UPGRADE

A practical guide

DALIBO

Feb. 2023

Contents

Anonymization & Data Masking for Postgres **13**
- Available on … 14
- Principles 14
- Quick Start 15
- Success Stories 16
- Support 16
- open a public issue: https://gitlab.com/dalibo/postgresql_anonymizer/issues 17

Backup Masking (aka Anonymous Dumps) **17**
- Transparent Anonymous Dumps 17
  1. Create a masked user 17
  2. Grant read access to that masked user 18
  3. Launch pg_dump with the masked user 18
- Anonymizing an SQL file 18
- Masking primary keys with Backup Masking 20
- DEPRECATED : pg_dump_anon.sh and pg_dump_anon 21

Definitions of the terms used in this project **21**

Configuration **22**
- anon.algorithm 22
- anon.restrict_to_trusted_schemas 22
- anon.salt 23
- anon.nosuperuser 23

Custom Fake Data **24**
- Alternative fake data packages 24
- Generate your own fake dataset 24
- Load your own fake data 24
- Using the PostgreSQL Faker extension 24
- Advanced Faking: masking_functions.md#advanced-faking 25

Put on your Masks ! **25**
Principles . . . 26
Escaping String literals . . . 26
Listing masking rules . . . 26
Debugging masking rules . . . 27
Removing a masking rule . . . 27
Custom Values for a Masking Policy . . . 27
Multiple Masking Policies . . . 28
Limitations . . . 29

Searching for Identifiers **29**
Limitations . . . 30
Contribute to the dictionaries . . . 30
[open an issue]: https://gitlab.com/dalibo/postgresql_anonymizer/-/issues . . . 30

Development Notes **30**

Hide sensitive data from a "masked" user **31**
Principles . . . 32
Generated columns . . . 32
Limitations . . . 32
How to unmask a role . . . 33
Transparent Dynamic Masking and Prepared Statements . . . 33

Masking Foreign Keys **33**
First of all : Avoid Natural Keys at all costs . . . 33
Example . . . 34
ON UPDATE CASCADE . . . 34
Pseudonymization . . . 35
Pseudonymization Is Not Anonymization: masking_functions.md#pseudonymization 35
Welcome to Paul's Boutique ! . . . 35
The Story . . . 35
Objectives . . . 36
About PostgreSQL Anonymizer . . . 36
About GDPR . . . 36
Requirements . . . 37
The Roles . . . 37
The Sample database . . . 38
Authors . . . 38
License . . . 38
Credits . . . 38

1 - Static Masking **38**
The story . . . 38
How it works . . . 39

Learning Objective . . . 39
The “customer” table . . . 39
The “payout” table . . . 39
Activate the extension . . . 40
Apply the rules permanently . . . 40
Exercises . . . 40
E101 - Mask the client’s first names . . . 40
E102 - Hide the last 3 digits of the postcode . . . 40
E103 - Count how many clients live in each postcode area? . . . 40
E104 - Keep only the year of each birth date . . . 41
E105 - Singling out a customer . . . 41
Solutions . . . 41
S101 . . . 41
S103 . . . 42
S104 . . . 42

2- How to use Dynamic Masking **43**
The Story . . . 43
How it works . . . 43
Objectives . . . 43
The “company” table . . . 43
The ”supplier” table . . . 44
Activate the extension . . . 44
Dynamic Masking . . . 44
Activate the masking engine . . . 44
Masking a role . . . 44
Masking the supplier names . . . 44
Exercises . . . 45
E201 - Guess who is the CEO of ”Johnny’s Shoe Store” . . . 45
E202 - Anonymize the companies . . . 45
E203 - Pseudonymize the company name . . . 45
Solutions . . . 46
S201 . . . 46
S202 . . . 46
S203 . . . 46
Now the fake company name is always the same. . . . 46

3- Anonymous Dumps **46**
The Story . . . 47
How it works . . . 47
Learning Objective . . . 47
Load the data . . . 47
Activate the extension . . . 47
Masking a JSON column . . . 48
Exercises . . . 49
E301 - Dump the anonymized data into a new database . . . 49

E302 - Pseudonymize the meta fields of the comments 49
Solutions 49
S301 49
S302 50

4 - Generalization **50**
The Story 50
How it works 50
Learning Objective 51
The ”employee” table 51
Data suppression 51
K-Anonymity 52
Range and Generalization functions 52
Declaring the indirect identifiers 53
Exercises 53
E401 - Simplify `v_staff_per_month` and decrease granularity . . 53
E402 - Staff progression over the years 53
E403 - Reaching 2-anonymity for the `v_staff_per_year` view . . 53
Solutions 53
S401 53
S402 54
S403 54

Conclusion **54**
Clean up ! 54
Many Masking Strategies 55
Many Masking Functions 55
Advantages 56
Drawbacks 56
Also… 56
Help Wanted! 56
This is a 4 hour workshop! 56
Questions? 56

PostgreSQL Anonymizer How To **56**
Write 57
Build 57
Type `make help` for more details 57

Import Export **57**
Principle 57
Compatibility of the import/export JSON schema between releases . . 57
Export 58
Import 60

Anonymization & Data Masking for Postgres **62**

Available on … 63
Quick Start 63
Success Stories 64
Support 65

Anonymization & Data Masking for Postgres 65
Available on … 65
Principles 65
Quick Start 66
Success Stories 67
Support 68
open a public issue: https://gitlab.com/dalibo/postgresql_anonymizer/issues 68

INSTALL 68
Install on RedHat / Rocky Linux / Alma Linux 69
Install on Fedora 69
Install on Debian / Ubuntu 70
Install on SUSE 70
Install with Ansible 71
Install With PGXN 71
Install From Source 71
Install with Docker 72
Install as a “Black Box” 73
Install With Django 73
Install on MacOS 73
Install on Windows 73
Install on PostgreSQL Forks 73
Install in the cloud 74
Addendum: Troubleshooting 74
Check that the extension is present 74
Check that the extension is loaded 74
Check that the extension is created 75
Check that the extension is initialized 75
Uninstall 75
Replace 17 by the version of your postgresql instance. 76

Ideas and Resources 76
Videos / Presentations 76
Similar Tools 76
Similar Implementations 76
GDPR 77
Concepts 77
Academic Research 77

Load the extension 77
Step 1: Load the extension 77

Alternative: `session_preload_libraries` . . . 78
Step 2: Create the extension . . . 78
All new connections to that database can now use the extension. . . . 78

Masking Data Wrappers **78**
Example . . . 78

Various Masking Strategies **80**
Destruction . . . 81
Adding Noise . . . 81
Local Differential Privacy (LDP) . . . 82
GRRM (Generalized Randomized Response Mechanism) . . . 82
Inspecting the privacy parameters . . . 82
Randomization . . . 82
Basic Random values . . . 83
Random between . . . 83
Random in Array . . . 83
Random in Enum . . . 83
Random in Range . . . 84
Random Sequence ID . . . 84
Faking . . . 85
Advanced Faking . . . 85
Pseudonymization . . . 88
OBSOLETE: pseudo_* functions . . . 89
Pseudonymizing primary keys . . . 89
Pseudonymization IS NOT Anonymization . . . 90
Generic hashing . . . 90
Partial Scrambling . . . 91
Conditional Masking . . . 91
Generalization . . . 92
Using `pg_catalog` functions . . . 93
Image bluring . . . 94
Write your own Masks ! . . . 94

Masking Views **97**
Generalization . . . 97
Example . . . 98
Generalization Functions . . . 99
Limitations . . . 99
Singling out and extreme values . . . 99
Generalization is not compatible with dynamic masking . . . 100
k-anonymity . . . 100
References . . . 100
How Google Anonymizes Data . . . 101

Performances **101**

Static Masking . . . 101
Dynamic Masking . . . 101
Anonymous Dumps . . . 102
How to speed things up ? . . . 102
Prefer `MASKED WITH VALUE` whenever possible . . . 102
Sampling . . . 102
Materialized Views . . . 102
Materialized Views: https://www.postgresql.org/docs/current/static/sql-creatematerializedview.html . . . 103

Privacy By Default **103**
Principle . . . 103
Example . . . 103
Unmasking columns . . . 104
Caveat: Add a DEFAULT to the NOT NULL columns . . . 104

Anonymous Replica **105**
Principle . . . 105
Preamble: Learn about logical replication ! . . . 105
Quick Setup . . . 106
Example . . . 106
A- On the publisher database . . . 106
B- On the subscriber database . . . 106
Changing the masking rules . . . 107
Anonymized Standby . . . 107
Security . . . 108
Limitations . . . 108
But I want to anonymize a primary key! . . . 109
Pseudonymization Is Not Anonymization: masking_functions.md#pseudonymization109

Welcome to Paul's Boutique ! **109**
The Story . . . 110
Objectives . . . 110
About GDPR . . . 110
Requirements . . . 110
The Roles . . . 111
The Sample database . . . 111

1- Static Masking **112**
Requirements . . . 112
The story . . . 112
How it works . . . 112
Learning Objective . . . 112
The "customer" table . . . 112
The "payout" table . . . 113
Activate the extension . . . 113

Declare the masking rules . . . 113
Exercises . . . 114
E101 - Mask the client's first names . . . 114
E102 - Hide the last 3 digits of the postcode . . . 114
E103 - Count how many clients live in each postcode area? . . . 114
E104 - Keep only the year of each birth date . . . 114
E105 - Singling out a customer . . . 114
Solutions . . . 115
S101 . . . 115
S103 . . . 115
S104 . . . 115

2- Dynamic Masking **116**
Requirements . . . 116
The Story . . . 116
How it works . . . 117
Objectives . . . 117
The `company` table . . . 117
The `supplier` table . . . 117
Activate the extension . . . 118
Dynamic Masking . . . 118
Activate the masking engine . . . 118
Masking a role . . . 118
Masking the supplier names . . . 118
Exercises . . . 118
E201 - Guess who is the CEO of "Johnny's Shoe Store" . . . 119
E202 - Anonymize the companies . . . 119
E203 - Pseudonymize the company name . . . 119
Solutions . . . 120
S201 . . . 120
S202 . . . 120
S203 . . . 120
Now the fake company name is always the same. . . . 120

3- Anonymous Dumps **120**
The Story . . . 121
How it works . . . 121
Learning Objective . . . 121
Load the data . . . 121
Activate the extension . . . 122
Masking a JSON column . . . 122
Exercises . . . 123
E301 - Dump the anonymized data into a new database . . . 123
E302 - Remove the email address . . . 123
E303 - Pseudonymize the IP address . . . 124
Solutions . . . 124

S301 . . . 124
S302 . . . 124
S303 . . . 125

4- Generalization **125**
The Story . . . 125
How it works . . . 125
Learning Objective . . . 125
The `employee` table . . . 126
Data suppression . . . 126
K-Anonymity . . . 127
Range and Generalization functions . . . 127
Declaring the indirect identifiers . . . 128
Exercises . . . 128
E401 - Simplify `v_staff_per_month` and decrease granularity . . . 128
E402 - Staff progression over the years . . . 128
E403 - Reaching 2-anonymity for the `v_staff_per_year` view . . . 128
Solutions . . . 129
S401 . . . 129
S402 . . . 129
S403 . . . 129

5- Anonymized Replica **129**
The Story . . . 130
How it works . . . 130
Learning Objective . . . 130
A note on this setup . . . 130
Prerequisites . . . 130
The source database . . . 131
Publication side: the replication role . . . 131
Replica side: create the database . . . 132
Replica side: initialize the extension and the tables . . . 132
Replica side: enable and subscribe . . . 133
Verification . . . 134
Beware of primary keys . . . 134
Exercises . . . 135
E501 - Add a masking rule after the fact . . . 135
E502 - A masked role on the replica? . . . 135
E503 - Verify the anonymized replication . . . 135
Solutions . . . 135
S501 . . . 135
S502 . . . 136
S503 . . . 136

Conclusion **136**
Clean up ! . . . 137

Also... 137
Help Wanted! 137

Sampling **137**
Principle 137
Sampling with TABLESAMPLE 138
Sampling with RLS policies 138
Maintaining Referential Integrity 139
Truncate Tables for the masked users 140

Security **140**
Permissions 140
Limit masking filters only to trusted schemas 141
Timing attacks in LDP functions 141
Security context of the functions 142
Hardening principles 142
Static masking good practices 143
Function with SQL standard body 143

Selective Masking (BETA) **144**
Principle 144
Example 144
Sampling: sampling.md 145

Permanently remove sensitive data **145**
Applying masking rules 145
Disabling Static Masking 147
Static Masking and Multiple Masking Policies 147
Parallel Static Masking 148
Faster masking by dropping indexes 149
Shuffling 150
Adding noise to a column 150

Support **150**
Supported versions 151
Compatibility 151
Support policy 151

Welcome to Paul's Boutique ! **151**
The Story 152
Objectives 152
About GDPR 152
Requirements 152
The Roles 153
The Sample database 153

1- Static Masking **154**

Requirements . . . 154
The story . . . 154
How it works . . . 154
Learning Objective . . . 154
The "customer" table . . . 154
The "payout" table . . . 155
Activate the extension . . . 155
Declare the masking rules . . . 155
Apply the rules permanently . . . 156
Exercises . . . 156
E101 - Mask the client's first names . . . 156
E102 - Hide the last 3 digits of the postcode . . . 156
E103 - Count how many clients live in each postcode area? . . . 156
E104 - Keep only the year of each birth date . . . 156
E105 - Singling out a customer . . . 157
Solutions . . . 157
S101 . . . 157
S102 . . . 158
S103 . . . 158
S104 . . . 158
S105 . . . 158

