pgAudit - Open Source PostgreSQL Audit Logging

Contents

pgAudit Open Source PostgreSQL Audit Logging **1**

Introduction . . . 2
Why pgAudit? . . . 2
Usage Considerations . . . 3
PostgreSQL Version Compatibility . . . 3
Compile and Install . . . 4
Settings . . . 4
pgaudit.log . . . 5
pgaudit.log_catalog . . . 5
pgaudit.log_client . . . 5
pgaudit.log_level . . . 6
pgaudit.log_parameter . . . 6
pgaudit.log_parameter_max_size . . . 6
pgaudit.log_relation . . . 6
pgaudit.log_rows . . . 6
pgaudit.log_statement . . . 6
pgaudit.log_statement_once . . . 7
pgaudit.role . . . 7
Session Audit Logging . . . 7
Configuration . . . 7
Example . . . 7
Object Audit Logging . . . 8
Configuration . . . 8
Example . . . 9
Format . . . 10
Caveats . . . 11
Authors . . . 12

pgAudit Open Source PostgreSQL Audit Logging

Introduction

The PostgreSQL Audit Extension (pgAudit) provides detailed session and/or object audit logging via the standard PostgreSQL logging facility.

The goal of pgAudit is to provide PostgreSQL users with capability to produce audit logs often required to comply with government, financial, or ISO certifications.

An audit is an official inspection of an individual's or organization's accounts, typically by an independent body. The information gathered by pgAudit is properly called an audit trail or audit log. The term audit log is used in this documentation.

Why pgAudit?

Basic statement logging can be provided by the standard logging facility with `log_statement = all`. This is acceptable for monitoring and other usages but does not provide the level of detail generally required for an audit. It is not enough to have a list of all the operations performed against the database. It must also be possible to find particular statements that are of interest to an auditor. The standard logging facility shows what the user requested, while pgAudit focuses on the details of what happened while the database was satisfying the request.

For example, an auditor may want to verify that a particular table was created inside a documented maintenance window. This might seem like a simple job for grep, but what if you are presented with something like this (intentionally obfuscated) example:

```
DO $$
BEGIN
    EXECUTE 'CREATE TABLE import' || 'ant_table (id INT)';
END $$;
```

Standard logging will give you this:

```
LOG:  statement: DO $$
BEGIN
    EXECUTE 'CREATE TABLE import' || 'ant_table (id INT)';
END $$;
```

It appears that finding the table of interest may require some knowledge of the code in cases where tables are created dynamically. This is not ideal since it would be preferable to just search on the table name. This is where pgAudit comes in. For the same input, it will produce this output in the log:

```
AUDIT: SESSION,33,1,FUNCTION,DO,,,"DO $$
BEGIN
    EXECUTE 'CREATE TABLE import' || 'ant_table (id INT)';
```

```
END $$;"
AUDIT: SESSION,33,2,DDL,CREATE TABLE,TABLE,public.important_table,CREATE TABLE important_tab
```

Not only is the `DO` block logged, but substatement 2 contains the full text of the `CREATE TABLE` with the statement type, object type, and full-qualified name to make searches easy.

When logging `SELECT` and `DML` statements, pgAudit can be configured to log a separate entry for each relation referenced in a statement. No parsing is required to find all statements that touch a particular table. In fact, the goal is that the statement text is provided primarily for deep forensics and should not be required for an audit.

Usage Considerations

Depending on settings, it is possible for pgAudit to generate an enormous volume of logging. Be careful to determine exactly what needs to be audit logged in your environment to avoid logging too much.

For example, when working in an OLAP environment it would probably not be wise to audit log inserts into a large fact table. The size of the log file will likely be many times the actual data size of the inserts because the log file is expressed as text. Since logs are generally stored with the OS this may lead to disk space being exhausted very quickly. In cases where it is not possible to limit audit logging to certain tables, be sure to assess the performance impact while testing and allocate plenty of space on the log volume. This may also be true for OLTP environments. Even if the insert volume is not as high, the performance impact of audit logging may still noticeably affect latency.

To limit the number of relations audit logged for `SELECT` and `DML` statements, consider using object audit logging (see Object Auditing). Object audit logging allows selection of the relations to be logged allowing for reduction of the overall log volume. However, when new relations are added they must be explicitly added to object audit logging. A programmatic solution where specified tables are excluded from logging and all others are included may be a good option in this case.

