example.

Function	Returns	Description
`set_limit(real)`	`real`	Sets the current similarity threshold that is used by the `%` operator. The threshold must be between 0 and 1 (default is 0.3). Returns the same value passed in.

Table F-26. `pg_trgm` Operators

Operator	Returns	Description
`text % text`	`boolean`	Returns `true` if its arguments have a similarity that is greater than the current similarity threshold set by `set_limit`.
`text <-> text`	`real`	Returns the "distance" between the arguments, that is one minus the `similarity()` value.

F.30.3. Index Support

The `pg_trgm` module provides GiST and GIN index operator classes that allow you to create an index over a text column for the purpose of very fast similarity searches. These index types support the above-described similarity operators, and additionally support trigram-based index searches for `LIKE`, `ILIKE`, `~` and `~*` queries. (These indexes do not support equality nor simple comparison operators, so you may need a regular B-tree index too.)

Example:

```
CREATE TABLE test_trgm (t text);
CREATE INDEX trgm_idx ON test_trgm USING GIST (t gist_trgm_ops);
```

or

```
CREATE INDEX trgm_idx ON test_trgm USING GIN (t gin_trgm_ops);
```

At this point, you will have an index on the `t` column that you can use for similarity searching. A typical query is

```
SELECT t, similarity(t, 'word') AS sml
  FROM test_trgm
  WHERE t % 'word'
  ORDER BY sml DESC, t;
```

This will return all values in the text column that are sufficiently similar to *word*, sorted from best match to worst. The index will be used to make this a fast operation even over very large data sets.

A variant of the above query is

```
SELECT t, t <-> 'word' AS dist
  FROM test_trgm
  ORDER BY dist LIMIT 10;
```

This can be implemented quite efficiently by GiST indexes, but not by GIN indexes. It will usually beat the first formulation when only a small number of the closest matches is wanted.

Beginning in PostgreSQL 9.1, these index types also support index searches for `LIKE` and `ILIKE`, for example

```
SELECT * FROM test_trgm WHERE t LIKE '%foo%bar';
```

The index search works by extracting trigrams from the search string and then looking these up in the index. The more trigrams in the search string, the more effective the index search is. Unlike B-tree based searches, the search string need not be left-anchored.

Beginning in PostgreSQL 9.3, these index types also support index searches for regular-expression matches (`~` and `~*` operators), for example

```
SELECT * FROM test_trgm WHERE t ~ '(foo|bar)';
```

The index search works by extracting trigrams from the regular expression and then looking these up in the index. The more trigrams that can be extracted from the regular expression, the more effective the index search is. Unlike B-tree based searches, the search string need not be left-anchored.

For both `LIKE` and regular-expression searches, keep in mind that a pattern with no extractable trigrams will degenerate to a full-index scan.

The choice between GiST and GIN indexing depends on the relative performance characteristics of GiST and GIN, which are discussed elsewhere. As a rule of thumb, a GIN index is faster to search than a GiST index, but slower to build or update; so GIN is better suited for static data and GiST for often-updated data.

F.30.4. Text Search Integration

Trigram matching is a very useful tool when used in conjunction with a full text index. In particular it can help to recognize misspelled input words that will not be matched directly by the full text search mechanism.

The first step is to generate an auxiliary table containing all the unique words in the documents:

```
CREATE TABLE words AS SELECT word FROM
        ts_stat('SELECT to_tsvector("simple", bodytext) FROM documents');
```

where `documents` is a table that has a text field `bodytext` that we wish to search. The reason for using the `simple` configuration with the `to_tsvector` function, instead of using a language-specific configuration, is that we want a list of the original (unstemmed) words.

Next, create a trigram index on the word column:

```
CREATE INDEX words_idx ON words USING GIN (word gin_trgm_ops);
```

Now, a SELECT query similar to the previous example can be used to suggest spellings for misspelled words in user search terms. A useful extra test is to require that the selected words are also of similar length to the misspelled word.

> **Note:** Since the words table has been generated as a separate, static table, it will need to be periodically regenerated so that it remains reasonably up-to-date with the document collection. Keeping it exactly current is usually unnecessary.

F.30.5. References

GiST Development Site http://www.sai.msu.su/~megera/postgres/gist/

Tsearch2 Development Site http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/

F.30.6. Authors

Oleg Bartunov <oleg@sai.msu.su>, Moscow, Moscow University, Russia

Teodor Sigaev <teodor@sigaev.ru>, Moscow, Delta-Soft Ltd.,Russia

Documentation: Christopher Kings-Lynne

This module is sponsored by Delta-Soft Ltd., Moscow, Russia.

F.31. postgres_fdw

The postgres_fdw module provides the foreign-data wrapper postgres_fdw, which can be used to access data stored in external PostgreSQL servers.

The functionality provided by this module overlaps substantially with the functionality of the older dblink module. But postgres_fdw provides more transparent and standards-compliant syntax for accessing remote tables, and can give better performance in many cases.

To prepare for remote access using postgres_fdw:

1. Install the postgres_fdw extension using CREATE EXTENSION.
2. Create a foreign server object, using CREATE SERVER, to represent each remote database you want to connect to. Specify connection information, except user and password, as options of the server object.
3. Create a user mapping, using CREATE USER MAPPING, for each database user you want to allow to access each foreign server. Specify the remote user name and password to use as user and password options of the user mapping.
4. Create a foreign table, using CREATE FOREIGN TABLE or IMPORT FOREIGN SCHEMA, for each remote table you want to access. The columns of the foreign table must match the referenced

remote table. You can, however, use table and/or column names different from the remote table's, if you specify the correct remote names as options of the foreign table object.

Now you need only `SELECT` from a foreign table to access the data stored in its underlying remote table. You can also modify the remote table using `INSERT`, `UPDATE`, or `DELETE`. (Of course, the remote user you have specified in your user mapping must have privileges to do these things.)

Note that `postgres_fdw` currently lacks support for `INSERT` statements with an `ON CONFLICT DO UPDATE` clause. However, the `ON CONFLICT DO NOTHING` clause is supported, provided a unique index inference specification is omitted.

It is generally recommended that the columns of a foreign table be declared with exactly the same data types, and collations if applicable, as the referenced columns of the remote table. Although `postgres_fdw` is currently rather forgiving about performing data type conversions at need, surprising semantic anomalies may arise when types or collations do not match, due to the remote server interpreting `WHERE` clauses slightly differently from the local server.

Note that a foreign table can be declared with fewer columns, or with a different column order, than its underlying remote table has. Matching of columns to the remote table is by name, not position.

F.31.1. FDW Options of postgres_fdw

F.31.1.1. Connection Options

A foreign server using the `postgres_fdw` foreign data wrapper can have the same options that libpq accepts in connection strings, as described in Section 31.1.2, except that these options are not allowed:

- `user` and `password` (specify these in a user mapping, instead)
- `client_encoding` (this is automatically set from the local server encoding)
- `fallback_application_name` (always set to `postgres_fdw`)

Only superusers may connect to foreign servers without password authentication, so always specify the `password` option for user mappings belonging to non-superusers.

F.31.1.2. Object Name Options

These options can be used to control the names used in SQL statements sent to the remote PostgreSQL server. These options are needed when a foreign table is created with names different from the underlying remote table's names.

`schema_name`

This option, which can be specified for a foreign table, gives the schema name to use for the foreign table on the remote server. If this option is omitted, the name of the foreign table's schema is used.

`table_name`

This option, which can be specified for a foreign table, gives the table name to use for the foreign table on the remote server. If this option is omitted, the foreign table's name is used.

`column_name`

This option, which can be specified for a column of a foreign table, gives the column name to use for the column on the remote server. If this option is omitted, the column's name is used.

F.31.1.3. Cost Estimation Options

`postgres_fdw` retrieves remote data by executing queries against remote servers, so ideally the estimated cost of scanning a foreign table should be whatever it costs to be done on the remote server, plus some overhead for communication. The most reliable way to get such an estimate is to ask the remote server and then add something for overhead — but for simple queries, it may not be worth the cost of an additional remote query to get a cost estimate. So `postgres_fdw` provides the following options to control how cost estimation is done:

`use_remote_estimate`

This option, which can be specified for a foreign table or a foreign server, controls whether `postgres_fdw` issues remote `EXPLAIN` commands to obtain cost estimates. A setting for a foreign table overrides any setting for its server, but only for that table. The default is `false`.

`fdw_startup_cost`

This option, which can be specified for a foreign server, is a numeric value that is added to the estimated startup cost of any foreign-table scan on that server. This represents the additional overhead of establishing a connection, parsing and planning the query on the remote side, etc. The default value is `100`.

`fdw_tuple_cost`

This option, which can be specified for a foreign server, is a numeric value that is used as extra cost per-tuple for foreign-table scans on that server. This represents the additional overhead of data transfer between servers. You might increase or decrease this number to reflect higher or lower network delay to the remote server. The default value is `0.01`.

When `use_remote_estimate` is true, `postgres_fdw` obtains row count and cost estimates from the remote server and then adds `fdw_startup_cost` and `fdw_tuple_cost` to the cost estimates. When `use_remote_estimate` is false, `postgres_fdw` performs local row count and cost estimation and then adds `fdw_startup_cost` and `fdw_tuple_cost` to the cost estimates. This local estimation is unlikely to be very accurate unless local copies of the remote table's statistics are available. Running ANALYZE on the foreign table is the way to update the local statistics; this will perform a scan of the remote table and then calculate and store statistics just as though the table were local. Keeping local statistics can be a useful way to reduce per-query planning overhead for a remote table — but if the remote table is frequently updated, the local statistics will soon be obsolete.

F.31.1.4. Updatability Options

By default all foreign tables using postgres_fdw are assumed to be updatable. This may be overridden using the following option:

updatable

This option controls whether postgres_fdw allows foreign tables to be modified using INSERT, UPDATE and DELETE commands. It can be specified for a foreign table or a foreign server. A table-level option overrides a server-level option. The default is true.

Of course, if the remote table is not in fact updatable, an error would occur anyway. Use of this option primarily allows the error to be thrown locally without querying the remote server. Note however that the information_schema views will report a postgres_fdw foreign table to be updatable (or not) according to the setting of this option, without any check of the remote server.

F.31.1.5. Importing Options

postgres_fdw is able to import foreign table definitions using IMPORT FOREIGN SCHEMA. This command creates foreign table definitions on the local server that match tables or views present on the remote server. If the remote tables to be imported have columns of user-defined data types, the local server must have compatible types of the same names.

Importing behavior can be customized with the following options (given in the IMPORT FOREIGN SCHEMA command):

import_collate

This option controls whether column COLLATE options are included in the definitions of foreign tables imported from a foreign server. The default is true. You might need to turn this off if the remote server has a different set of collation names than the local server does, which is likely to be the case if it's running on a different operating system.

import_default

This option controls whether column DEFAULT expressions are included in the definitions of foreign tables imported from a foreign server. The default is false. If you enable this option, be wary of defaults that might get computed differently on the local server than they would be on the remote server; nextval() is a common source of problems. The IMPORT will fail altogether if an imported default expression uses a function or operator that does not exist locally.

import_not_null

This option controls whether column NOT NULL constraints are included in the definitions of foreign tables imported from a foreign server. The default is true.

Note that constraints other than NOT NULL will never be imported from the remote tables. Although PostgreSQL does support CHECK constraints on foreign tables, there is no provision for importing them automatically, because of the risk that a constraint expression could evaluate differently on the local and remote servers. Any such inconsistency in the behavior of a CHECK constraint could lead to hard-to-detect errors in query optimization. So if you wish to import CHECK constraints, you must do so manually, and you should verify the semantics of each one carefully. For more detail about the treatment of CHECK constraints on foreign tables, see CREATE FOREIGN TABLE.

F.31.2. Connection Management

postgres_fdw establishes a connection to a foreign server during the first query that uses a foreign table associated with the foreign server. This connection is kept and re-used for subsequent queries in the same session. However, if multiple user identities (user mappings) are used to access the foreign server, a connection is established for each user mapping.

F.31.3. Transaction Management

During a query that references any remote tables on a foreign server, postgres_fdw opens a transaction on the remote server if one is not already open corresponding to the current local transaction. The remote transaction is committed or aborted when the local transaction commits or aborts. Savepoints are similarly managed by creating corresponding remote savepoints.

The remote transaction uses SERIALIZABLE isolation level when the local transaction has SERIALIZABLE isolation level; otherwise it uses REPEATABLE READ isolation level. This choice ensures that if a query performs multiple table scans on the remote server, it will get snapshot-consistent results for all the scans. A consequence is that successive queries within a single transaction will see the same data from the remote server, even if concurrent updates are occurring on the remote server due to other activities. That behavior would be expected anyway if the local transaction uses SERIALIZABLE or REPEATABLE READ isolation level, but it might be surprising for a READ COMMITTED local transaction. A future PostgreSQL release might modify these rules.

Note that it is currently not supported by postgres_fdw to prepare the remote transaction for two-phase commit.

F.31.4. Remote Query Optimization

postgres_fdw attempts to optimize remote queries to reduce the amount of data transferred from foreign servers. This is done by sending query WHERE clauses to the remote server for execution, and by not retrieving table columns that are not needed for the current query. To reduce the risk of misexecution of queries, WHERE clauses are not sent to the remote server unless they use only built-in data types, operators, and functions. Operators and functions in the clauses must be IMMUTABLE as well.

The query that is actually sent to the remote server for execution can be examined using EXPLAIN VERBOSE.

F.31.5. Remote Query Execution Environment

In the remote sessions opened by postgres_fdw, the search_path parameter is set to just pg_catalog, so that only built-in objects are visible without schema qualification. This is not an issue for queries generated by postgres_fdw itself, because it always supplies such qualification. However, this can pose a hazard for functions that are executed on the remote server via triggers or rules on remote tables. For example, if a remote table is actually a view, any functions used in that view will be executed with the restricted search path. It is recommended to schema-qualify all names in such functions, or else attach SET search_path options (see CREATE FUNCTION) to such functions to establish their expected search path environment.

postgres_fdw likewise establishes remote session settings for the parameters TimeZone, DateStyle, IntervalStyle, and extra_float_digits. These are less likely to be problematic than search_path, but can be handled with function SET options if the need arises.

It is *not* recommended that you override this behavior by changing the session-level settings of these parameters; that is likely to cause postgres_fdw to malfunction.

F.31.6. Cross-Version Compatibility

postgres_fdw can be used with remote servers dating back to PostgreSQL 8.3. Read-only capability is available back to 8.1. A limitation however is that postgres_fdw generally assumes that immutable built-in functions and operators are safe to send to the remote server for execution, if they appear in a WHERE clause for a foreign table. Thus, a built-in function that was added since the remote server's release might be sent to it for execution, resulting in "function does not exist" or a similar error. This type of failure can be worked around by rewriting the query, for example by embedding the foreign table reference in a sub-SELECT with OFFSET 0 as an optimization fence, and placing the problematic function or operator outside the sub-SELECT.

F.31.7. Examples

Here is an example of creating a foreign table with postgres_fdw. First install the extension:

```
CREATE EXTENSION postgres_fdw;
```

Then create a foreign server using CREATE SERVER. In this example we wish to connect to a PostgreSQL server on host 192.83.123.89 listening on port 5432. The database to which the connection is made is named foreign_db on the remote server:

```
CREATE SERVER foreign_server
        FOREIGN DATA WRAPPER postgres_fdw
        OPTIONS (host '192.83.123.89', port '5432', dbname 'foreign_db');
```

A user mapping, defined with CREATE USER MAPPING, is needed as well to identify the role that will be used on the remote server:

```
CREATE USER MAPPING FOR local_user
        SERVER foreign_server
        OPTIONS (user 'foreign_user', password 'password');
```

Now it is possible to create a foreign table with CREATE FOREIGN TABLE. In this example we wish to access the table named some_schema.some_table on the remote server. The local name for it will be foreign_table:

```
CREATE FOREIGN TABLE foreign_table (
        id integer NOT NULL,
        data text
```

```
)
        SERVER foreign_server
        OPTIONS (schema_name 'some_schema', table_name 'some_table');
```

It's essential that the data types and other properties of the columns declared in `CREATE FOREIGN TABLE` match the actual remote table. Column names must match as well, unless you attach `column_name` options to the individual columns to show how they are named in the remote table. In many cases, use of IMPORT FOREIGN SCHEMA is preferable to constructing foreign table definitions manually.

F.31.8. Author

Shigeru Hanada <shigeru.hanada@gmail.com>

F.32. seg

This module implements a data type `seg` for representing line segments, or floating point intervals. `seg` can represent uncertainty in the interval endpoints, making it especially useful for representing laboratory measurements.

F.32.1. Rationale

The geometry of measurements is usually more complex than that of a point in a numeric continuum. A measurement is usually a segment of that continuum with somewhat fuzzy limits. The measurements come out as intervals because of uncertainty and randomness, as well as because the value being measured may naturally be an interval indicating some condition, such as the temperature range of stability of a protein.

Using just common sense, it appears more convenient to store such data as intervals, rather than pairs of numbers. In practice, it even turns out more efficient in most applications.

Further along the line of common sense, the fuzziness of the limits suggests that the use of traditional numeric data types leads to a certain loss of information. Consider this: your instrument reads 6.50, and you input this reading into the database. What do you get when you fetch it? Watch:

```
test=> select 6.50 :: float8 as "pH";
 pH
---
6.5
(1 row)
```

In the world of measurements, 6.50 is not the same as 6.5. It may sometimes be critically different. The experimenters usually write down (and publish) the digits they trust. 6.50 is actually a fuzzy interval contained within a bigger and even fuzzier interval, 6.5, with their center points being (probably) the only common feature they share. We definitely do not want such different data items to appear the same.

Conclusion? It is nice to have a special data type that can record the limits of an interval with arbitrarily variable precision. Variable in the sense that each data element records its own precision.

Check this out:

```
test=> select '6.25 .. 6.50'::seg as "pH";
          pH
------------
6.25 .. 6.50
(1 row)
```

F.32.2. Syntax

The external representation of an interval is formed using one or two floating-point numbers joined by the range operator (`..` or `...`). Alternatively, it can be specified as a center point plus or minus a deviation. Optional certainty indicators (<, > or ~) can be stored as well. (Certainty indicators are ignored by all the built-in operators, however.) Table F-27 gives an overview of allowed representations; Table F-28 shows some examples.

In Table F-27, *x*, *y*, and *delta* denote floating-point numbers. *x* and *y*, but not *delta*, can be preceded by a certainty indicator.

Table F-27. `seg` External Representations

x	Single value (zero-length interval)
x `..` *y*	Interval from *x* to *y*
x `(+-)` *delta*	Interval from *x* - *delta* to *x* + *delta*
x `..`	Open interval with lower bound *x*
`..` *x*	Open interval with upper bound *x*

Table F-28. Examples of Valid `seg` Input

`5.0`	Creates a zero-length segment (a point, if you will)
`~5.0`	Creates a zero-length segment and records ~ in the data. ~ is ignored by `seg` operations, but is preserved as a comment.
`<5.0`	Creates a point at 5.0. < is ignored but is preserved as a comment.
`>5.0`	Creates a point at 5.0. > is ignored but is preserved as a comment.
`5(+-)0.3`	Creates an interval `4.7 .. 5.3`. Note that the `(+-)` notation isn't preserved.
`50 ..`	Everything that is greater than or equal to 50
`.. 0`	Everything that is less than or equal to 0
`1.5e-2 .. 2E-2`	Creates an interval `0.015 .. 0.02`
`1 ... 2`	The same as `1...2`, or `1 .. 2`, or `1..2` (spaces around the range operator are ignored)

Because the `...` operator is widely used in data sources, it is allowed as an alternative spelling of the `..` operator. Unfortunately, this creates a parsing ambiguity: it is not clear whether the upper bound in `0...23` is meant to be `23` or `0.23`. This is resolved by requiring at least one digit before the decimal point in all numbers in `seg` input.

As a sanity check, `seg` rejects intervals with the lower bound greater than the upper, for example `5 .. 2`.

F.32.3. Precision

`seg` values are stored internally as pairs of 32-bit floating point numbers. This means that numbers with more than 7 significant digits will be truncated.

Numbers with 7 or fewer significant digits retain their original precision. That is, if your query returns 0.00, you will be sure that the trailing zeroes are not the artifacts of formatting: they reflect the precision of the original data. The number of leading zeroes does not affect precision: the value 0.0067 is considered to have just 2 significant digits.

F.32.4. Usage

The `seg` module includes a GiST index operator class for `seg` values. The operators supported by the GiST operator class are shown in Table F-29.

Table F-29. Seg GiST Operators

Operator	Description
`[a, b] << [c, d]`	[a, b] is entirely to the left of [c, d]. That is, [a, b] << [c, d] is true if b < c and false otherwise.
`[a, b] >> [c, d]`	[a, b] is entirely to the right of [c, d]. That is, [a, b] >> [c, d] is true if a > d and false otherwise.
`[a, b] &< [c, d]`	Overlaps or is left of — This might be better read as "does not extend to right of". It is true when b <= d.
`[a, b] &> [c, d]`	Overlaps or is right of — This might be better read as "does not extend to left of". It is true when a >= c.
`[a, b] = [c, d]`	Same as — The segments [a, b] and [c, d] are identical, that is, a = c and b = d.
`[a, b] && [c, d]`	The segments [a, b] and [c, d] overlap.
`[a, b] @> [c, d]`	The segment [a, b] contains the segment [c, d], that is, a <= c and b >= d.
`[a, b] <@ [c, d]`	The segment [a, b] is contained in [c, d], that is, a >= c and b <= d.

(Before PostgreSQL 8.2, the containment operators `@>` and `<@` were respectively called `@` and `~`. These names are still available, but are deprecated and will eventually be retired. Notice that the old names are reversed from the convention formerly followed by the core geometric data types!)

The standard B-tree operators are also provided, for example

Operator	Description
`[a, b] < [c, d]`	Less than
`[a, b] > [c, d]`	Greater than

These operators do not make a lot of sense for any practical purpose but sorting. These operators first compare (a) to (c), and if these are equal, compare (b) to (d). That results in reasonably good sorting in most cases, which is useful if you want to use ORDER BY with this type.

F.32.5. Notes

For examples of usage, see the regression test `sql/seg.sql`.

The mechanism that converts `(+-)` to regular ranges isn't completely accurate in determining the number of significant digits for the boundaries. For example, it adds an extra digit to the lower boundary if the resulting interval includes a power of ten:

```
postgres=> select '10(+-)1'::seg as seg;
      seg
---------
9.0 .. 11             -- should be: 9 .. 11
```

The performance of an R-tree index can largely depend on the initial order of input values. It may be very helpful to sort the input table on the `seg` column; see the script `sort-segments.pl` for an example.

F.32.6. Credits

Original author: Gene Selkov, Jr. <`selkovjr@mcs.anl.gov`>, Mathematics and Computer Science Division, Argonne National Laboratory.

My thanks are primarily to Prof. Joe Hellerstein (https://dsf.berkeley.edu/jmh/) for elucidating the gist of the GiST (http://gist.cs.berkeley.edu/). I am also grateful to all Postgres developers, present and past, for enabling myself to create my own world and live undisturbed in it. And I would like to acknowledge my gratitude to Argonne Lab and to the U.S. Department of Energy for the years of faithful support of my database research.

F.33. sepgsql

`sepgsql` is a loadable module that supports label-based mandatory access control (MAC) based on SELinux security policy.

Warning

The current implementation has significant limitations, and does not enforce mandatory access control for all actions. See Section F.33.7.

F.33.1. Overview

This module integrates with SELinux to provide an additional layer of security checking above and beyond what is normally provided by PostgreSQL. From the perspective of SELinux, this module allows PostgreSQL to function as a user-space object manager. Each table or function access initiated by a DML query will be checked against the system security policy. This check is in addition to the usual SQL permissions checking performed by PostgreSQL.

SELinux access control decisions are made using security labels, which are represented by strings such as `system_u:object_r:sepgsql_table_t:s0`. Each access control decision involves two labels: the label of the subject attempting to perform the action, and the label of the object on which the operation is to be performed. Since these labels can be applied to any sort of object, access control decisions for objects stored within the database can be (and, with this module, are) subjected to the same general criteria used for objects of any other type, such as files. This design is intended to allow a centralized security policy to protect information assets independent of the particulars of how those assets are stored.

The SECURITY LABEL statement allows assignment of a security label to a database object.

F.33.2. Installation

`sepgsql` can only be used on Linux 2.6.28 or higher with SELinux enabled. It is not available on any other platform. You will also need libselinux 2.1.10 or higher and selinux-policy 3.9.13 or higher (although some distributions may backport the necessary rules into older policy versions).

The `sestatus` command allows you to check the status of SELinux. A typical display is:

```
$ sestatus
SELinux status:                 enabled
SELinuxfs mount:                /selinux
Current mode:                   enforcing
Mode from config file:          enforcing
Policy version:                 24
Policy from config file:        targeted
```

If SELinux is disabled or not installed, you must set that product up first before installing this module.

To build this module, include the option `--with-selinux` in your PostgreSQL `configure` command. Be sure that the `libselinux-devel` RPM is installed at build time.

To use this module, you must include `sepgsql` in the shared_preload_libraries parameter in `postgresql.conf`. The module will not function correctly if loaded in any other manner. Once the module is loaded, you should execute `sepgsql.sql` in each database. This will install functions needed for security label management, and assign initial security labels.

Here is an example showing how to initialize a fresh database cluster with `sepgsql` functions and security labels installed. Adjust the paths shown as appropriate for your installation:

```
$ export PGDATA=/path/to/data/directory
$ initdb
$ vi $PGDATA/postgresql.conf
  change
    #shared_preload_libraries = ''                # (change requires restart)
  to
    shared_preload_libraries = 'sepgsql'          # (change requires restart)
$ for DBNAME in template0 template1 postgres; do
    postgres --single -F -c exit_on_error=true $DBNAME \
      </usr/local/pgsql/share/contrib/sepgsql.sql >/dev/null
  done
```

Please note that you may see some or all of the following notifications depending on the particular versions you have of libselinux and selinux-policy:

```
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 33 has invalid object type db_blobs
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 36 has invalid object type db_langua
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 37 has invalid object type db_langua
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 38 has invalid object type db_langua
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 39 has invalid object type db_langua
/etc/selinux/targeted/contexts/sepgsql_contexts:  line 40 has invalid object type db_langua
```

These messages are harmless and should be ignored.

If the installation process completes without error, you can now start the server normally.

F.33.3. Regression Tests

Due to the nature of SELinux, running the regression tests for `sepgsql` requires several extra configuration steps, some of which must be done as root. The regression tests will not be run by an ordinary `make check` or `make installcheck` command; you must set up the configuration and then invoke the test script manually. The tests must be run in the `contrib/sepgsql` directory of a configured PostgreSQL build tree. Although they require a build tree, the tests are designed to be executed against an installed server, that is they are comparable to `make installcheck` not `make check`.