2- Dynamic Masking **159**
Requirements . . . 159
The Story . . . 160
How it works . . . 160
Objectives . . . 160
The `company` table . . . 160
The `supplier` table . . . 161
Activate the extension . . . 161
Dynamic Masking . . . 161
Activate the masking engine . . . 161
Masking a role . . . 161
Masking the supplier names . . . 162
Exercises . . . 162
E201 - Guess who is the CEO of "Johnny's Shoe Store" . . . 162
E202 - Anonymize the companies . . . 163
E203 - Pseudonymize the company name . . . 163
Solutions . . . 163
S201 . . . 163
S202 . . . 164
S203 . . . 164
Now the fake company name is always the same. . . . 165

3- Anonymous Dumps **165**
The Story . . . 165

How it works . . . 165
Learning Objective . . . 165
Load the data . . . 166
Activate the extension . . . 166
Masking a JSON column . . . 166
Exercises . . . 168
E301 - Dump the anonymized data into a new database . . . 168
E302 - Remove the email address . . . 168
E303 - Pseudonymize the IP address . . . 169
Solutions . . . 169
S301 . . . 169
S302 . . . 169
S303 . . . 170

4- Generalization **170**
The Story . . . 170
How it works . . . 171
Learning Objective . . . 171
The `employee` table . . . 171
Data suppression . . . 172
K-Anonymity . . . 172
Range and Generalization functions . . . 173
Declaring the indirect identifiers . . . 174
Exercises . . . 174
E401 - Simplify `v_staff_per_month` and decrease granularity . . . 174
E402 - Staff progression over the years . . . 174
E403 - Reaching 2-anonymity for the `v_staff_per_year` view . . . 175
Solutions . . . 175
S401 . . . 175
S402 . . . 175
S403 . . . 176

5- Anonymized Replica **176**
The Story . . . 176
How it works . . . 176
Learning Objective . . . 176
A note on this setup . . . 177
Prerequisites . . . 177
The source database . . . 177
Publication side: the replication role . . . 178
Publication side: create the publication . . . 178
Replica side: create the database . . . 179
Replica side: initialize the extension and the tables . . . 179
Replica side: masking rules . . . 180
Replica side: enable and subscribe . . . 180
Verification . . . 181

Beware of primary keys . . . 182
Exercises . . . 182
E501 - Add a masking rule after the fact . . . 182
E502 - A masked role on the replica? . . . 182
E503 - Verify the anonymized replication . . . 182
Solutions . . . 183
S501 . . . 183
S502 . . . 183
S503 . . . 184

Conclusion **184**
Clean up ! . . . 184
Also... . . . 185
Help Wanted! . . . 185

DO NOT MODIFY THESE FILES **185**

Upgrade **185**
Upgrade to version 3.3 and further versions . . . 185
Salt Rotation . . . 186
Subqueries in masking rules . . . 186
Inlining of functions . . . 186
Functions with SQL standard body and transparent dynamic masking . . . 186
Upgrade to version 3.2 and further versions . . . 186
Security Barrier . . . 186
New import / export format . . . 187
Upgrade to version 3.0 and further versions . . . 187
PostgreSQL 13 is not supported anymore . . . 187
Legacy Dynamic Masking is fully removed . . . 187
Breaking changes in internal catalogs . . . 187
Upgrade to version 2.0 and further versions . . . 187
Upgrade to version 1.3 and further versions . . . 188
Using custom masking functions . . . 188
Using `pg_catalog` functions . . . 188
Operators . . . 189
Conditional masking rules . . . 189

Anonymization & Data Masking for Postgres

`PostgreSQL Anonymizer` is an extension to mask or replace personally identifiable information (PII) or commercially sensitive data from a Postgres database.



Figure 1: PostgreSQL Anonymizer

Available on ...

PostgreSQL Anonymizer is available on multiple linux distributions, cloud service providers, development frameworks and PostgreSQL forks:

See the INSTALL section for more details.

Principles

The project has a **declarative approach** of anonymization. This means you can declare the masking rules using the PostgreSQL Data Definition Language (DDL) and specify your anonymization policy inside the table definition itself.

The main goal of this extension is to offer **anonymization by design**. We firmly believe that data masking rules should be written by the people who develop the application because they have the best knowledge of how the data model works. Therefore masking rules must be implemented directly inside the database schema.

Once the masking rules are defined, you can apply them using 6 different **masking methods** :

-
-
-
-
-
-

Each method has its pros and cons. Different masking methods may be used in different contexts. In any case, masking the data directly inside the PostgreSQL instance without using an external tool is crucial to limit the exposure and the risks of data leak.

In addition, various Masking Functions are available : randomization, faking, partial scrambling, shuffling, noise, generalization or even your own custom function!

Finally, the extension offers a panel of detection functions that will try to guess which columns need to be anonymized.

Quick Start

Step 0. Launch docker image of the project

```
ANON_IMG=registry.gitlab.com/dalibo/postgresql_anonymizer
docker run --name anon_quickstart --detach -e POSTGRES_PASSWORD=x $ANON_IMG
docker exec -it anon_quickstart psql -U postgres
```

Step 1. Create a database and load the extension in it

```
CREATE DATABASE demo;
ALTER DATABASE demo SET session_preload_libraries = 'anon'

\connect demo
You are now connected to database "demo" as user "postgres".
```

Step 2. Create a table

```
CREATE TABLE people AS
    SELECT  153478       AS id,
            'Sarah'      AS firstname,
            'Conor'      AS lastname,
            '0609110911' AS phone
;

SELECT * FROM people;
   id    | firstname | lastname |   phone
---------+-----------+----------+------------
 153478 | Sarah     | Conor    | 0609110911
```

Step 3. Create the extension and activate the masking engine

```
CREATE EXTENSION anon;
ALTER DATABASE demo SET anon.transparent_dynamic_masking TO true;
```

Step 4. Declare a masked user

```
CREATE ROLE skynet LOGIN;

SECURITY LABEL FOR anon ON ROLE skynet IS 'MASKED';

GRANT pg_read_all_data to skynet;
```

Step 5. Declare the masking rules

```
SECURITY LABEL FOR anon ON COLUMN people.lastname
  IS 'MASKED WITH FUNCTION anon.dummy_last_name()';
```

```
SECURITY LABEL FOR anon ON COLUMN people.phone
  IS 'MASKED WITH FUNCTION anon.partial(phone,2,$$******$$,2)';
```

Step 6. Connect with the masked user

```
\connect - skynet
You are now connected to database "demo" as user "skynet"

SELECT * FROM people;
   id   | firstname | lastname  |   phone
--------+-----------+-----------+------------
 153478 | Sarah     | Stranahan | 06******11
```

Success Stories

> With PostgreSQL Anonymizer we integrate, from the design of the database, the principle that outside production the data must be anonymized. Thus we can reinforce the GDPR rules, without affecting the quality of the tests during version upgrades for example.

— **Thierry Aimé, Office of Architecture and Standards in the French Public Finances Directorate General (DGFiP)**

> Thanks to PostgreSQL Anonymizer we were able to define complex masking rules in order to implement full pseudonymization of our databases without losing functionality. Testing on realistic data while guaranteeing the confidentiality of patient data is a key point to improve the robustness of our functionalities and the quality of our customer service.

— **Julien Biaggi, Product Owner at bioMérieux**

> I just discovered your postgresql_anonymizer extension and used it at my company for anonymizing our user for local development. Nice work!

— **Max Metcalfe**

If this extension is useful to you, please let us know !

Support

We need your feedback and ideas ! Let us know what you think of this tool, how it fits your needs and what features are missing.

Want to talk directly with us ? Join us on Matrix or Discord

- Matrix: https://matrix.to/#/#anon:dalibo.com

- Discord: https://discord.com/channels/710918545906597938/1427672533104070807

You can also open a public issue on gitlab

If you need enterprise support or if you want to sponsor the project, send us a message at contact@dalibo.com.

open a public issue: https://gitlab.com/dalibo/postgresql_anonymizer/issues

title: anonymous_dumps draft: false toc: true —

Backup Masking (aka Anonymous Dumps)

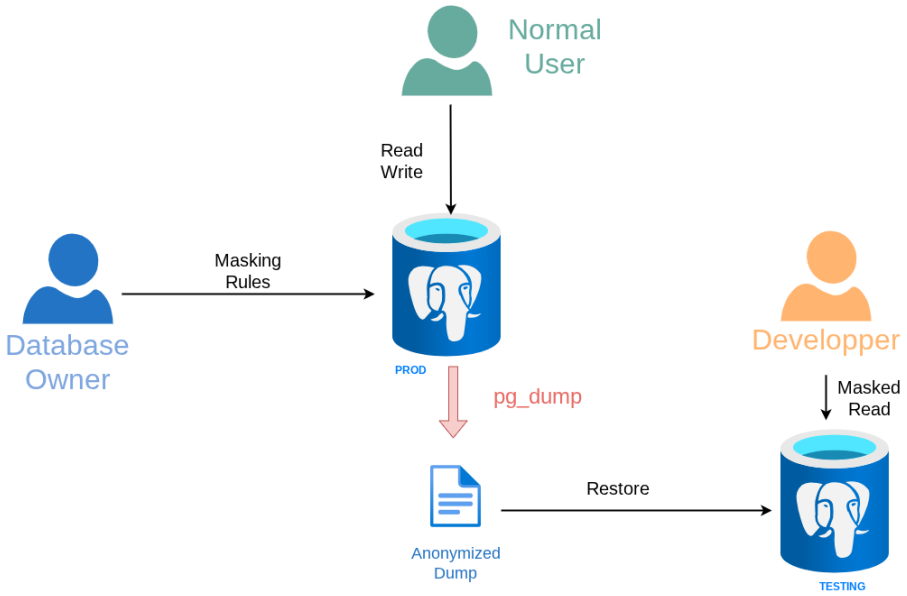


Figure 2: PostgreSQL Anonymous Dumps

Transparent Anonymous Dumps

To export the anonymized data from a database, follow these 3 steps:

1. Create a masked user

```
CREATE ROLE anon_dumper LOGIN PASSWORD 'x';
ALTER ROLE anon_dumper SET anon.transparent_dynamic_masking = True;
SECURITY LABEL FOR anon ON ROLE anon_dumper IS 'MASKED';
```

NOTE: You can replace the name `anon_dumper` by another name.

2. Grant read access to that masked user

```
GRANT pg_read_all_data to anon_dumper;
```

NOTE: If you want a more fine-grained access policy you can grant access more precisely, for instance:

```
GRANT USAGE ON SCHEMA public TO anon_dumper;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO anon_dumper;
GRANT SELECT ON ALL SEQUENCES IN SCHEMA public TO anon_dumper;

GRANT USAGE ON SCHEMA foo TO anon_dumper;
GRANT SELECT ON ALL TABLES IN SCHEMA foo TO anon_dumper;
GRANT SELECT ON ALL SEQUENCES IN SCHEMA foo TO anon_dumper;
```

3. Launch pg_dump with the masked user

Now to export the anonymous data from a database named `foo`, let's use `pg_dump`:

```
pg_dump foo \
        --user anon_dumper \
        --no-security-labels \
        --exclude-extension="anon" \
        --file=foo_anonymized.sql
```

NOTES:

- linebreaks are here for readability
- `--no-security-labels` will remove the masking rules from the anonymous dump. This is really important because masked users should not have access to the masking policy.
- `--exclude-extension` is only available with `pg_dump 17` and later. As an alternative you can use `--extension plpgsql`.
- `--format=custom` is supported

Anonymizing an SQL file

> In previous versions of the documentation, this method was also called « anonymizing black box ».

You can also apply masking rules directly on a database backup file !

The PostgreSQL Anonymizer docker image contains a specific entrypoint script called `/dump.sh`. You pass the original data and the masking rules to to that `/dump.sh` script and it will return an anonymized dump.

Here's an example in 4 steps:

Step 1: Dump your original data (for instance `dump.sql`)



Figure 3: Anon Files image

```
pg_dump --format=plain [...] my_db > dump.sql
```

Note this method only works with plain sql format (`-Fp`). You **cannot** use the custom format (`-Fc`) and the directory format (`-Fd`) here.

If you want to maintain the owners and grants, you need export them with `pg_dumpall --roles-only` like this:

```
(pg_dumpall -Fp [...] --roles-only && pg_dump -Fp [...] my_db ) > dump.sql
```

Step 2: Write your masking rules in a separate file (for instance `rules.sql`)

```
RESET search_path;

SECURITY LABEL FOR anon ON COLUMN people.lastname
  IS 'MASKED WITH FUNCTION anon.dummy_last_name()';

-- etc.
```

Step 3: Pass the dump and the rules through the docker image and receive an anonymized dump !

```
IMG=registry.gitlab.com/dalibo/postgresql_anonymizer:stable
ANON="docker run --rm --interactive $IMG /dump.sh"
cat dump.sql rules.sql | $ANON > anon_dump.sql
```

(this last step is written on 3 lines for clarity)

NB: You can also gather *step 1* and *step 3* in a single command:

```
(pg_dumpall --roles-only && pg_dump my_db && cat rules.sql) | $ANON > anon_dump.sql
```

NOTES:

You can use most the pg_dump output options with the `/dump.sh` script, for instance:

```
cat dump.sql rules.sql | $ANON --data-only --inserts > anon_dump.sql
```

The `RESET search_path` command in `rules.sql` is required because pg_dump will disable the search_path in `dump.sql` for security reasons. Alternatively you can use fully-qualified column names in `rules.sql`, for instance `public.people.lastname` instead of `people.lastname`.

Masking primary keys with Backup Masking

Primary keys (such as `SERIAL`) are often masked with the `anon.random_id()` function which will generate a unique random identifier every it is called.

However this function will not work with Backup Masking because `pg_dump` will * connect in read-only mode to the database (`default_transaction_read_only=on;`) and the `anon.random_id()` function needs to update a sequence to avoid generating the same value twice.