PostgreSQL Version Compatibility

pgAudit supports PostgreSQL 14 or greater.

In order to support new functionality introduced in each PostgreSQL release, pgAudit maintains a separate branch for each PostgreSQL major version (currently PostgreSQL 14 - 19) which will be maintained in a manner similar to the PostgreSQL project.

Aside from bug fixes, no further development is allowed for stable branches. New development, if any, will be strictly for the next unreleased major version of PostgreSQL.

pgAudit versions relate to PostgreSQL major versions as follows:

- **pgAudit v19.X** is intended to support PostgreSQL 19.
- **pgAudit v18.X** is intended to support PostgreSQL 18.
- **pgAudit v17.X** is intended to support PostgreSQL 17.
- **pgAudit v16.X** is intended to support PostgreSQL 16.
- **pgAudit v1.7.X** is intended to support PostgreSQL 15.
- **pgAudit v1.6.X** is intended to support PostgreSQL 14.

Compile and Install

pgAudit can be compiled against an installed copy of PostgreSQL with development packages using `PGXS`. The following instructions should work on most Unix-like operating systems.

Clone the pgAudit extension:

```
git clone https://github.com/pgaudit/pgaudit.git
```

Change to pgAudit directory:

```
cd pgaudit
```

Checkout `REL_19_STABLE` branch (note that the stable branch may not exist for unreleased versions of PostgreSQL):

```
git checkout REL_19_STABLE
```

Build and install pgAudit:

```
make install USE_PGXS=1 PG_CONFIG=/usr/pgsql-19/bin/pg_config
```

Instructions for testing and development may be found in `test`.

Settings

Settings may be modified only by a superuser. Allowing normal users to change their settings would defeat the point of an audit log.

Settings can be specified globally (in `postgresql.conf` or using `ALTER SYSTEM ... SET`), at the database level (using `ALTER DATABASE ... SET`), or at the role level (using `ALTER ROLE ... SET`). Note that settings are not inherited through normal role inheritance and `SET ROLE` will not alter a user's pgAudit settings. This is a limitation of the roles system and not inherent to pgAudit.

The pgAudit extension must be loaded in shared_preload_libraries. Otherwise, an error will be raised at load time and no audit logging will occur.

In addition, `CREATE EXTENSION pgaudit` must be called before `pgaudit.log` is set to ensure proper pgaudit functionality. The extension installs event triggers

which add additional auditing for DDL. pgAudit will work without the extension installed but DDL statements will not have information about the object type and name.

If the `pgaudit` extension is dropped and needs to be recreated then `pgaudit.log` must be unset first otherwise an error will be raised.

pgaudit.log

Specifies which classes of statements will be logged by session audit logging. Possible values are:

- **READ**: `SELECT` and `COPY` when the source is a relation or a query.
- **WRITE**: `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE`, and `COPY` when the destination is a relation.
- **FUNCTION**: Function calls and `DO` blocks.
- **ROLE**: Statements related to roles and privileges: `GRANT`, `REVOKE`, `CREATE/ALTER/DROP ROLE`.
- **DDL**: All `DDL` that is not included in the `ROLE` class.
- **MISC**: Miscellaneous commands, e.g. `DISCARD`, `FETCH`, `CHECKPOINT`, `VACUUM`, `SET`.
- **MISC_SET**: Miscellaneous `SET` commands, e.g. `SET ROLE`.
- **ALL**: Include all of the above.

Multiple classes can be provided using a comma-separated list and classes can be subtracted by prefacing the class with a `-` sign (see Session Audit Logging).

The default is `none`.

pgaudit.log_catalog

Specifies that session logging should be enabled in the case where all relations in a statement are in pg_catalog. Disabling this setting will reduce noise in the log from tools like psql and PgAdmin that query the catalog heavily.

The default is `on`.

pgaudit.log_client

Specifies whether log messages will be visible to a client process such as psql. This setting should generally be left disabled but may be useful for debugging or other purposes.

Note that `pgaudit.log_level` is only enabled when `pgaudit.log_client` is `on`.

The default is `off`.

pgaudit.log_level

Specifies the log level that will be used for log entries (see Message Severity Levels for valid levels) but note that `ERROR`, `FATAL`, and `PANIC` are not allowed). This setting is used for regression testing and may also be useful to end users for testing or other purposes.

Note that `pgaudit.log_level` is only enabled when `pgaudit.log_client` is `on`; otherwise the default will be used.