First, set up `sepgsql` in a working database according to the instructions in Section F.33.2. Note that the current operating system user must be able to connect to the database as superuser without password authentication.

Second, build and install the policy package for the regression test. The `sepgsql-regtest` policy is a special purpose policy package which provides a set of rules to be allowed during the regression tests. It should be built from the policy source file `sepgsql-regtest.te`, which is done using `make` with a Makefile supplied by SELinux. You will need to locate the appropriate Makefile on your system; the path shown below is only an example. Once built, install this policy package using the `semodule` command, which loads supplied policy packages into the kernel. If the package is correctly installed, `semodule -l` should list `sepgsql-regtest` as an available policy package:

```
$ cd .../contrib/sepgsql
$ make -f /usr/share/selinux/devel/Makefile
$ sudo semodule -u sepgsql-regtest.pp
$ sudo semodule -l | grep sepgsql
sepgsql-regtest 1.07
```

Third, turn on `sepgsql_regression_test_mode`. For security reasons, the rules in `sepgsql-regtest` are not enabled by default; the `sepgsql_regression_test_mode` parameter enables the rules needed to launch the regression tests. It can be turned on using the `setsebool` command:

```
$ sudo setsebool sepgsql_regression_test_mode on
$ getsebool sepgsql_regression_test_mode
sepgsql_regression_test_mode --> on
```

Fourth, verify your shell is operating in the `unconfined_t` domain:

```
$ id -Z
unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
```

See Section F.33.8 for details on adjusting your working domain, if necessary.

Finally, run the regression test script:

```
$ ./test_sepgsql
```

This script will attempt to verify that you have done all the configuration steps correctly, and then it will run the regression tests for the `sepgsql` module.

After completing the tests, it's recommended you disable the `sepgsql_regression_test_mode` parameter:

```
$ sudo setsebool sepgsql_regression_test_mode off
```

You might prefer to remove the `sepgsql-regtest` policy entirely:

```
$ sudo semodule -r sepgsql-regtest
```

F.33.4. GUC Parameters

`sepgsql.permissive` (`boolean`)

This parameter enables `sepgsql` to function in permissive mode, regardless of the system setting. The default is off. This parameter can only be set in the `postgresql.conf` file or on the server command line.

When this parameter is on, `sepgsql` functions in permissive mode, even if SELinux in general is working in enforcing mode. This parameter is primarily useful for testing purposes.

`sepgsql.debug_audit` (`boolean`)

This parameter enables the printing of audit messages regardless of the system policy settings. The default is off, which means that messages will be printed according to the system settings.

The security policy of SELinux also has rules to control whether or not particular accesses are logged. By default, access violations are logged, but allowed accesses are not.

This parameter forces all possible logging to be turned on, regardless of the system policy.

F.33.5. Features

F.33.5.1. Controlled Object Classes

The security model of SELinux describes all the access control rules as relationships between a subject entity (typically, a client of the database) and an object entity (such as a database object), each of which is identified by a security label. If access to an unlabeled object is attempted, the object is treated as if it were assigned the label `unlabeled_t`.

Currently, `sepgsql` allows security labels to be assigned to schemas, tables, columns, sequences, views, and functions. When `sepgsql` is in use, security labels are automatically assigned to supported database objects at creation time. This label is called a default security label, and is decided according to the system security policy, which takes as input the creator's label, the label assigned to the new object's parent object and optionally name of the constructed object.

A new database object basically inherits the security label of the parent object, except when the security policy has special rules known as type-transition rules, in which case a different label may be applied. For schemas, the parent object is the current database; for tables, sequences, views, and functions, it is the containing schema; for columns, it is the containing table.

F.33.5.2. DML Permissions

For tables, `db_table:select`, `db_table:insert`, `db_table:update` or `db_table:delete` are checked for all the referenced target tables depending on the kind of statement; in addition, `db_table:select` is also checked for all the tables that contain columns referenced in the `WHERE` or `RETURNING` clause, as a data source for `UPDATE`, and so on.

Column-level permissions will also be checked for each referenced column. `db_column:select` is checked on not only the columns being read using `SELECT`, but those being referenced in other DML statements; `db_column:update` or `db_column:insert` will also be checked for columns being modified by `UPDATE` or `INSERT`.

For example, consider:

```
UPDATE t1 SET x = 2, y = md5sum(y) WHERE z = 100;
```

Here, `db_column:update` will be checked for `t1.x`, since it is being updated, `db_column:{select update}` will be checked for `t1.y`, since it is both updated and referenced, and `db_column:select` will be checked for `t1.z`, since it is only referenced. `db_table:{select update}` will also be checked at the table level.

For sequences, `db_sequence:get_value` is checked when we reference a sequence object using `SELECT`; however, note that we do not currently check permissions on execution of corresponding functions such as `lastval()`.

For views, `db_view:expand` will be checked, then any other required permissions will be checked on the objects being expanded from the view, individually.

For functions, `db_procedure:{execute}` will be checked when user tries to execute a function as a part of query, or using fast-path invocation. If this function is a trusted procedure, it also checks `db_procedure:{entrypoint}` permission to check whether it can perform as entry point of trusted procedure.

In order to access any schema object, `db_schema:search` permission is required on the containing schema. When an object is referenced without schema qualification, schemas on which this permission is not present will not be searched (just as if the user did not have `USAGE` privilege on the schema). If an explicit schema qualification is present, an error will occur if the user does not have the requisite permission on the named schema.

The client must be allowed to access all referenced tables and columns, even if they originated from views which were then expanded, so that we apply consistent access control rules independent of the manner in which the table contents are referenced.

The default database privilege system allows database superusers to modify system catalogs using DML commands, and reference or modify toast tables. These operations are prohibited when `sepgsql` is enabled.

F.33.5.3. DDL Permissions

SELinux defines several permissions to control common operations for each object type; such as creation, alter, drop and relabel of security label. In addition, several object types have special permissions to control their characteristic operations; such as addition or deletion of name entries within a particular schema.

Creating a new database object requires `create` permission. SELinux will grant or deny this permission based on the client's security label and the proposed security label for the new object. In some cases, additional privileges are required:

- CREATE DATABASE additionally requires `getattr` permission for the source or template database.
- Creating a schema object additionally requires `add_name` permission on the parent schema.
- Creating a table additionally requires permission to create each individual table column, just as if each table column were a separate top-level object.
- Creating a function marked as `LEAKPROOF` additionally requires `install` permission. (This permission is also checked when `LEAKPROOF` is set for an existing function.)

When `DROP` command is executed, `drop` will be checked on the object being removed. Permissions will be also checked for objects dropped indirectly via `CASCADE`. Deletion of objects contained within a particular schema (tables, views, sequences and procedures) additionally requires `remove_name` on the schema.

When `ALTER` command is executed, `setattr` will be checked on the object being modified for each object types, except for subsidiary objects such as the indexes or triggers of a table, where permissions are instead checked on the parent object. In some cases, additional permissions are required:

- Moving an object to a new schema additionally requires `remove_name` permission on the old schema and `add_name` permission on the new one.
- Setting the `LEAKPROOF` attribute on a function requires `install` permission.
- Using SECURITY LABEL on an object additionally requires `relabelfrom` permission for the object in conjunction with its old security label and `relabelto` permission for the object in conjunction with its new security label. (In cases where multiple label providers are installed and the user tries to set a security label, but it is not managed by SELinux, only `setattr` should be checked here. This is currently not done due to implementation restrictions.)

F.33.5.4. Trusted Procedures

Trusted procedures are similar to security definer functions or setuid commands. SELinux provides a feature to allow trusted code to run using a security label different from that of the client, generally for the purpose of providing highly controlled access to sensitive data (e.g., rows might be omitted, or the precision of stored values might be reduced). Whether or not a function acts as a trusted procedure is controlled by its security label and the operating system security policy. For example:

```
postgres=# CREATE TABLE customer (
               cid     int primary key,
               cname   text,
               credit  text
           );
CREATE TABLE
postgres=# SECURITY LABEL ON COLUMN customer.credit
               IS 'system_u:object_r:sepgsql_secret_table_t:s0';
SECURITY LABEL
postgres=# CREATE FUNCTION show_credit(int) RETURNS text
             AS 'SELECT regexp_replace(credit, "-[0-9]+$", "-xxxx", "g")
                         FROM customer WHERE cid = $1'
           LANGUAGE sql;
CREATE FUNCTION
postgres=# SECURITY LABEL ON FUNCTION show_credit(int)
               IS 'system_u:object_r:sepgsql_trusted_proc_exec_t:s0';
SECURITY LABEL
```

The above operations should be performed by an administrative user.

```
postgres=# SELECT * FROM customer;
ERROR:  SELinux: security policy violation
postgres=# SELECT cid, cname, show_credit(cid) FROM customer;
 cid | cname  |     show_credit
-----+--------+---------------------
   1 | taro   | 1111-2222-3333-xxxx
   2 | hanako | 5555-6666-7777-xxxx
(2 rows)
```

In this case, a regular user cannot reference `customer.credit` directly, but a trusted procedure `show_credit` allows him to print the credit card numbers of customers with some of the digits masked out.

F.33.5.5. Dynamic Domain Transitions

It is possible to use SELinux's dynamic domain transition feature to switch the security label of the client process, the client domain, to a new context, if that is allowed by the security policy. The client domain needs the `setcurrent` permission and also `dyntransition` from the old to the new domain.

Dynamic domain transitions should be considered carefully, because they allow users to switch their label, and therefore their privileges, at their option, rather than (as in the case of a trusted procedure) as mandated by the system. Thus, the `dyntransition` permission is only considered safe when used to switch to a domain with a smaller set of privileges than the original one. For example:

```
regression=# select sepgsql_getcon();
                    sepgsql_getcon
-----------------------------------------------------
 unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
(1 row)

regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-s0:c1.c4');
 sepgsql_setcon
----------------
 t
(1 row)

regression=# SELECT sepgsql_setcon('unconfined_u:unconfined_r:unconfined_t:s0-s0:c1.c1023')
ERROR:  SELinux: security policy violation
```

In this example above we were allowed to switch from the larger MCS range `c1.c1023` to the smaller range `c1.c4`, but switching back was denied.

A combination of dynamic domain transition and trusted procedure enables an interesting use case that fits the typical process life-cycle of connection pooling software. Even if your connection pooling software is not allowed to run most of SQL commands, you can allow it to switch the security label of the client using the `sepgsql_setcon()` function from within a trusted procedure; that should take some credential to authorize the request to switch the client label. After that, this session will have the privileges of the target user, rather than the connection pooler. The connection pooler can later revert the security label change by again using `sepgsql_setcon()` with `NULL` argument, again invoked from within a trusted procedure with appropriate permissions checks. The point here is that only the trusted procedure actually has permission to change the effective security label, and only does so when given proper credentials. Of course, for secure operation, the credential store (table, procedure definition, or whatever) must be protected from unauthorized access.

F.33.5.6. Miscellaneous

We reject the LOAD command across the board, because any module loaded could easily circumvent security policy enforcement.

F.33.6. Sepgsql Functions

Table F-30 shows the available functions.

Table F-30. Sepgsql Functions

`sepgsql_getcon() returns text`	Returns the client domain, the current security label of the client.
`sepgsql_setcon(text) returns bool`	Switches the client domain of the current session to the new domain, if allowed by the security policy. It also accepts `NULL` input as a request to transition to the client's original domain.

`sepgsql_mcstrans_in(text) returns text`	Translates the given qualified MLS/MCS range into raw format if the mcstrans daemon is running.
`sepgsql_mcstrans_out(text) returns text`	Translates the given raw MLS/MCS range into qualified format if the mcstrans daemon is running.
`sepgsql_restorecon(text) returns bool`	Sets up initial security labels for all objects within the current database. The argument may be NULL, or the name of a specfile to be used as alternative of the system default.

F.33.7. Limitations

Data Definition Language (DDL) Permissions

Due to implementation restrictions, some DDL operations do not check permissions.

Data Control Language (DCL) Permissions

Due to implementation restrictions, DCL operations do not check permissions.

Row-level access control

PostgreSQL supports row-level access, but `sepgsql` does not.

Covert channels

`sepgsql` does not try to hide the existence of a certain object, even if the user is not allowed to reference it. For example, we can infer the existence of an invisible object as a result of primary key conflicts, foreign key violations, and so on, even if we cannot obtain the contents of the object. The existence of a top secret table cannot be hidden; we only hope to conceal its contents.

F.33.8. External Resources

SE-PostgreSQL Introduction[5]

This wiki page provides a brief overview, security design, architecture, administration and upcoming features.

Fedora SELinux User Guide[6]

This document provides a wide spectrum of knowledge to administer SELinux on your systems. It focuses primarily on Fedora, but is not limited to Fedora.

5. https://wiki.postgresql.org/wiki/SEPostgreSQL
6. http://docs.fedoraproject.org/selinux-user-guide/

Fedora SELinux FAQ[7]

This document answers frequently asked questions about SELinux. It focuses primarily on Fedora, but is not limited to Fedora.

F.33.9. Author

KaiGai Kohei <kaigai@ak.jp.nec.com>

F.34. spi

The spi module provides several workable examples of using the Server Programming Interface (SPI) and triggers. While these functions are of some value in their own right, they are even more useful as examples to modify for your own purposes. The functions are general enough to be used with any table, but you have to specify table and field names (as described below) while creating a trigger.

Each of the groups of functions described below is provided as a separately-installable extension.

F.34.1. refint — Functions for Implementing Referential Integrity

`check_primary_key()` and `check_foreign_key()` are used to check foreign key constraints. (This functionality is long since superseded by the built-in foreign key mechanism, of course, but the module is still useful as an example.)

`check_primary_key()` checks the referencing table. To use, create a `BEFORE INSERT OR UPDATE` trigger using this function on a table referencing another table. Specify as the trigger arguments: the referencing table's column name(s) which form the foreign key, the referenced table name, and the column names in the referenced table which form the primary/unique key. To handle multiple foreign keys, create a trigger for each reference.

`check_foreign_key()` checks the referenced table. To use, create a `BEFORE DELETE OR UPDATE` trigger using this function on a table referenced by other table(s). Specify as the trigger arguments: the number of referencing tables for which the function has to perform checking, the action if a referencing key is found (`cascade` — to delete the referencing row, `restrict` — to abort transaction if referencing keys exist, `setnull` — to set referencing key fields to null), the triggered table's column names which form the primary/unique key, then the referencing table name and column names (repeated for as many referencing tables as were specified by first argument). Note that the primary/unique key columns should be marked NOT NULL and should have a unique index.

There are examples in `refint.example`.

7. http://docs.fedoraproject.org/selinux-faq

F.34.2. timetravel — Functions for Implementing Time Travel

Long ago, PostgreSQL had a built-in time travel feature that kept the insert and delete times for each tuple. This can be emulated using these functions. To use these functions, you must add to a table two columns of `abstime` type to store the date when a tuple was inserted (start_date) and changed/deleted (stop_date):

```
CREATE TABLE mytab (
        ...             ...
        start_date      abstime,
        stop_date       abstime
        ...             ...
);
```

The columns can be named whatever you like, but in this discussion we'll call them start_date and stop_date.

When a new row is inserted, start_date should normally be set to current time, and stop_date to `infinity`. The trigger will automatically substitute these values if the inserted data contains nulls in these columns. Generally, inserting explicit non-null data in these columns should only be done when re-loading dumped data.

Tuples with stop_date equal to `infinity` are "valid now", and can be modified. Tuples with a finite stop_date cannot be modified anymore — the trigger will prevent it. (If you need to do that, you can turn off time travel as shown below.)

For a modifiable row, on update only the stop_date in the tuple being updated will be changed (to current time) and a new tuple with the modified data will be inserted. Start_date in this new tuple will be set to current time and stop_date to `infinity`.

A delete does not actually remove the tuple but only sets its stop_date to current time.

To query for tuples "valid now", include `stop_date = 'infinity'` in the query's WHERE condition. (You might wish to incorporate that in a view.) Similarly, you can query for tuples valid at any past time with suitable conditions on start_date and stop_date.

`timetravel()` is the general trigger function that supports this behavior. Create a `BEFORE INSERT OR UPDATE OR DELETE` trigger using this function on each time-traveled table. Specify two trigger arguments: the actual names of the start_date and stop_date columns. Optionally, you can specify one to three more arguments, which must refer to columns of type `text`. The trigger will store the name of the current user into the first of these columns during INSERT, the second column during UPDATE, and the third during DELETE.

`set_timetravel()` allows you to turn time-travel on or off for a table. `set_timetravel('mytab', 1)` will turn TT ON for table `mytab`. `set_timetravel('mytab', 0)` will turn TT OFF for table `mytab`. In both cases the old status is reported. While TT is off, you can modify the start_date and stop_date columns freely. Note that the on/off status is local to the current database session — fresh sessions will always start out with TT ON for all tables.

`get_timetravel()` returns the TT state for a table without changing it.

There is an example in `timetravel.example`.

F.34.3. autoinc — Functions for Autoincrementing Fields

`autoinc()` is a trigger that stores the next value of a sequence into an integer field. This has some overlap with the built-in "serial column" feature, but it is not the same: `autoinc()` will override attempts to substitute a different field value during inserts, and optionally it can be used to increment the field during updates, too.

To use, create a `BEFORE INSERT` (or optionally `BEFORE INSERT OR UPDATE`) trigger using this function. Specify two trigger arguments: the name of the integer column to be modified, and the name of the sequence object that will supply values. (Actually, you can specify any number of pairs of such names, if you'd like to update more than one autoincrementing column.)

There is an example in `autoinc.example`.

F.34.4. insert_username — Functions for Tracking Who Changed a Table

`insert_username()` is a trigger that stores the current user's name into a text field. This can be useful for tracking who last modified a particular row within a table.

To use, create a `BEFORE INSERT` and/or `UPDATE` trigger using this function. Specify a single trigger argument: the name of the text column to be modified.

There is an example in `insert_username.example`.

F.34.5. moddatetime — Functions for Tracking Last Modification Time

`moddatetime()` is a trigger that stores the current time into a `timestamp` field. This can be useful for tracking the last modification time of a particular row within a table.

To use, create a `BEFORE UPDATE` trigger using this function. Specify a single trigger argument: the name of the column to be modified. The column must be of type `timestamp` or `timestamp with time zone`.

There is an example in `moddatetime.example`.

F.35. sslinfo

The `sslinfo` module provides information about the SSL certificate that the current client provided when connecting to PostgreSQL. The module is useless (most functions will return NULL) if the current connection does not use SSL.

This extension won't build at all unless the installation was configured with `--with-openssl`.

F.35.1. Functions Provided

`ssl_is_used() returns boolean`

Returns TRUE if current connection to server uses SSL, and FALSE otherwise.

`ssl_version() returns text`

Returns the name of the protocol used for the SSL connection (e.g., SSLv2, SSLv3, or TLSv1).

`ssl_cipher() returns text`

Returns the name of the cipher used for the SSL connection (e.g., DHE-RSA-AES256-SHA).

`ssl_client_cert_present() returns boolean`

Returns TRUE if current client has presented a valid SSL client certificate to the server, and FALSE otherwise. (The server might or might not be configured to require a client certificate.)

`ssl_client_serial() returns numeric`

Returns serial number of current client certificate. The combination of certificate serial number and certificate issuer is guaranteed to uniquely identify a certificate (but not its owner — the owner ought to regularly change his keys, and get new certificates from the issuer).

So, if you run your own CA and allow only certificates from this CA to be accepted by the server, the serial number is the most reliable (albeit not very mnemonic) means to identify a user.

`ssl_client_dn() returns text`

Returns the full subject of the current client certificate, converting character data into the current database encoding. It is assumed that if you use non-ASCII characters in the certificate names, your database is able to represent these characters, too. If your database uses the SQL_ASCII encoding, non-ASCII characters in the name will be represented as UTF-8 sequences.

The result looks like `/CN=Somebody /C=Some country/O=Some organization`.

`ssl_issuer_dn() returns text`

Returns the full issuer name of the current client certificate, converting character data into the current database encoding. Encoding conversions are handled the same as for `ssl_client_dn`.

The combination of the return value of this function with the certificate serial number uniquely identifies the certificate.

This function is really useful only if you have more than one trusted CA certificate in your server's `root.crt` file, or if this CA has issued some intermediate certificate authority certificates.

`ssl_client_dn_field(fieldname text) returns text`

This function returns the value of the specified field in the certificate subject, or NULL if the field is not present. Field names are string constants that are converted into ASN1 object identifiers using the OpenSSL object database. The following values are acceptable:

```
commonName (alias CN)
surname (alias SN)
name
givenName (alias GN)
countryName (alias C)
localityName (alias L)
stateOrProvinceName (alias ST)
```

```
organizationName (alias O)
organizationalUnitName (alias OU)
title
description
initials
postalCode
streetAddress
generationQualifier
description
dnQualifier
x500UniqueIdentifier
pseudonym
role
emailAddress
```

All of these fields are optional, except `commonName`. It depends entirely on your CA's policy which of them would be included and which wouldn't. The meaning of these fields, however, is strictly defined by the X.500 and X.509 standards, so you cannot just assign arbitrary meaning to them.

`ssl_issuer_field(fieldname text) returns text`

Same as `ssl_client_dn_field`, but for the certificate issuer rather than the certificate subject.

F.35.2. Author

Victor Wagner <`vitus@cryptocom.ru`>, Cryptocom LTD

E-Mail of Cryptocom OpenSSL development group: <`openssl@cryptocom.ru`>

F.36. tablefunc

The `tablefunc` module includes various functions that return tables (that is, multiple rows). These functions are useful both in their own right and as examples of how to write C functions that return multiple rows.

F.36.1. Functions Provided

Table F-31 shows the functions provided by the `tablefunc` module.

Table F-31. `tablefunc` Functions

Function	Returns	Description
`normal_rand(int numvals, float8 mean, float8 stddev)`	`setof float8`	Produces a set of normally distributed random values

Function	Returns	Description
crosstab(text sql)	setof record	Produces a "pivot table" containing row names plus *N* value columns, where *N* is determined by the row type specified in the calling query
crosstab*N*(text sql)	setof table_crosstab_*N*	Produces a "pivot table" containing row names plus *N* value columns. crosstab2, crosstab3, and crosstab4 are predefined, but you can create additional crosstab*N* functions as described below
crosstab(text source_sql, text category_sql)	setof record	Produces a "pivot table" with the value columns specified by a second query
crosstab(text sql, int N)	setof record	Obsolete version of crosstab(text). The parameter *N* is now ignored, since the number of value columns is always determined by the calling query
connectby(text relname, text keyid_fld, text parent_keyid_fld [, text orderby_fld], text start_with, int max_depth [, text branch_delim])	setof record	Produces a representation of a hierarchical tree structure

F.36.1.1. normal_rand

```
normal_rand(int numvals, float8 mean, float8 stddev) returns setof float8
```

normal_rand produces a set of normally distributed random values (Gaussian distribution).

numvals is the number of values to be returned from the function. mean is the mean of the normal distribution of values and stddev is the standard deviation of the normal distribution of values.

For example, this call requests 1000 values with a mean of 5 and a standard deviation of 3:

```
test=# SELECT * FROM normal_rand(1000, 5, 3);
     normal_rand
---------------------
     1.56556322244898
     9.10040991424657
     5.36957140345079
    -0.369151492880995
     0.283600703686639
```

```
        .
        .
        .
     4.82992125404908
     9.71308014517282
     2.49639286969028
(1000 rows)
```

F.36.1.2. `crosstab(text)`

```
crosstab(text sql)
crosstab(text sql, int N)
```

The `crosstab` function is used to produce "pivot" displays, wherein data is listed across the page rather than down. For example, we might have data like

```
row1    val11
row1    val12
row1    val13
...
row2    val21
row2    val22
row2    val23
...
```

which we wish to display like

```
row1    val11   val12   val13   ...
row2    val21   val22   val23   ...
...
```

The `crosstab` function takes a text parameter that is a SQL query producing raw data formatted in the first way, and produces a table formatted in the second way.

The `sql` parameter is a SQL statement that produces the source set of data. This statement must return one `row_name` column, one `category` column, and one `value` column. `N` is an obsolete parameter, ignored if supplied (formerly this had to match the number of output value columns, but now that is determined by the calling query).

For example, the provided query might produce a set something like:

```
 row_name    cat    value
----------+-------+-------
  row1      cat1    val1
  row1      cat2    val2
  row1      cat3    val3
  row1      cat4    val4
  row2      cat1    val5
  row2      cat2    val6
  row2      cat3    val7
  row2      cat4    val8
```

The `crosstab` function is declared to return `setof record`, so the actual names and types of the output columns must be defined in the `FROM` clause of the calling `SELECT` statement, for example:

```
SELECT * FROM crosstab('...') AS ct(row_name text, category_1 text, category_2 text);
```

This example produces a set something like:

```
           <== value  columns  ==>
 row_name   category_1   category_2
----------+------------+------------
  row1        val1         val2
  row2        val5         val6
```

The `FROM` clause must define the output as one `row_name` column (of the same data type as the first result column of the SQL query) followed by N `value` columns (all of the same data type as the third result column of the SQL query). You can set up as many output value columns as you wish. The names of the output columns are up to you.

The `crosstab` function produces one output row for each consecutive group of input rows with the same `row_name` value. It fills the output `value` columns, left to right, with the `value` fields from these rows. If there are fewer rows in a group than there are output `value` columns, the extra output columns are filled with nulls; if there are more rows, the extra input rows are skipped.

In practice the SQL query should always specify `ORDER BY 1,2` to ensure that the input rows are properly ordered, that is, values with the same `row_name` are brought together and correctly ordered within the row. Notice that `crosstab` itself does not pay any attention to the second column of the query result; it's just there to be ordered by, to control the order in which the third-column values appear across the page.

Here is a complete example:

```
CREATE TABLE ct(id SERIAL, rowid TEXT, attribute TEXT, value TEXT);
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att1','val1');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att2','val2');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att3','val3');
INSERT INTO ct(rowid, attribute, value) VALUES('test1','att4','val4');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att1','val5');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att2','val6');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att3','val7');
INSERT INTO ct(rowid, attribute, value) VALUES('test2','att4','val8');

SELECT *
FROM crosstab(
  'select rowid, attribute, value
   from ct
   where attribute = "att2" or attribute = "att3"
   order by 1,2')
AS ct(row_name text, category_1 text, category_2 text, category_3 text);

 row_name | category_1 | category_2 | category_3
----------+------------+------------+------------
 test1    | val2       | val3       |
```

```
 test2    | val6       | val7       |
(2 rows)
```

You can avoid always having to write out a FROM clause to define the output columns, by setting up a custom crosstab function that has the desired output row type wired into its definition. This is described in the next section. Another possibility is to embed the required FROM clause in a view definition.