See issue #529 for more details:

https://gitlab.com/dalibo/postgresql_anonymizer/-/issues/529

Therefore if you use 'anon.random_id() in some rules, the backup masking process will throw the following error :

```
pg_dump: detail: Error message from server:
ERROR:  permission denied for sequence random_id_seq
```

The solution is to rewrite the masking rules based on `anon.random_id()` and use `anon.pseudo_shift(BIGINT)` or `anon.pseudo_xor(BIGINT)` instead.

For instance the masking rule below:

```
SECURITY LABEL FOR anon ON COLUMN people.id
  IS 'MASKED WITH FUNCTION anon.random_id()';
```

would become

```
SECURITY LABEL FOR anon ON COLUMN people.id
  IS 'MASKED WITH FUNCTION anon.pseudo_xor(id)';
```

The `anon.pseudo_shift(BIGINT)` and `anon.pseudo_xor(BIGINT)` functions use a secret value (`anon.shift`) to pseudonymize the primary key. The secret value can be initialized randomly with `anon.set_shift()` or defined with `anon.set_shift(INT)`.

WARNING: Remember that Pseudonymization is not Anonymization !

DEPRECATED : pg_dump_anon.sh and pg_dump_anon

In version 0.x, the anonymous dumps were done with a shell script named `pg_dump_anon.sh`. In version 1.x it was done with a golang script named `pg_dump_anon`. **Both commands are now deprecated.** — title: concepts draft: false toc: true —

Definitions of the terms used in this project

Two main strategies are used:

- **Dynamic Masking** offers an altered view of the real data without modifying it. Some users may only read the masked data, others may access the authentic version.
- **Permanent Destruction** is the definitive action of substituting the sensitive information with uncorrelated data. Once processed, the authentic data cannot be retrieved.

The data can be altered with several techniques:

- **Deletion** or **Nullification** simply removes data.
- **Static Substitution** consistently replaces the data with a generic value. For instance: replacing all values of a TEXT column with the value "CONFIDENTIAL".
- **Variance** is the action of "shifting" dates and numeric values. For example, by applying a +/- 10% variance to a salary column, the dataset will remain meaningful.
- **Generalization** reduces the accuracy of the data by replacing it with a range of values. Instead of saying "Bob is 28 years old", you can say "Bob is between 20 and 30 years old". This is useful for analytics because the data remains true.
- **Shuffling** mixes values within the same columns. This method is open to being reversed if the shuffling algorithm can be deciphered.
- **Randomization** replaces sensitive data with **random-but-plausible** values. The goal is to avoid any identification from the data record while remaining suitable for testing, data analysis and data processing.
- **Partial scrambling** is similar to static substitution but leaves out some part of the data. For instance : a credit card number can be replaced by '40XX XXXX XXXX XX96'
- **Custom rules** are designed to alter data following specific needs. For instance, randomizing simultaneously a zipcode and a city name while keeping them coherent.

- **Pseudonymization** is a way to **protect** personal information by hiding it using additional information. **Encryption** and **Hashing** are two examples of pseudonymization techniques. However a pseudonymizated data is still linked to the original data.
- **Local Differential Privacy (LDP)** perturbs each data point individually before collection, so that no raw value is ever exposed to anyone. The amount of perturbation is controlled by a privacy parameter called **epsilon**: a smaller epsilon gives stronger privacy guarantees at the cost of accuracy.
- **GRRM** (Generalized Randomized Response Mechanism) is an LDP method for categorical values. Each value is either reported truthfully or replaced with a random alternative. The probability of keeping the true value depends on epsilon and the total number of possible categories. — title: configure draft: false toc: true —

Configuration

The extension has currently a few options that be defined for the entire instance (inside `postgresql.conf` or with `ALTER SYSTEM`).

It is also possible and often a good idea to define them at the database level like this:

```
ALTER DATABASE customers SET anon.restrict_to_trusted_schemas = on;
```

Only superuser can change the parameters below :

anon.algorithm

Type	Text
Default value	'sha256'
Visible	only to superusers

This is the hashing method used by pseudonymizing functions. Checkout the pgcrypto documentation for the list of available options.

See `anon.salt` to learn why this parameter is a very sensitive information.

anon.restrict_to_trusted_schemas

Type	Boolean
Default value	off
Visible	to all users

By enabling this parameter, masking rules must be defined using functions located in a limited list of namespaces. By default, only the `anon` schema is trusted.

This improves security by preventing users from declaring their custom masking filters. This also means that the schema must be explicit inside the masking rules.

For more details, check out the Write your own masks section of the Masking functions chapter.

anon.salt

Type	Text
Default value	(empty)
Visible	only to superusers

This is the salt used by pseudonymizing functions. It is very important to define a custom salt for each database like this:

```
ALTER DATABASE foo SET anon.salt = 'This_Is_A_Very_Secret_Salt';
```

If a masked user can read the salt, he/she can run a brute force attack to retrieve the original data based on the 3 elements:

- The pseudonymized data
- The hashing algorithm (see `anon.algorithm`)
- The salt

The GDPR considered that the salt and the name of the hashing algorithm should be protected with the same level of security that the data itself. This is why you should store the salt directly within the database with `ALTER DATABASE`.

anon.nosuperuser

Type	Boolean
Default value	'on'
Visible	only to superusers

By enabling this parameter, masking rules can be applyed only by users without the `SUPERUSER` privilege. — title: custom_fake_data draft: false toc: true —

Custom Fake Data

This extension is delivered with a small set of fake data by default. For each fake function (`fake_email()`, `fake_first_name()`) we provide only 1000 unique values, and they are only in English.

The data is stored in dedicated tables inside the `anon` schema.

Here's how you can create your own set of fake data!

Alternative fake data packages

The project is offering alternative fake datasets (currently only French). You can download the zip file containing the dataset and load it into the extension like this:

1. Go to https://gitlab.com/dalibo/postgresql_anonymizer/-/packages
2. Click on "data"
3. Choose your preferred zip file and download it on your server
4. Unzip the file into a folder (for example `/path/to/custom_csv_files/`)
5. Run `SELECT anon.init('/path/to/custom_csv_files/')`

Generate your own fake dataset

Just add data inside the fake data tables.

For instance, You can generate fake data using the `dummy_*` functions:

```
INSERT INTO anon.email(val,locale)
SELECT anon.dummy_email('fr_FR'), 'fr_FR';

SELECT setval('anon.email_oid_seq', max(oid))
FROM anon.email;

CLUSTER anon.email;
```

Load your own fake data

If you want to use your own dataset, you can import custom CSV files with :

```
SELECT anon.init('/path/to/custom_csv_files/')
```

Look at the `data` folder to find the format of the CSV files.

Using the PostgreSQL Faker extension

If you need more specialized fake data sets, please read the Advanced Faking section.

Advanced Faking: masking_functions.md#advanced-faking

title: datamodel draft: false toc: true —

```
classDiagram

  class identifier_category{
        INTEGER id,
        TEXT name
        BOOL direct_identifier
        TEXT anon_function
  }

  class field_name{
        TEXT attname
        TEXT lang
        INTEGER fk_identifiers_category
  }

  field_name "1..N" --> "1" identifier_category
```

Put on your Masks !

The main idea of this extension is to implement the concept of **Privacy by Design**, which is principle imposed by the Article 25 of the GDPR.

With PostgreSQL Anonymizer, you can declare a **masking policy** which is a set of **masking rules** stored inside the database model and applied to various database objects.

The data masking rules should be written by the people who develop the application because they have the best knowledge of how the data model works. Therefore masking rules must be implemented directly inside the database schema.

This allows to mask the data directly inside the PostgreSQL instance without using an external tool and thus limiting the exposure and the risks of data leak.

The data masking rules are declared simply by using security labels:

When designing your masking policy, follow the *Principle of Least Privilege* (PoLP) and avoid using superuser accounts to execute masking tasks or dynamic masking operations. Instead, use dedicated roles with only the permissions required to perform these operations.

For more information on applying the *Principle of Least Privilege* to masking and other related topics, please refer to our **Security** page.

```
CREATE TABLE player( id SERIAL, name TEXT, total_points INT, highest_score INT);
```

```
INSERT INTO player VALUES
  ( 1, 'Kareem Abdul-Jabbar', 38387, 55),
  ( 5, 'Michael Jordan', 32292, 69);

SECURITY LABEL FOR anon ON COLUMN player.name
  IS 'MASKED WITH FUNCTION anon.fake_last_name()';

SECURITY LABEL FOR anon ON COLUMN player.id
  IS 'MASKED WITH VALUE NULL';
```

Principles

- You can mask tables in multiple schemas
- Generated columns are respected.
- Row Security Policies aka `RLS` are respected.
- A masking rule may break data integrity. For instance, you can mask a `NOT NULL` column with the value `NULL`. This is up to you to decide whether or not the masked users need data integrity.
- You need to declare masking rules on views. By default, the masking rules declared on the underlying tables are **NOT APPLIED** on the view. For instance, if a view `v_foo` is based upon a table `foo`, then the masking rules of table `foo` will not be applied to `v_foo`. You will need to declare specific masking rules for `v_foo`. Remember that PostgreSQL uses the view owner (not the current user) to check permissions on the underlying tables.

Escaping String literals

As you may have noticed the masking rule definitions are placed between single quotes. Therefore if you need to use a string inside a masking rule, you need to use C-Style escapes like this:

```
SECURITY LABEL FOR anon ON COLUMN player.name
  IS E'MASKED WITH VALUE \'CONFIDENTIAL\'';
```

Or use dollar quoting which is easier to read:

```
SECURITY LABEL FOR anon ON COLUMN player.name
  IS 'MASKED WITH VALUE $$CONFIDENTIAL$$';
```

Listing masking rules

To display all the masking rules declared in the current database, check out the `anon.{all|sys|user}_rules` views:

```
SELECT * FROM anon.all_rules;
SELECT * FROM anon.sys_rules;
SELECT * FROM anon.user_rules;
```

Debugging masking rules

When an error occurs to due a wrong masking rule, you can get more detailed information about the problem by setting `client_min_messages` to `DEBUG` and you will get useful details

```
postgres=# SET client_min_messages=DEBUG;
SET
postgres=# SELECT anon.anonymize_database();
DEBUG:  Anonymize table public.bar with firstname = anon.fake_first_name()
DEBUG:  Anonymize table public.foo with id = NULL
ERROR:  Cannot mask a "NOT NULL" column with a NULL value
HINT:  If privacy_by_design is enabled, add a default value to the column
CONTEXT:  PL/pgSQL function anon.anonymize_table(regclass) line 47 at RAISE
SQL function "anonymize_database" statement 1
```

Removing a masking rule

You can simply erase a masking rule like this:

```
SECURITY LABEL FOR anon ON COLUMN player.name IS NULL;
```

To remove all rules at once, you can use:

```
SELECT anon.remove_masks_for_all_columns();
```

Custom Values for a Masking Policy

In some case, you may have to apply variations to masking policy based on a set of parameters. In other words, custom values in PostgreSQL Anonymizer allow you to parameterize masking rules so that the same masking policy can behave differently based on context - such as the country, environment, or customer profile.

Define the custom values as a JSON dictionary:

```
SET anon.custom_values
TO '{ "country": "France", "locale": "fr_FR" }';
```

You can then use the `anon.custom_value()` function in your masking rule Alternavely there's also a `anon.custom_value(key,default)` variant.

```
SECURITY LABEL FOR anon ON COLUMN player.name
IS 'MASKED WITH FUNCTION anon.fake_last_name_locale(anon.custom_value('locale'))';
```

Custom values defined this way are returned as `text`, depending on the context, you might have to use the appropriate cast to remain compatible with some functions and assignments or keep your predicate indexable.

Finally you can define the custom_values on a per-database or on a per-user basis.

```
ALTER DATABASE french_stores
SET anon.custom_values
TO '{ "country": "France", "locale": "fr_FR" }';

ALTER DATABASE italian_stores
SET anon.custom_values
TO '{ "country": "Italia", "locale": "it_IT" }';
```

Multiple Masking Policies

By default, there is only one masking policy named 'anon'. Most of the times, a single policy is enough. However in more complex situations, the database owner may want to define different sets of masking rules for different use cases.

This can be achieved by declaring multiple masking policies.

For instance, we can add 2 new policies with:

```
ALTER DATABASE foo SET anon.masking_policies TO 'devtests, analytics';
```

> Important: You need to reconnect to the database so that the change takes effect !

We can now define a "devtests" policy for a developer name "devin". Devin wants to run CI tests on his code using fake/random data.

```
SECURITY LABEL FOR devtests ON COLUMN player.name
  IS 'MASKED WITH FUNCTION anon.fake_last_name()';

SECURITY LABEL FOR devtests ON COLUMN player.highest_score
  IS 'MASKED WITH FUNCTION anon.random_int_between(0,50)';

SECURITY LABEL FOR devtests ON ROLE devin IS 'MASKED';
```

We can also define an "analytics" for a data scientist name "Anna". Anna needs to run global stats over the dataset, she want to keep the real value on the `highest_score` column but she does not need to know the players names

```
SECURITY LABEL FOR analytics ON COLUMN player.name
  IS 'MASKED WITH VALUE NULL';

SECURITY LABEL FOR analytics ON ROLE anna IS 'MASKED';
```

Only one policy can be applied to a role. If you define that a role is masked in several masking policies, only the first one in the list will be applied.

The "anon" policy is always declared and cannot be removed.

If you declare a function as `TRUSTED`, it will be trusted for all masking policies.

Limitations

- The masking rules are **NOT INHERITED** ! If you have split a table into multiple partitions, you need to declare the masking rules for each partition.
- Masking identity columns is tricky. If an identity column is defined as `GENERATED ALWAYS`, then static masking will not work on that column. Note identity columns are used most of the time for surrogate keys (also known as "factless keys") and in general those keys should not required to be masked. However if you really need to mask and identity column you can redefine it as `GENERATED DEFAULT`. — title: detection draft: false toc: true —

Searching for Identifiers

> WARNING : This feature is at an early stage of development.