The default is `log`.

pgaudit.log_parameter

Specifies that audit logging should include the parameters that were passed with the statement. When parameters are present they will be included in `CSV` format after the statement text.

The default is `off`.

pgaudit.log_parameter_max_size

Specifies that parameter values longer than this setting (in bytes) should not be logged, but replaced with `<long param suppressed>`. This is set in bytes, not characters, so does not account for multi-byte characters in a text parameters's encoding. This setting has no effect if `log_parameter` is `off`. If this setting is 0 (the default), all parameters are logged regardless of length

The default is `0`.

pgaudit.log_relation

Specifies whether session audit logging should create a separate log entry for each relation (`TABLE`, `VIEW`, etc.) referenced in a `SELECT` or `DML` statement. This is a useful shortcut for exhaustive logging without using object audit logging.

The default is `off`.

pgaudit.log_rows

Specifies that audit logging should include the number of rows retrieved or affected by a statement. When enabled the rows field will be included after the parameter field.

The default is `off`.

pgaudit.log_statement

Specifies whether logging will include the statement text and parameters (if enabled). Depending on requirements, an audit log might not require this and it makes the logs less verbose.

The default is `on`.

pgaudit.log_statement_once

Specifies whether logging will include the statement text and parameters with the first log entry for a statement/substatement combination or with every entry. Enabling this setting will result in less verbose logging but may make it more difficult to determine the statement that generated a log entry, though the statement/substatement pair along with the process id should suffice to identify the statement text logged with a previous entry.

The default is `off`.

pgaudit.role

Specifies the master role to use for object audit logging. Multiple audit roles can be defined by granting them to the master role. This allows multiple groups to be in charge of different aspects of audit logging.

There is no default.

Session Audit Logging

Session audit logging provides detailed logs of all statements executed by a user in the backend.

Configuration

Session logging is enabled with the pgaudit.log setting.

Enable session logging for all `DML` and `DDL` and log all relations in `DML` statements:

```
set pgaudit.log = 'write, ddl';
set pgaudit.log_relation = on;
```

Enable session logging for all commands except `MISC` and raise audit log messages as `NOTICE`:

```
set pgaudit.log = 'all, -misc';
set pgaudit.log_level = notice;
```

Example

In this example session audit logging is used for logging `DDL` and `SELECT` statements. Note that the insert statement is not logged since the `WRITE` class is not enabled

SQL:

```
set pgaudit.log = 'read, ddl';
```

```
create table account
(
    id int,
    name text,
    password text,
    description text
);

insert into account (id, name, password, description)
             values (1, 'user1', 'HASH1', 'blah, blah');

select *
    from account;
```

Log Output:

```
AUDIT: SESSION,1,1,DDL,CREATE TABLE,TABLE,public.account,create table account
(
    id int,
    name text,
    password text,
    description text
);,<not logged>
AUDIT: SESSION,2,1,READ,SELECT,,,select *
    from account,,<not logged>
```

Object Audit Logging

Object audit logging logs statements that affect a particular relation. Only `SELECT`, `INSERT`, `UPDATE` and `DELETE` commands are supported. `TRUNCATE` is not included in object audit logging.

Object audit logging is intended to be a finer-grained replacement for `pgaudit.log = 'read, write'`. As such, it may not make sense to use them in conjunction but one possible scenario would be to use session logging to capture each statement and then supplement that with object logging to get more detail about specific relations.

Configuration

Object-level audit logging is implemented via the roles system. The pgaudit.role setting defines the role that will be used for audit logging. A relation (`TABLE`, `VIEW`, etc.) will be audit logged when the audit role has permissions for the command executed or inherits the permissions from another role. This allows you to effectively have multiple audit roles even though there is a single master role in any context.

Set pgaudit.role to `auditor` and grant `SELECT` and `DELETE` privileges on the

account table. Any SELECT or DELETE statements on the account table will now be logged:

```
set pgaudit.role = 'auditor';

grant select, delete
   on public.account
   to auditor;
```

Example

In this example object audit logging is used to illustrate how a granular approach may be taken towards logging of SELECT and DML statements. Note that logging on the account table is controlled by column-level permissions, while logging on the account_role_map table is table-level.