F.36.1.3. crosstab*N*(text)

```
crosstabN(text sql)
```

The crosstab*N* functions are examples of how to set up custom wrappers for the general crosstab function, so that you need not write out column names and types in the calling SELECT query. The tablefunc module includes crosstab2, crosstab3, and crosstab4, whose output row types are defined as

```
CREATE TYPE tablefunc_crosstab_N AS (
    row_name TEXT,
    category_1 TEXT,
    category_2 TEXT,
        .
        .
        .
    category_N TEXT
);
```

Thus, these functions can be used directly when the input query produces row_name and value columns of type text, and you want 2, 3, or 4 output values columns. In all other ways they behave exactly as described above for the general crosstab function.

For instance, the example given in the previous section would also work as

```
SELECT *
FROM crosstab3(
  'select rowid, attribute, value
   from ct
   where attribute = "att2" or attribute = "att3"
   order by 1,2');
```

These functions are provided mostly for illustration purposes. You can create your own return types and functions based on the underlying crosstab() function. There are two ways to do it:

- Create a composite type describing the desired output columns, similar to the examples in contrib/tablefunc/tablefunc--1.0.sql. Then define a unique function name accepting one text parameter and returning setof your_type_name, but linking to the same underlying crosstab C function. For example, if your source data produces row names that are text, and values that are float8, and you want 5 value columns:

  ```
  CREATE TYPE my_crosstab_float8_5_cols AS (
  ```

```
    my_row_name text,
    my_category_1 float8,
    my_category_2 float8,
    my_category_3 float8,
    my_category_4 float8,
    my_category_5 float8
);

CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(text)
    RETURNS setof my_crosstab_float8_5_cols
    AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;
```

- Use `OUT` parameters to define the return type implicitly. The same example could also be done this way:

```
CREATE OR REPLACE FUNCTION crosstab_float8_5_cols(
    IN text,
    OUT my_row_name text,
    OUT my_category_1 float8,
    OUT my_category_2 float8,
    OUT my_category_3 float8,
    OUT my_category_4 float8,
    OUT my_category_5 float8)
  RETURNS setof record
  AS '$libdir/tablefunc','crosstab' LANGUAGE C STABLE STRICT;
```

F.36.1.4. `crosstab(text, text)`

```
crosstab(text source_sql, text category_sql)
```

The main limitation of the single-parameter form of `crosstab` is that it treats all values in a group alike, inserting each value into the first available column. If you want the value columns to correspond to specific categories of data, and some groups might not have data for some of the categories, that doesn't work well. The two-parameter form of `crosstab` handles this case by providing an explicit list of the categories corresponding to the output columns.

`source_sql` is a SQL statement that produces the source set of data. This statement must return one `row_name` column, one `category` column, and one `value` column. It may also have one or more "extra" columns. The `row_name` column must be first. The `category` and `value` columns must be the last two columns, in that order. Any columns between `row_name` and `category` are treated as "extra". The "extra" columns are expected to be the same for all rows with the same `row_name` value.

For example, `source_sql` might produce a set something like:

```
SELECT row_name, extra_col, cat, value FROM foo ORDER BY 1;

 row_name    extra_col   cat    value
----------+------------+-----+---------
  row1         extra1    cat1    val1
  row1         extra1    cat2    val2
  row1         extra1    cat4    val4
  row2         extra2    cat1    val5
```

```
  row2         extra2    cat2    val6
  row2         extra2    cat3    val7
  row2         extra2    cat4    val8
```

category_sql is a SQL statement that produces the set of categories. This statement must return only one column. It must produce at least one row, or an error will be generated. Also, it must not produce duplicate values, or an error will be generated. category_sql might be something like:

```
SELECT DISTINCT cat FROM foo ORDER BY 1;
    cat
  -------
    cat1
    cat2
    cat3
    cat4
```

The crosstab function is declared to return setof record, so the actual names and types of the output columns must be defined in the FROM clause of the calling SELECT statement, for example:

```
SELECT * FROM crosstab('...', '...')
    AS ct(row_name text, extra text, cat1 text, cat2 text, cat3 text, cat4 text);
```

This will produce a result something like:

```
                  <==  value  columns   ==>
row_name   extra   cat1   cat2   cat3   cat4
---------+-------+------+------+------+------
  row1     extra1  val1   val2          val4
  row2     extra2  val5   val6   val7   val8
```

The FROM clause must define the proper number of output columns of the proper data types. If there are *N* columns in the source_sql query's result, the first *N*-2 of them must match up with the first *N*-2 output columns. The remaining output columns must have the type of the last column of the source_sql query's result, and there must be exactly as many of them as there are rows in the category_sql query's result.

The crosstab function produces one output row for each consecutive group of input rows with the same row_name value. The output row_name column, plus any "extra" columns, are copied from the first row of the group. The output value columns are filled with the value fields from rows having matching category values. If a row's category does not match any output of the category_sql query, its value is ignored. Output columns whose matching category is not present in any input row of the group are filled with nulls.

In practice the source_sql query should always specify ORDER BY 1 to ensure that values with the same row_name are brought together. However, ordering of the categories within a group is not important. Also, it is essential to be sure that the order of the category_sql query's output matches the specified output column order.

Here are two complete examples:

```
create table sales(year int, month int, qty int);
insert into sales values(2007, 1, 1000);
insert into sales values(2007, 2, 1500);
insert into sales values(2007, 7, 500);
insert into sales values(2007, 11, 1500);
insert into sales values(2007, 12, 2000);
insert into sales values(2008, 1, 1000);

select * from crosstab(
  'select year, month, qty from sales order by 1',
  'select m from generate_series(1,12) m'
) as (
  year int,
  "Jan" int,
  "Feb" int,
  "Mar" int,
  "Apr" int,
  "May" int,
  "Jun" int,
  "Jul" int,
  "Aug" int,
  "Sep" int,
  "Oct" int,
  "Nov" int,
  "Dec" int
);
 year | Jan  | Feb  | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov  | Dec
------+------+------+-----+-----+-----+-----+-----+-----+-----+-----+------+------
 2007 | 1000 | 1500 |     |     |     |     | 500 |     |     |     | 1500 | 2000
 2008 | 1000 |      |     |     |     |     |     |     |     |     |      |
(2 rows)

CREATE TABLE cth(rowid text, rowdt timestamp, attribute text, val text);
INSERT INTO cth VALUES('test1','01 March 2003','temperature','42');
INSERT INTO cth VALUES('test1','01 March 2003','test_result','PASS');
INSERT INTO cth VALUES('test1','01 March 2003','volts','2.6987');
INSERT INTO cth VALUES('test2','02 March 2003','temperature','53');
INSERT INTO cth VALUES('test2','02 March 2003','test_result','FAIL');
INSERT INTO cth VALUES('test2','02 March 2003','test_startdate','01 March 2003');
INSERT INTO cth VALUES('test2','02 March 2003','volts','3.1234');

SELECT * FROM crosstab
(
  'SELECT rowid, rowdt, attribute, val FROM cth ORDER BY 1',
  'SELECT DISTINCT attribute FROM cth ORDER BY 1'
)
AS
(
       rowid text,
       rowdt timestamp,
       temperature int4,
```

```
       test_result text,
       test_startdate timestamp,
       volts float8
);
 rowid |          rowdt           | temperature | test_result |      test_startdate      |
-------+--------------------------+-------------+-------------+--------------------------+-
 test1 | Sat Mar 01 00:00:00 2003 |          42 | PASS        |                          |
 test2 | Sun Mar 02 00:00:00 2003 |          53 | FAIL        | Sat Mar 01 00:00:00 2003 |
(2 rows)
```

You can create predefined functions to avoid having to write out the result column names and types in each query. See the examples in the previous section. The underlying C function for this form of `crosstab` is named `crosstab_hash`.

F.36.1.5. `connectby`

```
connectby(text relname, text keyid_fld, text parent_keyid_fld
          [, text orderby_fld ], text start_with, int max_depth
          [, text branch_delim ])
```

The `connectby` function produces a display of hierarchical data that is stored in a table. The table must have a key field that uniquely identifies rows, and a parent-key field that references the parent (if any) of each row. `connectby` can display the sub-tree descending from any row.

Table F-32 explains the parameters.

Table F-32. `connectby` Parameters

Parameter	Description
`relname`	Name of the source relation
`keyid_fld`	Name of the key field
`parent_keyid_fld`	Name of the parent-key field
`orderby_fld`	Name of the field to order siblings by (optional)
`start_with`	Key value of the row to start at
`max_depth`	Maximum depth to descend to, or zero for unlimited depth
`branch_delim`	String to separate keys with in branch output (optional)

The key and parent-key fields can be any data type, but they must be the same type. Note that the `start_with` value must be entered as a text string, regardless of the type of the key field.

The `connectby` function is declared to return `setof record`, so the actual names and types of the output columns must be defined in the `FROM` clause of the calling `SELECT` statement, for example:

```
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0, '~')
    AS t(keyid text, parent_keyid text, level int, branch text, pos int);
```

The first two output columns are used for the current row's key and its parent row's key; they must match the type of the table's key field. The third output column is the depth in the tree and must be of type `integer`. If a `branch_delim` parameter was given, the next output column is the branch display and must be of type `text`. Finally, if an `orderby_fld` parameter was given, the last output column is a serial number, and must be of type `integer`.

The "branch" output column shows the path of keys taken to reach the current row. The keys are separated by the specified `branch_delim` string. If no branch display is wanted, omit both the `branch_delim` parameter and the branch column in the output column list.

If the ordering of siblings of the same parent is important, include the `orderby_fld` parameter to specify which field to order siblings by. This field can be of any sortable data type. The output column list must include a final integer serial-number column, if and only if `orderby_fld` is specified.

The parameters representing table and field names are copied as-is into the SQL queries that `connectby` generates internally. Therefore, include double quotes if the names are mixed-case or contain special characters. You may also need to schema-qualify the table name.

In large tables, performance will be poor unless there is an index on the parent-key field.

It is important that the `branch_delim` string not appear in any key values, else `connectby` may incorrectly report an infinite-recursion error. Note that if `branch_delim` is not provided, a default value of `~` is used for recursion detection purposes.

Here is an example:

```
CREATE TABLE connectby_tree(keyid text, parent_keyid text, pos int);

INSERT INTO connectby_tree VALUES('row1',NULL, 0);
INSERT INTO connectby_tree VALUES('row2','row1', 0);
INSERT INTO connectby_tree VALUES('row3','row1', 0);
INSERT INTO connectby_tree VALUES('row4','row2', 1);
INSERT INTO connectby_tree VALUES('row5','row2', 0);
INSERT INTO connectby_tree VALUES('row6','row4', 0);
INSERT INTO connectby_tree VALUES('row7','row3', 0);
INSERT INTO connectby_tree VALUES('row8','row6', 0);
INSERT INTO connectby_tree VALUES('row9','row5', 0);

-- with branch, without orderby_fld (order of results is not guaranteed)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0, '~')
 AS t(keyid text, parent_keyid text, level int, branch text);
 keyid | parent_keyid | level |       branch
-------+--------------+-------+---------------------
 row2  |              |     0 | row2
 row4  | row2         |     1 | row2~row4
 row6  | row4         |     2 | row2~row4~row6
 row8  | row6         |     3 | row2~row4~row6~row8
 row5  | row2         |     1 | row2~row5
 row9  | row5         |     2 | row2~row5~row9
(6 rows)

-- without branch, without orderby_fld (order of results is not guaranteed)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'row2', 0)
 AS t(keyid text, parent_keyid text, level int);
 keyid | parent_keyid | level
```

```
-------+--------------+-------
 row2  |              |     0
 row4  | row2         |     1
 row6  | row4         |     2
 row8  | row6         |     3
 row5  | row2         |     1
 row9  | row5         |     2
(6 rows)

-- with branch, with orderby_fld (notice that row5 comes before row4)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0, '~')
 AS t(keyid text, parent_keyid text, level int, branch text, pos int);
 keyid | parent_keyid | level |       branch        | pos
-------+--------------+-------+---------------------+-----
 row2  |              |     0 | row2                |   1
 row5  | row2         |     1 | row2~row5           |   2
 row9  | row5         |     2 | row2~row5~row9      |   3
 row4  | row2         |     1 | row2~row4           |   4
 row6  | row4         |     2 | row2~row4~row6      |   5
 row8  | row6         |     3 | row2~row4~row6~row8 |   6
(6 rows)

-- without branch, with orderby_fld (notice that row5 comes before row4)
SELECT * FROM connectby('connectby_tree', 'keyid', 'parent_keyid', 'pos', 'row2', 0)
 AS t(keyid text, parent_keyid text, level int, pos int);
 keyid | parent_keyid | level | pos
-------+--------------+-------+-----
 row2  |              |     0 |   1
 row5  | row2         |     1 |   2
 row9  | row5         |     2 |   3
 row4  | row2         |     1 |   4
 row6  | row4         |     2 |   5
 row8  | row6         |     3 |   6
(6 rows)
```

F.36.2. Author

Joe Conway

F.37. tcn

The `tcn` module provides a trigger function that notifies listeners of changes to any table on which it is attached. It must be used as an `AFTER` trigger `FOR EACH ROW`.

Only one parameter may be supplied to the function in a `CREATE TRIGGER` statement, and that is optional. If supplied it will be used for the channel name for the notifications. If omitted `tcn` will be used for the channel name.

The payload of the notifications consists of the table name, a letter to indicate which type of operation was performed, and column name/value pairs for primary key columns. Each part is separated from the next by a comma. For ease of parsing using regular expressions, table and column names are always wrapped in double quotes, and data values are always wrapped in single quotes. Embedded quotes are doubled.

A brief example of using the extension follows.

```
test=# create table tcndata
test-#   (
test(#     a int not null,
test(#     b date not null,
test(#     c text,
test(#     primary key (a, b)
test(#   );
CREATE TABLE
test=# create trigger tcndata_tcn_trigger
test-#   after insert or update or delete on tcndata
test-#   for each row execute procedure triggered_change_notification();
CREATE TRIGGER
test=# listen tcn;
LISTEN
test=# insert into tcndata values (1, date '2012-12-22', 'one'),
test-#                            (1, date '2012-12-23', 'another'),
test-#                            (2, date '2012-12-23', 'two');
INSERT 0 3
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='1',"b"='2012-12-22'" receive
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='1',"b"='2012-12-23'" receive
Asynchronous notification "tcn" with payload ""tcndata",I,"a"='2',"b"='2012-12-23'" receive
test=# update tcndata set c = 'uno' where a = 1;
UPDATE 2
Asynchronous notification "tcn" with payload ""tcndata",U,"a"='1',"b"='2012-12-22'" receive
Asynchronous notification "tcn" with payload ""tcndata",U,"a"='1',"b"='2012-12-23'" receive
test=# delete from tcndata where a = 1 and b = date '2012-12-22';
DELETE 1
Asynchronous notification "tcn" with payload ""tcndata",D,"a"='1',"b"='2012-12-22'" receive
```

F.38. test_decoding

`test_decoding` is an example of a logical decoding output plugin. It doesn't do anything especially useful, but can serve as a starting point for developing your own output plugin.

`test_decoding` receives WAL through the logical decoding mechanism and decodes it into text representations of the operations performed.

Typical output from this plugin, used over the SQL logical decoding interface, might be:

```
postgres=# SELECT * FROM pg_logical_slot_get_changes('test_slot', NULL, NULL, 'include-xids
 location   | xid |                       data
------------+-----+------------------------------------------------
 0/16D30F8 | 691 | BEGIN
 0/16D32A0 | 691 | table public.data: INSERT: id[int4]:2 data[text]:'arg'
 0/16D32A0 | 691 | table public.data: INSERT: id[int4]:3 data[text]:'demo'
 0/16D32A0 | 691 | COMMIT
 0/16D32D8 | 692 | BEGIN
 0/16D3398 | 692 | table public.data: DELETE: id[int4]:2
 0/16D3398 | 692 | table public.data: DELETE: id[int4]:3
 0/16D3398 | 692 | COMMIT
(8 rows)
```

F.39. tsearch2

The tsearch2 module provides backwards-compatible text search functionality for applications that used tsearch2 before text searching was integrated into core PostgreSQL in release 8.3.

F.39.1. Portability Issues

Although the built-in text search features were based on tsearch2 and are largely similar to it, there are numerous small differences that will create portability issues for existing applications:

- Some functions' names were changed, for example `rank` to `ts_rank`. The replacement `tsearch2` module provides aliases having the old names.
- The built-in text search data types and functions all exist within the system schema `pg_catalog`. In an installation using tsearch2, these objects would usually have been in the `public` schema, though some users chose to place them in a separate schema of their own. Explicitly schema-qualified references to the objects will therefore fail in either case. The replacement `tsearch2` module provides alias objects that are stored in `public` (or another schema if necessary) so that such references will still work.
- There is no concept of a "current parser" or "current dictionary" in the built-in text search features, only of a current search configuration (set by the `default_text_search_config` parameter). While the current parser and current dictionary were used only by functions intended for debugging, this might still pose a porting obstacle in some cases. The replacement `tsearch2` module emulates these additional state variables and provides backwards-compatible functions for setting and retrieving them.

There are some issues that are not addressed by the replacement `tsearch2` module, and will therefore require application code changes in any case:

- The old `tsearch2` trigger function allowed items in its argument list to be names of functions to be invoked on the text data before it was converted to `tsvector` format. This was removed as being a security hole, since it was not possible to guarantee that the function invoked was the one intended. The recommended approach if the data must be massaged before being indexed is to write a custom trigger that does the work for itself.

- Text search configuration information has been moved into core system catalogs that are noticeably different from the tables used by tsearch2. Any applications that examined or modified those tables will need adjustment.
- If an application used any custom text search configurations, those will need to be set up in the core catalogs using the new text search configuration SQL commands. The replacement `tsearch2` module offers a little bit of support for this by making it possible to load an old set of tsearch2 configuration tables into PostgreSQL 8.3. (Without the module, it is not possible to load the configuration data because values in the `regprocedure` columns cannot be resolved to functions.) While those configuration tables won't actually *do* anything, at least their contents will be available to be consulted while setting up an equivalent custom configuration in 8.3.
- The old `reset_tsearch()` and `get_covers()` functions are not supported.
- The replacement `tsearch2` module does not define any alias operators, relying entirely on the built-in ones. This would only pose an issue if an application used explicitly schema-qualified operator names, which is very uncommon.

F.39.2. Converting a pre-8.3 Installation

The recommended way to update a pre-8.3 installation that uses tsearch2 is:

1. Make a dump from the old installation in the usual way, but be sure not to use `-c` (`--clean`) option of pg_dump or pg_dumpall.
2. In the new installation, create empty database(s) and install the replacement `tsearch2` module into each database that will use text search. This must be done *before* loading the dump data! If your old installation had the tsearch2 objects in a schema other than `public`, be sure to adjust the `CREATE EXTENSION` command so that the replacement objects are created in that same schema.
3. Load the dump data. There will be quite a few errors reported due to failure to recreate the original tsearch2 objects. These errors can be ignored, but this means you cannot restore the dump in a single transaction (eg, you cannot use pg_restore's `-1` switch).
4. Examine the contents of the restored tsearch2 configuration tables (`pg_ts_cfg` and so on), and create equivalent built-in text search configurations as needed. You may drop the old configuration tables once you've extracted all the useful information from them.
5. Test your application.

At a later time you may wish to rename application references to the alias text search objects, so that you can eventually uninstall the replacement `tsearch2` module.

F.39.3. References

Tsearch2 Development Site http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/

F.40. tsm_system_rows

The tsm_system_rows module provides the table sampling method SYSTEM_ROWS, which can be used in the TABLESAMPLE clause of a SELECT command.

This table sampling method accepts a single integer argument that is the maximum number of rows to read. The resulting sample will always contain exactly that many rows, unless the table does not contain enough rows, in which case the whole table is selected.

Like the built-in SYSTEM sampling method, SYSTEM_ROWS performs block-level sampling, so that the sample is not completely random but may be subject to clustering effects, especially if only a small number of rows are requested.

SYSTEM_ROWS does not support the REPEATABLE clause.

F.40.1. Examples

Here is an example of selecting a sample of a table with SYSTEM_ROWS. First install the extension:

```
CREATE EXTENSION tsm_system_rows;
```

Then you can use it in a SELECT command, for instance:

```
SELECT * FROM my_table TABLESAMPLE SYSTEM_ROWS(100);
```

This command will return a sample of 100 rows from the table my_table (unless the table does not have 100 visible rows, in which case all its rows are returned).

F.41. tsm_system_time

The tsm_system_time module provides the table sampling method SYSTEM_TIME, which can be used in the TABLESAMPLE clause of a SELECT command.

This table sampling method accepts a single floating-point argument that is the maximum number of milliseconds to spend reading the table. This gives you direct control over how long the query takes, at the price that the size of the sample becomes hard to predict. The resulting sample will contain as many rows as could be read in the specified time, unless the whole table has been read first.

Like the built-in SYSTEM sampling method, SYSTEM_TIME performs block-level sampling, so that the sample is not completely random but may be subject to clustering effects, especially if only a small number of rows are selected.

SYSTEM_TIME does not support the REPEATABLE clause.

F.41.1. Examples

Here is an example of selecting a sample of a table with SYSTEM_TIME. First install the extension:

```
CREATE EXTENSION tsm_system_time;
```

Then you can use it in a `SELECT` command, for instance:

```
SELECT * FROM my_table TABLESAMPLE SYSTEM_TIME(1000);
```

This command will return as large a sample of `my_table` as it can read in 1 second (1000 milliseconds). Of course, if the whole table can be read in under 1 second, all its rows will be returned.

F.42. unaccent

`unaccent` is a text search dictionary that removes accents (diacritic signs) from lexemes. It's a filtering dictionary, which means its output is always passed to the next dictionary (if any), unlike the normal behavior of dictionaries. This allows accent-insensitive processing for full text search.

The current implementation of `unaccent` cannot be used as a normalizing dictionary for the `thesaurus` dictionary.

F.42.1. Configuration

An `unaccent` dictionary accepts the following options:

- `RULES` is the base name of the file containing the list of translation rules. This file must be stored in `$SHAREDIR/tsearch_data/` (where `$SHAREDIR` means the PostgreSQL installation's shared-data directory). Its name must end in `.rules` (which is not to be included in the `RULES` parameter).

The rules file has the following format:

- Each line represents one translation rule, consisting of a character with accent followed by a character without accent. The first is translated into the second. For example,

  ```
  À       A
  Á       A
  Â       A
  Ã       A
  Ä       A
  Å       A
  Æ       A
  ```

  The two characters must be separated by whitespace, and any leading or trailing whitespace on a line is ignored.
- Alternatively, if only one character is given on a line, instances of that character are deleted; this is useful in languages where accents are represented by separate characters.
- Actually, each "character" can be any string not containing whitespace, so `unaccent` dictionaries could be used for other sorts of substring substitutions besides diacritic removal.

- As with other PostgreSQL text search configuration files, the rules file must be stored in UTF-8 encoding. The data is automatically translated into the current database's encoding when loaded. Any lines containing untranslatable characters are silently ignored, so that rules files can contain rules that are not applicable in the current encoding.

A more complete example, which is directly useful for most European languages, can be found in `unaccent.rules`, which is installed in `$SHAREDIR/tsearch_data/` when the `unaccent` module is installed.

F.42.2. Usage

Installing the `unaccent` extension creates a text search template `unaccent` and a dictionary `unaccent` based on it. The `unaccent` dictionary has the default parameter setting `RULES='unaccent'`, which makes it immediately usable with the standard `unaccent.rules` file. If you wish, you can alter the parameter, for example

```
mydb=# ALTER TEXT SEARCH DICTIONARY unaccent (RULES='my_rules');
```

or create new dictionaries based on the template.

To test the dictionary, you can try:

```
mydb=# select ts_lexize('unaccent','Hôtel');
 ts_lexize
-----------
 {Hotel}
(1 row)
```

Here is an example showing how to insert the `unaccent` dictionary into a text search configuration:

```
mydb=# CREATE TEXT SEARCH CONFIGURATION fr ( COPY = french );
mydb=# ALTER TEXT SEARCH CONFIGURATION fr
        ALTER MAPPING FOR hword, hword_part, word
        WITH unaccent, french_stem;
mydb=# select to_tsvector('fr','Hôtels de la Mer');
    to_tsvector
-------------------
 'hotel':1 'mer':4
(1 row)

mydb=# select to_tsvector('fr','Hôtel de la Mer') @@ to_tsquery('fr','Hotels');
 ?column?
----------
 t
(1 row)

mydb=# select ts_headline('fr','Hôtel de la Mer',to_tsquery('fr','Hotels'));
      ts_headline
------------------------
 <b>Hôtel</b> de la Mer
```

```
(1 row)
```

F.42.3. Functions

The `unaccent()` function removes accents (diacritic signs) from a given string. Basically, it's a wrapper around `unaccent`-type dictionaries, but it can be used outside normal text search contexts.

```
unaccent([dictionary regdictionary, ] string text) returns text
```

If the *`dictionary`* argument is omitted, the text search dictionary named `unaccent` and appearing in the same schema as the `unaccent()` function itself is used.

For example:

```
SELECT unaccent('unaccent', 'Hôtel');
SELECT unaccent('Hôtel');
```

F.43. uuid-ossp

The `uuid-ossp` module provides functions to generate universally unique identifiers (UUIDs) using one of several standard algorithms. There are also functions to produce certain special UUID constants.

F.43.1. `uuid-ossp` Functions

Table F-33 shows the functions available to generate UUIDs. The relevant standards ITU-T Rec. X.667, ISO/IEC 9834-8:2005, and RFC 4122 specify four algorithms for generating UUIDs, identified by the version numbers 1, 3, 4, and 5. (There is no version 2 algorithm.) Each of these algorithms could be suitable for a different set of applications.

Table F-33. Functions for UUID Generation

Function	Description
`uuid_generate_v1()`	This function generates a version 1 UUID. This involves the MAC address of the computer and a time stamp. Note that UUIDs of this kind reveal the identity of the computer that created the identifier and the time at which it did so, which might make it unsuitable for certain security-sensitive applications.