As we've seen previously, this extension makes it very easy to declare masking rules.

However, when you create an anonymization strategy, the hard part is scanning the database model to find which columns contains direct and indirect identifiers, and then decide how these identifiers should be masked.

The extension provides a `detect()` function that will search for common identifier names based on a dictionary. For now, 2 dictionaries are available: english ('en_US') and french ('fr_FR'). By default, the english dictionary is used:

```
# SELECT anon.detect('en_US');
 table_name |  column_name   | identifiers_category | direct
------------+----------------+----------------------+--------
 customer   | CreditCard     | creditcard           | t
 vendor     | Firstname      | firstname            | t
 customer   | firstname      | firstname            | t
 customer   | id             | account_id           | t
```

The identifier categories are based on the HIPAA classification.

Limitations

This is an heuristic method in the sense that it may report useful information, but it is based on a pragmatic approach that can lead to detection mistakes, especially:

- `false positive`: a column is reported as an identifier, but it is not.
- `false negative`: a column contains identifiers, but it is not reported

The second one is of course more problematic. In any case, you should only consider this function as a helping tool, and acknowledge that you still need to review the entire database model in search of hidden identifiers.

Contribute to the dictionaries

This detection tool is based on dictionaries of identifiers. Currently these dictionaries contain only a few entries.

For instance, you can see the english identifier dictionary in file `data/identifier_en_US.csv`.

You can help us improve this feature by sending us a list of direct and indirect identifiers you have found in your own data models ! Send us an email at contact@dalibo.com or [open an issue] in the project.

[open an issue]: https://gitlab.com/dalibo/postgresql_anonymizer/-/issues

title: dev/README draft: false toc: true —

Development Notes

This folders contains weird ideas, failed tests and dodgy dead ends.

We use jupyter to write these notebooks. Most of them are probably outdated.

Here's how you can install jupyter:

```
pip3 install --upgrade pip
pip3 install --r docs/dev/requirements
export PATH=$PATH:~/.local/bin
```

And then launch jupyter:

```
jupyter notebook
# or
jupyter notebook --no-browser --port 9999
```

Or convert the notebooks

```
jupyter nbconvert docs/dev/*.ipynb --to markdown
```

Hide sensitive data from a “masked” user

You can hide some data from a role by declaring this role as “MASKED”.

Other roles will still access the original data.

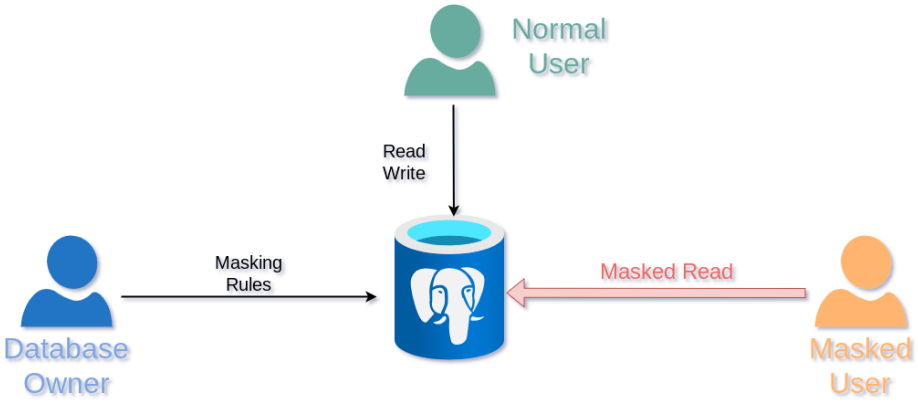


Figure 4: PostgreSQL Dynamic Masking

Example:

```
CREATE TABLE people ( id TEXT, firstname TEXT, lastname TEXT, phone TEXT);
INSERT INTO people VALUES ('T1','Sarah', 'Conor','0609110911');
SELECT * FROM people;
```

```
=# SELECT * FROM people;
 id | firstname | lastname |   phone
----+-----------+----------+------------
 T1 | Sarah     | Conor    | 0609110911
(1 row)
```

Step 1 : Activate the dynamic masking engine

```
=# CREATE EXTENSION IF NOT EXISTS anon CASCADE;
=# ALTER DATABASE foo SET anon.transparent_dynamic_masking TO true;
```

Step 2 : Declare the masking rules

```
SECURITY LABEL FOR anon ON COLUMN people.name
IS 'MASKED WITH FUNCTION anon.dummy_last_name()';

SECURITY LABEL FOR anon ON COLUMN people.phone
IS 'MASKED WITH FUNCTION anon.partial(phone,2,$$******$$,2)';
```

Step 3 : Declare a masked user with read access

```
=# CREATE ROLE skynet LOGIN;
=# SECURITY LABEL FOR anon ON ROLE skynet IS 'MASKED';

GRANT pg_read_all_data to skynet;
```

NOTE: If you want a more fine-grained access policy you can grant access more precisely, for instance:

```
GRANT USAGE ON SCHEMA public TO skynet;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO skynet;
-- etc.
```

Step 4 : Connect with the masked user

```
=# \c - skynet
=> SELECT * FROM people;
 id | firstname | lastname  |   phone
----+-----------+-----------+------------
 T1 | Sarah     | Stranahan | 06******11
(1 row)
```

Principles

- Masked roles should not be allowed to insert, update or delete data.
- You can mask table in multiple schemas.
- Generated columns are respected.
- You can apply Row Security Policies aka `RLS` to a masked role.
- A masking rule may break data integrity. For instance, you can mask a column having a UNIQUE constraint with the value `NULL`. This is up to you to decide whether or not the mask users need data integrity.

Generated columns

Generated columns are recomputed from the masked values of their source columns by default. To return a generated column's original stored value instead, label it as `NOT MASKED` in the active masking policy:

```
SECURITY LABEL FOR anon ON COLUMN users.email_hash IS 'NOT MASKED';
```

This keeps `email_hash` unchanged while `email` remains dynamically masked. Removing the label restores recomputation from masked values.

Limitations

Masked roles are not allowed to :

- use `EXPLAIN`
- define `SECURITY LABELS`
- use cached plans prepared by another role (see below)
- read views defined as `SECURITY INVOKER`

How to unmask a role

Simply remove the security label like this:

```
SECURITY LABEL FOR anon ON ROLE bob IS NULL;
```

Transparent Dynamic Masking and Prepared Statements

Transparent dynamic masking works by rewriting a query when PostgreSQL analyzes it. Prepared statements (`PREPARE`), PL/pgSQL functions and other cached plans are analyzed once and then reused, so the masking decision must be re-evaluated whenever it could change.

To guarantee this, the extension marks every query analyzed on the current role, in the same way PostgreSQL does for Row Level Security.

As a result, a cached plan is analyzed again when:

- the current role changes (for instance with `SET ROLE`), so a plan prepared by an unmasked role is never reused by a masked role;
- a masking rule is declared, changed or removed, or a role is masked or unmasked with `SECURITY LABEL`, so plans prepared before the change are rebuilt with the new rules.

These steps ensure that prepared statements and PL/pgSQL functions behave the same way as regular queries when run by a masked user.

The trade-off is that a new execution plan must be regenerated, which may be less performant, each time a cached plan is reused by a different role in the session or the masking rules change. The impact is usually small, but applications that frequently switch roles and rely heavily on prepared statements may notice it. — title: foreign_keys draft: false toc: true —

Masking Foreign Keys

TLDR; Masking a foreign key is hard. There is no silver buller but this chapter describes multiple strategies you can apply depending on your context.

First of all : Avoid Natural Keys at all costs

This is the best advice you will find here.

A primary or a foreign key should not have any meaning outside of the database itself. Therefore you should not have to anonymize it.

If you really need to anonymize a primary/foreign key in a table, this means that it is a natural key (as opposed to a surrogate key).

Natural keys are problematic for many reasons:

- they can change over time (like email addresses or product codes), forcing cascading updates throughout related tables
- they're often not truly unique in practice, even seemingly unique values like SSNs can have duplicates or exceptions
- they tend to be longer and more complex than simple integers
- they make joins slower and indexes larger
- they can contain sensitive information that you might not want exposed in URLs or logs.
- they may change whenever business rules evolve, requiring database restructuring.

Surrogate keys (i.e. auto-incrementing integers) avoid these issues by providing stable, meaningless identifiers that never need to change.

If you have implemented surrogate keys, you don't need to read this chapter any further ;-)

Example

Let's consider this (dumb) example

```
CREATE TABLE author (
  id SERIAL UNIQUE,
  name TEXT
);

CREATE TABLE book (
  id SERIAL UNIQUE,
  name TEXT,
  fk_author_id INTEGER,
  FOREIGN KEY (fk_author_id) REFERENCES author(id)
);
```

ON UPDATE CASCADE

For static masking, the best solution is add the `ON UPDATE CASCADE` action to the foreign key :

```
ALTER TABLE book DROP CONSTRAINT book_fk_author_name_fkey;

ALTER TABLE book
ADD CONSTRAINT book_fk_author_name_fkey
    FOREIGN KEY (fk_author_name)
    REFERENCES author(name)
    ON UPDATE CASCADE
;
```

And then define a masking rule only for the referenced column

```
SECURITY LABEL FOR anon ON COLUMN author.name
  IS 'MASKED WITH FUNCTION anon.dummy_name()';
```

Now whenever the `author` table will be anonymized, the `book.fk_author_name` column will be altered automatically. Therefore you should not declare a masking rule on `book.fk_author_name` !

Pseudonymization

However the `ON UPDATE CASCADE` action is only triggered when the real data is actually altered, so it will not work for dynamic masking, backup masking and replica masking.

In these situations, the best approach is to use a pseudonymization function

First of all declared that the foerign key is deferrable:

```
ALTER TABLE book DROP CONSTRAINT book_fk_author_name_fkey;

ALTER TABLE book
ADD CONSTRAINT book_fk_author_name_fkey
    FOREIGN KEY (fk_author_name)
    REFERENCES author(name)
    DEFERRABLE
;
```

Then declare a similar masking rule for each column:

```
SECURITY LABEL FOR anon ON COLUMN author.name
  IS 'MASKED WITH FUNCTION pg_catalog.md5(name)';

SECURITY LABEL FOR anon ON COLUMN book.fk_author_name
  IS 'MASKED WITH FUNCTION pg_catalog.md5(fk_author_name)';
```

However keep in mind that that Pseudonymization Is Not Anonymization !

Pseudonymization Is Not Anonymization: masking_functions.md#pseudonymizat

title: how-to/0-masking_data_with_postgresql_anonymizer draft: false toc: true —

Welcome to Paul's Boutique !

This is a 4 hours workshop that demonstrates various anonymization techniques using the PostgreSQL Anonymizer extension.

The Story

Paul's boutique

Paul's boutique has a lot of customers. Paul asks his friend Pierre, a Data Scientist, to make some statistics about his clients : average age, etc…

Pierre wants a direct access to the database in order to write SQL queries.

Jack is an employee of Paul. He's in charge of relationship with the various suppliers of the shop.

Paul respects his suppliers privacy. He needs to hide the personal information to Pierre, but Jack needs read and write access the real data.

Objectives

Using the simple example above, we will learn:

- How to write masking rules
- The difference between static and dynamic masking
- Implementing advanced masking techniques

About PostgreSQL Anonymizer

Logo image

`postgresql_anonymizer` is an extension to mask or replace personally identifiable information (PII) or commercially sensitive data from a PostgreSQL database.

The project has a **declarative approach** of anonymization. This means you can declare the masking rules using the PostgreSQL Data Definition Language (DDL) and specify your anonymization strategy inside the table definition itself.

Once the maskings rules are defined, you can access the anonymized data in 4 different ways:

- Anonymous Dumps : Simply export the masked data into an SQL file
- Static Masking : Remove the PII according to the rules
- Dynamic Masking : Hide PII only for the masked users
- Generalization : Create "blurred views" of the original data

About GDPR

This presentation **does not** go into the details of the GPDR act and the general concepts of anonymization.

For more information about it, please refer to the talk below:

- Anonymisation, Au-delà du RGPD (Video / French)
- Anonymization, Beyond GDPR (PDF / english)

Requirements

In order to make this workshop, you will need:

- A Linux VM (preferably `Debian 11 bullseye` or `Ubuntu 22.04`)
- A PostgreSQL instance (preferably `PostgreSQL 14`)
- The PostgreSQL Anonymizer (anon) extension, installed and initialized by a superuser
- A database named "boutique" owned by a **superuser** called "paul"
- A role "pierre" and a role "jack", both allowed to connect to the database "boutique"

A simple way to deploy a workshop environment is to install Docker Desktop and download the image below:

```
docker pull registry.gitlab.com/dalibo/postgresql_anonymizer:stable
```

Check out the INSTALL section in the documentation to learn how to install the extension in your PostgreSQL instance.

The Roles

We will with 3 different users:

```
text { .run-postgres user=postgres dbname=postgres show_result=false
} CREATE ROLE paul LOGIN SUPERUSER PASSWORD 'CHANGEME'; CREATE
ROLE pierre LOGIN PASSWORD 'CHANGEME'; CREATE ROLE jack LOGIN
PASSWORD 'CHANGEME';
```

Unless stated otherwise, all commands must be executed with the role `paul`.

Setup a `.pgpass` file to simplify the connections !

```
cat > ~/.pgpass << EOL
*:*:boutique:paul:CHANGEME
*:*:boutique:pierre:CHANGEME
*:*:boutique:jack:CHANGEME
EOL
```

```
chmod 0600 ~/.pgpass
```

The Sample database

We will work on a database called "boutique":

```
text { .run-postgres user=postgres dbname=postgres } CREATE
DATABASE boutique OWNER paul;
```

We need to activate the `anon` library inside that database:

```
text { .run-postgres user=postgres dbname=postgres } ALTER DATABASE
boutique   SET session_preload_libraries = 'anon';
```

Authors

This workshop is a collective work from Damien Clochard, Be Hai Tran, Florent Jardin, Frédéric Yhuel.