SQL:

```
set pgaudit.role = 'auditor';

create table account
(
    id int,
    name text,
    password text,
    description text
);

grant select (password)
   on public.account
   to auditor;

select id, name
  from account;

select password
  from account;

grant update (name, password)
   on public.account
   to auditor;

update account
   set description = 'yada, yada';

update account
   set password = 'HASH2';
```

```
create table account_role_map
(
    account_id int,
    role_id int
);

grant select
   on public.account_role_map
   to auditor;

select account.password,
       account_role_map.role_id
  from account
       inner join account_role_map
            on account.id = account_role_map.account_id
```

Log Output:

```
AUDIT: OBJECT,1,1,READ,SELECT,TABLE,public.account,select password
  from account,<not logged>
AUDIT: OBJECT,2,1,WRITE,UPDATE,TABLE,public.account,update account
   set password = 'HASH2',<not logged>
AUDIT: OBJECT,3,1,READ,SELECT,TABLE,public.account,select account.password,
       account_role_map.role_id
  from account
       inner join account_role_map
            on account.id = account_role_map.account_id,<not logged>
AUDIT: OBJECT,3,1,READ,SELECT,TABLE,public.account_role_map,select account.password,
       account_role_map.role_id
  from account
       inner join account_role_map
            on account.id = account_role_map.account_id,<not logged>
```

Format

Audit entries are written to the standard logging facility and contain the following columns in comma-separated format. Output is compliant CSV format only if the log line prefix portion of each log entry is removed.

- **AUDIT_TYPE** - `SESSION` or `OBJECT`.
- **STATEMENT_ID** - Unique statement ID for this session. Each statement ID represents a backend call. Statement IDs are sequential even if some statements are not logged. There may be multiple entries for a statement ID when more than one relation is logged.
- **SUBSTATEMENT_ID** - Sequential ID for each sub-statement within

the main statement. For example, calling a function from a query. Sub-statement IDs are continuous even if some sub-statements are not logged. There may be multiple entries for a sub-statement ID when more than one relation is logged.

- **CLASS** - e.g. `READ`, `ROLE` (see pgaudit.log).
- **COMMAND** - e.g. `ALTER TABLE`, `SELECT`.
- **OBJECT_TYPE** - `TABLE`, `INDEX`, `VIEW`, etc. Available for `SELECT`, `DML` and most `DDL` statements.
- **OBJECT_NAME** - The fully-qualified object name (e.g. public.account). Available for `SELECT`, `DML` and most `DDL` statements.
- **STATEMENT** - Statement executed on the backend.
- **PARAMETER** - If `pgaudit.log_parameter` is set then this field will contain the statement parameters as quoted CSV or `<none>` if there are no parameters. Otherwise, the field is `<not logged>`.

Use log_line_prefix to add any other fields that are needed to satisfy your audit log requirements. A typical log line prefix might be `'%m %u %d [%p]: '` which would provide the date/time, user name, database name, and process id for each audit log.

Caveats

Audit logging is best-effort and not transactional. pgAudit writes audit entries through the standard PostgreSQL logging facility, which does not flush each entry to disk synchronously with the transaction that produced it, nor does it propagate write errors back to the session. There is no guarantee that a committed transaction will have a corresponding audit log entry. If the server crashes or loses power, or the log destination becomes unavailable (for example, the log volume fills up) after a transaction commits but before its audit entries are durably written, those entries may be lost. Conversely, a statement is logged when it executes, so an entry may be written even if its transaction later rolls back.

Object renames are logged under the name they were renamed to. For example, renaming a table will produce the following result:

```
ALTER TABLE test RENAME TO test2;

AUDIT: SESSION,36,1,DDL,ALTER TABLE,TABLE,public.test2,ALTER TABLE test RENAME TO test2,<not
```

It is possible to have a command logged more than once. For example, when a table is created with a primary key specified at creation time the index for the primary key will be logged independently and another audit log will be made for the index under the create entry. The multiple entries will however be contained within one statement ID.

Autovacuum and Autoanalyze are not logged.

Statements that are executed after a transaction enters an aborted state will not be audit logged. However, the statement that caused the error and any subsequent statements executed in the aborted transaction will be logged as ERRORs by the standard logging facility.

It is not possible to reliably audit superusers with pgAudit. One solution is to restrict access to superuser accounts and use the set_user extension to escalate permissions when required.

Authors

The PostgreSQL Audit Extension is based on the 2ndQuadrant pgaudit project authored by Simon Riggs, Abhijit Menon-Sen, and Ian Barwick and submitted as an extension to PostgreSQL core. Additional development has been done by David Steele of Crunchy Data.