Function	Description
`uuid_generate_v1mc()`	This function generates a version 1 UUID but uses a random multicast MAC address instead of the real MAC address of the computer.
`uuid_generate_v3(namespace uuid, name text)`	This function generates a version 3 UUID in the given namespace using the specified input name. The namespace should be one of the special constants produced by the `uuid_ns_*()` functions shown in Table F-34. (It could be any UUID in theory.) The name is an identifier in the selected namespace. For example: `SELECT uuid_generate_v3(uuid_ns_url(), 'http://www.` The name parameter will be MD5-hashed, so the cleartext cannot be derived from the generated UUID. The generation of UUIDs by this method has no random or environment-dependent element and is therefore reproducible.
`uuid_generate_v4()`	This function generates a version 4 UUID, which is derived entirely from random numbers.
`uuid_generate_v5(namespace uuid, name text)`	This function generates a version 5 UUID, which works like a version 3 UUID except that SHA-1 is used as a hashing method. Version 5 should be preferred over version 3 because SHA-1 is thought to be more secure than MD5.

Table F-34. Functions Returning UUID Constants

`uuid_nil()`	A “nil” UUID constant, which does not occur as a real UUID.
`uuid_ns_dns()`	Constant designating the DNS namespace for UUIDs.
`uuid_ns_url()`	Constant designating the URL namespace for UUIDs.
`uuid_ns_oid()`	Constant designating the ISO object identifier (OID) namespace for UUIDs. (This pertains to ASN.1 OIDs, which are unrelated to the OIDs used in PostgreSQL.)
`uuid_ns_x500()`	Constant designating the X.500 distinguished name (DN) namespace for UUIDs.

F.43.2. Building `uuid-ossp`

Historically this module depended on the OSSP UUID library, which accounts for the module's name. While the OSSP UUID library can still be found at http://www.ossp.org/pkg/lib/uuid/, it is not well maintained, and is becoming increasingly difficult to port to newer platforms. `uuid-ossp` can now be built without the OSSP library on some platforms. On FreeBSD, NetBSD, and some other BSD-derived platforms, suitable UUID creation functions are included in the core `libc` library. On Linux, OS X, and some other platforms, suitable functions are provided in the `libuuid` library, which originally came from the `e2fsprogs` project (though on modern Linux it is considered part of `util-linux-ng`). When invoking `configure`, specify `--with-uuid=bsd` to use the BSD functions, or `--with-uuid=e2fs` to use `e2fsprogs`' `libuuid`, or `--with-uuid=ossp` to use the OSSP UUID library. More than one of these libraries might be available on a particular machine, so `configure` does not automatically choose one.

> **Note:** If you only need randomly-generated (version 4) UUIDs, consider using the `gen_random_uuid()` function from the pgcrypto module instead.

F.43.3. Author

Peter Eisentraut <`peter_e@gmx.net`>

F.44. xml2

The `xml2` module provides XPath querying and XSLT functionality.

F.44.1. Deprecation Notice

From PostgreSQL 8.3 on, there is XML-related functionality based on the SQL/XML standard in the core server. That functionality covers XML syntax checking and XPath queries, which is what this module does, and more, but the API is not at all compatible. It is planned that this module will be removed in a future version of PostgreSQL in favor of the newer standard API, so you are encouraged to try converting your applications. If you find that some of the functionality of this module is not available in an adequate form with the newer API, please explain your issue to <`pgsql-hackers@lists.postgresql.org`> so that the deficiency can be addressed.

F.44.2. Description of Functions

Table F-35 shows the functions provided by this module. These functions provide straightforward XML parsing and XPath queries. All arguments are of type `text`, so for brevity that is not shown.

Table F-35. Functions

Function	Returns	Description
`xml_is_well_formed(document)`	`bool`	This parses the document text in its parameter and returns true if the document is well-formed XML. (Note: before PostgreSQL 8.2, this function was called `xml_valid()`. That is the wrong name since validity and well-formedness have different meanings in XML. The old name is still available, but is deprecated.)
`xpath_string(document, query)`	`text`	These functions evaluate the XPath query on the supplied
`xpath_number(document, query)`	`float4`	document, and cast the result to the specified type.
`xpath_bool(document, query)`	`bool`	
`xpath_nodeset(document, query, toptag, itemtag)`	`text`	This evaluates query on document and wraps the result in XML tags. If the result is multivalued, the output will look like: `<toptag> <itemtag>Value 1 which could be an XMI <itemtag>Value 2....</itemtag> </toptag>` If either `toptag` or `itemtag` is an empty string, the relevant tag is omitted.
`xpath_nodeset(document, query)`	`text`	Like `xpath_nodeset(document, query, toptag, itemtag)` but result omits both tags.
`xpath_nodeset(document, query, itemtag)`	`text`	Like `xpath_nodeset(document, query, toptag, itemtag)` but result omits `toptag`.
`xpath_list(document, query, separator)`	`text`	This function returns multiple values separated by the specified separator, for example `Value 1,Value 2,Value 3` if separator is `,`.

Function	Returns	Description
xpath_list(document, query)	text	This is a wrapper for the above function that uses , as the separator.

F.44.3. xpath_table

```
xpath_table(text key, text document, text relation, text xpaths, text criteria) returns set
```

xpath_table is a table function that evaluates a set of XPath queries on each of a set of documents and returns the results as a table. The primary key field from the original document table is returned as the first column of the result so that the result set can readily be used in joins. The parameters are described in Table F-36.

Table F-36. xpath_table Parameters

Parameter	Description
key	the name of the "key" field — this is just a field to be used as the first column of the output table, i.e., it identifies the record from which each output row came (see note below about multiple values)
document	the name of the field containing the XML document
relation	the name of the table or view containing the documents
xpaths	one or more XPath expressions, separated by |
criteria	the contents of the WHERE clause. This cannot be omitted, so use true or 1=1 if you want to process all the rows in the relation

These parameters (except the XPath strings) are just substituted into a plain SQL SELECT statement, so you have some flexibility — the statement is

```
SELECT <key>, <document> FROM <relation> WHERE <criteria>
```

so those parameters can be *anything* valid in those particular locations. The result from this SELECT needs to return exactly two columns (which it will unless you try to list multiple fields for key or document). Beware that this simplistic approach requires that you validate any user-supplied values to avoid SQL injection attacks.

The function has to be used in a FROM expression, with an AS clause to specify the output columns; for example

```
SELECT * FROM
xpath_table('article_id',
            'article_xml',
            'articles',
            '/article/author|/article/pages|/article/title',
```

```
            'date_entered > "2003-01-01" ')
AS t(article_id integer, author text, page_count integer, title text);
```

The `AS` clause defines the names and types of the columns in the output table. The first is the "key" field and the rest correspond to the XPath queries. If there are more XPath queries than result columns, the extra queries will be ignored. If there are more result columns than XPath queries, the extra columns will be NULL.

Notice that this example defines the `page_count` result column as an integer. The function deals internally with string representations, so when you say you want an integer in the output, it will take the string representation of the XPath result and use PostgreSQL input functions to transform it into an integer (or whatever type the `AS` clause requests). An error will result if it can't do this — for example if the result is empty — so you may wish to just stick to `text` as the column type if you think your data has any problems.

The calling `SELECT` statement doesn't necessarily have be just `SELECT *` — it can reference the output columns by name or join them to other tables. The function produces a virtual table with which you can perform any operation you wish (e.g., aggregation, joining, sorting etc). So we could also have:

```
SELECT t.title, p.fullname, p.email
FROM xpath_table('article_id', 'article_xml', 'articles',
                 '/article/title|/article/author/@id',
                 'xpath_string(article_xml,"/article/@date") > "2003-03-20" ')
       AS t(article_id integer, title text, author_id integer),
     tblPeopleInfo AS p
WHERE t.author_id = p.person_id;
```

as a more complicated example. Of course, you could wrap all of this in a view for convenience.

F.44.3.1. Multivalued Results

The `xpath_table` function assumes that the results of each XPath query might be multivalued, so the number of rows returned by the function may not be the same as the number of input documents. The first row returned contains the first result from each query, the second row the second result from each query. If one of the queries has fewer values than the others, null values will be returned instead.

In some cases, a user will know that a given XPath query will return only a single result (perhaps a unique document identifier) — if used alongside an XPath query returning multiple results, the single-valued result will appear only on the first row of the result. The solution to this is to use the key field as part of a join against a simpler XPath query. As an example:

```
CREATE TABLE test (
    id int PRIMARY KEY,
    xml text
);

INSERT INTO test VALUES (1, '<doc num="C1">
<line num="L1"><a>1</a><b>2</b><c>3</c></line>
<line num="L2"><a>11</a><b>22</b><c>33</c></line>
</doc>');

INSERT INTO test VALUES (2, '<doc num="C2">
```

```
<line num="L1"><a>111</a><b>222</b><c>333</c></line>
<line num="L2"><a>111</a><b>222</b><c>333</c></line>
</doc>');

SELECT * FROM
  xpath_table('id','xml','test',
              '/doc/@num|/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
              'true')
  AS t(id int, doc_num varchar(10), line_num varchar(10), val1 int, val2 int, val3 int)
WHERE id = 1 ORDER BY doc_num, line_num

 id | doc_num | line_num | val1 | val2 | val3
----+---------+----------+------+------+------
  1 | C1      | L1       |    1 |    2 |    3
  1 |         | L2       |   11 |   22 |   33
```

To get `doc_num` on every line, the solution is to use two invocations of `xpath_table` and join the results:

```
SELECT t.*,i.doc_num FROM
  xpath_table('id', 'xml', 'test',
              '/doc/line/@num|/doc/line/a|/doc/line/b|/doc/line/c',
              'true')
    AS t(id int, line_num varchar(10), val1 int, val2 int, val3 int),
  xpath_table('id', 'xml', 'test', '/doc/@num', 'true')
    AS i(id int, doc_num varchar(10))
WHERE i.id=t.id AND i.id=1
ORDER BY doc_num, line_num;

 id | line_num | val1 | val2 | val3 | doc_num
----+----------+------+------+------+---------
  1 | L1       |    1 |    2 |    3 | C1
  1 | L2       |   11 |   22 |   33 | C1
(2 rows)
```

F.44.4. XSLT Functions

The following functions are available if libxslt is installed:

F.44.4.1. `xslt_process`

```
xslt_process(text document, text stylesheet, text paramlist) returns text
```

This function applies the XSL stylesheet to the document and returns the transformed result. The `paramlist` is a list of parameter assignments to be used in the transformation, specified in the form `a=1,b=2`. Note that the parameter parsing is very simple-minded: parameter values cannot contain commas!

There is also a two-parameter version of `xslt_process` which does not pass any parameters to the transformation.

F.44.5. Author

John Gray <`jgray@azuli.co.uk`>

Development of this module was sponsored by Torchbox Ltd. (www.torchbox.com). It has the same BSD license as PostgreSQL.

Appendix G. Additional Supplied Programs

This appendix and the previous one contain information regarding the modules that can be found in the `contrib` directory of the PostgreSQL distribution. See Appendix F for more information about the `contrib` section in general and server extensions and plug-ins found in `contrib` specifically.

This appendix covers utility programs found in `contrib`. Once installed, either from source or a packaging system, they are found in the `bin` directory of the PostgreSQL installation and can be used like any other program.

G.1. Client Applications

This section covers PostgreSQL client applications in `contrib`. They can be run from anywhere, independent of where the database server resides. See also Reference II, *PostgreSQL Client Applications* for information about client applications that part of the core PostgreSQL distribution.

oid2name

Name

`oid2name` — resolve OIDs and file nodes in a PostgreSQL data directory

Synopsis

`oid2name` [*option*...]

Description

oid2name is a utility program that helps administrators to examine the file structure used by PostgreSQL. To make use of it, you need to be familiar with the database file structure, which is described in Chapter 63.

> **Note:** The name "oid2name" is historical, and is actually rather misleading, since most of the time when you use it, you will really be concerned with tables' filenode numbers (which are the file names visible in the database directories). Be sure you understand the difference between table OIDs and table filenodes!

oid2name connects to a target database and extracts OID, filenode, and/or table name information. You can also have it show database OIDs or tablespace OIDs.

Options

oid2name accepts the following command-line arguments:

`-f` *`filenode`*

show info for table with filenode *`filenode`*

`-i`

include indexes and sequences in the listing

`-o` *`oid`*

show info for table with OID *`oid`*

`-q`

omit headers (useful for scripting)

`-s`

show tablespace OIDs

`-S`

include system objects (those in `information_schema`, `pg_toast` and `pg_catalog` schemas)

`-t` *`tablename_pattern`*

show info for table(s) matching *`tablename_pattern`*

`-V`
`--version`

Print the oid2name version and exit.

`-x`

display more information about each object shown: tablespace name, schema name, and OID

`-?`
`--help`

Show help about oid2name command line arguments, and exit.

oid2name also accepts the following command-line arguments for connection parameters:

`-d` *`database`*

database to connect to

`-H` *`host`*

database server's host

`-p` *port*

database server's port

`-U` *username*

user name to connect as

`-P` *password*

password (deprecated — putting this on the command line is a security hazard)

To display specific tables, select which tables to show by using `-o`, `-f` and/or `-t`. `-o` takes an OID, `-f` takes a filenode, and `-t` takes a table name (actually, it's a `LIKE` pattern, so you can use things like `foo%`). You can use as many of these options as you like, and the listing will include all objects matched by any of the options. But note that these options can only show objects in the database given by `-d`.

If you don't give any of `-o`, `-f` or `-t`, but do give `-d`, it will list all tables in the database named by `-d`. In this mode, the `-S` and `-i` options control what gets listed.

If you don't give `-d` either, it will show a listing of database OIDs. Alternatively you can give `-s` to get a tablespace listing.

Notes

oid2name requires a running database server with non-corrupt system catalogs. It is therefore of only limited use for recovering from catastrophic database corruption situations.

Examples

```
$ # what's in this database server, anyway?
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default

$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
-------------------------
    1663       pg_default
    1664        pg_global
  155151         fastdisk
  155152          bigdisk

$ # OK, let's look into database alvherre
$ cd $PGDATA/base/17228
```

```
$ # get top 10 db objects in the default tablespace, ordered by size
$ ls -lS * | head -10
-rw-------  1 alvherre alvherre 136536064 sep 14 09:51 155173
-rw-------  1 alvherre alvherre  17965056 sep 14 09:51 1155291
-rw-------  1 alvherre alvherre   1204224 sep 14 09:51 16717
-rw-------  1 alvherre alvherre    581632 sep  6 17:51 1255
-rw-------  1 alvherre alvherre    237568 sep 14 09:50 16674
-rw-------  1 alvherre alvherre    212992 sep 14 09:51 1249
-rw-------  1 alvherre alvherre    204800 sep 14 09:51 16684
-rw-------  1 alvherre alvherre    196608 sep 14 09:50 16700
-rw-------  1 alvherre alvherre    163840 sep 14 09:50 16699
-rw-------  1 alvherre alvherre    122880 sep  6 17:51 16751

$ # I wonder what file 155173 is ...
$ oid2name -d alvherre -f 155173
From database "alvherre":
  Filenode  Table Name
----------------------
    155173    accounts

$ # you can ask for more than one object
$ oid2name -d alvherre -f 155173 -f 1155291
From database "alvherre":
  Filenode     Table Name
-------------------------
    155173       accounts
   1155291  accounts_pkey

$ # you can mix the options, and get more details with -x
$ oid2name -d alvherre -t accounts -f 1155291 -x
From database "alvherre":
  Filenode     Table Name      Oid  Schema  Tablespace
-----------------------------------------------------
    155173       accounts   155173  public  pg_default
   1155291  accounts_pkey  1155291  public  pg_default

$ # show disk space for every db object
$ du [0-9]* |
> while read SIZE FILENODE
> do
>   echo "$SIZE       `oid2name -q -d alvherre -i -f $FILENODE`"
> done
16            1155287  branches_pkey
16            1155289  tellers_pkey
17561            1155291  accounts_pkey
...

$ # same, but sort by size
$ du [0-9]* | sort -rn | while read SIZE FN
> do
>   echo "$SIZE   `oid2name -q -d alvherre -f $FN`"
> done
```

```
133466             155173    accounts
17561             1155291  accounts_pkey
1177               16717  pg_proc_proname_args_nsp_index
...

$ # If you want to see what's in tablespaces, use the pg_tblspc directory
$ cd $PGDATA/pg_tblspc
$ oid2name -s
All tablespaces:
     Oid  Tablespace Name
------------------------
    1663       pg_default
    1664        pg_global
  155151         fastdisk
  155152          bigdisk

$ # what databases have objects in tablespace "fastdisk"?
$ ls -d 155151/*
155151/17228/  155151/PG_VERSION

$ # Oh, what was database 17228 again?
$ oid2name
All databases:
    Oid  Database Name  Tablespace
----------------------------------
  17228       alvherre  pg_default
  17255     regression  pg_default
  17227      template0  pg_default
      1      template1  pg_default

$ # Let's see what objects does this database have in the tablespace.
$ cd 155151/17228
$ ls -l
total 0
-rw-------  1 postgres postgres 0 sep 13 23:20 155156

$ # OK, this is a pretty small table ... but which one is it?
$ oid2name -d alvherre -f 155156
From database "alvherre":
  Filenode  Table Name
----------------------
    155156         foo
```

Author

B. Palmer <bpalmer@crimelabs.net>

vacuumlo

Name

vacuumlo — remove orphaned large objects from a PostgreSQL database

Synopsis

vacuumlo [*option*...] *dbname*...

Description

vacuumlo is a simple utility program that will remove any "orphaned" large objects from a PostgreSQL database. An orphaned large object (LO) is considered to be any LO whose OID does not appear in any oid or lo data column of the database.

If you use this, you may also be interested in the lo_manage trigger in the lo module. lo_manage is useful to try to avoid creating orphaned LOs in the first place.

All databases named on the command line are processed.

Options

vacuumlo accepts the following command-line arguments:

-l *limit*

Remove no more than *limit* large objects per transaction (default 1000). Since the server acquires a lock per LO removed, removing too many LOs in one transaction risks exceeding max_locks_per_transaction. Set the limit to zero if you want all removals done in a single transaction.

-n

Don't remove anything, just show what would be done.

-v

Write a lot of progress messages.

-V
--version

Print the vacuumlo version and exit.

-?
--help

Show help about vacuumlo command line arguments, and exit.

vacuumlo also accepts the following command-line arguments for connection parameters:

`-h` *`hostname`*

Database server's host.

`-p` *`port`*

Database server's port.

`-U` *`username`*

User name to connect as.

`-w`
`--no-password`

Never issue a password prompt. If the server requires password authentication and a password is not available by other means such as a `.pgpass` file, the connection attempt will fail. This option can be useful in batch jobs and scripts where no user is present to enter a password.

`-W`

Force vacuumlo to prompt for a password before connecting to a database.

This option is never essential, since vacuumlo will automatically prompt for a password if the server demands password authentication. However, vacuumlo will waste a connection attempt finding out that the server wants a password. In some cases it is worth typing `-W` to avoid the extra connection attempt.

Notes

vacuumlo works by the following method: First, vacuumlo builds a temporary table which contains all of the OIDs of the large objects in the selected database. It then scans through all columns in the database that are of type `oid` or `lo`, and removes matching entries from the temporary table. (Note: Only types with these names are considered; in particular, domains over them are not considered.) The remaining entries in the temporary table identify orphaned LOs. These are removed.

Author

Peter Mount <`peter@retep.org.uk`>

G.2. Server Applications

This section covers PostgreSQL server-related applications in `contrib`. They are typically run on the host where the database server resides. See also Reference III, *PostgreSQL Server Applications* for information about server applications that part of the core PostgreSQL distribution.

pg_standby

Name

`pg_standby` — supports the creation of a PostgreSQL warm standby server

Synopsis

`pg_standby` [*option*...] *archivelocation nextwalfile xlogfilepath* [*restartwalfile*]

Description

pg_standby supports creation of a "warm standby" database server. It is designed to be a production-ready program, as well as a customizable template should you require specific modifications.

pg_standby is designed to be a waiting `restore_command`, which is needed to turn a standard archive recovery into a warm standby operation. Other configuration is required as well, all of which is described in the main server manual (see Section 25.2).

To configure a standby server to use pg_standby, put this into its `recovery.conf` configuration file:

```
restore_command = 'pg_standby archiveDir %f %p %r'
```

where *archiveDir* is the directory from which WAL segment files should be restored.

If *restartwalfile* is specified, normally by using the `%r` macro, then all WAL files logically preceding this file will be removed from *archivelocation*. This minimizes the number of files that need to be retained, while preserving crash-restart capability. Use of this parameter is appropriate if the *archivelocation* is a transient staging area for this particular standby server, but *not* when the *archivelocation* is intended as a long-term WAL archive area.

pg_standby assumes that *archivelocation* is a directory readable by the server-owning user. If *restartwalfile* (or `-k`) is specified, the *archivelocation* directory must be writable too.

There are two ways to fail over to a "warm standby" database server when the master server fails:

Smart Failover

> In smart failover, the server is brought up after applying all WAL files available in the archive. This results in zero data loss, even if the standby server has fallen behind, but if there is a lot of unapplied WAL it can be a long time before the standby server becomes ready. To trigger a smart failover, create a trigger file containing the word `smart`, or just create it and leave it empty.

Fast Failover

In fast failover, the server is brought up immediately. Any WAL files in the archive that have not yet been applied will be ignored, and all transactions in those files are lost. To trigger a fast failover, create a trigger file and write the word `fast` into it. pg_standby can also be configured to execute a fast failover automatically if no new WAL file appears within a defined interval.

Options

pg_standby accepts the following command-line arguments:

`-c`

Use `cp` or `copy` command to restore WAL files from archive. This is the only supported behavior so this option is useless.

`-d`

Print lots of debug logging output on `stderr`.

`-k`

Remove files from *`archivelocation`* so that no more than this many WAL files before the current one are kept in the archive. Zero (the default) means not to remove any files from *`archivelocation`*. This parameter will be silently ignored if *`restartwalfile`* is specified, since that specification method is more accurate in determining the correct archive cut-off point. Use of this parameter is *deprecated* as of PostgreSQL 8.3; it is safer and more efficient to specify a *`restartwalfile`* parameter. A too small setting could result in removal of files that are still needed for a restart of the standby server, while a too large setting wastes archive space.

`-r` *`maxretries`*

Set the maximum number of times to retry the copy command if it fails (default 3). After each failure, we wait for *`sleeptime`* * *`num_retries`* so that the wait time increases progressively. So by default, we will wait 5 secs, 10 secs, then 15 secs before reporting the failure back to the standby server. This will be interpreted as end of recovery and the standby will come up fully as a result.

`-s` *`sleeptime`*

Set the number of seconds (up to 60, default 5) to sleep between tests to see if the WAL file to be restored is available in the archive yet. The default setting is not necessarily recommended; consult Section 25.2 for discussion.

`-t` *`triggerfile`*

Specify a trigger file whose presence should cause failover. It is recommended that you use a structured file name to avoid confusion as to which server is being triggered when multiple servers exist on the same system; for example `/tmp/pgsql.trigger.5432`.

`-V`
`--version`

Print the pg_standby version and exit.

`-w` *`maxwaittime`*

Set the maximum number of seconds to wait for the next WAL file, after which a fast failover will be performed. A setting of zero (the default) means wait forever. The default setting is not necessarily recommended; consult Section 25.2 for discussion.

`-?`
`--help`

Show help about pg_standby command line arguments, and exit.

Notes

pg_standby is designed to work with PostgreSQL 8.2 and later.

PostgreSQL 8.3 provides the `%r` macro, which is designed to let pg_standby know the last file it needs to keep. With PostgreSQL 8.2, the `-k` option must be used if archive cleanup is required. This option remains available in 8.3, but its use is deprecated.

PostgreSQL 8.4 provides the `recovery_end_command` option. Without this option a leftover trigger file can be hazardous.

pg_standby is written in C and has an easy-to-modify source code, with specifically designated sections to modify for your own needs

Examples

On Linux or Unix systems, you might use:

```
archive_command = 'cp %p .../archive/%f'

restore_command = 'pg_standby -d -s 2 -t /tmp/pgsql.trigger.5442 .../archive %f %p %r 2>>st

recovery_end_command = 'rm -f /tmp/pgsql.trigger.5442'
```

where the archive directory is physically located on the standby server, so that the `archive_command` is accessing it across NFS, but the files are local to the standby (enabling use of `ln`). This will:

- produce debugging output in `standby.log`
- sleep for 2 seconds between checks for next WAL file availability
- stop waiting only when a trigger file called `/tmp/pgsql.trigger.5442` appears, and perform failover according to its content
- remove the trigger file when recovery ends
- remove no-longer-needed files from the archive directory

On Windows, you might use:

```
archive_command = 'copy %p ...\\archive\\%f'

restore_command = 'pg_standby -d -s 5 -t C:\pgsql.trigger.5442 ...\archive %f %p %r 2>>stan

recovery_end_command = 'del C:\pgsql.trigger.5442'
```

Note that backslashes need to be doubled in the `archive_command`, but *not* in the `restore_command` or `recovery_end_command`. This will:

- use the `copy` command to restore WAL files from archive
- produce debugging output in `standby.log`
- sleep for 5 seconds between checks for next WAL file availability
- stop waiting only when a trigger file called `C:\pgsql.trigger.5442` appears, and perform failover according to its content
- remove the trigger file when recovery ends
- remove no-longer-needed files from the archive directory

The `copy` command on Windows sets the final file size before the file is completely copied, which would ordinarily confuse pg_standby. Therefore pg_standby waits `sleeptime` seconds once it sees the proper file size. GNUWin32's `cp` sets the file size only after the file copy is complete.

Since the Windows example uses `copy` at both ends, either or both servers might be accessing the archive directory across the network.

Author

Simon Riggs <`simon@2ndquadrant.com`>

See Also

pg_archivecleanup

Appendix H. External Projects

PostgreSQL is a complex software project, and managing the project is difficult. We have found that many enhancements to PostgreSQL can be more efficiently developed separately from the core project.

H.1. Client Interfaces

There are only two client interfaces included in the base PostgreSQL distribution:

- libpq is included because it is the primary C language interface, and because many other client interfaces are built on top of it.
- ECPG is included because it depends on the server-side SQL grammar, and is therefore sensitive to changes in PostgreSQL itself.

All other language interfaces are external projects and are distributed separately. Table H-1 includes a list of some of these projects. Note that some of these packages might not be released under the same license as PostgreSQL. For more information on each language interface, including licensing terms, refer to its website and documentation.

Table H-1. Externally Maintained Client Interfaces

Name	Language	Comments	Website
DBD::Pg	Perl	Perl DBI driver	https://metacpan.org/release/DBD-Pg/
JDBC	Java	Type 4 JDBC driver	https://jdbc.postgresql.org/
libpqxx	C++	C++ interface	https://pqxx.org/
node-postgres	JavaScript	Node.js driver	https://node-postgres.com/
Npgsql	.NET	.NET data provider	https://www.npgsql.org/
pgtcl	Tcl		https://github.com/flightaware/Pgtcl
pgtclng	Tcl		http://sourceforge.net/projects/pgtclng/
pq	Go	Pure Go driver for Go's database/sql	https://github.com/lib/pq
psqlODBC	ODBC	ODBC driver	https://odbc.postgresql.org/
psycopg	Python	DB API 2.0-compliant	https://www.psycopg.org/

H.2. Administration Tools

There are several administration tools available for PostgreSQL. The most popular is pgAdmin[1], and there are several commercially available ones as well.