License

This document is distributed under the PostgreSQL license.

The source is available at

https://gitlab.com/dalibo/postgresql_anonymizer/-/tree/master/docs/how-to

Credits

- Cover photo by Alex Conchillos from Pexels (CC Zero)
- "Paul's Boutique" is the second studio album by American hip hop group Beastie Boys, released on July 25, 1989 by Capitol Records — title: how-to/1-static_masking draft: false toc: true —

1 - Static Masking

> Static Masking is the simplest way to hide personal information! This idea is simply to destroy the original data or replace it with an artificial one.

The story

Over the years, Paul has collected data about his customers and their purchases in a simple database. He recently installed a brand new sales application and the old database is now obsolete. He wants to save it and he would like to remove all personal information before archiving it.

How it works

Anon Static image

Learning Objective

In this section, we will learn:

- How to write simple masking rules
- The advantage and limitations of static masking
- The concept of “Singling Out” a person

The “customer” table

“‘text { .run-postgres parse_query=False } DROP TABLE IF EXISTS customer CASCADE;

DROP TABLE IF EXISTS payout CASCADE;

CREATE TABLE customer (id SERIAL PRIMARY KEY, firstname TEXT, lastname TEXT, phone TEXT, birth DATE, postcode TEXT);

Insert a few persons:

```text { .run-postgres  }
INSERT INTO customer
VALUES
(107,'Sarah','Conor','060-911-0911', '1965-10-10', '90016'),
(258,'Luke', 'Skywalker', NULL, '1951-09-25', '90120'),
(341,'Don', 'Draper','347-515-3423', '1926-06-01', '04520')
;
```

```text { .run-postgres } SELECT * FROM customer;```

The “payout” table

Sales are tracked in a simple table:

```text { .run-postgres  parse_query=False  } CREATE TABLE payout
(     id SERIAL PRIMARY KEY,     fk_customer_id INT REFERENCES
customer(id),     order_date DATE,     payment_date DATE,     amount
INT );```

Let’s add some orders:

```text { .run-postgres  } INSERT INTO payout VALUES (1,107,'2021-10-01','2021-10-01',
'7'), (2,258,'2021-10-02','2021-10-03', '20'), (3,341,'2021-10-02','2021-10-02',
'543'), (4,258,'2021-10-05','2021-10-05', '12'), (5,258,'2021-10-06','2021-10-06',
'92') ;```
```

Activate the extension

“‘text { .run-postgres } CREATE EXTENSION IF NOT EXISTS anon CASCADE;

SELECT anon.init();

SELECT setseed(0);

```
## Declare the masking rules

Paul wants to hide the last name and the phone numbers of his clients.
He will use the `fake_last_name()` and `partial()` functions for that:

```text { .run-postgres  }
SECURITY LABEL FOR anon ON COLUMN customer.lastname
IS 'MASKED WITH FUNCTION anon.fake_last_name()';

SECURITY LABEL FOR anon ON COLUMN customer.phone
IS 'MASKED WITH FUNCTION anon.partial(phone,2,$$X-XXX-XX$$,2)';
```

Apply the rules permanently

```
text { .run-postgres  } SELECT anon.anonymize_table('customer');

text { .run-postgres  } SELECT id, firstname, lastname, phone
FROM customer;
```

> This is called `Static Masking` because the **real data has been permanently replaced**. We'll see later how we can use dynamic anonymization or anonymous exports.

Exercises

E101 - Mask the client's first names

Declare a new masking rule and run the static anonymization function again.

E102 - Hide the last 3 digits of the postcode

Paul realizes that the postcode gives a clear indication of where his customers live. However he would like to have statistics based on their ”postcode area”.

Add a new masking rule to replace the last 3 digits by ‘x’.

E103 - Count how many clients live in each postcode area?

Aggregate the customers based on their anonymized postcode.
```

E104 - Keep only the year of each birth date

Paul wants age-based statistic. But he also wants to hide the real birth date of the customers.

Replace all the birth dates by January 1rst, while keeping the real year.

HINT: You can use the make_date function !

E105 - Singling out a customer

Even if the "customer" is properly anonymized, we can still isolate a given individual based on data stored outside of the table. For instance, we can identify the best client of Paul's boutique with a query like this:

```
text { .run-postgres  } WITH best_client AS (     SELECT SUM(amount),
fk_customer_id     FROM payout     GROUP BY fk_customer_id     ORDER
BY 1 DESC     LIMIT 1 ) SELECT c.* FROM customer c JOIN best_client
b ON (c.id = b.fk_customer_id)
```

This is called **Singling Out a person.**

We need to anonymize even further by removing the link between a person and its company. In the "order" table, this link is materialized by a foreign key on the field "fk_company_id". However we can't remove values from this column or insert fake identifiers because if would break the foreign key constraint.

How can we separate the customers from their payouts while respecting the integrity of the data?

Find a function that will shuffle the column "fk_company_id" of the "payout" table

HINT: Check out the static masking section of the documentation

Solutions

S101

"'text { .run-postgres } SECURITY LABEL FOR anon ON COLUMN customer.firstname IS 'MASKED WITH FUNCTION anon.fake_first_name()';

SELECT anon.anonymize_table('customer');

SELECT id, firstname, lastname FROM customer;

S102

```text { .run-postgres  }
SECURITY LABEL FOR anon ON COLUMN customer.postcode
IS 'MASKED WITH FUNCTION anon.partial(postcode,2,$$xxx$$,0)';
```

```
SELECT anon.anonymize_table('customer');

SELECT id, firstname, lastname, postcode
FROM customer;
```

S103

```
text { .run-postgres  } SELECT postcode, COUNT(id) FROM customer
GROUP BY postcode;
```

S104

“‘text { .run-postgres } SECURITY LABEL FOR anon ON COLUMN customer.birth IS ‘MASKED WITH FUNCTION make_date(EXTRACT(YEAR FROM birth)::INT,1,1)’;

SELECT anon.anonymize_table(‘customer’);

SELECT id, firstname, lastname, birth FROM customer;

```
### S105

Let\'s mix up the values of the fk_customer_id:

```text { .run-postgres  }
SELECT anon.shuffle_column('payout','fk_customer_id','id');
```

Now let’s try to single out the best client again :

```
text { .run-postgres  } WITH best_client AS (     SELECT SUM(amount),
fk_customer_id     FROM payout     GROUP BY fk_customer_id     ORDER
BY 1 DESC     LIMIT 1 ) SELECT c.* FROM customer c JOIN best_client
b ON (c.id = b.fk_customer_id);
```

WARNING: Note that the link between a `customer` and its `payout` is now completely false. For instance, if a customer A had 2 payouts. One of these payout may be linked to a customer B, while the second one is linked to a customer C.

In other words, this shuffling method with respect the foreign key constraint (aka the referential integrity) but it will break the data integrity. For some use case, this may be a problem.

In this case, Pierre will not be able to produce a BI report with the shuffle data, because the links between the customers and their payments are fake. — title: how-to/2-dynamic_masking draft: false toc: true —
```

2- How to use Dynamic Masking

> With Dynamic Masking, the database owner can hide personal data for some users, while other users are still allowed to read and write the authentic data.

The Story

Paul has 2 employees:

- Jack is operating the new sales application, he needs access to the real data. He is what the GPDR would call a **"data processor"**.
- Pierre is a data analyst who runs statistic queries on the database. He should not have access to any personal data.

How it works

Anon Dynamic image

Objectives

In this section, we will learn:

- How to write simple masking rules
- The advantage and limitations of dynamic masking
- The concept of "Linkability" of a person

The "company" table

"'text { .run-postgres parse_query=False }

DROP TABLE IF EXISTS supplier CASCADE;

DROP TABLE IF EXISTS company CASCADE;

CREATE TABLE company (id SERIAL PRIMARY KEY, name TEXT, vat_id TEXT UNIQUE);

```run-postgres
INSERT INTO company
VALUES
(952,'Shadrach', 'FR62684255667'),
(194,E'Johnny\'s Shoe Store','CHE670945644'),
(346,'Capitol Records','GB663829617823')
;

SELECT * FROM company;
```

The ”supplier” table

```
text { .run-postgres parse_query=False } CREATE TABLE supplier
(     id SERIAL PRIMARY KEY,     fk_company_id INT REFERENCES
company(id),     contact TEXT,     phone TEXT,     job_title TEXT
);
```

```
INSERT INTO supplier
VALUES
(299,194,'Johnny Ryall','597-500-569','CEO'),
(157,346,'George Clinton', '131-002-530','Sales manager')
;
```

```
SELECT * FROM supplier;
```

Activate the extension

```
CREATE EXTENSION IF NOT EXISTS anon CASCADE;

SELECT anon.init();

SELECT setseed(0);
```

Dynamic Masking

Activate the masking engine

```
SELECT anon.start_dynamic_masking();
```

Masking a role

```
SECURITY LABEL FOR anon ON ROLE pierre IS 'MASKED';

GRANT SELECT ON supplier TO pierre;
GRANT ALL ON SCHEMA public TO jack;
GRANT ALL ON ALL TABLES IN SCHEMA public TO jack;
```

Now connect as Pierre and try to read the supplier table:

```
text { .run-postgres user=pierre} SELECT * FROM supplier;
```

For the moment, there is no masking rule so Pierre can see the original data in each table.

Masking the supplier names

Connect as Paul and define a masking rule on the supplier table:

```
SECURITY LABEL FOR  anon ON COLUMN supplier.contact
IS 'MASKED WITH VALUE $$CONFIDENTIAL$$';
```

Now connect as Pierre and try to read the supplier table again:

```
text { .run-postgres user=pierre} SELECT * FROM supplier;
```

Now connect as Jack and try to read the real data:

```
text { .run-postgres user=jack } SELECT * FROM supplier;
```

Exercises

E201 - Guess who is the CEO of "Johnny's Shoe Store"

Masking the supplier name is clearly not enough to provide anonymity.

Connect as Pierre and write a simple SQL query that would reindentify some suppliers based on their job and their company.

Company names and job positions are available in many public datasets. A simple search on Linkedin or Google, would give you the names of the top executives of most companies..

> This is called **Linkability**: the ability to connect multiple records concerning the same data subject.

E202 - Anonymize the companies

We need to anonymize the "company" table, too. Even if they don't contain personal information, some fields can be used to **infer** the identity of their employees...

Write 2 masking rules for the company table. The first one will replace the "name" field with a fake name. The second will replace the "vat_id" with a random sequence of 10 characters

> HINT: Go to the documentation and look at the faking functions and random functions!

Connect as Pierre and check that he cannot view the real company info:

E203 - Pseudonymize the company name

Because of dynamic masking, the fake values will be different every time Pierre tries to read the table.

Pierre would like to have always the same fake values for a given company. **This is called pseudonymization.**

Write a new masking rule over the "vat_id" field by generating 10 random characters using the md5() function.

Write a new masking rule over the "name" field by using a pseudonymizing function.

Solutions

S201

```
text { .run-postgres user=pierre } SELECT s.id, s.contact, s.job_title,
c.name FROM supplier s JOIN company c ON s.fk_company_id = c.id;
```

S202

```
SECURITY LABEL FOR anon ON COLUMN company.name
IS 'MASKED WITH FUNCTION anon.fake_company()';

SECURITY LABEL FOR anon ON COLUMN company.vat_id
IS 'MASKED WITH FUNCTION anon.random_string(10)';
```

Now connect as Pierre and read the table again:

```
text { .run-postgres user=pierre } SELECT * FROM company;
```

Pierre will see different "fake data" every time he reads the table:

```
text { .run-postgres user=pierre } SELECT * FROM company;
```

S203

```
ALTER FUNCTION anon.pseudo_company SECURITY DEFINER;

SECURITY LABEL FOR  anon ON COLUMN company.name
IS 'MASKED WITH FUNCTION anon.pseudo_company(id)';
```

Connect as Pierre and read the table multiple times:

```
text { .run-postgres user=pierre } SELECT * FROM company;
```

```
text { .run-postgres user=pierre } SELECT * FROM company;
```

Now the fake company name is always the same.

title: how-to/3-anonymous_dumps draft: false toc: true —

3- Anonymous Dumps

> In many situation, what we want is simply to export the anonymized data into another database (for testing or to produce statistics). This is what **pg_dump_anon** does!

The Story

Paul has a website and a comment section where customers can express their views.

He hired a web agency to develop a new design for his website. The agency asked for a SQL export (dump) of the current website database. Paul wants to "clean" the database export and remove any personal information contained in the comment section.

How it works

Anon Dump image

Learning Objective

- Extract the anonymized data from the database
- Write a custom masking function to handle a JSON field.

Load the data

```
DROP TABLE IF EXISTS website_comment CASCADE;

CREATE TABLE website_comment (
  id SERIAL PRIMARY KEY,
  message JSONB
);
```

```
curl -Ls https://dali.bo/website_comment -o /tmp/website_comment.tsv
head /tmp/website_comment.tsv

COPY website_comment
FROM '/tmp/website_comment.tsv'
```

```
SELECT
  message->'meta'->'name' AS name,
  message->'content' AS content
FROM website_comment
ORDER BY id ASC
```

Activate the extension

```
CREATE EXTENSION IF NOT EXISTS anon CASCADE;
SELECT anon.init();
SELECT setseed(0);
```

Masking a JSON column

The "comment" field is filled with personal information and the fact the field does not have a standard schema makes our tasks harder.

In general, unstructured data are difficult to mask.

As we can see, web visitors can write any kind of information in the comment section. Our best option is to remove this key entirely because there's no way to extract personal data properly.

We can *clean* the comment column simply by removing the "content" key!

```
SELECT message - ARRAY['content']
FROM website_comment
WHERE id=1;
```

First let's create a dedicated schema and declare it as trusted. This means the "anon" extension will accept the functions located in this schema as valid masking functions. Only a superuser should be able to add functions in this schema.

```
CREATE SCHEMA IF NOT EXISTS my_masks;

SECURITY LABEL FOR anon ON SCHEMA my_masks IS 'TRUSTED';
```

Now we can write a function that remove the message content:

```
CREATE OR REPLACE FUNCTION my_masks.remove_content(j JSONB)
RETURNS JSONB
AS $func$
  SELECT j - ARRAY['content']
$func$
LANGUAGE SQL
;
```

Let's try it!

```
SELECT my_masks.remove_content(message)
FROM website_comment
```

And now we can use it in a masking rule:

```
SECURITY LABEL FOR anon ON COLUMN website_comment.message
```

```
IS 'MASKED WITH FUNCTION my_masks.remove_content(message)';
```

Finally we can export an **anonymous dump** of the table with `pg_dump_anon`:

```
export PATH=$PATH:$(pg_config --bindir)
pg_dump_anon --help
```

```
export PATH=$PATH:$(pg_config --bindir)
export PGHOST=localhost
export PGUSER=paul
pg_dump_anon boutique --table=website_comment > /tmp/dump.sql
```

Exercises

E301 - Dump the anonymized data into a new database

Create a database named "boutique_anon" and transfer the entire database into it.

E302 - Pseudonymize the meta fields of the comments

Pierre plans to extract general information from the metadata. For instance, he wants to calculate the number of unique visitors based on the different IP addresses. But an IP address is an **indirect identifier**, so Paul needs to anonymize this field while maintaining the fact that some values appear multiple times.

Replace the `remove_content` function with a better one called `clean_comment` that will:

- Remove the content key
- Replace the "name" value with a fake last name
- Replace the "ip_address" value with its MD5 signature
- Nullify the "email" key

  HINT: Look at the `jsonb_set()` and `jsonb_build_object()` functions

Solutions

S301

```
export PATH=$PATH:$(pg_config --bindir)
export PGHOST=localhost
export PGUSER=paul
dropdb --if-exists boutique_anon
createdb boutique_anon --owner paul
pg_dump_anon boutique | psql --quiet boutique_anon
```

```
export PGHOST=localhost
export PGUSER=paul
psql boutique_anon -c 'SELECT COUNT(*) FROM company'
```

S302

```
CREATE OR REPLACE FUNCTION my_masks.clean_comment(message JSONB)
RETURNS JSONB
VOLATILE
LANGUAGE SQL
AS $func$
SELECT
  jsonb_set(
    message,
    ARRAY['meta'],
    jsonb_build_object(
        'name',anon.fake_last_name(),
        'ip_address', md5((message->'meta'->'ip_addr')::TEXT),
        'email', NULL
    )
  ) - ARRAY['content'];
$func$;
```

```
SELECT my_masks.clean_comment(message)
FROM website_comment;
```

```
SECURITY LABEL FOR anon ON COLUMN website_comment.message
IS 'MASKED WITH FUNCTION my_masks.clean_comment(message)';
```

4 - Generalization

> The main idea of generalization is to ”blur” the original data. For example, instead of saying ”Mister X was born on July 25, 1989”, we can say ”Mister X was born is the 80’s”. The information is still true, but it is less precise and it can’t be used to reidentify the subject.

The Story

Paul hired dozens of employees over the years. He kept a record of their hair color, size and medical condition.

Paul wants to extract weird stats from these details. He provides generalized views to Pierre.

How it works

Anon Generalized image

Learning Objective

In this section, we will learn:

- The difference between masking and generalization
- The concept of "K-anonymity"

The "employee" table

"'text { .run-postgres parse_query=False } DROP TABLE IF EXISTS employee CASCADE;

CREATE TABLE employee (id INT PRIMARY KEY, full_name TEXT, first_day DATE, last_day DATE, height INT, hair TEXT, eyes TEXT, size TEXT, asthma BOOLEAN, CHECK(hair = ANY(ARRAY['bald','blond','dark','red'])), CHECK(eyes = ANY(ARRAY['blue','green','brown'])) , CHECK(size = ANY(ARRAY['S','M','L','XL','XXL'])));

```
::: caution

This is awkward and illegal.

:::
```

Loading the data:

```
``` bash
curl -Ls https://dali.bo/employee -o /tmp/employee.tsv
head -n3 /tmp/employee.tsv

COPY employee FROM '/tmp/employee.tsv'

SELECT count(*) FROM employee;

SELECT full_name,first_day, hair, size, asthma
FROM employee
LIMIT 3;
```

Data suppression

Paul wants to find if there's a correlation between asthma and the eyes color.

He provides the following view to Pierre.

```
DROP MATERIALIZED VIEW IF EXISTS v_asthma_eyes;
```
```

```
CREATE MATERIALIZED VIEW v_asthma_eyes AS
SELECT eyes, asthma
FROM employee;
```

```
SELECT *
FROM v_asthma_eyes
LIMIT 3;
```

Pierre can now write queries over this view.

```
SELECT
  eyes,
  100*COUNT(1) FILTER (WHERE asthma) / COUNT(1) AS asthma_rate
FROM v_asthma_eyes
GROUP BY eyes;
```

Pierre just proved that asthma is caused by green eyes.

K-Anonymity

The 'asthma' and 'eyes' are considered as indirect identifiers.

```
SECURITY LABEL FOR k_anonymity ON COLUMN v_asthma_eyes.eyes
IS 'INDIRECT IDENTIFIER';

SECURITY LABEL FOR k_anonymity ON COLUMN v_asthma_eyes.asthma
IS 'INDIRECT IDENTIFIER';
```

```
SELECT anon.k_anonymity('v_asthma_eyes');
```

The v_asthma_eyes has '2-anonymity'. This means that each quasi-identifier combination (the 'eyes-asthma' tuples) occurs in at least 2 records for a dataset.

In other words, it means that each individual in the view cannot be distinguished from at least 1 (k-1) other individual.

Range and Generalization functions

```
DROP MATERIALIZED VIEW IF EXISTS v_staff_per_month;
CREATE MATERIALIZED VIEW v_staff_per_month AS
SELECT
    anon.generalize_daterange(first_day,'month') AS first_day,
    anon.generalize_daterange(last_day,'month') AS last_day
FROM employee;
```

```
SELECT *
FROM v_staff_per_month
LIMIT 3;
```

Pierre can write a query to find how many employees were hired in november 2021.

```
SELECT COUNT(1)
       FILTER (
            WHERE make_date(2019,11,1)
            BETWEEN lower(first_day)
            AND COALESCE(upper(last_day),now())
       )
FROM v_staff_per_month;
```

Declaring the indirect identifiers

Now let's check the k-anonymity of this view by declaring which columns are indirect identifiers.

```
SECURITY LABEL FOR anon ON COLUMN v_staff_per_month.first_day
IS 'INDIRECT IDENTIFIER';

SECURITY LABEL FOR anon ON COLUMN v_staff_per_month.last_day
IS 'INDIRECT IDENTIFIER';

SELECT anon.k_anonymity('v_staff_per_month');
```

In this case, the k factor is 1 which means that at least one unique individual can be identified directly by his/her first and last dates.

Note that the security label provider is `k_anonymity` and not `anon`.

Exercises

E401 - Simplify `v_staff_per_month` and decrease granularity

Generalizing dates per month is not enough. Write another view called 'v_staff_per_year' that will generalize dates per year.

Also simplify the view by using a range of int to store the years instead of a date range.

E402 - Staff progression over the years

How many people worked for Paul for each year between 2018 and 2021?

E403 - Reaching 2-anonymity for the `v_staff_per_year` view

What is the k-anonymity of 'v_staff_per_month_years'?

Solutions

S401

```
DROP MATERIALIZED VIEW IF EXISTS v_staff_per_year;
```

```
CREATE MATERIALIZED VIEW v_staff_per_year AS
SELECT
  int4range(
    extract(year from first_day)::INT,
    extract(year from last_day)::INT,
    '[]'
  ) AS period
FROM employee;
```

‘[]’ will include the upper bound

```
SELECT *
FROM v_staff_per_year
LIMIT 3;
```

S402

```
SELECT
  year,
  COUNT(1) FILTER (
      WHERE year <@ period
  )
FROM
    generate_series(2018,2021) year,
    v_staff_per_year
GROUP BY year
ORDER BY year ASC;
```

S403

```
SECURITY LABEL FOR anon ON COLUMN v_staff_per_year.period
IS 'INDIRECT IDENTIFIER';

SELECT anon.k_anonymity('v_staff_per_year');
```

Conclusion

Clean up !

```
text { .run-postgres user=postgres } DROP EXTENSION anon CASCADE;
```

“‘text { .run-postgres user=postgres dbname=postgres }

REASSIGN OWNED BY jack TO postgres; REVOKE ALL ON SCHEMA public FROM jack;

REASSIGN OWNED BY paul TO postgres;

REASSIGN OWNED BY pierre TO postgres;

<!---
We can't use run-postgres here, because the pandoc-run-postgres filter is already connected to the "boutique" database. Therefore we can't remove it at this point of the document generation
-->

```text { .run-postgres user=postgres dbname=postgres }
DROP DATABASE IF EXISTS boutique;
```

```
text { .run-postgres user=postgres dbname=postgres } DROP ROLE
IF EXISTS jack; DROP ROLE IF EXISTS paul; DROP ROLE IF EXISTS
pierre;
```

Many Masking Strategies

- Static Masking : perfect for ”once-and-for-all” anonymization
- Dynamic Masking : useful when one user is untrusted
- Anonymous Dumps : can be used in CI/CD workflows
- **Generalization** good for statistics and data science

Many Masking Functions

- Destruction and partial destruction
- Adding Noise
- Randomization
- Faking and Advanced Faking
- Pseudonymization
- Generic Hashing
- Custom masking

RTFM -> Masking Functions

Advantages

- Masking rules written in SQL
- Masking rules stored in the database schema
- No need for an external ETL
- Works with all current versions of PostgreSQL
- Multiple strategies, multiple functions

Drawbacks

- Does not work with other databases (hence the name)
- Lack of feedback for huge tables (> 10 TB)

Also...

Other projects you may like

- pg_sample : extract a small dataset from a larger PostgreSQL database
- PostgreSQL Faker : An advanced faking extension based on the python Faker lib

Help Wanted!

This is a free and open project!

labs.dalibo.com/postgresql_anonymizer

Please send us feedback on how you use it, how it fits your needs (or not), etc.

This is a 4 hour workshop!

Sources are here: gitlab.com/dalibo/postgresql_anonymizer

Download the PDF Handout

Questions?

Deal With It image :::

PostgreSQL Anonymizer How To

This is a 4 hours workshop that demonstrates various anonymization techniques.

Write

This workshop is written with jupyter-notebook. The `*.ipynb` files are mixing markdown content with live SQL statements that are executed on a PostgreSQL instance.

```
pip install -r requirements.txt
jupyter notebook
```

Build

The source files are converted to markdown and then exported to pdf, slides, epub, etc.

```
make
```

The export files will be available in the `_build` folder.

Type `make help` for more details

title: impexp draft: false toc: true —

Import Export

Principle

Rules can be imported and exported in jsonb format via the functions :

- `anon.export_current_database_rules(provider text DEFAULT 'anon')`
- `anon.export_roles_rules(provider text DEFAULT 'anon')`
- `anon.import_database_rules(database_rules jsonb, provider text DEFAULT 'anon')`
- `anon.import_roles_rules(role_rules jsonb, provider text DEFAULT 'anon'`

Since roles are instance wide objects they must be managed separately.

Compatibility of the import/export JSON schema between releases

The JSON schema used for imports / exports can change between versions and may not be backward compatible. All the changes to the format are detailed in the CHANGELOG.

It is recommended to re-export the rules after an upgrade.

The migration path for export files is:

- Import the rules in **anon** version N;

- Upgrade **anon** from version N to N+1;
- Export the rules using the new schema.

Export

We will create the following objects in the database **anon** to present the feature.

```
CREATE EXTENSION IF NOT EXISTS anon CASCADE;
ALTER DATABASE anon SET anon.masking_policies = 'impexp';
ALTER DATABASE anon SET session_preload_libraries=anon;
\c - -- reconnect

CREATE ROLE anna LOGIN;
CREATE SCHEMA test;
CREATE TABLE test.t1(c int, t text);
CREATE TABLE test.t2(c int, t text);
CREATE TABLE test.t3(c int, t text);
CREATE VIEW test.v AS SELECT * FROM test.t3;
SECURITY LABEL FOR impexp ON ROLE anna
                IS 'MASKED';
SECURITY LABEL FOR impexp ON SCHEMA test
                IS 'TRUSTED';
SECURITY LABEL FOR impexp ON TABLE test.t1
                IS 'MASKED WHEN c > 10';
SECURITY LABEL FOR impexp ON COLUMN test.t1.t
                IS 'MASKED WITH FUNCTION pg_catalog.md5(t)';
SECURITY LABEL FOR impexp ON COLUMN test.t2.t
                IS 'MASKED WITH VALUE NULL';
SECURITY LABEL FOR impexp ON VIEW test.v
                IS 'MASKED WHEN c > 10';
SECURITY LABEL FOR impexp ON COLUMN test.v.t
                IS 'MASKED WITH FUNCTION pg_catalog.md5(t)';
SECURITY LABEL FOR impexp ON FUNCTION pg_catalog.md5(text)
                IS 'TRUSTED';
```

Exporting all role rules for the `impexp` masking policy (details about the format are described in the Import section of this page):

```
SELECT jsonb_pretty(x) FROM anon.export_roles_rules('impexp') AS F(x);
```

```
        jsonb_pretty
-----------------------------
 {                           +
     "anna": "MASKED"+
 }
(1 row)
```

Export the current database's roles for the `impexp` provider:

```
SELECT jsonb_pretty(x) FROM anon.export_current_database_rules('impexp') AS F(x);
                              jsonb_pretty
-----------------------------------------------------------------------
 {                                                                    +
     "schemas": {                                                     +
         "test": {                                                    +
             "mask": "TRUSTED",                                       +
             "views": {                                               +
                 "v": {                                               +
                     "mask": "MASKED WHEN c > 10",                    +
                     "columns": {                                     +
                         "t": "MASKED WITH FUNCTION pg_catalog.md5(t)"+
                     }                                                +
                 }                                                    +
             },                                                       +
             "tables": {                                              +
                 "t1": {                                              +
                     "mask": "MASKED WHEN c > 10",                    +
                     "columns": {                                     +
                         "t": "MASKED WITH FUNCTION pg_catalog.md5(t)"+
                     }                                                +
                 },                                                   +
                 "t2": {                                              +
                     "columns": {                                     +
                         "t": "MASKED WITH VALUE NULL"                +
                     }                                                +
                 }                                                    +
             }                                                        +
         },                                                           +
         "pg_catalog": {                                              +
             "functions": {                                           +
                 "md5": [                                             +
                     {                                                +
                         "mask": "TRUSTED",                           +
                         "parameters": [                              +
                             "text"                                   +
                         ]                                            +
                     }                                                +
                 ],                                                   +
             }                                                        +
         }                                                            +
     }                                                                +
 }
(1 row)
```

If no masking policy is provided, `anon` is the default. In this case all the masking

policies will be exported, including those added by the extension (functions in the `pg_catalog` or `anon` schema).