H.3. Procedural Languages

PostgreSQL includes several procedural languages with the base distribution: PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python.

In addition, there are a number of procedural languages that are developed and maintained outside the core PostgreSQL distribution. Table H-2 lists some of these packages. Note that some of these projects might not be released under the same license as PostgreSQL. For more information on each procedural language, including licensing information, refer to its website and documentation.

Table H-2. Externally Maintained Procedural Languages

Name	Language	Website
PL/Java	Java	https://github.com/tada/pljava
PL/PHP	PHP	https://public.commandprompt.com/projects/plphp
PL/Py	Python	http://python.projects.postgresql.org/backend/
PL/R	R	https://github.com/postgres-plr/plr
PL/Ruby	Ruby	http://raa.ruby-lang.org/project/pl-ruby/
PL/Scheme	Scheme	http://plscheme.projects.postgresql.org/
PL/sh	Unix shell	https://github.com/petere/plsh

H.4. Extensions

PostgreSQL is designed to be easily extensible. For this reason, extensions loaded into the database can function just like features that are built in. The `contrib/` directory shipped with the source code contains several extensions, which are described in Appendix F. Other extensions are developed independently, like PostGIS[2]. Even PostgreSQL replication solutions can be developed externally. For example, Slony-I[3] is a popular master/standby replication solution that is developed independently from the core project.

1. http://www.pgadmin.org/
2. https://postgis.net/
3. https://www.slony.info

Appendix I. The Source Code Repository

The PostgreSQL source code is stored and managed using the Git version control system. A public mirror of the master repository is available; it is updated within a minute of any change to the master repository.

Our wiki, https://wiki.postgresql.org/wiki/Working_with_Git, has some discussion on working with Git.

Note that building PostgreSQL from the source repository requires reasonably up-to-date versions of bison, flex, and Perl. These tools are not needed to build from a distribution tarball, because the files that these tools are used to build are included in the tarball. Other tool requirements are the same as shown in Section 15.2.

I.1. Getting The Source via Git

With Git you will make a copy of the entire code repository on your local machine, so you will have access to all history and branches offline. This is the fastest and most flexible way to develop or test patches.

Git

1. You will need an installed version of Git, which you can get from https://git-scm.com. Many systems already have a recent version of Git installed by default, or available in their package distribution system.
2. To begin using the Git repository, make a clone of the official mirror:

   ```
   git clone https://git.postgresql.org/git/postgresql.git
   ```

   This will copy the full repository to your local machine, so it may take a while to complete, especially if you have a slow Internet connection. The files will be placed in a new subdirectory `postgresql` of your current directory.

   The Git mirror can also be reached via the Git protocol. Just change the URL prefix to `git`, as in:

   ```
   git clone git://git.postgresql.org/git/postgresql.git
   ```
3. Whenever you want to get the latest updates in the system, `cd` into the repository, and run:

   ```
   git fetch
   ```

Git can do a lot more things than just fetch the source. For more information, consult the Git man pages, or see the website at https://git-scm.com.

Appendix J. Documentation

PostgreSQL has four primary documentation formats:

- Plain text, for pre-installation information
- HTML, for on-line browsing and reference
- PDF or PostScript, for printing
- man pages, for quick reference.

Additionally, a number of plain-text `README` files can be found throughout the PostgreSQL source tree, documenting various implementation issues.

HTML documentation and man pages are part of a standard distribution and are installed by default. PDF and PostScript format documentation is available separately for download.

J.1. DocBook

The documentation sources are written in *DocBook*, which is a markup language superficially similar to HTML. Both of these languages are applications of the *Standard Generalized Markup Language*, SGML, which is essentially a language for describing other languages. In what follows, the terms DocBook and SGML are both used, but technically they are not interchangeable.

DocBook allows an author to specify the structure and content of a technical document without worrying about presentation details. A document style defines how that content is rendered into one of several final forms. DocBook is maintained by the OASIS group[1]. The official DocBook site[2] has good introductory and reference documentation and a complete O'Reilly book for your online reading pleasure. The NewbieDoc Docbook Guide[3] is very helpful for beginners. The FreeBSD Documentation Project[4] also uses DocBook and has some good information, including a number of style guidelines that might be worth considering.

J.2. Tool Sets

The following tools are used to process the documentation. Some might be optional, as noted.

DocBook DTD[5]

This is the definition of DocBook itself. We currently use version 4.2; you cannot use later or earlier

1. http://www.oasis-open.org
2. http://www.oasis-open.org/docbook/
3. http://newbiedoc.sourceforge.net/metadoc/docbook-guide.html
4. http://www.freebsd.org/docproj/docproj.html
5. http://www.oasis-open.org/docbook/

versions. You need the SGML variant of the DocBook DTD, but to build man pages you also need the XML variant of the same version.

ISO 8879 character entities[6]

These are required by DocBook but are distributed separately because they are maintained by ISO.

DocBook DSSSL Stylesheets[7]

These contain the processing instructions for converting the DocBook sources to other formats, such as HTML.

DocBook XSL Stylesheets[8]

This is another stylesheet for converting DocBook to other formats. We currently use this to produce man pages and optionally HTMLHelp. You can also use this toolchain to produce HTML or PDF output, but official PostgreSQL releases use the DSSSL stylesheets for that.

The minimum required version is currently 1.74.0.

OpenJade[9]

This is the base package of SGML processing. It contains an SGML parser, a DSSSL processor (that is, a program to convert SGML to other formats using DSSSL stylesheets), as well as a number of related tools. Jade is now being maintained by the OpenJade group, no longer by James Clark.

Libxml2[10] for `xmllint`

This library and the `xmllint` tool it contains are used for processing XML. Many developers will already have Libxml2 installed, because it is also used when building the PostgreSQL code. Note, however, that `xmllint` might need to be installed from a separate subpackage.

Libxslt[11] for `xsltproc`

This is the processing tool to use with the XSLT stylesheets (like `jade` is the processing tool for DSSSL stylesheets).

JadeTeX[12]

If you want to, you can also install JadeTeX to use TeX as a formatting backend for Jade. JadeTeX can create PostScript or PDF files (the latter with bookmarks).

However, the output from JadeTeX is inferior to what you get from the RTF backend. Particular problem areas are tables and various artifacts of vertical and horizontal spacing. Also, there is no opportunity to manually polish the results.

We have documented experience with several installation methods for the various tools that are needed to process the documentation. These will be described below. There might be some other packaged distributions for these tools. Please report package status to the documentation mailing list, and we will include that information here.

6. http://www.oasis-open.org/cover/ISOEnts.zip
7. http://wiki.docbook.org/DocBookDssslStylesheetDocs
8. http://wiki.docbook.org/DocBookXslStylesheets
9. http://openjade.sourceforge.net
10. http://xmlsoft.org/
11. http://xmlsoft.org/XSLT/
12. http://jadetex.sourceforge.net

J.2.1. Installation on Fedora, RHEL, and Derivatives

To install the required packages, use:

```
yum install docbook-dtds docbook-style-dsssl docbook-style-xsl libxslt openjade
```

J.2.2. Installation on FreeBSD

The FreeBSD Documentation Project is itself a heavy user of DocBook, so it comes as no surprise that there is a full set of "ports" of the documentation tools available on FreeBSD. The following ports need to be installed to build the documentation on FreeBSD.

- `textproc/docbook-sgml`
- `textproc/docbook-xml`
- `textproc/docbook-xsl`
- `textproc/dsssl-docbook-modular`
- `textproc/libxslt`
- `textproc/openjade`

A number of things from `/usr/ports/print` (`tex`, `jadetex`) might also be of interest.

More information about the FreeBSD documentation tools can be found in the FreeBSD Documentation Project's instructions[13].

J.2.3. Debian Packages

There is a full set of packages of the documentation tools available for Debian GNU/Linux. To install, simply use:

```
apt-get install docbook docbook-dsssl docbook-xsl libxml2-utils openjade1.3 opensp xsltproc
```

J.2.4. OS X

If you use MacPorts, the following will get you set up:

```
sudo port install docbook-dsssl docbook-sgml-4.2 docbook-xml-4.2 docbook-xsl libxslt openja
```

13. http://www.freebsd.org/doc/en_US.ISO8859-1/books/fdp-primer/tools.html

J.2.5. Manual Installation from Source

The manual installation process of the DocBook tools is somewhat complex, so if you have pre-built packages available, use them. We describe here only a standard setup, with reasonably standard installation paths, and no "fancy" features. For details, you should study the documentation of the respective package, and read SGML introductory material.

J.2.5.1. Installing OpenJade

1. The installation of OpenJade offers a GNU-style `./configure; make; make install` build process. Details can be found in the OpenJade source distribution. In a nutshell:

   ```
   ./configure --enable-default-catalog=/usr/local/share/sgml/catalog
   make
   make install
   ```

   Be sure to remember where you put the "default catalog"; you will need it below. You can also leave it off, but then you will have to set the environment variable `SGML_CATALOG_FILES` to point to the file whenever you use jade later on. (This method is also an option if OpenJade is already installed and you want to install the rest of the toolchain locally.)

   > **Note:** Some users have reported encountering a segmentation fault using OpenJade 1.4devel to build the PDFs, with a message like:
   >
   > ```
   > openjade:./stylesheet.dsl:664:2:E: flow object not accepted by port; only display flow objects
   > make: *** [postgres-A4.tex-pdf] Segmentation fault
   > ```
   >
   > Downgrading to OpenJade 1.3 should get rid of this error.

2. Additionally, you should install the files `dsssl.dtd`, `fot.dtd`, `style-sheet.dtd`, and `catalog` from the `dsssl` directory somewhere, perhaps into `/usr/local/share/sgml/dsssl`. It's probably easiest to copy the entire directory:

   ```
   cp -R dsssl /usr/local/share/sgml
   ```

3. Finally, create the file `/usr/local/share/sgml/catalog` and add this line to it:

   ```
   CATALOG "dsssl/catalog"
   ```

   (This is a relative path reference to the file installed in step 2. Be sure to adjust it if you chose your installation layout differently.)

J.2.5.2. Installing the DocBook DTD Kit

1. Obtain the DocBook V4.2 distribution[14].
2. Create the directory `/usr/local/share/sgml/docbook-4.2` and change to it. (The exact location is irrelevant, but this one is reasonable within the layout we are following here.)

   ```
   $ mkdir /usr/local/share/sgml/docbook-4.2
   $ cd /usr/local/share/sgml/docbook-4.2
   ```

14. http://www.docbook.org/sgml/4.2/docbook-4.2.zip

3. Unpack the archive:

   ```
   $ unzip -a ...../docbook-4.2.zip
   ```
   (The archive will unpack its files into the current directory.)

4. Edit the file `/usr/local/share/sgml/catalog` (or whatever you told jade during installation) and put a line like this into it:

   ```
   CATALOG "docbook-4.2/docbook.cat"
   ```

5. Download the ISO 8879 character entities archive[15], unpack it, and put the files in the same directory you put the DocBook files in:

   ```
   $ cd /usr/local/share/sgml/docbook-4.2
   $ unzip ...../ISOEnts.zip
   ```

6. Run the following command in the directory with the DocBook and ISO files:

   ```
   perl -pi -e 's/iso-(.*).gml/ISO\1/g' docbook.cat
   ```
   (This fixes a mixup between the names used in the DocBook catalog file and the actual names of the ISO character entity files.)

J.2.5.3. Installing the DocBook DSSSL Style Sheets

To install the style sheets, unzip and untar the distribution and move it to a suitable place, for example `/usr/local/share/sgml`. (The archive will automatically create a subdirectory.)

```
$ gunzip docbook-dsssl-1.xx.tar.gz
$ tar -C /usr/local/share/sgml -xf docbook-dsssl-1.xx.tar
```

The usual catalog entry in `/usr/local/share/sgml/catalog` can also be made:

```
CATALOG "docbook-dsssl-1.xx/catalog"
```

Because stylesheets change rather often, and it's sometimes beneficial to try out alternative versions, PostgreSQL doesn't use this catalog entry. See Section J.2.6 for information about how to select the stylesheets instead.

J.2.5.4. Installing JadeTeX

To install and use JadeTeX, you will need a working installation of TeX and LaTeX2e, including the supported tools and graphics packages, Babel, AMS fonts and AMS-LaTeX, the PSNFSS extension and companion kit of "the 35 fonts", the dvips program for generating PostScript, the macro packages fancyhdr, hyperref, minitoc, url and ot2enc. All of these can be found on your friendly neighborhood CTAN site[16]. The installation of the TeX base system is far beyond the scope of this introduction. Binary packages should be available for any system that can run TeX.

Before you can use JadeTeX with the PostgreSQL documentation sources, you will need to increase the size of TeX's internal data structures. Details on this can be found in the JadeTeX installation instructions.

15. http://www.oasis-open.org/cover/ISOEnts.zip
16. http://www.ctan.org

Once that is finished you can install JadeTeX:

```
$ gunzip jadetex-xxx.tar.gz
$ tar xf jadetex-xxx.tar
$ cd jadetex
$ make install
$ mktexlsr
```

The last two need to be done as root.

J.2.6. Detection by `configure`

Before you can build the documentation you need to run the `configure` script as you would when building the PostgreSQL programs themselves. Check the output near the end of the run, it should look something like this:

```
checking for onsgmls... onsgmls
checking for openjade... openjade
checking for DocBook V4.2... yes
checking for DocBook stylesheets... /usr/share/sgml/docbook/stylesheet/dsssl/modular
checking for collateindex.pl... /usr/bin/collateindex.pl
checking for xsltproc... xsltproc
checking for osx... osx
```

If neither `onsgmls` nor `nsgmls` were found then some of the following tests will be skipped. `nsgmls` is part of the Jade package. You can pass the environment variables `JADE` and `NSGMLS` to configure to point to the programs if they are not found automatically. If "DocBook V4.2" was not found then you did not install the DocBook DTD kit in a place where Jade can find it, or you have not set up the catalog files correctly. See the installation hints above. The DocBook stylesheets are looked for in a number of relatively standard places, but if you have them some other place then you should set the environment variable `DOCBOOKSTYLE` to the location and rerun `configure` afterwards.

J.3. Building The Documentation

Once you have everything set up, change to the directory `doc/src/sgml` and run one of the commands described in the following subsections to build the documentation. (Remember to use GNU make.)

J.3.1. HTML

To build the HTML version of the documentation:

```
doc/src/sgml$ make html
```

This is also the default target. The output appears in the subdirectory `html`.

To create a proper index, the build might process several identical stages. If you do not care about the index, and just want to proof-read the output, use `draft`:

```
doc/src/sgml$ make draft
```

To build the documentation as a single HTML page, use:

```
doc/src/sgml$ make postgres.html
```

J.3.2. Manpages

We use the DocBook XSL stylesheets to convert DocBook `refentry` pages to *roff output suitable for man pages. The man pages are also distributed as a tar archive, similar to the HTML version. To create the man pages, use the commands:

```
cd doc/src/sgml
make man
```

J.3.3. Print Output via JadeTeX

If you want to use JadeTex to produce a printable rendition of the documentation, you can use one of the following commands:

- To generate PostScript via DVI in A4 format:

  ```
  doc/src/sgml$ make postgres-A4.ps
  ```

  In U.S. letter format:

  ```
  doc/src/sgml$ make postgres-US.ps
  ```

- To make a PDF:

  ```
  doc/src/sgml$ make postgres-A4.pdf
  ```

  or:

  ```
  doc/src/sgml$ make postgres-US.pdf
  ```

  (Of course you can also make a PDF version from the PostScript, but if you generate PDF directly, it will have hyperlinks and other enhanced features.)

When using JadeTeX to build the PostgreSQL documentation, you will probably need to increase some of TeX's internal parameters. These can be set in the file `texmf.cnf`. The following settings worked at the time of this writing:

```
hash_extra.jadetex  = 200000
```

```
hash_extra.pdfjadetex  = 200000
pool_size.jadetex = 2000000
pool_size.pdfjadetex = 2000000
string_vacancies.jadetex = 150000
string_vacancies.pdfjadetex = 150000
max_strings.jadetex = 300000
max_strings.pdfjadetex = 300000
save_size.jadetex = 15000
save_size.pdfjadetex = 15000
```

J.3.4. Overflow Text

Occasionally text is too wide for the printed margins, and in extreme cases, too wide for the printed page, e.g., non-wrapped text, wide tables. Overly wide text generates "Overfull hbox" messages in the TeX log output file, e.g., `postgres-US.log` or `postgres-A4.log`. There are 72 points in an inch so anything reported as over 72 points too wide will probably not fit on the printed page (assuming one inch margins). To find the SGML text causing the overflow, find the first page number mentioned above the overflow message, e.g., `[50 ###]` (page 50), and look at the page after that (e.g., page 51) in the PDF file to see the overflow text and adjust the SGML accordingly.

J.3.5. Print Output via RTF

You can also create a printable version of the PostgreSQL documentation by converting it to RTF and applying minor formatting corrections using an office suite. Depending on the capabilities of the particular office suite, you can then convert the documentation to PostScript of PDF. The procedure below illustrates this process using Applixware.

> **Note:** It appears that current versions of the PostgreSQL documentation trigger some bug in or exceed the size limit of OpenJade. If the build process of the RTF version hangs for a long time and the output file still has size 0, then you might have hit that problem. (But keep in mind that a normal build takes 5 to 10 minutes, so don't abort too soon.)

Applixware RTF Cleanup

OpenJade omits specifying a default style for body text. In the past, this undiagnosed problem led to a long process of table of contents generation. However, with great help from the Applixware folks the symptom was diagnosed and a workaround is available.

1. Generate the RTF version by typing:

   ```
   doc/src/sgml$ make postgres.rtf
   ```

2. Repair the RTF file to correctly specify all styles, in particular the default style. If the document contains `refentry` sections, one must also replace formatting hints which tie a preceding paragraph

to the current paragraph, and instead tie the current paragraph to the following one. A utility, `fixrtf`, is available in `doc/src/sgml` to accomplish these repairs:

```
doc/src/sgml$ ./fixrtf --refentry postgres.rtf
```

The script adds `{\s0 Normal;}` as the zeroth style in the document. According to Applixware, the RTF standard would prohibit adding an implicit zeroth style, though Microsoft Word happens to handle this case. For repairing `refentry` sections, the script replaces `\keepn` tags with `\keep`.

3. Open a new document in Applixware Words and then import the RTF file.
4. Generate a new table of contents (ToC) using Applixware.

    a. Select the existing ToC lines, from the beginning of the first character on the first line to the last character of the last line.

    b. Build a new ToC using Tools⟶Book Building⟶Create Table of Contents. Select the first three levels of headers for inclusion in the ToC. This will replace the existing lines imported in the RTF with a native Applixware ToC.

    c. Adjust the ToC formatting by using Format⟶Style, selecting each of the three ToC styles, and adjusting the indents for `First` and `Left`. Use the following values:

Style	First Indent (inches)	Left Indent (inches)
TOC-Heading 1	0.4	0.4
TOC-Heading 2	0.8	0.8
TOC-Heading 3	1.2	1.2

5. Work through the document to:

    - Adjust page breaks.
    - Adjust table column widths.

6. Replace the right-justified page numbers in the Examples and Figures portions of the ToC with correct values. This only takes a few minutes.
7. Delete the index section from the document if it is empty.
8. Regenerate and adjust the table of contents.

    a. Select the ToC field.

    b. Select Tools⟶Book Building⟶Create Table of Contents.

    c. Unbind the ToC by selecting Tools⟶Field Editing⟶Unprotect.

    d. Delete the first line in the ToC, which is an entry for the ToC itself.

9. Save the document as native Applixware Words format to allow easier last minute editing later.
10. “Print” the document to a file in PostScript format.

J.3.6. Plain Text Files

The installation instructions are also distributed as plain text, in case they are needed in a situation where better reading tools are not available. The `INSTALL` file corresponds to Chapter 15, with some minor changes to account for the different context. To recreate the file, change to the directory `doc/src/sgml` and enter **`make INSTALL`**.

In the past, the release notes and regression testing instructions were also distributed as plain text, but this practice has been discontinued.

J.3.7. Syntax Check

Building the documentation can take very long. But there is a method to just check the correct syntax of the documentation files, which only takes a few seconds:

```
doc/src/sgml$ make check
```

J.4. Documentation Authoring

SGML and DocBook do not suffer from an oversupply of open-source authoring tools. The most common tool set is the Emacs/XEmacs editor with appropriate editing mode. On some systems these tools are provided in a typical full installation.

J.4.1. Emacs/PSGML

PSGML is the most common and most powerful mode for editing SGML documents. When properly configured, it will allow you to use Emacs to insert tags and check markup consistency. You could use it for HTML as well. Check the PSGML web site[17] for downloads, installation instructions, and detailed documentation.

There is one important thing to note with PSGML: its author assumed that your main SGML DTD directory would be `/usr/local/lib/sgml`. If, as in the examples in this chapter, you use `/usr/local/share/sgml`, you have to compensate for this, either by setting `SGML_CATALOG_FILES` environment variable, or you can customize your PSGML installation (its manual tells you how).

Put the following in your `~/.emacs` environment file (adjusting the path names to be appropriate for your system):

```
; ********** for SGML mode (psgml)

(setq sgml-omittag t)
(setq sgml-shorttag t)
(setq sgml-minimize-attributes nil)
(setq sgml-always-quote-attributes t)
```

17. http://www.lysator.liu.se/projects/about_psgml.html

```
(setq sgml-indent-step 1)
(setq sgml-indent-data t)
(setq sgml-parent-document nil)
(setq sgml-exposed-tags nil)
(setq sgml-catalog-files '("/usr/local/share/sgml/catalog"))

(autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t )
```

and in the same file add an entry for SGML into the (existing) definition for `auto-mode-alist`:

```
(setq
  auto-mode-alist
  '(("\\.sgml$" . sgml-mode)
   ))
```

You might find that when using PSGML, a comfortable way of working with these separate files of book parts is to insert a proper `DOCTYPE` declaration while you're editing them. If you are working on this source, for instance, it is an appendix chapter, so you would specify the document as an "appendix" instance of a DocBook document by making the first line look like this:

```
<!DOCTYPE appendix PUBLIC "-//OASIS//DTD DocBook V4.2//EN">
```

This means that anything and everything that reads SGML will get it right, and I can verify the document with `nsgmls -s docguide.sgml`. (But you need to take out that line before building the entire documentation set.)

J.4.2. Other Emacs Modes

GNU Emacs ships with a different SGML mode, which is not quite as powerful as PSGML, but it's less confusing and lighter weight. Also, it offers syntax highlighting (font lock), which can be very helpful. `src/tools/editors/emacs.samples` contains sample settings for this mode.

Norm Walsh offers a major mode[18] specifically for DocBook which also has font-lock and a number of features to reduce typing.

J.5. Style Guide

J.5.1. Reference Pages

Reference pages should follow a standard layout. This allows users to find the desired information more quickly, and it also encourages writers to document all relevant aspects of a command. Consistency is not only desired among PostgreSQL reference pages, but also with reference pages provided by the operating system and other packages. Hence the following guidelines have been developed. They are for the most part consistent with similar guidelines established by various operating systems.

18. http://nwalsh.com/emacs/docbookide/index.html

Reference pages that describe executable commands should contain the following sections, in this order. Sections that do not apply can be omitted. Additional top-level sections should only be used in special circumstances; often that information belongs in the "Usage" section.

Name

This section is generated automatically. It contains the command name and a half-sentence summary of its functionality.

Synopsis

This section contains the syntax diagram of the command. The synopsis should normally not list each command-line option; that is done below. Instead, list the major components of the command line, such as where input and output files go.

Description

Several paragraphs explaining what the command does.

Options

A list describing each command-line option. If there are a lot of options, subsections can be used.

Exit Status

If the program uses 0 for success and non-zero for failure, then you do not need to document it. If there is a meaning behind the different non-zero exit codes, list them here.

Usage

Describe any sublanguage or run-time interface of the program. If the program is not interactive, this section can usually be omitted. Otherwise, this section is a catch-all for describing run-time features. Use subsections if appropriate.

Environment

List all environment variables that the program might use. Try to be complete; even seemingly trivial variables like `SHELL` might be of interest to the user.

Files

List any files that the program might access implicitly. That is, do not list input and output files that were specified on the command line, but list configuration files, etc.

Diagnostics

Explain any unusual output that the program might create. Refrain from listing every possible error message. This is a lot of work and has little use in practice. But if, say, the error messages have a standard format that the user can parse, this would be the place to explain it.

Notes

Anything that doesn't fit elsewhere, but in particular bugs, implementation flaws, security considerations, compatibility issues.

Examples

Examples

History

If there were some major milestones in the history of the program, they might be listed here. Usually, this section can be omitted.

Author

Author (only used in the contrib section)

See Also

Cross-references, listed in the following order: other PostgreSQL command reference pages, PostgreSQL SQL command reference pages, citation of PostgreSQL manuals, other reference pages (e.g., operating system, other packages), other documentation. Items in the same group are listed alphabetically.

Reference pages describing SQL commands should contain the following sections: Name, Synopsis, Description, Parameters, Outputs, Notes, Examples, Compatibility, History, See Also. The Parameters section is like the Options section, but there is more freedom about which clauses of the command can be listed. The Outputs section is only needed if the command returns something other than a default command-completion tag. The Compatibility section should explain to what extent this command conforms to the SQL standard(s), or to which other database system it is compatible. The See Also section of SQL commands should list SQL commands before cross-references to programs.

Appendix K. Acronyms

This is a list of acronyms commonly used in the PostgreSQL documentation and in discussions about PostgreSQL.