Import

The import function take a json string and creates the security labels. All objects must exist beforehand with the correct type.

A masking rule is a string.

The role rules are imported with `anon.import_roles_rules(rule, provider)`. where `provider` is the masking policy and defaults to `anon` and `rule` is a json string composed of a hashmap where:

- each key is a role name;
- each object is a masking rule.

All identifiers (object names and types) must preserve exact case because they are quoted during security label import to prevent SQL injection.

```
SELECT * FROM anon.import_roles_rules(
$$
{
    "anna": "MASKED"
}
$$, 'impexp'
);
```

The command outputs the `SECURITY LABEL` statements that where executed.

```
 import_roles_rules
--------------------


(1 row)
```

For a database the json follows this schema:

- `mask`: an optional key, it contains the masking rule
- `schemas`:
  - `mask`: an optional key, it contains the masking rule
  - `functions`: an optional key, it contains a hasmap where:
    - each key is a valid function name, function with duplicate names should include the parameter list;
    - each object is a set (a collection that doesn't allow duplicates):
      - `parameters`: an array with all the parameter types in order of apparition;
      - `mask`: the masking rule;
  - `tables`|`views`|`foreign tables`: optional keys key, each can contain a hasmap where:
    - each key is a valid relation name of the specified kind;

- each object contains:
  - `mask`: an optional key, it contains the masking rule;
  - `columns`: a hashmap of columns where:
    - each key is a valid column name;
    - each object is a masking rule.

The security labels are created in the current database and the provided masking policy (or `anon` otherwise) with the `anon.import_database_rules(rules, provider)` function.

```
SELECT * FROM anon.import_database_rules(
$$
{
    "schemas": {
        "test": {
            "mask": "TRUSTED",
            "tables": {
                "t1": {
                    "mask": "MASKED WHEN c > 10",
                    "columns": {
                        "t": "MASKED WITH FUNCTION pg_catalog.md5(t)"
                    }
                },
                "t2": {
                    "columns": {
                        "t": "MASKED WITH VALUE NULL"
                    }
                }
            },
            "views": {
                "v": {
                    "mask": "MASKED WHEN c > 10",
                    "columns": {
                        "t": "MASKED WITH FUNCTION pg_catalog.md5(t)"
                    }
                }
            }
        },
        "pg_catalog": {
            "functions": {
                "md5": [{
                  "parameters": ["text"],
                  "mask": "TRUSTED"
                }]
            }
        }
    }
```

```
}
$$, 'impexp'
);
```

The rules are now visible in the `pg_masking_rules` view:

```
SELECT *
  FROM anon.user_rules
 WHERE full_objname LIKE '%t1';
```

```
 provider | objtype | full_objname |        label
----------+---------+--------------+----------------------
 impexp   | table   | test.t1      |  MASKED WHEN c > 10
(1 row)
```



Figure 5: PostgreSQL Anonymizer

Anonymization & Data Masking for Postgres

`PostgreSQL Anonymizer` is an extension to mask or replace personally identifiable information (PII) or commercially sensitive data from a Postgres database.

The project has a **declarative approach** of anonymization. This means you can declare the masking rules using the PostgreSQL Data Definition Language (DDL) and specify your anonymization policy inside the table definition itself.

The main goal of this extension is to offer **anonymization by design**. We firmly believe that data masking rules should be written by the people who develop the application because they have the best knowledge of how the data model works. Therefore masking rules must be implemented directly inside the database schema.

Once the masking rules are defined, you can apply them using 6 different **masking methods** :

-
-
-
-
-

-

Each method has its pros and cons. Different masking methods may be used in different contexts. In any case, masking the data directly inside the PostgreSQL instance without using an external tool is crucial to limit the exposure and the risks of data leak.

In addition, various Masking Functions are available : randomization, faking, partial scrambling, shuffling, noise, generalization or even your own custom function!

Finally, the extension offers a panel of detection functions that will try to guess which columns need to be anonymized.

Available on ...

PostgreSQL Anonymizer is available on multiple linux distributions, cloud service providers, development frameworks and PostgreSQL forks:

See the INSTALL section for more details.

Quick Start

Step 0. Launch docker image of the project

```
ANON_IMG=registry.gitlab.com/dalibo/postgresql_anonymizer
docker run --name anon_quickstart --detach -e POSTGRES_PASSWORD=x $ANON_IMG
docker exec -it anon_quickstart psql -U postgres
```

Step 1. Create a database and load the extension in it

```
CREATE DATABASE demo;
-- the docker image already preloads anon in every database
\connect demo
You are now connected to database "demo" as user "postgres".
```

Step 2. Create a table

```
CREATE TABLE people AS
    SELECT  153478       AS id,
            'Sarah'      AS firstname,
            'Conor'      AS lastname,
            '0609110911' AS phone
;
SELECT * FROM people;
   id    | firstname | lastname |   phone
---------+-----------+----------+------------
 153478 | Sarah     | Conor    | 0609110911
```

Step 3. Create the extension and activate the masking engine

```
CREATE EXTENSION anon;
ALTER DATABASE demo SET anon.transparent_dynamic_masking TO true;
```

Step 4. Declare a masked user

```
CREATE ROLE skynet LOGIN;

SECURITY LABEL FOR anon ON ROLE skynet IS 'MASKED';

GRANT pg_read_all_data to skynet;
```

Step 5. Declare the masking rules

```
SECURITY LABEL FOR anon ON COLUMN people.lastname
  IS 'MASKED WITH FUNCTION anon.dummy_last_name()';

SECURITY LABEL FOR anon ON COLUMN people.phone
  IS 'MASKED WITH FUNCTION anon.partial(phone,2,$$******$$,2)';
```

Step 6. Connect with the masked user

```
\connect - skynet
You are now connected to database "demo" as user "skynet"

SELECT * FROM people;
   id   | firstname | lastname  |   phone
--------+-----------+-----------+------------
 153478 | Sarah     | Stranahan | 06******11
```

Success Stories

> With PostgreSQL Anonymizer we integrate, from the design of the database, the principle that outside production the data must be anonymized. Thus we can reinforce the GDPR rules, without affecting the quality of the tests during version upgrades for example.

— **Thierry Aimé, Office of Architecture and Standards in the French Public Finances Directorate General (DGFiP)**

> Thanks to PostgreSQL Anonymizer we were able to define complex masking rules in order to implement full pseudonymization of our databases without losing functionality. Testing on realistic data while guaranteeing the confidentiality of patient data is a key point to improve the robustness of our functionalities and the quality of our customer service.

— **Julien Biaggi, Product Owner at bioMérieux**

> I just discovered your postgresql_anonymizer extension and used it at my company for anonymizing our user for local development. Nice work!
>
> — **Max Metcalfe**

If this extension is useful to you, please let us know !

Support

We need your feedback and ideas ! Let us know what you think of this tool, how it fits your needs and what features are missing.

Want to talk directly with us ? Join us on Matrix or Discord

- Matrix: https://matrix.to/#/#anon:dalibo.com
- Discord: https://discord.com/channels/710918545906597938/1427672533104070807

You can also open a public issue on gitlab

If you need enterprise support or if you want to sponsor the project, send us a message at contact@dalibo.com.



Figure 6: PostgreSQL Anonymizer

Anonymization & Data Masking for Postgres

`PostgreSQL Anonymizer` is an extension to mask or replace personally identifiable information (PII) or commercially sensitive data from a Postgres database.

Available on ...

PostgreSQL Anonymizer is available on multiple linux distributions, cloud service providers, development frameworks and PostgreSQL forks:

See the INSTALL section for more details.

Principles

The project has a **declarative approach** of anonymization. This means you can declare the masking rules using the PostgreSQL Data Definition Language

(DDL) and specify your anonymization policy inside the table definition itself.

The main goal of this extension is to offer **anonymization by design**. We firmly believe that data masking rules should be written by the people who develop the application because they have the best knowledge of how the data model works. Therefore masking rules must be implemented directly inside the database schema.

Once the masking rules are defined, you can apply them using 6 different **masking methods** :

-
-
-
-
-
-

Each method has its pros and cons. Different masking methods may be used in different contexts. In any case, masking the data directly inside the PostgreSQL instance without using an external tool is crucial to limit the exposure and the risks of data leak.

In addition, various Masking Functions are available : randomization, faking, partial scrambling, shuffling, noise, generalization or even your own custom function!

Finally, the extension offers a panel of detection functions that will try to guess which columns need to be anonymized.

Quick Start

Step 0. Launch docker image of the project

```
ANON_IMG=registry.gitlab.com/dalibo/postgresql_anonymizer
docker run --name anon_quickstart --detach -e POSTGRES_PASSWORD=x $ANON_IMG
docker exec -it anon_quickstart psql -U postgres
```

Step 1. Create a database and load the extension in it

```
CREATE DATABASE demo;
ALTER DATABASE demo SET session_preload_libraries = 'anon'

\connect demo
You are now connected to database "demo" as user "postgres".
```

Step 2. Create a table

```
CREATE TABLE people AS
    SELECT  153478       AS id,
            'Sarah'      AS firstname,
```

```
            'Conor'      AS lastname,
            '0609110911' AS phone
;

SELECT * FROM people;
   id    | firstname | lastname |   phone
---------+-----------+----------+------------
 153478 | Sarah     | Conor    | 0609110911
```

Step 3. Create the extension and activate the masking engine

```
CREATE EXTENSION anon;
ALTER DATABASE demo SET anon.transparent_dynamic_masking TO true;
```

Step 4. Declare a masked user

```
CREATE ROLE skynet LOGIN;

SECURITY LABEL FOR anon ON ROLE skynet IS 'MASKED';

GRANT pg_read_all_data to skynet;
```

Step 5. Declare the masking rules

```
SECURITY LABEL FOR anon ON COLUMN people.lastname
  IS 'MASKED WITH FUNCTION anon.dummy_last_name()';

SECURITY LABEL FOR anon ON COLUMN people.phone
  IS 'MASKED WITH FUNCTION anon.partial(phone,2,$$******$$,2)';
```

Step 6. Connect with the masked user

```
\connect - skynet
You are now connected to database "demo" as user "skynet"

SELECT * FROM people;
   id    | firstname | lastname  |   phone
---------+-----------+-----------+------------
 153478 | Sarah     | Stranahan | 06******11
```

Success Stories

> With PostgreSQL Anonymizer we integrate, from the design of the database, the principle that outside production the data must be anonymized. Thus we can reinforce the GDPR rules, without affecting the quality of the tests during version upgrades for example.

— Thierry Aimé, Office of Architecture and Standards in the French Public Finances Directorate General (DGFiP)

> Thanks to PostgreSQL Anonymizer we were able to define complex masking rules in order to implement full pseudonymization of our databases without losing functionality. Testing on realistic data while guaranteeing the confidentiality of patient data is a key point to improve the robustness of our functionalities and the quality of our customer service.

— **Julien Biaggi, Product Owner at bioMérieux**

> I just discovered your postgresql_anonymizer extension and used it at my company for anonymizing our user for local development. Nice work!

— **Max Metcalfe**

If this extension is useful to you, please let us know !

Support

We need your feedback and ideas ! Let us know what you think of this tool, how it fits your needs and what features are missing.

Want to talk directly with us ? Join us on Matrix or Discord

- Matrix: https://matrix.to/#/#anon:dalibo.com
- Discord: https://discord.com/channels/710918545906597938/1427672533104070807

You can also open a public issue on gitlab

If you need enterprise support or if you want to sponsor the project, send us a message at contact@dalibo.com.

open a public issue: https://gitlab.com/dalibo/postgresql_anonymizer/issues

title: INSTALL draft: false toc: true —

INSTALL

The installation process is composed of 3 basic steps:

- Step 1: **Deploy** the extension into the host server
- Step 2: **Load** the extension in the PostgreSQL instance
- Step 3: **Create** and **Initialize** the extension inside the database

Two versions are available — `stable` (production) and `latest` (new features).
For the list of supported PostgreSQL versions and compatibility, see Support.

There are multiple ways to install the extension :

- Install on RedHat / Rocky Linux

- Install on Fedora
- Install on Debian / Ubuntu
- Install on SUSE
- Install with Ansible
- Install With PGXN
- Install from source
- Install with docker
- Install as a black box
- Install with Django
- Install on MacOS
- Install on Windows
- Install on PostgreSQL Forks
- Install in the cloud
- Load the extension
- Uninstall

Loading and creating the extension is covered on a dedicated page and works the same way whatever the installation method you choose: see Load the extension.