ANSI

American National Standards Institute[1]

API

Application Programming Interface[2]

ASCII

American Standard Code for Information Interchange[3]

BKI

Backend Interface

CA

Certificate Authority[4]

CIDR

Classless Inter-Domain Routing[5]

CPAN

Comprehensive Perl Archive Network[6]

CRL

Certificate Revocation List[7]

CSV

Comma Separated Values[8]

CTE

Common Table Expression

CVE

Common Vulnerabilities and Exposures[9]

1. http://en.wikipedia.org/wiki/American_National_Standards_Institute
2. http://en.wikipedia.org/wiki/API
3. http://en.wikipedia.org/wiki/Ascii
4. http://en.wikipedia.org/wiki/Certificate_authority
5. http://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing
6. https://www.cpan.org/
7. http://en.wikipedia.org/wiki/Certificate_revocation_list
8. http://en.wikipedia.org/wiki/Comma-separated_values
9. https://cve.mitre.org/

DBA

Database Administrator[10]

DBI

Database Interface (Perl)[11]

DBMS

Database Management System[12]

DDL

Data Definition Language[13], SQL commands such as `CREATE TABLE`, `ALTER USER`

DML

Data Manipulation Language[14], SQL commands such as `INSERT`, `UPDATE`, `DELETE`

DST

Daylight Saving Time[15]

ECPG

Embedded C for PostgreSQL

ESQL

Embedded SQL[16]

FAQ

Frequently Asked Questions[17]

FSM

Free Space Map

GEQO

Genetic Query Optimizer

GIN

Generalized Inverted Index

GiST

Generalized Search Tree

Git

Git[18]

10. http://en.wikipedia.org/wiki/Database_administrator
11. http://dbi.perl.org/
12. http://en.wikipedia.org/wiki/Dbms
13. http://en.wikipedia.org/wiki/Data_Definition_Language
14. http://en.wikipedia.org/wiki/Data_Manipulation_Language
15. http://en.wikipedia.org/wiki/Daylight_saving_time
16. http://en.wikipedia.org/wiki/Embedded_SQL
17. http://en.wikipedia.org/wiki/FAQ
18. http://en.wikipedia.org/wiki/Git_(software)

GMT

Greenwich Mean Time[19]

GSSAPI

Generic Security Services Application Programming Interface[20]

GUC

Grand Unified Configuration, the PostgreSQL subsystem that handles server configuration

HBA

Host-Based Authentication

HOT

Heap-Only Tuples[21]

IEC

International Electrotechnical Commission[22]

IEEE

Institute of Electrical and Electronics Engineers[23]

IPC

Inter-Process Communication[24]

ISO

International Organization for Standardization[25]

ISSN

International Standard Serial Number[26]

JDBC

Java Database Connectivity[27]

LDAP

Lightweight Directory Access Protocol[28]

MSVC

Microsoft Visual C[29]

MVCC

Multi-Version Concurrency Control

19. http://en.wikipedia.org/wiki/GMT
20. http://en.wikipedia.org/wiki/Generic_Security_Services_Application_Program_Interface
21. https://git.postgresql.org/gitweb?p=postgresql.git;a=blob;f=src/backend/access/heap/README.HOT;hb=HEAD
22. http://en.wikipedia.org/wiki/International_Electrotechnical_Commission
23. https://standards.ieee.org/
24. http://en.wikipedia.org/wiki/Inter-process_communication
25. http://www.iso.org/iso/home.htm
26. http://en.wikipedia.org/wiki/Issn
27. http://en.wikipedia.org/wiki/Java_Database_Connectivity
28. http://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol
29. http://en.wikipedia.org/wiki/Visual_C++

NLS

National Language Support[30]

ODBC

Open Database Connectivity[31]

OID

Object Identifier

OLAP

Online Analytical Processing[32]

OLTP

Online Transaction Processing[33]

ORDBMS

Object-Relational Database Management System[34]

PAM

Pluggable Authentication Modules[35]

PGSQL

PostgreSQL

PGXS

PostgreSQL Extension System

PID

Process Identifier[36]

PITR

Point-In-Time Recovery (Continuous Archiving)

PL

Procedural Languages (server-side)

POSIX

Portable Operating System Interface[37]

RDBMS

Relational Database Management System[38]

30. http://en.wikipedia.org/wiki/Internationalization_and_localization
31. http://en.wikipedia.org/wiki/Open_Database_Connectivity
32. http://en.wikipedia.org/wiki/Olap
33. http://en.wikipedia.org/wiki/OLTP
34. http://en.wikipedia.org/wiki/ORDBMS
35. http://en.wikipedia.org/wiki/Pluggable_Authentication_Modules
36. http://en.wikipedia.org/wiki/Process_identifier
37. http://en.wikipedia.org/wiki/POSIX
38. http://en.wikipedia.org/wiki/Relational_database_management_system

RFC

Request For Comments[39]

SGML

Standard Generalized Markup Language[40]

SPI

Server Programming Interface

SP-GiST

Space-Partitioned Generalized Search Tree

SQL

Structured Query Language[41]

SRF

Set-Returning Function

SSH

Secure Shell[42]

SSL

Secure Sockets Layer[43]

SSPI

Security Support Provider Interface[44]

SYSV

Unix System V[45]

TCP/IP

Transmission Control Protocol (TCP) / Internet Protocol (IP)[46]

TID

Tuple Identifier

TOAST

The Oversized-Attribute Storage Technique

TPC

Transaction Processing Performance Council[47]

39. http://en.wikipedia.org/wiki/Request_for_Comments
40. http://en.wikipedia.org/wiki/SGML
41. http://en.wikipedia.org/wiki/SQL
42. http://en.wikipedia.org/wiki/Secure_Shell
43. http://en.wikipedia.org/wiki/Secure_Sockets_Layer
44. http://msdn.microsoft.com/en-us/library/aa380493%28VS.85%29.aspx
45. http://en.wikipedia.org/wiki/System_V
46. http://en.wikipedia.org/wiki/Transmission_Control_Protocol
47. http://www.tpc.org/

URL

Uniform Resource Locator[48]

UTC

Coordinated Universal Time[49]

UTF

Unicode Transformation Format[50]

UTF8

Eight-Bit Unicode Transformation Format[51]

UUID

Universally Unique Identifier

WAL

Write-Ahead Log

XID

Transaction Identifier

XML

Extensible Markup Language[52]

48. http://en.wikipedia.org/wiki/URL
49. http://en.wikipedia.org/wiki/Coordinated_Universal_Time
50. http://www.unicode.org/
51. http://en.wikipedia.org/wiki/Utf8
52. http://en.wikipedia.org/wiki/XML

Bibliography

Selected references and readings for SQL and PostgreSQL.

Some white papers and technical reports from the original POSTGRES development team are available at the University of California, Berkeley, Computer Science Department web site[1].

SQL Reference Books

Judith Bowman, Sandra Emerson, and Marcy Darnovsky, *The Practical SQL Handbook: Using SQL Variants*, Fourth Edition, Addison-Wesley Professional, ISBN 0-201-70309-2, 2001.

C. J. Date and Hugh Darwen, *A Guide to the SQL Standard: A user's guide to the standard database language SQL*, Fourth Edition, Addison-Wesley, ISBN 0-201-96426-0, 1997.

C. J. Date, *An Introduction to Database Systems*, Eighth Edition, Addison-Wesley, ISBN 0-321-19784-4, 2003.

Ramez Elmasri and Shamkant Navathe, *Fundamentals of Database Systems*, Fourth Edition, Addison-Wesley, ISBN 0-321-12226-7, 2003.

Jim Melton and Alan R. Simon, *Understanding the New SQL: A complete guide*, Morgan Kaufmann, ISBN 1-55860-245-3, 1993.

Jeffrey D. Ullman, *Principles of Database and Knowledge-Base Systems: Classical Database Systems*, Volume 1, Computer Science Press, 1988.

PostgreSQL-specific Documentation

Stefan Simkovics, *Enhancement of the ANSI SQL Implementation of PostgreSQL*, Department of Information Systems, Vienna University of Technology, November 29, 1998.

Discusses SQL history and syntax, and describes the addition of `INTERSECT` and `EXCEPT` constructs into PostgreSQL. Prepared as a Master's Thesis with the support of O. Univ. Prof. Dr. Georg Gottlob and Univ. Ass. Mag. Katrin Seyr at Vienna University of Technology.

A. Yu and J. Chen, The POSTGRES Group, *The Postgres95 User Manual*, University of California, Sept. 5, 1995.

1. https://dsf.berkeley.edu/papers/

Zelaine Fong, *The design and implementation of the POSTGRES query optimizer* [2], University of California, Berkeley, Computer Science Department.

Proceedings and Articles

D. Ports and K. Grittner, "Serializable Snapshot Isolation in PostgreSQL[3]", Proc. VLDB Conference, August 2012.

H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A Critique of ANSI SQL Isolation Levels[4]", Proc. ACM-SIGMOD Conference on Management of Data, June 1995.

Nels Olson, *Partial indexing in POSTGRES: research project*, University of California, UCB Engin T7.49.1993 O676, 1993.

L. Ong and J. Goh, "A Unified Framework for Version Modeling Using Production Rules in a Database System", *ERL Technical Memorandum M90/33*, University of California, April, 1990.

L. Rowe and M. Stonebraker, " The POSTGRES data model [5]", Proc. VLDB Conference, Sept. 1987.

P. Seshadri and A. Swami, "Generalized Partial Indexes (cached version) [6] ", Proc. Eleventh International Conference on Data Engineering, 6-10 March 1995, IEEE Computer Society Press, Cat. No.95CH35724, 1995, 420-7.

M. Stonebraker and L. Rowe, " The design of POSTGRES [7]", Proc. ACM-SIGMOD Conference on Management of Data, May 1986.

M. Stonebraker, E. Hanson, and C. H. Hong, "The design of the POSTGRES rules system", Proc. IEEE Conference on Data Engineering, Feb. 1987.

M. Stonebraker, " The design of the POSTGRES storage system [8]", Proc. VLDB Conference, Sept. 1987.

M. Stonebraker, M. Hearst, and S. Potamianos, " A commentary on the POSTGRES rules system [9]", *SIGMOD Record 18(3)*, Sept. 1989.

M. Stonebraker, " The case for partial indexes [10]", *SIGMOD Record 18(4)*, Dec. 1989, 4-11.

M. Stonebraker, L. A. Rowe, and M. Hirohama, " The implementation of POSTGRES [11]", *Transactions on Knowledge and Data Engineering 2(1)*, IEEE, March 1990.

2. https://dsf.berkeley.edu/papers/UCB-MS-zfong.pdf
3. https://arxiv.org/pdf/1208.4179
4. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-95-51.pdf
5. https://dsf.berkeley.edu/papers/ERL-M87-13.pdf
6. https://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.5740
7. https://dsf.berkeley.edu/papers/ERL-M85-95.pdf
8. https://dsf.berkeley.edu/papers/ERL-M87-06.pdf
9. https://dsf.berkeley.edu/papers/ERL-M89-82.pdf
10. https://dsf.berkeley.edu/papers/ERL-M89-17.pdf
11. https://dsf.berkeley.edu/papers/ERL-M90-34.pdf

M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, " On Rules, Procedures, Caching and Views in Database Systems [12]", Proc. ACM-SIGMOD Conference on Management of Data, June 1990.

12. https://dsf.berkeley.edu/papers/ERL-M90-36.pdf

Index

Symbols

$, 38
$libdir, 1023
$libdir/plugins, 568, 1720
*, 116
.pgpass, 787
.pg_service.conf, 787
::, 45
_PG_fini, 1022
_PG_init, 1022
_PG_output_plugin_init, 1312

A

abbrev, 267
ABORT, 1325
abs, 200
acos, 203
administration tools
  externally maintained, 2601
adminpack, 2424
advisory lock, 435
age, 248
aggregate function, 12
  built-in, 303
  invocation, 40
  moving aggregate, 1048
  ordered set, 1051
  polymorphic, 1049
  support functions for, 1052
  user-defined, 1046
  variadic, 1049
AIX
  installation on, 476
  IPC configuration, 500
akeys, 2484
alias
  for table name in query, 12
  in the FROM clause, 105
  in the select list, 117
ALL, 313, 316
allow_system_table_mods configuration parameter, 576
ALTER AGGREGATE, 1327
ALTER COLLATION, 1330
ALTER CONVERSION, 1332
ALTER DATABASE, 1334
ALTER DEFAULT PRIVILEGES, 1337
ALTER DOMAIN, 1340
ALTER EVENT TRIGGER, 1344
ALTER EXTENSION, 1345
ALTER FOREIGN DATA WRAPPER, 1349
ALTER FOREIGN TABLE, 1351
ALTER FUNCTION, 1357
ALTER GROUP, 1361
ALTER INDEX, 1363
ALTER LANGUAGE, 1366
ALTER LARGE OBJECT, 1367
ALTER MATERIALIZED VIEW, 1368
ALTER OPERATOR, 1370
ALTER OPERATOR CLASS, 1372
ALTER OPERATOR FAMILY, 1374
ALTER POLICY, 1378
ALTER ROLE, 599, 1380
ALTER RULE, 1385
ALTER SCHEMA, 1387
ALTER SEQUENCE, 1388
ALTER SERVER, 1391
ALTER SYSTEM, 1393
ALTER TABLE, 1395
ALTER TABLESPACE, 1408
ALTER TEXT SEARCH CONFIGURATION, 1410
ALTER TEXT SEARCH DICTIONARY, 1412
ALTER TEXT SEARCH PARSER, 1414
ALTER TEXT SEARCH TEMPLATE, 1415
ALTER TRIGGER, 1416
ALTER TYPE, 1418
ALTER USER, 1422
ALTER USER MAPPING, 1424
ALTER VIEW, 1426
ANALYZE, 624, 1429
AND (operator), 197
anonymous code blocks, 1614
any, 195, 305, 313, 316
anyarray, 195
anyelement, 195
anyenum, 195
anynonarray, 195

anyrange, 195
applicable role, 935
application_name configuration parameter, 553
arbitrary precision numbers, 130
archive_cleanup_command recovery parameter, 673
archive_command configuration parameter, 539
archive_mode configuration parameter, 539
archive_timeout configuration parameter, 540
area, 263
armor, 2518
ARRAY, 47, 171
  accessing, 173
  constant, 172
  constructor, 47
  declaration, 171
  determination of result type, 367
  I/O, 179
  modifying, 175
  of user-defined type, 1056
  searching, 178
array_agg, 304, 2490
array_append, 298
array_cat, 298
array_dims, 298
array_fill, 298
array_length, 298
array_lower, 298
array_ndims, 298
array_nulls configuration parameter, 571
array_position, 298
array_positions, 298
array_prepend, 298
array_remove, 298
array_replace, 298
array_to_json, 286
array_to_string, 298
array_upper, 298
ascii, 206
asin, 203
ASSERT
  in PL/pgSQL, 1182
assertions
  in PL/pgSQL, 1182
asynchronous commit, 713
AT TIME ZONE, 257
atan, 203
atan2, 203
authentication_timeout configuration parameter, 526
auth_delay, 2425
auth_delay.milliseconds configuration parameter, 2425
auto-increment
  (see serial)
autocommit
  bulk-loading data, 455
  psql, 1929
autovacuum
  configuration parameters, 560
  general information, 629
autovacuum configuration parameter, 560
autovacuum_analyze_scale_factor configuration parameter, 561
autovacuum_analyze_threshold configuration parameter, 561
autovacuum_freeze_max_age configuration parameter, 561
autovacuum_max_workers configuration parameter, 560
autovacuum_multixact_freeze_max_age configuration parameter, 561
autovacuum_naptime configuration parameter, 560
autovacuum_vacuum_cost_delay configuration parameter, 561
autovacuum_vacuum_cost_limit configuration parameter, 562
autovacuum_vacuum_scale_factor configuration parameter, 561
autovacuum_vacuum_threshold configuration parameter, 560
autovacuum_work_mem configuration parameter, 531
auto_explain, 2425
auto_explain.log_analyze configuration parameter, 2426
auto_explain.log_buffers configuration parameter, 2426
auto_explain.log_format configuration parameter, 2427
auto_explain.log_min_duration configuration parameter, 2426

auto_explain.log_nested_statements configuration parameter, 2427
auto_explain.log_timing configuration parameter, 2426
auto_explain.log_triggers configuration parameter, 2427
auto_explain.log_verbose configuration parameter, 2427
avals, 2484
average, 304
avg, 304

B

B-tree
  (see index)
Background workers, 1309
backslash escapes, 29
backslash_quote configuration parameter, 571
backup, 337, 633
base type, 1001
BASE_BACKUP, 2121
BEGIN, 1432
BETWEEN, 198
BETWEEN SYMMETRIC, 198
bgwriter_delay configuration parameter, 533
bgwriter_lru_maxpages configuration parameter, 533
bgwriter_lru_multiplier configuration parameter, 533
bigint, 33, 129
bigserial, 132
binary data, 136
  functions, 220
binary string
  concatenation, 220
  length, 221
bison, 463
bit string
  constant, 32
  data type, 157
bit strings
  functions, 222
bitmap scan, 375, 544
bit_and, 304
bit_length, 204
bit_or, 304
BLOB
  (see large object)
block_size configuration parameter, 574
bonjour configuration parameter, 526
bonjour_name configuration parameter, 526
Boolean
  data type, 149
  operators
    (see operators, logical)
bool_and, 304
bool_or, 304
booting
  starting the server during, 495
box, 264
box (data type), 154
BRIN
  (see index)
brin_metapage_info, 2509
brin_page_items, 2509
brin_page_type, 2509
brin_revmap_data, 2509
brin_summarize_new_values, 349
broadcast, 267
btree_gin, 2428
btree_gist, 2428
btrim, 206, 221
bt_metap, 2508
bt_page_items, 2508
bt_page_stats, 2508
bytea, 136
bytea_output configuration parameter, 565

C

C, 730, 818
C++, 1046
canceling
  SQL command, 767
cardinality, 298
CASCADE
  with DROP, 94
  foreign key action, 64
Cascading Replication, 649
CASE, 294
  determination of result type, 367
case sensitivity
  of SQL commands, 28

cast
I/O conversion, 1466
cbrt, 200
ceil, 200
ceiling, 200
center, 263
Certificate, 595
char, 134
character, 134
character set, 567, 575, 614
character string
concatenation, 204
constant, 29
data types, 134
length, 204
character varying, 134
char_length, 204
check constraint, 57
CHECK OPTION, 1599
checkpoint, 715, 1434
checkpoint_completion_target configuration parameter, 539
checkpoint_timeout configuration parameter, 538
checkpoint_warning configuration parameter, 539
check_function_bodies configuration parameter, 564
chkpass, 2430
chr, 206
cid, 193
cidr, 156
circle, 155, 264
citext, 2431
client authentication, 580
timeout during, 526
client_encoding configuration parameter, 567
client_min_messages configuration parameter, 562
clock_timestamp, 248
CLOSE, 1435
CLUSTER, 1437
of databases
(see database cluster)
clusterdb, 1818
clustering, 649
cluster_name configuration parameter, 558
cmax, 66
cmin, 66
COALESCE, 296
COLLATE, 45
collation, 611
in PL/pgSQL, 1148
in SQL functions, 1018
collation for, 329
column, 6, 55
adding, 67
removing, 68
renaming, 70
system column, 66
column data type
changing, 69
column reference, 38
col_description, 334
COMMENT, 1440
about database objects, 334
in SQL, 35
COMMIT, 1444
COMMIT PREPARED, 1446
commit_delay configuration parameter, 538
commit_siblings configuration parameter, 538
common table expression
(see WITH)
comparison
composite type, 316
operators, 197
row constructor, 316
subquery result row, 313
compiling
libpq applications, 794
composite type, 180, 1001
comparison, 316
constant, 182
constructor, 48
computed field, 186
concat, 206
concat_ws, 206
concurrency, 423
conditional expression, 294
configuration
of recovery
of a standby server, 673
of the server, 519
of the server
functions, 336
configure, 463

config_file configuration parameter, 523
conjunction, 197
connectby, 2564, 2572
connection service file, 787
conninfo, 736
constant, 29
constraint, 57
  adding, 68
  check, 57
  exclusion, 65
  foreign key, 62
  name, 58
  NOT NULL, 60
  primary key, 62
  removing, 68
  unique, 60
constraint exclusion, 90, 548
constraint_exclusion configuration parameter, 548
CONTINUE
  in PL/pgSQL, 1165
continuous archiving, 633
  in standby, 661
control file, 1077
convert, 206
convert_from, 206
convert_to, 206
COPY, 8, 1448
  with libpq, 770
corr, 306
correlation, 306
cos, 203
cot, 203
count, 304
covariance
  population, 306
  sample, 306
covar_pop, 306
covar_samp, 306
cpu_index_tuple_cost configuration parameter, 546
cpu_operator_cost configuration parameter, 546
cpu_tuple_cost configuration parameter, 546
CREATE AGGREGATE, 1459
CREATE CAST, 1466
CREATE COLLATION, 1471
CREATE CONVERSION, 1473
CREATE DATABASE, 603, 1475
CREATE DOMAIN, 1479
CREATE EVENT TRIGGER, 1482
CREATE EXTENSION, 1484
CREATE FOREIGN DATA WRAPPER, 1487
CREATE FOREIGN TABLE, 1489
CREATE FUNCTION, 1493
CREATE GROUP, 1501
CREATE INDEX, 1502
CREATE LANGUAGE, 1509
CREATE MATERIALIZED VIEW, 1513
CREATE OPERATOR, 1515
CREATE OPERATOR CLASS, 1519
CREATE OPERATOR FAMILY, 1523
CREATE POLICY, 1525
CREATE ROLE, 597, 1531
CREATE RULE, 1536
CREATE SCHEMA, 1539
CREATE SEQUENCE, 1542
CREATE SERVER, 1546
CREATE TABLE, 6, 1548
CREATE TABLE AS, 1564
CREATE TABLESPACE, 607, 1567
CREATE TEXT SEARCH CONFIGURATION, 1569
CREATE TEXT SEARCH DICTIONARY, 1571
CREATE TEXT SEARCH PARSER, 1573
CREATE TEXT SEARCH TEMPLATE, 1575
CREATE TRANSFORM, 1577
CREATE TRIGGER, 1580
CREATE TYPE, 1586
CREATE USER, 1596
CREATE USER MAPPING, 1597
CREATE VIEW, 1599
createdb, 2, 604, 1821
createlang, 1825
createuser, 597, 1828
CREATE_REPLICATION_SLOT, 2118
cross compilation, 469
cross join, 101
crosstab, 2566, 2568, 2569
crypt, 2514
cstring, 195
ctid, 66, 1116
CUBE, 113
cube (extension), 2433
cume_dist, 311

hypothetical, 309
current_catalog, 324
current_database, 324
current_date, 248
current_query, 324
current_role, 324
current_schema, 324
current_schemas, 324
current_setting, 336
current_time, 248
current_timestamp, 248
current_user, 324
currval, 292
cursor
CLOSE, 1435
DECLARE, 1605
FETCH, 1697
in PL/pgSQL, 1173
MOVE, 1724
showing the query plan, 1691
cursor_tuple_fraction configuration parameter, 548
custom scan provider
handler for, 2181
Cygwin
installation on, 479

D

data area
(see database cluster)
data partitioning, 649
data type, 127
base, 1001
category, 358
composite, 1001
constant, 34
conversion, 357
enumerated (enum), 150
internal organization, 1023
numeric, 128
type cast, 45
user-defined, 1053
database, 603
creating, 2
privilege to create, 598
database activity
monitoring, 677
database cluster, 6, 493
data_checksums configuration parameter, 574
data_directory configuration parameter, 523
data_sync_retry configuration parameter, 574
date, 138, 140
constants, 143
current, 258
output format, 144
(see also formatting)
DateStyle configuration parameter, 566
date_part, 248, 253
date_trunc, 248, 256
dblink, 2437, 2443
dblink_build_sql_delete, 2469
dblink_build_sql_insert, 2467
dblink_build_sql_update, 2471
dblink_cancel_query, 2464
dblink_close, 2453
dblink_connect, 2438
dblink_connect_u, 2441
dblink_disconnect, 2442
dblink_error_message, 2456
dblink_exec, 2447
dblink_fetch, 2451
dblink_get_connections, 2455
dblink_get_notify, 2459
dblink_get_pkey, 2465
dblink_get_result, 2461
dblink_is_busy, 2458
dblink_open, 2449
dblink_send_query, 2457
db_user_namespace configuration parameter, 528
deadlock, 434
timeout during, 570
deadlock_timeout configuration parameter, 570
DEALLOCATE, 1604
dearmor, 2518
debug_assertions configuration parameter, 574
debug_deadlocks configuration parameter, 578
debug_pretty_print configuration parameter, 553
debug_print_parse configuration parameter, 553
debug_print_plan configuration parameter, 553

debug_print_rewritten configuration parameter, 553
decimal
  (see numeric)
DECLARE, 1605
decode, 206, 221
decode_bytea
  in PL/Perl, 1225
decrypt, 2522
decrypt_iv, 2522
default value, 56
  changing, 69
default_statistics_target configuration parameter, 547
default_tablespace configuration parameter, 563
default_text_search_config configuration parameter, 568
default_transaction_deferrable configuration parameter, 564
default_transaction_isolation configuration parameter, 564
default_transaction_read_only configuration parameter, 564
default_with_oids configuration parameter, 571
deferrable transaction
  setting, 1794
  setting default, 564
defined, 2484
degrees, 200
delay, 260
DELETE, 14, 98, 1609, 2484
  RETURNING, 98
deleting, 98
dense_rank, 311
  hypothetical, 309
diameter, 263
dict_int, 2473
dict_xsyn, 2473
difference, 2480
digest, 2513
dirty read, 423
DISCARD, 1612
disjunction, 197
disk drive, 718
disk space, 623
disk usage, 709
DISTINCT, 10, 117
div, 200
dmetaphone, 2482
dmetaphone_alt, 2482
DO, 1614
document
  text search, 385
dollar quoting, 32
double precision, 131
DROP AGGREGATE, 1616
DROP CAST, 1618
DROP COLLATION, 1620
DROP CONVERSION, 1622
DROP DATABASE, 606, 1624
DROP DOMAIN, 1625
DROP EVENT TRIGGER, 1627
DROP EXTENSION, 1629
DROP FOREIGN DATA WRAPPER, 1631
DROP FOREIGN TABLE, 1633
DROP FUNCTION, 1635
DROP GROUP, 1637
DROP INDEX, 1638
DROP LANGUAGE, 1640
DROP MATERIALIZED VIEW, 1642
DROP OPERATOR, 1644
DROP OPERATOR CLASS, 1646
DROP OPERATOR FAMILY, 1648
DROP OWNED, 1650
DROP POLICY, 1652
DROP ROLE, 597, 1654
DROP RULE, 1656
DROP SCHEMA, 1658
DROP SEQUENCE, 1660
DROP SERVER, 1662
DROP TABLE, 7, 1664
DROP TABLESPACE, 1666
DROP TEXT SEARCH CONFIGURATION, 1668
DROP TEXT SEARCH DICTIONARY, 1670
DROP TEXT SEARCH PARSER, 1672
DROP TEXT SEARCH TEMPLATE, 1674
DROP TRANSFORM, 1676
DROP TRIGGER, 1678
DROP TYPE, 1680
DROP USER, 1682
DROP USER MAPPING, 1683
DROP VIEW, 1685
dropdb, 606, 1833

droplang, 1836
dropuser, 597, 1839
DROP_REPLICATION_SLOT, 2121
DTD, 162
DTrace, 470, 696
duplicate, 10
duplicates, 117
dynamic loading, 569, 1022
dynamic_library_path, 1023
dynamic_library_path configuration parameter, 569
dynamic_shared_memory_type configuration parameter, 531

E

each, 2484
earth, 2476
earthdistance, 2475
earth_box, 2476
earth_distance, 2476
ECPG, 818, 1842
effective_cache_size configuration parameter, 546
effective_io_concurrency configuration parameter, 534
elog, 2145
  in PL/Perl, 1224
  in PL/Python, 1248
  in PL/Tcl, 1212
embedded SQL
  in C, 818
enabled role, 960
enable_bitmapscan configuration parameter, 544
enable_hashagg configuration parameter, 544
enable_hashjoin configuration parameter, 544
enable_indexonlyscan configuration parameter, 544
enable_indexscan configuration parameter, 544
enable_material configuration parameter, 544
enable_mergejoin configuration parameter, 544
enable_nestloop configuration parameter, 544
enable_seqscan configuration parameter, 545
enable_sort configuration parameter, 545
enable_tidscan configuration parameter, 545
encode, 206, 221
encode_array_constructor
  in PL/Perl, 1225
encode_array_literal
  in PL/Perl, 1225
encode_bytea
  in PL/Perl, 1225
encode_typed_literal
  in PL/Perl, 1225
encrypt, 2522
encryption, 512
  for specific columns, 2513
encrypt_iv, 2522
END, 1687
enumerated types, 150
enum_first, 261
enum_last, 261
enum_range, 261
environment variable, 785
ereport, 2145
error codes
  libpq, 752
  list of, 2258
error message, 744
escape string syntax, 29
escape_string_warning configuration parameter, 572
escaping strings
  in libpq, 759
event log
  event log, 517
event trigger, 1098
  in C, 1104
  in PL/Tcl, 1215
event_source configuration parameter, 551
event_trigger, 195
every, 304
EXCEPT, 118
exceptions
  in PL/pgSQL, 1169
exclusion constraint, 65
EXECUTE, 1689
exist, 2484
EXISTS, 313
EXIT
  in PL/pgSQL, 1164
exit_on_error configuration parameter, 573

exp, 200
EXPLAIN, 440, 1691
expression
  order of evaluation, 50
  syntax, 37
extending SQL, 1001
extension, 1077
  externally maintained, 2601
external_pid_file configuration parameter, 523
extract, 248, 253
extra_float_digits configuration parameter, 567

F

factorial, 200
failover, 649
false, 149
family, 267
fast path, 768
fdw_handler, 195
FETCH, 1697
field
  computed, 186
field selection, 39
file system mount points, 494
file_fdw, 2477
FILTER, 40
first_value, 311
flex, 463
float4
  (see real)
float8
  (see double precision)
floating point, 131
floating-point
  display, 567
floor, 200
foreign data, 93
foreign data wrapper
  handler for, 2162
foreign key, 16, 62
foreign table, 93
format, 206, 218
  use in PL/pgSQL, 1154
formatting, 239
format_type, 329
Free Space Map, 2242
FreeBSD
  IPC configuration, 500
  shared library, 1033
  start script, 496
from_collapse_limit configuration parameter, 548
FSM
  (see Free Space Map)
fsm_page_contents, 2510
fsync configuration parameter, 535
full text search, 384
  data types, 158
  functions and operators, 158
full_page_writes configuration parameter, 537
function, 197
  default values for arguments, 1012
  in the FROM clause, 106
  internal, 1021
  invocation, 40
  mixed notation, 53
  named argument, 1005
  named notation, 52
  output parameter, 1010
  polymorphic, 1002
  positional notation, 52
  RETURNS TABLE, 1016
  type resolution in an invocation, 362
  user-defined, 1003
    in C, 1022
    in SQL, 1004
  variadic, 1011
  with SETOF, 1014
functional dependency, 112
fuzzystrmatch, 2479

G

gc_to_sec, 2476
generate_series, 319
generate_subscripts, 321
genetic query optimization, 547
gen_random_bytes, 2523
gen_random_uuid, 2523
gen_salt, 2515
GEQO
  (see genetic query optimization)
geqo configuration parameter, 547

geqo_effort configuration parameter, 547
geqo_generations configuration parameter, 547
geqo_pool_size configuration parameter, 547
geqo_seed configuration parameter, 547
geqo_selection_bias configuration parameter, 547
geqo_threshold configuration parameter, 547
get_bit, 221
get_byte, 221
get_current_ts_config, 269
get_raw_page, 2507
GIN
  (see index)
gin_fuzzy_search_limit configuration parameter, 570
gin_leafpage_items, 2510
gin_metapage_info, 2510
gin_page_opaque_info, 2510
gin_pending_list_limit configuration parameter, 566
GiST
  (see index)
global data
  in PL/Python, 1241
  in PL/Tcl, 1210
GRANT, 70, 1701
GREATEST, 297
  determination of result type, 367
Gregorian calendar, 2274
GROUP BY, 13, 111
grouping, 111, 310
GROUPING SETS, 113
GSSAPI, 589
GUID, 161

H

hash
  (see index)
has_any_column_privilege, 326
has_column_privilege, 326
has_database_privilege, 326
has_foreign_data_wrapper_privilege, 326
has_function_privilege, 326
has_language_privilege, 326
has_schema_privilege, 326
has_sequence_privilege, 326
has_server_privilege, 326
has_tablespace_privilege, 326
has_table_privilege, 326
has_type_privilege, 326
HAVING, 13, 112
hba_file configuration parameter, 523
heap_page_items, 2507
height, 263
hierarchical database, 6
high availability, 649
history
  of PostgreSQL, xlvii
hmac, 2513
host, 267
host name, 738
hostmask, 267
Hot Standby, 649
hot_standby configuration parameter, 542
hot_standby_feedback configuration parameter, 543
HP-UX
  installation on, 480
  IPC configuration, 501
  shared library, 1033
hstore, 2482, 2484
hstore_to_array, 2484
hstore_to_json, 2484
hstore_to_jsonb, 2484
hstore_to_jsonb_loose, 2484
hstore_to_json_loose, 2484
hstore_to_matrix, 2484
huge_pages configuration parameter, 529
hypothetical-set aggregate
  built-in, 309

I

icount, 2491
ident, 591
identifier
  length, 28
  syntax of, 27
IDENTIFY_SYSTEM, 2117
ident_file configuration parameter, 523
idx, 2491
IFNULL, 296

ignore_checksum_failure configuration parameter, 578
ignore_system_indexes configuration parameter, 576
IMMUTABLE, 1020
IMPORT FOREIGN SCHEMA, 1709
IN, 313, 316
include
  in configuration file, 522
include_dir
  in configuration file, 522
include_if_exists
  in configuration file, 522
index, 370, 2502
  and ORDER BY, 374
  B-tree, 371
  BRIN, 373, 2231
  building concurrently, 1505
  combining multiple indexes, 375
  examining usage, 382
  on expressions, 376
  for user-defined data type, 1063
  GIN, 373, 2224
    text search, 418
  GiST, 372, 2203
    text search, 418
  hash, 371
  locks, 438
  multicolumn, 373
  partial, 377
  SP-GiST, 372, 2214
  unique, 376
index scan, 544
index-only scan, 544
inet (data type), 155
inet_client_addr, 324
inet_client_port, 324
inet_merge, 267
inet_same_family, 267
inet_server_addr, 324
inet_server_port, 324
information schema, 933
inheritance, 22, 81, 572
initcap, 206
initdb, 493, 1949
Initialization Fork, 2243
input function, 1053
INSERT, 7, 96, 1711
  RETURNING, 98
inserting, 96
installation, 461
  on Windows, 486
instr function, 1204
int2
  (see smallint)
int4
  (see integer)
int8
  (see bigint)
intagg, 2490
intarray, 2491
integer, 33, 129
integer_datetimes configuration parameter, 574
interfaces
  externally maintained, 2600
internal, 195
INTERSECT, 118
interval, 138, 146
  output format, 148
    (see also formatting)
IntervalStyle configuration parameter, 566
intset, 2491
int_array_aggregate, 2490
int_array_enum, 2490
inverse distribution, 308
IS DISTINCT FROM, 198, 316
IS DOCUMENT, 277
IS FALSE, 199
IS NOT DISTINCT FROM, 198, 316
IS NOT DOCUMENT, 277
IS NOT FALSE, 199
IS NOT NULL, 199
IS NOT TRUE, 199
IS NOT UNKNOWN, 199
IS NULL, 199, 573
IS TRUE, 199
IS UNKNOWN, 199
isclosed, 263
isempty, 303
isfinite, 248
isn, 2494
ISNULL, 199
isn_weak, 2496
isopen, 263
is_array_ref

in PL/Perl, 1226
is_valid, 2496

J

join, 10, 101
controlling the order, 452
cross, 101
left, 102
natural, 103
outer, 11, 102
right, 102
self, 12
join_collapse_limit configuration parameter, 548
JSON, 164
functions and operators, 283
JSONB, 164
containment, 167
existence, 167
indexes on, 168
jsonb_agg, 304
jsonb_array_elements, 287
jsonb_array_elements_text, 287
jsonb_array_length, 287
jsonb_build_array, 286
jsonb_build_object, 286
jsonb_each, 287
jsonb_each_text, 287
jsonb_extract_path, 287
jsonb_extract_path_text, 287
jsonb_object, 286
jsonb_object_agg, 304
jsonb_object_keys, 287
jsonb_populate_record, 287
jsonb_populate_recordset, 287
jsonb_pretty, 287
jsonb_set, 287
jsonb_strip_nulls, 287
jsonb_to_record, 287
jsonb_to_recordset, 287
jsonb_typeof, 287
json_agg, 304
json_array_elements, 287
json_array_elements_text, 287
json_array_length, 287
json_build_array, 286
json_build_object, 286
json_each, 287
json_each_text, 287
json_extract_path, 287
json_extract_path_text, 287
json_object, 286
json_object_agg, 304
json_object_keys, 287
json_populate_record, 287
json_populate_recordset, 287
json_strip_nulls, 287
json_to_record, 287
json_to_recordset, 287
json_typeof, 287
Julian date, 2274
justify_days, 248
justify_hours, 248
justify_interval, 248

K

key word
list of, 2276
syntax of, 27
krb_caseins_users configuration parameter, 528
krb_server_keyfile configuration parameter, 528

L

label
(see alias)
lag, 311
language_handler, 195
large object, 806
lastval, 292
last_value, 311
LATERAL
in the FROM clause, 108
latitude, 2476
lca, 2502
lc_collate configuration parameter, 574
lc_ctype configuration parameter, 574
lc_messages configuration parameter, 567

lc_monetary configuration parameter, 567
lc_numeric configuration parameter, 567
lc_time configuration parameter, 567
LDAP, 466, 592
LDAP connection parameter lookup, 788
ldconfig, 474
lead, 311
LEAST, 297
  determination of result type, 367
left, 206
left join, 102
length, 206, 221, 263, 269
  of a binary string
    (see binary strings, length)
  of a character string
    (see character string, length)
length(tsvector), 397
levenshtein, 2480
levenshtein_less_equal, 2480
lex, 463
libedit, 461
libperl, 462
libpq, 730
  single-row mode, 766
libpq-fe.h, 730, 742
libpq-int.h, 742
libpython, 462
library finalization function, 1022
library initialization function, 1022
LIKE, 224
  and locales, 611
LIMIT, 119
line, 153
line segment, 153
linear regression, 306
Linux
  IPC configuration, 501
  shared library, 1033
  start script, 496
LISTEN, 1718
listen_addresses configuration parameter, 524
ll_to_earth, 2476
ln, 200
lo, 2498
LOAD, 1720
load balancing, 649
locale, 494, 609
localtime, 248
localtimestamp, 248
local_preload_libraries configuration parameter, 568
lock, 430, 430, 1721
  advisory, 435
  monitoring, 696
lock_timeout configuration parameter, 564
log, 200
log shipping, 649
logging_collector configuration parameter, 549
Logical Decoding, 1312, 1314
login privilege, 598
log_autovacuum_min_duration configuration parameter, 560
log_btree_build_stats configuration parameter, 578
log_checkpoints configuration parameter, 554
log_connections configuration parameter, 554
log_destination configuration parameter, 549
log_directory configuration parameter, 550
log_disconnections configuration parameter, 554
log_duration configuration parameter, 554
log_error_verbosity configuration parameter, 554
log_executor_stats configuration parameter, 559
log_filename configuration parameter, 550
log_file_mode configuration parameter, 550
log_hostname configuration parameter, 554
log_line_prefix configuration parameter, 555
log_lock_waits configuration parameter, 556
log_min_duration_statement configuration parameter, 552
log_min_error_statement configuration parameter, 552
log_min_messages configuration parameter, 551
log_parser_stats configuration parameter, 559
log_planner_stats configuration parameter, 559
log_replication_commands configuration parameter, 556
log_rotation_age configuration parameter, 550
log_rotation_size configuration parameter, 551
log_statement configuration parameter, 556
log_statement_stats configuration parameter, 559
log_temp_files configuration parameter, 556

log_timezone configuration parameter, 556
log_truncate_on_rotation configuration parameter, 551
longitude, 2476
looks_like_number
  in PL/Perl, 1225
loop
  in PL/pgSQL, 1164
lower, 204, 303
  and locales, 611
lower_inc, 303
lower_inf, 303
lo_close, 810
lo_compat_privileges configuration parameter, 572
lo_creat, 807, 811
lo_create, 807
lo_export, 808, 811
lo_from_bytea, 811
lo_get, 811
lo_import, 807, 811
lo_import_with_oid, 808
lo_lseek, 809
lo_lseek64, 809
lo_open, 808
lo_put, 811
lo_read, 809
lo_tell, 810
lo_tell64, 810
lo_truncate, 810
lo_truncate64, 810
lo_unlink, 811, 811
lo_write, 809
lpad, 206
lseg, 153, 264
ltree, 2500
ltree2text, 2502
ltrim, 206

M

MAC address
  (see macaddr)
macaddr (data type), 157
macOS
  installation on, 481
magic block, 1022
maintenance, 622
maintenance_work_mem configuration parameter, 530
make, 461
make_date, 248
make_interval, 248
make_time, 248
make_timestamp, 248
make_timestamptz, 248
make_valid, 2496
MANPATH, 475
masklen, 267
materialized view
  implementation through rules, 1117
materialized views, 2086
max, 304
max_connections configuration parameter, 524
max_files_per_process configuration parameter, 532
max_function_args configuration parameter, 575
max_identifier_length configuration parameter, 575
max_index_keys configuration parameter, 575
max_locks_per_transaction configuration parameter, 570
max_pred_locks_per_transaction configuration parameter, 571
max_prepared_transactions configuration parameter, 530
max_replication_slots configuration parameter, 541
max_stack_depth configuration parameter, 531
max_standby_archive_delay configuration parameter, 542
max_standby_streaming_delay configuration parameter, 543
max_wal_senders configuration parameter, 540
max_wal_size configuration parameter, 539
max_worker_processes configuration parameter, 534
md5, 206, 221, 588
median, 42
  (see also percentile)
memory context
  in SPI, 1294
metaphone, 2481

min, 304
MinGW
  installation on, 481
min_wal_size configuration parameter, 539
mod, 200
mode
  statistical, 308
monitoring
  database activity, 677
MOVE, 1724
moving-aggregate mode, 1048
Multiversion Concurrency Control, 423
MultiXactId, 628
MVCC, 423

N

name
  qualified, 77
  syntax of, 27
  unqualified, 78
NaN
  (see not a number)
natural join, 103
negation, 197
NetBSD
  IPC configuration, 501
  shared library, 1033
  start script, 496
netmask, 267
network, 267
  data types, 155
Network Attached Storage (NAS)
  (see Network File Systems)
Network File Systems, 495
nextval, 292
NFS
  (see Network File Systems)
nlevel, 2502
non-durable, 458
nonblocking connection, 732, 762
nonrepeatable read, 423
normal_rand, 2565
NOT (operator), 197
not a number
  double precision, 131
  numeric (data type), 130
NOT IN, 313, 316
not-null constraint, 60
notation
  functions, 51
notice processing
  in libpq, 778
notice processor, 778
notice receiver, 778
NOTIFY, 1726
  in libpq, 769
NOTNULL, 199
now, 248
npoints, 263
nth_value, 311
ntile, 311
null value
  with check constraints, 59
  comparing, 199
  default value, 56
  in DISTINCT, 117
  in libpq, 757
  in PL/Perl, 1218
  in PL/Python, 1237
  with unique constraints, 61
NULLIF, 297
number
  constant, 33
numeric, 33
numeric (data type), 130
numnode, 269, 397
NVL, 296

O

object identifier
  data type, 193
object-oriented database, 6
obj_description, 334
octet_length, 204, 220
OFFSET, 119
oid, 193
  column, 66
  in libpq, 759
oid2name, 2589
ON CONFLICT, 1711
ONLY, 101
opaque, 195

OpenBSD
  IPC configuration, 501
  shared library, 1033
  start script, 496
OpenSSL, 466
  (see also SSL)
operator, 197
  invocation, 40
  logical, 197
  precedence, 36
  syntax, 34
  type resolution in an invocation, 358
  user-defined, 1058
operator class, 380, 1063
operator family, 380, 1072
operator_precedence_warning configuration parameter, 572
OR (operator), 197
Oracle
  porting from PL/SQL to PL/pgSQL, 1197
ORDER BY, 9, 118
  and locales, 610
ordered-set aggregate, 40
  built-in, 308
ordering operator, 1075
ordinality, 322
OS X
  IPC configuration, 502
  shared library, 1033
outer join, 102
output function, 1053
OVER clause, 43
OVERLAPS, 251
overlay, 204, 220
overloading
  functions, 1019
  operators, 1058
owner, 70

P

pageinspect, 2507
page_header, 2507
palloc, 1032
PAM, 466, 595
parameter
  syntax, 38
parenthesis, 38
partitioning, 85
password, 599
  authentication, 588
  of the superuser, 494
password file, 787
passwordcheck, 2510
password_encryption configuration parameter, 528
path, 264, 475
  for schemas, 562
path (data type), 154
pattern matching, 223
patterns
  in psql and pg_dump, 1928
pclose, 263
peer, 592
percentile
  continuous, 308
  discrete, 308
percent_rank, 311
  hypothetical, 309
performance, 440
perl, 463, 1217
permission
  (see privilege)
pfree, 1032
PGAPPNAME, 786
pgbench, 1852
PGcancel, 767
PGCLIENTENCODING, 786
PGconn, 730
PGCONNECT_TIMEOUT, 786
pgcrypto, 2513
PGDATA, 493
PGDATABASE, 786
PGDATESTYLE, 787
PGEventProc, 781
PGGEQO, 787
PGGSSLIB, 786
PGHOST, 786
PGHOSTADDR, 786
PGKRBSRVNAME, 786
PGLOCALEDIR, 787
PGOPTIONS, 786
PGPASSFILE, 786
PGPASSWORD, 786
PGPORT, 786

pgp_armor_headers, 2518
pgp_key_id, 2518
pgp_pub_decrypt, 2517
pgp_pub_decrypt_bytea, 2517
pgp_pub_encrypt, 2517
pgp_pub_encrypt_bytea, 2517
pgp_sym_decrypt, 2517
pgp_sym_decrypt_bytea, 2517
pgp_sym_encrypt, 2517
pgp_sym_encrypt_bytea, 2517
PGREQUIREPEER, 786
PGREQUIRESSL, 786
PGresult, 751
pgrowlocks, 2528, 2528
PGSERVICE, 786
PGSERVICEFILE, 786
PGSSLCERT, 786
PGSSLCOMPRESSION, 786
PGSSLCRL, 786
PGSSLKEY, 786
PGSSLMODE, 786
PGSSLROOTCERT, 786
pgstatginindex, 2537
pgstatindex, 2536
pgstattuple, 2534, 2535
pgstattuple_approx, 2537
PGSYSCONFDIR, 787
PGTZ, 787
PGUSER, 786
pgxs, 1085
pg_advisory_lock, 352
pg_advisory_lock_shared, 352
pg_advisory_unlock, 352
pg_advisory_unlock_all, 352
pg_advisory_unlock_shared, 352
pg_advisory_xact_lock, 352
pg_advisory_xact_lock_shared, 352
pg_aggregate, 2004
pg_am, 2007
pg_amop, 2009
pg_amproc, 2010
pg_archivecleanup, 1953
pg_attrdef, 2011
pg_attribute, 2012
pg_authid, 2015
pg_auth_members, 2017
pg_available_extensions, 2078
pg_available_extension_versions, 2079
pg_backend_pid, 324
pg_backup_start_time, 337
pg_basebackup, 1845
pg_buffercache, 2511
pg_buffercache_pages, 2511
pg_cancel_backend, 337
pg_cast, 2017
pg_class, 2019
pg_client_encoding, 206
pg_collation, 2023
pg_collation_is_visible, 329
pg_column_size, 346
pg_config, 1864
  with ecpg, 879
  with libpq, 795
  with user-defined C functions, 1032
pg_conf_load_time, 325
pg_constraint, 2024
pg_controldata, 1956
pg_conversion, 2027
pg_conversion_is_visible, 329
pg_create_logical_replication_slot, 342
pg_create_physical_replication_slot, 342
pg_create_restore_point, 337
pg_ctl, 493, 495, 1957
pg_current_xlog_insert_location, 337
pg_current_xlog_location, 337
pg_cursors, 2079
pg_database, 605, 2028
pg_database_size, 346
pg_db_role_setting, 2030
pg_ddl_command, 195
pg_default_acl, 2031
pg_depend, 2032
pg_describe_object, 333
pg_description, 2033
pg_drop_replication_slot, 342
pg_dump, 1867
pg_dumpall, 1880
  use during upgrade, 510
pg_enum, 2034
pg_event_trigger, 2035
pg_event_trigger_ddl_commands, 353
pg_event_trigger_dropped_objects, 354
pg_event_trigger_table_rewrite_oid, 356
pg_event_trigger_table_rewrite_reason, 356
pg_export_snapshot, 341
pg_extension, 2035

pg_extension_config_dump, 1080
pg_filenode_relation, 348
pg_file_rename, 2424
pg_file_settings, 2080
pg_file_unlink, 2424
pg_file_write, 2424
pg_foreign_data_wrapper, 2036
pg_foreign_server, 2037
pg_foreign_table, 2038
pg_freespace, 2526
pg_freespacemap, 2526
pg_function_is_visible, 329
pg_get_constraintdef, 329
pg_get_expr, 329
pg_get_functiondef, 329
pg_get_function_arguments, 329
pg_get_function_identity_arguments, 329
pg_get_function_result, 329
pg_get_indexdef, 329
pg_get_keywords, 329
pg_get_object_address, 333
pg_get_ruledef, 329
pg_get_serial_sequence, 329
pg_get_triggerdef, 329
pg_get_userbyid, 329
pg_get_viewdef, 329
pg_group, 2081
pg_has_role, 326
pg_hba.conf, 580
pg_ident.conf, 587
pg_identify_object, 333
pg_identify_object_as_address, 333
pg_index, 2038
pg_indexes, 2081
pg_indexes_size, 346
pg_inherits, 2041
pg_isready, 1886
pg_is_in_backup, 337
pg_is_in_recovery, 339
pg_is_other_temp_schema, 324
pg_is_xlog_replay_paused, 341
pg_language, 2042
pg_largeobject, 2043
pg_largeobject_metadata, 2044
pg_last_committed_xact, 335
pg_last_xact_replay_timestamp, 339
pg_last_xlog_receive_location, 339
pg_last_xlog_replay_location, 339
pg_listening_channels, 324
pg_locks, 2082
pg_logdir_ls, 2424
pg_logical_slot_get_binary_changes, 342
pg_logical_slot_get_changes, 342
pg_logical_slot_peek_binary_changes, 342
pg_logical_slot_peek_changes, 342
pg_lsn, 195
pg_ls_dir, 350
pg_matviews, 2086
pg_my_temp_schema, 324
pg_namespace, 2044
pg_notify, 1727
pg_opclass, 2045
pg_opclass_is_visible, 329
pg_operator, 2046
pg_operator_is_visible, 329
pg_opfamily, 2047
pg_opfamily_is_visible, 329
pg_options_to_table, 329
pg_pltemplate, 2047
pg_policies, 2086
pg_policy, 2048
pg_postmaster_start_time, 324
pg_prepared_statements, 2087
pg_prepared_xacts, 2088
pg_prewarm, 2527
pg_proc, 2049
pg_range, 2054
pg_read_binary_file, 350
pg_read_file, 350
pg_receivexlog, 1889
pg_recvlogical, 1893
pg_relation_filenode, 348
pg_relation_filepath, 348
pg_relation_size, 346
pg_reload_conf, 337
pg_relpages, 2537
pg_replication_origin, 2055
pg_replication_origin_advance, 342
pg_replication_origin_create, 342
pg_replication_origin_drop, 342
pg_replication_origin_oid, 342
pg_replication_origin_progress, 342
pg_replication_origin_session_is_setup, 342
pg_replication_origin_session_progress, 342
pg_replication_origin_session_reset, 342
pg_replication_origin_session_setup, 342

pg_replication_origin_status, 2089
pg_replication_origin_xact_reset, 342
pg_replication_origin_xact_setup, 342
pg_replication_slots, 2089
pg_resetxlog, 1963
pg_restore, 1897
pg_rewind, 1966
pg_rewrite, 2055
pg_roles, 2091
pg_rotate_logfile, 337
pg_rules, 2092
pg_seclabel, 2056
pg_seclabels, 2093
pg_service.conf, 787
pg_settings, 2094
pg_shadow, 2096
pg_shdepend, 2057
pg_shdescription, 2058
pg_shseclabel, 2059
pg_size_pretty, 346
pg_sleep, 260
pg_sleep_for, 260
pg_sleep_until, 260
pg_standby, 2596
pg_start_backup, 337
pg_statio_all_indexes, 680
pg_statio_all_sequences, 680
pg_statio_all_tables, 680
pg_statio_sys_indexes, 680
pg_statio_sys_sequences, 680
pg_statio_sys_tables, 680
pg_statio_user_indexes, 680
pg_statio_user_sequences, 680
pg_statio_user_tables, 680
pg_statistic, 452, 2059
pg_stats, 452, 2097
pg_stat_activity, 679
pg_stat_all_indexes, 680
pg_stat_all_tables, 680
pg_stat_archiver, 680
pg_stat_bgwriter, 680
pg_stat_clear_snapshot, 694
pg_stat_database, 680
pg_stat_database_conflicts, 680
pg_stat_file, 350
pg_stat_get_activity, 694
pg_stat_get_snapshot_timestamp, 694
pg_stat_replication, 679
pg_stat_reset, 694
pg_stat_reset_shared, 694
pg_stat_reset_single_function_counters, 694
pg_stat_reset_single_table_counters, 694
pg_stat_ssl, 679
pg_stat_statements, 2529
  function, 2532
pg_stat_statements_reset, 2532
pg_stat_sys_indexes, 680
pg_stat_sys_tables, 680
pg_stat_user_functions, 680
pg_stat_user_indexes, 680
pg_stat_user_tables, 680
pg_stat_xact_all_tables, 680
pg_stat_xact_sys_tables, 680
pg_stat_xact_user_functions, 680
pg_stat_xact_user_tables, 680
pg_stop_backup, 337
pg_switch_xlog, 337
pg_tables, 2100
pg_tablespace, 2062
pg_tablespace_databases, 329
pg_tablespace_location, 329
pg_tablespace_size, 346
pg_table_is_visible, 329
pg_table_size, 346
pg_temp, 562
  securing functions, 1499
pg_terminate_backend, 337
pg_test_fsync, 1969
pg_test_timing, 1971
pg_timezone_abbrevs, 2101
pg_timezone_names, 2101
pg_total_relation_size, 346
pg_transform, 2062
pg_trgm, 2538
pg_trigger, 2063
pg_try_advisory_lock, 352
pg_try_advisory_lock_shared, 352
pg_try_advisory_xact_lock, 352
pg_try_advisory_xact_lock_shared, 352
pg_ts_config, 2065
pg_ts_config_is_visible, 329
pg_ts_config_map, 2066
pg_ts_dict, 2066
pg_ts_dict_is_visible, 329
pg_ts_parser, 2067
pg_ts_parser_is_visible, 329

pg_ts_template, 2068
pg_ts_template_is_visible, 329
pg_type, 2068
pg_typeof, 329
pg_type_is_visible, 329
pg_upgrade, 1975
pg_user, 2102
pg_user_mapping, 2077
pg_user_mappings, 2102
pg_views, 2103
pg_xact_commit_timestamp, 335
pg_xlogdump, 1983
pg_xlogfile_name, 337
pg_xlogfile_name_offset, 337
pg_xlog_location_diff, 337
pg_xlog_replay_pause, 341
pg_xlog_replay_resume, 341
phantom read, 424
pi, 200
PIC, 1032
PID
  determining PID of server process
    in libpq, 745
PITR, 633
PITR standby, 649
pkg-config
  with ecpg, 879
  with libpq, 795
PL/Perl, 1217
PL/PerlU, 1227
PL/pgSQL, 1140
PL/Python, 1233
PL/SQL (Oracle)
  porting to PL/pgSQL, 1197
PL/Tcl, 1208
plainto_tsquery, 269, 392
plperl.on_init configuration parameter, 1230
plperl.on_plperlu_init configuration parameter, 1231
plperl.on_plperl_init configuration parameter, 1231
plperl.use_strict configuration parameter, 1231
plpgsql.check_asserts configuration parameter, 1182
plpgsql.variable_conflict configuration parameter, 1192
point, 153, 264
point-in-time recovery, 633
policy, 71
polygon, 154, 264
polymorphic function, 1002
polymorphic type, 1002
popen, 263
populate_record, 2484
port, 739
port configuration parameter, 524
position, 204, 220
POSTGRES, xlvii, 1, 495, 604, 1986
postgres user, 493
Postgres95, xlviii
postgresql.auto.conf, 520
postgresql.conf, 519
postgres_fdw, 2542
postmaster, 1994
post_auth_delay configuration parameter, 576
power, 200
PQbackendPID, 745
PQbinaryTuples, 756
  with COPY, 770
PQcancel, 767
PQclear, 754
PQclientEncoding, 774
PQcmdStatus, 758
PQcmdTuples, 758
PQconndefaults, 734
PQconnectdb, 731
PQconnectdbParams, 730
PQconnectionNeedsPassword, 745
PQconnectionUsedPassword, 745
PQconnectPoll, 732
PQconnectStart, 732
PQconnectStartParams, 732
PQconninfo, 734
PQconninfoFree, 776
PQconninfoParse, 735
PQconsumeInput, 764
PQcopyResult, 776
PQdb, 742
PQdescribePortal, 750
PQdescribePrepared, 750
PQencryptPassword, 776
PQendcopy, 774
PQerrorMessage, 744
PQescapeBytea, 761
PQescapeByteaConn, 761
PQescapeIdentifier, 759

PQescapeLiteral, 759
PQescapeString, 760
PQescapeStringConn, 760
PQexec, 747
PQexecParams, 747
PQexecPrepared, 750
PQfformat, 756
  with COPY, 770
PQfinish, 735
PQfireResultCreateEvents, 776
PQflush, 765
PQfmod, 756
PQfn, 768
PQfname, 755
PQfnumber, 755
PQfreeCancel, 767
PQfreemem, 775
PQfsize, 756
PQftable, 755
PQftablecol, 755
PQftype, 756
PQgetCancel, 767
PQgetCopyData, 772
PQgetisnull, 757
PQgetlength, 757
PQgetline, 772
PQgetlineAsync, 773
PQgetResult, 764
PQgetssl, 746
PQgetvalue, 757
PQhost, 742
PQinitOpenSSL, 793
PQinitSSL, 793
PQinstanceData, 782
PQisBusy, 765
PQisnonblocking, 765
PQisthreadsafe, 794
PQlibVersion, 777
  (see also PQserverVersion)
PQmakeEmptyPGresult, 776
PQnfields, 755
  with COPY, 770
PQnotifies, 769
PQnparams, 757
PQntuples, 754
PQoidStatus, 759
PQoidValue, 759
PQoptions, 743
PQparameterStatus, 743
PQparamtype, 757
PQpass, 742
PQping, 736
PQpingParams, 736
PQport, 742
PQprepare, 749
PQprint, 758
PQprotocolVersion, 744
PQputCopyData, 771
PQputCopyEnd, 771
PQputline, 773
PQputnbytes, 773
PQregisterEventProc, 782
PQrequestCancel, 768
PQreset, 735
PQresetPoll, 735
PQresetStart, 735
PQresStatus, 752
PQresultAlloc, 777
PQresultErrorField, 752
PQresultErrorMessage, 752
PQresultInstanceData, 783
PQresultSetInstanceData, 782
PQresultStatus, 751
PQsendDescribePortal, 763
PQsendDescribePrepared, 763
PQsendPrepare, 763
PQsendQuery, 762
PQsendQueryParams, 762
PQsendQueryPrepared, 763
PQserverVersion, 744
PQsetClientEncoding, 774
PQsetdb, 732
PQsetdbLogin, 731
PQsetErrorVerbosity, 775
PQsetInstanceData, 782
PQsetnonblocking, 765
PQsetNoticeProcessor, 778
PQsetNoticeReceiver, 778
PQsetResultAttrs, 777
PQsetSingleRowMode, 766
PQsetvalue, 777
PQsocket, 744
PQsslAttribute, 745
PQsslAttributeNames, 746
PQsslInUse, 745
PQsslStruct, 746

PQstatus, 743
PQtrace, 775
PQtransactionStatus, 743
PQtty, 743
PQunescapeBytea, 761
PQuntrace, 775
PQuser, 742
predicate locking, 427
PREPARE, 1729
PREPARE TRANSACTION, 1732
prepared statements
  creating, 1729
  executing, 1689
  removing, 1604
  showing the query plan, 1691
preparing a query
  in PL/pgSQL, 1193
  in PL/Python, 1244
  in PL/Tcl, 1211
pre_auth_delay configuration parameter, 576
primary key, 62
primary_conninfo recovery parameter, 675
primary_slot_name recovery parameter, 676
privilege, 70
  querying, 325
  with rules, 1131
  for schemas, 79
  with views, 1131
procedural language, 1137
  externally maintained, 2601
  handler for, 2159
protocol
  frontend-backend, 2104
ps
  to monitor activity, 677
psql, 3, 1906
Python, 1233

Q

qualified name, 77
query, 8, 100
query plan, 440
query tree, 1108
querytree, 269, 397
quotation marks
  and identifiers, 28
  escaping, 29
quote_all_identifiers configuration parameter, 572
quote_ident, 206
  in PL/Perl, 1225
  use in PL/pgSQL, 1154
quote_literal, 206
  in PL/Perl, 1225
  use in PL/pgSQL, 1154
quote_nullable, 206
  in PL/Perl, 1225
  use in PL/pgSQL, 1154

R

radians, 200
radius, 263, 594
RAISE
  in PL/pgSQL, 1180
random, 203
random_page_cost configuration parameter, 545
range table, 1108
range type, 187
  exclude, 192
  indexes on, 192
rank, 311
  hypothetical, 309
read committed, 424
read-only transaction
  setting, 1794
  setting default, 564
readline, 461
real, 131
REASSIGN OWNED, 1734
record, 195
recovery.conf, 673
recovery_end_command recovery parameter, 674
recovery_min_apply_delay recovery parameter, 676
recovery_target recovery parameter, 674
recovery_target_action recovery parameter, 675
recovery_target_inclusive recovery parameter, 674
recovery_target_name recovery parameter, 674

recovery_target_time recovery parameter, 674
recovery_target_timeline recovery parameter, 675
recovery_target_xid recovery parameter, 674
rectangle, 154
referential integrity, 16, 62
REFRESH MATERIALIZED VIEW, 1736
regclass, 193
regconfig, 193
regdictionary, 193
regexp_matches, 206, 226
regexp_replace, 206, 226
regexp_split_to_array, 206, 226
regexp_split_to_table, 206, 226
regoper, 193
regoperator, 193
regproc, 193
regprocedure, 193
regression intercept, 306
regression slope, 306
regression test, 472
regression tests, 720
regr_avgx, 306
regr_avgy, 306
regr_count, 306
regr_intercept, 306
regr_r2, 306
regr_slope, 306
regr_sxx, 306
regr_sxy, 306
regr_syy, 306
regtype, 193
regular expression, 225, 226
  (see also pattern matching)
regular expressions
  and locales, 611
reindex, 630, 1738
reindexdb, 1940
relation, 6
relational database, 6
RELEASE SAVEPOINT, 1741
repeat, 206
repeatable read, 426
replace, 206
replication, 649
Replication Origins, 1321
Replication Progress Tracking, 1321
replication slot
  logical replication, 1314
  streaming replication, 657
reporting errors
  in PL/pgSQL, 1180
RESET, 1743
restartpoint, 716
restart_after_crash configuration parameter, 573
restore_command recovery parameter, 673
RESTRICT
  with DROP, 94
  foreign key action, 64
RETURN NEXT
  in PL/pgSQL, 1159
RETURN QUERY
  in PL/pgSQL, 1159
RETURNING, 98
RETURNING INTO
  in PL/pgSQL, 1151
reverse, 206
REVOKE, 70, 1745
right, 206
right join, 102
role, 597
  applicable, 935
  enabled, 960
  membership in, 599
  privilege to create, 598
  privilege to initiate replication, 598
ROLLBACK, 1749
  psql, 1931
ROLLBACK PREPARED, 1751
ROLLBACK TO SAVEPOINT, 1753
ROLLUP, 113
round, 200
routine maintenance, 622
row, 6, 48, 55
row estimation
  planner, 2250
row type, 180
  constructor, 48
row-level security, 71
row-wise comparison, 316
row_number, 311
row_security configuration parameter, 563
row_security_active, 326
row_to_json, 286
rpad, 206

rtrim, 206
rule, 1108
  and materialized views, 1117
  and views, 1110
  for DELETE, 1120
  for INSERT, 1120
  for SELECT, 1110
  compared with triggers, 1134
  for UPDATE, 1120

S

SAVEPOINT, 1755
savepoints
  defining, 1755
  releasing, 1741
  rolling back, 1753
scalar
  (see expression)
schema, 76, 603
  creating, 77
  current, 78, 324
  public, 78
  removing, 77
SCO
  installation on, 482
SCO OpenServer
  IPC configuration, 502
search path, 78
  current, 324
  object visibility, 328
search_path configuration parameter, 78, 562
  use in securing functions, 1499
SECURITY LABEL, 1757
sec_to_gc, 2476
seg, 2548
segment_size configuration parameter, 575
SELECT, 8, 100, 1760
  select list, 116
SELECT INTO, 1782
  in PL/pgSQL, 1151
semaphores, 498
sepgsql, 2551
sepgsql.debug_audit configuration parameter, 2554
sepgsql.permissive configuration parameter, 2554
sequence, 292
  and serial type, 132
sequential scan, 545
seq_page_cost configuration parameter, 545
serial, 132
serial2, 132
serial4, 132
serial8, 132
serializable, 427
Serializable Snapshot Isolation, 423
serialization anomaly, 424, 427
server log, 549
  log file maintenance, 631
server spoofing, 511
server_encoding configuration parameter, 575
server_version configuration parameter, 575
server_version_num configuration parameter, 575
session_preload_libraries configuration parameter, 568
session_replication_role configuration parameter, 564
session_user, 324
SET, 336, 1784
SET CONSTRAINTS, 1788
set difference, 118
set intersection, 118
set operation, 118
set returning functions
  functions, 319
SET ROLE, 1790
SET SESSION AUTHORIZATION, 1792
SET TRANSACTION, 1794
set union, 118
SET XML OPTION, 566
setseed, 203
setval, 292
setweight, 269, 396
set_bit, 221
set_byte, 221
set_config, 336
set_limit, 2539
set_masklen, 267
shared library, 474, 1032
shared memory, 498
shared_buffers configuration parameter, 529
shared_preload_libraries, 1045

shared_preload_libraries configuration parameter, 569
shobj_description, 334
SHOW, 336, 1797
show_limit, 2539
show_trgm, 2539
shutdown, 508
SIGHUP, 520, 585, 587
SIGINT, 508
sign, 200
signal
  backend processes, 337
significant digits, 567
SIGQUIT, 508
SIGTERM, 508
SIMILAR TO, 225
similarity, 2539
sin, 203
single-user mode, 1990
skeys, 2484
sleep, 260
slice, 2484
sliced bread
  (see TOAST)
smallint, 129
smallserial, 132
Solaris
  installation on, 483
  IPC configuration, 503
  shared library, 1033
  start script, 496
SOME, 305, 313, 316
sort, 2491
sorting, 118
sort_asc, 2491
sort_desc, 2491
soundex, 2480
SP-GiST
  (see index)
SPI, 1250
  examples, 2560
SPI_connect, 1250
SPI_copytuple, 1298
SPI_cursor_close, 1283
  in PL/Perl, 1222
SPI_cursor_fetch, 1279
SPI_cursor_find, 1278
SPI_cursor_move, 1280
SPI_cursor_open, 1273
SPI_cursor_open_with_args, 1275
SPI_cursor_open_with_paramlist, 1277
SPI_exec, 1259
SPI_execp, 1272
SPI_execute, 1255
SPI_execute_plan, 1269
SPI_execute_plan_with_paramlist, 1271
SPI_execute_with_args, 1260
spi_exec_prepared
  in PL/Perl, 1223
spi_exec_query
  in PL/Perl, 1221
spi_fetchrow
  in PL/Perl, 1222
SPI_finish, 1252
SPI_fname, 1286
SPI_fnumber, 1287
SPI_freeplan, 1304
  in PL/Perl, 1223
SPI_freetuple, 1302
SPI_freetuptable, 1303
SPI_getargcount, 1266
SPI_getargtypeid, 1267
SPI_getbinval, 1289
SPI_getnspname, 1293
SPI_getrelname, 1292
SPI_gettype, 1290
SPI_gettypeid, 1291
SPI_getvalue, 1288
SPI_is_cursor_plan, 1268
SPI_keepplan, 1284
spi_lastoid
  in PL/Tcl, 1212
SPI_modifytuple, 1300
SPI_palloc, 1294
SPI_pfree, 1297
SPI_pop, 1254
SPI_prepare, 1262
  in PL/Perl, 1223
SPI_prepare_cursor, 1264
SPI_prepare_params, 1265
SPI_push, 1253
spi_query
  in PL/Perl, 1222
spi_query_prepared
  in PL/Perl, 1223
SPI_repalloc, 1296

SPI_returntuple, 1299
SPI_saveplan, 1285
SPI_scroll_cursor_fetch, 1281
SPI_scroll_cursor_move, 1282
split_part, 206
SQL/CLI, 2301
SQL/Foundation, 2301
SQL/Framework, 2301
SQL/JRT, 2301
SQL/MED, 2301
SQL/OLB, 2301
SQL/PSM, 2301
SQL/Schemata, 2301
SQL/XML, 2301
sql_inheritance configuration parameter, 572
sqrt, 200
ssh, 516
SSI, 423
SSL, 513, 789
  with libpq, 740, 746
ssl configuration parameter, 527
sslinfo, 2562
ssl_ca_file configuration parameter, 527
ssl_cert_file configuration parameter, 527
ssl_cipher, 2563
ssl_ciphers configuration parameter, 527
ssl_client_cert_present, 2563
ssl_client_dn, 2563
ssl_client_dn_field, 2563
ssl_client_serial, 2563
ssl_crl_file configuration parameter, 527
ssl_ecdh_curve configuration parameter, 528
ssl_issuer_dn, 2563
ssl_issuer_field, 2564
ssl_is_used, 2562
ssl_key_file configuration parameter, 527
ssl_prefer_server_ciphers configuration parameter, 528
ssl_version, 2563
SSPI, 590
STABLE, 1020
standard deviation, 306
  population, 306
  sample, 306
standard_conforming_strings configuration parameter, 572
standby server, 649
standby_mode recovery parameter, 675
START TRANSACTION, 1800
START_REPLICATION, 2118
statement_timeout configuration parameter, 564
statement_timestamp, 248
statistics, 306, 678
  of the planner, 451, 624
stats_temp_directory configuration parameter, 559
stddev, 306
stddev_pop, 306
stddev_samp, 306
STONITH, 649
storage parameters, 1556
Streaming Replication, 649
string
  (see character string)
strings
  backslash quotes, 571
  escape warning, 572
  standard conforming, 572
string_agg, 304
string_to_array, 298
strip, 269, 397
strpos, 206
subarray, 2491
subltree, 2502
subpath, 2502
subquery, 12, 46, 106, 313
subscript, 39
substr, 206
substring, 204, 220, 225, 226
sum, 304
superuser, 4, 598
superuser_reserved_connections configuration parameter, 525
suppress_redundant_updates_trigger, 352
svals, 2484
synchronize_seqscans configuration parameter, 573
synchronous commit, 713
Synchronous Replication, 649
synchronous_commit configuration parameter, 536
synchronous_standby_names configuration parameter, 541
syntax
  SQL, 27

syslog_facility configuration parameter, 551
syslog_ident configuration parameter, 551
system catalog
  schema, 80
systemd
  RemoveIPC, 504

T

table, 6, 55
  creating, 55
  inheritance, 81
  modifying, 67
  partitioning, 85
  removing, 56
  renaming, 70
TABLE command, 1760
table expression, 100
table function, 106
table sampling method, 2177
tablefunc, 2564
tableoid, 66
TABLESAMPLE method, 2177
tablespace, 606
  default, 563
  temporary, 563
tan, 203
target list, 1109
Tcl, 1208
tcn, 2574
tcp_keepalives_count configuration parameter, 526
tcp_keepalives_idle configuration parameter, 526
tcp_keepalives_interval configuration parameter, 526
template0, 605
template1, 604, 604
temp_buffers configuration parameter, 530
temp_file_limit configuration parameter, 531
temp_tablespaces configuration parameter, 563
test, 720
test_decoding, 2575
text, 134, 267
text search, 384
  data types, 158
  functions and operators, 158
  indexes, 418
text2ltree, 2502
threads
  with libpq, 793
tid, 193
time, 138, 141
  constants, 143
  current, 258
  output format, 144
    (see also formatting)
time span, 138
time with time zone, 138, 141
time without time zone, 138, 141
time zone, 145, 567
  conversion, 257
  input abbreviations, 2270
  POSIX-style specification, 2272
time zone data, 469
time zone names, 567
timelines, 633
TIMELINE_HISTORY, 2118
timeofday, 248
timeout
  client authentication, 526
  deadlock, 570
timestamp, 138, 142
timestamp with time zone, 138, 142
timestamp without time zone, 138, 142
timestamptz, 138
TimeZone configuration parameter, 567
timezone_abbreviations configuration parameter, 567
TOAST, 2239
  and user-defined types, 1056
  per-column storage settings, 1397
  versus large objects, 806
token, 27
to_ascii, 206
to_char, 240
  and locales, 611
to_date, 240
to_hex, 206
to_json, 286
to_jsonb, 286
to_number, 240
to_regclass, 329
to_regnamespace, 329
to_regoper, 329

to_regoperator, 329
to_regproc, 329
to_regprocedure, 329
to_regrole, 329
to_regtype, 329
to_timestamp, 240
to_tsquery, 269, 391
to_tsvector, 269, 390
trace_locks configuration parameter, 577
trace_lock_oidmin configuration parameter, 577
trace_lock_table configuration parameter, 578
trace_lwlocks configuration parameter, 577
trace_notify configuration parameter, 576
trace_recovery_messages configuration parameter, 576
trace_sort configuration parameter, 577
trace_userlocks configuration parameter, 577
track_activities configuration parameter, 559
track_activity_query_size configuration parameter, 559
track_commit_timestamp configuration parameter, 541
track_counts configuration parameter, 559
track_functions configuration parameter, 559
track_io_timing configuration parameter, 559
transaction, 17
transaction ID
  wraparound, 625
transaction isolation, 423
transaction isolation level, 424
  read committed, 424
  repeatable read, 426
  serializable, 427
  setting, 1794
  setting default, 564
transaction log
  (see WAL)
transaction_timestamp, 248
transform_null_equals configuration parameter, 573
translate, 206
trigger, 195, 1088
  arguments for trigger functions, 1090
  for updating a derived tsvector column, 399
  in C, 1091
  in PL/pgSQL, 1182
  in PL/Python, 1242
  in PL/Tcl, 1213
  compared with rules, 1134
triggered_change_notification, 2574
trigger_file recovery parameter, 676
trim, 204, 220
true, 149
trunc, 200, 268
TRUNCATE, 1801
trusted
  PL/Perl, 1227
tsearch2, 2576
tsm_handler, 195
tsm_system_rows, 2577
tsm_system_time, 2578
tsquery (data type), 159
tsvector (data type), 158
tsvector concatenation, 396
tsvector_update_trigger, 269
tsvector_update_trigger_column, 269
ts_debug, 271, 414
ts_headline, 269, 394
ts_lexize, 271, 417
ts_parse, 271, 416
ts_rank, 269, 392
ts_rank_cd, 269, 392
ts_rewrite, 269, 397
ts_stat, 271, 400
ts_token_type, 271, 416
txid_current, 334
txid_current_snapshot, 334
txid_snapshot_xip, 334
txid_snapshot_xmax, 334
txid_snapshot_xmin, 334
txid_visible_in_snapshot, 334
type
  (see data type)
  polymorphic, 1002
type cast, 33, 45

U

UESCAPE, 28, 31
unaccent, 2579, 2581
Unicode escape
  in identifiers, 28
  in string constants, 31
UNION, 118

determination of result type, 367
uniq, 2491
unique constraint, 60
Unix domain socket, 738
UnixWare
installation on, 482
IPC configuration, 503
shared library, 1034
unix_socket_directories configuration parameter, 525
unix_socket_group configuration parameter, 525
unix_socket_permissions configuration parameter, 525
UNLISTEN, 1804
unnest, 298
unqualified name, 78
updatable views, 1601
UPDATE, 14, 97, 1806
RETURNING, 98
update_process_title configuration parameter, 558
updating, 97
upgrading, 509
upper, 204, 303
and locales, 611
upper_inc, 303
upper_inf, 303
UPSERT, 1711
URI, 736
user, 324, 597
current, 324
user mapping, 93
User name maps, 586
UUID, 161, 467
uuid-ossp, 2581
uuid_generate_v1, 2581
uuid_generate_v1mc, 2581
uuid_generate_v3, 2581

V

vacuum, 622, 1811
vacuumdb, 1944
vacuumlo, 2594
vacuum_cost_delay configuration parameter, 532
vacuum_cost_limit configuration parameter, 533
vacuum_cost_page_dirty configuration parameter, 532
vacuum_cost_page_hit configuration parameter, 532
vacuum_cost_page_miss configuration parameter, 532
vacuum_defer_cleanup_age configuration parameter, 542
vacuum_freeze_min_age configuration parameter, 565
vacuum_freeze_table_age configuration parameter, 565
vacuum_multixact_freeze_min_age configuration parameter, 565
vacuum_multixact_freeze_table_age configuration parameter, 565
value expression, 37
VALUES, 120, 1814
determination of result type, 367
varchar, 134
variadic function, 1011
variance, 306
population, 306
sample, 306
var_pop, 306
var_samp, 306
version, 4, 325
compatibility, 509
view, 16
implementation through rules, 1110
materialized, 1117
updating, 1125
Visibility Map, 2243
VM
(see Visibility Map)
void, 195
VOLATILE, 1020
volatility
functions, 1020
VPATH, 463, 1087

W

WAL, 711
wal_block_size configuration parameter, 575
wal_buffers configuration parameter, 537
wal_compression configuration parameter, 537
wal_debug configuration parameter, 578
wal_keep_segments configuration parameter, 541
wal_level configuration parameter, 534
wal_log_hints configuration parameter, 537
wal_receiver_status_interval configuration parameter, 543
wal_receiver_timeout configuration parameter, 543
wal_retrieve_retry_interval configuration parameter, 543
wal_segment_size configuration parameter, 575
wal_sender_timeout configuration parameter, 541
wal_sync_method configuration parameter, 536
wal_writer_delay configuration parameter, 538
warm standby, 649
WHERE, 110
where to log, 549
WHILE
  in PL/pgSQL, 1166
width, 263
width_bucket, 200
window function, 19
  built-in, 311
  invocation, 43
  order of execution, 115
WITH
  in SELECT, 121, 1760
WITH CHECK OPTION, 1599
WITHIN GROUP, 40
witness server, 649
work_mem configuration parameter, 530
wraparound
  of multixact IDs, 628
  of transaction IDs, 625

X

xid, 193
xmax, 66
xmin, 66
XML, 161
XML export, 280
XML option, 162, 566
xml2, 2583
xmlagg, 276, 304
xmlbinary configuration parameter, 565
xmlcomment, 273
xmlconcat, 273
xmlelement, 273
XMLEXISTS, 277
xmlforest, 275
xmloption configuration parameter, 566
xmlparse, 162
xmlpi, 276
xmlroot, 276
xmlserialize, 162
xml_is_well_formed, 278
xml_is_well_formed_content, 278
xml_is_well_formed_document, 278
XPath, 279
xpath_exists, 280
xpath_table, 2585
xslt_process, 2587

Y

yacc, 463

Z

zero_damaged_pages configuration parameter, 578
zlib, 462, 469