If you're having any problem, check the Troubleshooting section.

Install on RedHat / Rocky Linux / Alma Linux

!!! warning "New RPM repository !"

```
DO NOT use the package provided by the PGDG RPM repository.
It is obsolete.
```

Step 0: Add the DaLibo Labs RPM repository to your system.

```
curl -fsSL https://repo.dalibo.com/install.sh | sh
```

Alternatively you can download the `latest` version from the Gitlab Package Registry.

Step 1: Deploy

```
sudo yum install postgresql_anonymizer_16
```

(Replace `16` with the major version of your PostgreSQL instance.)

Then **load the extension** to finish the setup.

Install on Fedora

The extension is packaged natively for Fedora, so there is no repository to add.

Step 1: Deploy

```
sudo dnf install postgresql16-anonymizer
```

This also pulls in the matching `postgresql16-server` package. Fedora packages the extension for PostgreSQL 16 only, under versioned package names; this is independent of the default PostgreSQL version of your Fedora release (Fedora 44 ships PostgreSQL 18). For another major version, use our own RPM packages or install from source.

!!! note

`Fedora ships the version that was current when the release was built, so it may lag behind the latest upstream release. For a more recent version, install [our own RPM packages](#install-on-redhat-rocky-linux-alma-linux) or [from source](#install-from-source).`

Then **load the extension** to finish the setup.

Install on Debian / Ubuntu

This is the recommended way to install the `stable` version

Step 0: Add the DaLibo Labs DEB Repo to your system.

```
curl -fsSL https://repo.dalibo.com/install.sh | sh
```

Alternatively you can download the `latest` version from the Gitlab Package Registry.

Step 1: Deploy

```
sudo apt install postgresql_anonymizer_16
```

(Replace `16` with the major version of your PostgreSQL instance.)

Then **load the extension** to finish the setup.

Install on SUSE

This procedure works with the official PostgreSQL packages from the Postgres Zypper Repository.

Step 0: Download the latest version from the PostgreSQL Anonymizer Package Registry

Rename the downloaded file as `postgresql_anonymizer_16.rpm`

(Replace `16` with the major version of your PostgreSQL instance.)

Step 1: Deploy

```
sudo zypper install postgresql_anonymizer_16.rpm
```

Then **load the extension** to finish the setup.

Install with Ansible

This method will install the `stable` extension

Step 1a: Install the Dalibo PostgreSQL Essential Ansible Collection

```
ansible-galaxy collection install dalibo.advanced
```

Step 1b: Write a playbook (e.g. `anon.yml`) to the `postgresql_anonymizer` role to the database servers. For instance:

```
---
- name: Install the PostgreSQL Anonymizer extension on all hosts of the pgsql group
  hosts: pgsql
  roles:
    - dalibo.advanced.anon
```

Step 1c: Launch the playbook

```
ansible-playbook anon.yml
```

Then **load the extension** to finish the setup.

Install With PGXN

!!! warning

```
This method is not available currently but you can use the
"Install From Source" method below which is very similar.
```

Install From Source

This is the recommended way to install the `latest` extension

!!! Important

```
Building the extension requires a full Rust development
environment. It is not recommended to build it on a production server.
```

Before anything else, you need to install the PGRX System Requirements and install and initialise PGRX itself using

```
cargo install cargo-pgrx --version 0.14.3 --locked
cargo pgrx init
```

!!! note

```
You may need to specify your pg_config location in the second command
by using the `--pg{version}` flag
(e.g. `--pg16 /usr/lib/postgresql/16/bin/pg_config`).
```

Step 0: Download the source from the official repository on Gitlab, either the archive of the latest release, or clone the `latest` branch:

```
git clone https://gitlab.com/dalibo/postgresql_anonymizer.git
```

Step 1: Build the project like any other PostgreSQL extension:

```
make extension
sudo make install
```

NOTE: If you have multiple versions of PostgreSQL on the server or if the package does not build/install correctly, you may need to specify which version is your target by defining the `PG_CONFIG` and `PGVER` env variable like this:

```
make extension PG_CONFIG=/usr/lib/postgresql/14/bin/pg_config PGVER=pg14
sudo make install PG_CONFIG=/usr/lib/postgresql/14/bin/pg_config PGVER=pg14
```

Then **load the extension** to finish the setup.

Install with Docker

If you can't (or don't want to) install the PostgreSQL Anonymizer extension directly inside your instance, then you can use the docker image :

```
docker pull registry.gitlab.com/dalibo/postgresql_anonymizer:stable
```

The image is available with 2 two tags:

- `latest` (default) contains the current developments
- `stable` is the based on the previous release

You can run the docker image like the regular postgres docker image.

For example:

Launch a postgres docker container

```
docker run -d -e POSTGRES_PASSWORD=x -p 6543:5432 registry.gitlab.com/dalibo/postgresql_anon
```

then connect:

```
export PGPASSWORD=x
psql --host=localhost --port=6543 --user=postgres
```

The extension is already created and initialized, you can use it directly:

```
# SELECT anon.partial_email('daamien@gmail.com');
     partial_email
-----------------------
 da******@gm******.com
(1 row)
```

Note: The docker image is based on the latest PostgreSQL version and we do not plan to provide a docker image for each version of PostgreSQL. However you can build your own image based on the version you need like this:

```
DOCKER_PG_MAJOR_VERSION=16 make docker_image
```

Install as a "Black Box"

see Anonymous Dumps

Install With Django

Django PostgreSQL Anonymizer provides seamless integration with the PostgreSQL Anonymizer extension, enabling you to anonymize data at the database level with zero performance overhead. Ideal for development workflows, safe data sharing, and reducing privacy risks.

See https://django-postgres-anonymizer.readthedocs.io/

Install on MacOS

WE DO NOT PROVIDE COMMUNITY SUPPORT FOR THIS EXTENSION ON MACOS SYSTEMS.

However it should be possible to build the extension if you install the PGRX Mac OS system requirements and then follow the regular install from source procedure.

Install on Windows

At the moment there's no native build of PostgreSQL Anonymizer for Windows.

However it is possible to run PostgreSQL inside a WSL2 container, which is basically an Ubuntu subsystem running on Windows. You can then install PostgreSQL Anonymizer inside the WSL2 container like you would on a regular Ubuntu server.

Please read the Windows documentation for more details:

- Install WSL2
- Install PostgreSQL in WSL2

If you need native builds for Windows, please consider sponsoring the project !

Install on PostgreSQL Forks

The extension is also available for PostgreSQL forks and PostgreSQL-compatible software, such as:

- EDB Postgres
- Postgres Pro Enterprise
- Tanzu Greenplum
- Yugabytes

Please refer to their own documentation on how to activate the extension as they might have a platform-specific install procedure.

Install in the cloud

This extension must be installed with superuser privileges, which is something that most Database As A Service platforms (DBaaS), such as Amazon RDS or Microsoft Azure SQL, do not allow. They must add the extension to their catalog in order for you to use it.

At the time we are writing this (Jan. 2026), the following platforms provide PostgreSQL Anonymizer:

- Aiven
- Alibaba Cloud
- Crunchy Bridge
- Google Cloud SQL
- IBM Cloud
- Microsoft Azure Database
- Neon
- Postgres.ai
- Scalingo
- Yandex

Please refer to their own documentation on how to activate the extension as they might have a platform-specific install procedure.

If your favorite DBaaS provider is not present in the list above, there is not much we can do about it… Although we have open discussions with most major actors in this domain, we DO NOT have internal knowledge on whether or not they will support it in the near future. If privacy and anonymity are a concern to you, we encourage you to contact the customer service of these platforms and ask them directly if they plan to add this extension to their catalog.

Addendum: Troubleshooting

If you are having difficulties, you may have missed a step during the installation processes. Here's a quick checklist to help you:

Check that the extension is present

First, let's see if the extension was correctly deployed:

```
ls $(pg_config --sharedir)/extension/anon
ls $(pg_config --pkglibdir)/anon.so
```

If you get an error, the extension is probably not present on host server. Go back to step 1.

Check that the extension is loaded

Now connect to your database and look at the configuration with:

```
SHOW shared_preload_libraries;
SHOW session_preload_libraries;
SHOW local_preload_libraries;
```

If you don't see `anon` in any of these parameters, go back to step 2.

Check that the extension is created

Again connect to your database and type:

```
SELECT * FROM pg_extension WHERE extname= 'anon';
```

If the result is empty, the extension is not declared in your database. Go back to step 3.

Check that the extension is initialized

Finally, look at the state of the extension:

```
SELECT anon.is_initialized();
```

If the result is not `t`, the extension data is not present. Go back to step 3.

Uninstall

Step 1: Remove all rules

```
SELECT anon.remove_masks_for_all_columns();
SELECT anon.remove_masks_for_all_roles();
```

Although this step is not mandatory, it is highly recommended.

In some situations ever, it may be useful to keep the masking rules inside the database schema even if the anon extension is removed ! Keep in mind that pg_dump and pg_restore both have an option `--no-security-labels` to exclude the masking rules when you want to import/export the database.

Step 2: Drop the extension

```
DROP EXTENSION anon;
```

Step 3: Unload the extension

Reverse the load step. If you preloaded `anon` globally with `shared_preload_libraries`:

```
ALTER SYSTEM RESET shared_preload_libraries;
```

then restart PostgreSQL. If you loaded it per-database instead:

```
ALTER DATABASE foo RESET session_preload_libraries;
```

Step 4: Uninstall the extension

For Redhat / Rocky:

```
sudo yum remove postgresql_anonymizer_17
```

Replace 17 by the version of your postgresql instance.

title: links draft: false toc: true —

Ideas and Resources

Videos / Presentations

- French: https://www.youtube.com/watch?v=KGSlp4UygdU
- English: https://www.youtube.com/watch?v=niIIFL4s-L8
- Chinese: https://www.youtube.com/watch?v=n9atI31FcSM

Similar Tools

Here's a list of **open-source** projects with similar goals. AS the PostgreSQL Anonymizer extension is often compared to some of them, we try to maintain the feature matrix below:

Name	rules syntax	static masking	dynamic masking	backup masking	replica masking	FDW masking
PostgreSQL Anonymizer	SQL					
database anonymizer	YAML	-	-		-	-
greenmask	YAML	-	-		-	-
pg_anon	JSON	-	-		-	-
pg_anonymize	SQL	-			-	-
pg_diffix	SQL	-		-	-	-
pg-anonymizer	JS	-	-		-	-
pg-mask	SQL	-	-		-	-
pganonymize	YAML	-	-		-	-
pgantomizer	YAML	-	-		-	-
pgEdge Anonymizer	YAML		-	-	-	-
pgstream	YAML	-	-	-		-

Similar Implementations

- Dynamic Data Masking With MS SQL Server
- Citus : Using search_path and views to hide columns for reporting with Postgres
- MariaDB : Masking with maxscale

GDPR

- Ultimate Guide to Data Anonymization
- UK ICO Anonymisation Code of Practice
- L. Sweeney, Simple Demographics Often Identify People Uniquely, 2000
- How Google anonymizes data
- IAPP's Guide To Anonymisation

Concepts

- Differential_Privacy
- K-Anonymity

Academic Research

- L. Sweeney. k-anonymity: a model for protecting privacy. International Journal on Uncertainty, Fuzziness and Knowledge-based Systems, 10 (5), 2002, pp. 557-570. https://epic.org/wp-content/uploads/privacy/reidentification/Sweeney_Article.pdf
- A. Narayanan and V. Shmatikov, "Robust de-anonymization of large sparse datasets," in 29th IEEE Symposium on Security and Privacy, 2008, pp. 111–125. https://www.cs.cornell.edu/~shmat/shmat_oak08netflix.pdf — title: load draft: false toc: true —

Load the extension

Once the extension is installed on the host server, you need to **load** it into your PostgreSQL instance and then **create** it inside your database.

This page is the same whatever the installation method you used.

Step 1: Load the extension

The `anon` library must be preloaded by PostgreSQL. The **recommended** way is to load it globally with `shared_preload_libraries`:

```
ALTER SYSTEM SET shared_preload_libraries = 'anon';
```

Then **restart** the PostgreSQL instance.

This loads `anon` for **every backend**, including background workers. In particular it is **required for Replica Masking**, whose masking runs inside the logical-replication apply workers — and those workers do not read `session_preload_libraries`.

Alternative: `session_preload_libraries`

If you cannot restart the instance, or you want to enable `anon` on a per-database basis only, load it with `session_preload_libraries` instead:

```
ALTER DATABASE foo SET session_preload_libraries = 'anon';
```

(If you already load extensions that way, just add `anon` to the list.)

This method has its own benefits:

- it applies to the **next** connections, so **no restart** is needed — you just have to reconnect ;
- it is dumped by `pg_dump` with the `-C` option, so the database dump is self-contained ;
- it is propagated to a standby by streaming replication.

!!! warning

```
`session_preload_libraries` does **not** cover background workers, so it is
**not enough for [Replica Masking]**. Use `shared_preload_libraries` if you
need it.
```

Step 2: Create the extension

Close your session and open a new one (so the library is loaded), then create the extension inside your database:

```
CREATE EXTENSION anon;
SELECT anon.init();
```

`anon.init()` loads the dataset required by the `fake_*` faking functions.

All new connections to that database can now use the extension.

title: masking_data_wrappers draft: false toc: true —

Masking Data Wrappers

The principle of a masking data wrappers is to use Postgres as a "masking proxy" in front of any type of external data source. Using Foreign Data Wrappers, we can apply masking rules to data stored in CSV files, in another RDBM, in a NoSQL store, in a LDAP directory, etc.

Of course the remote data source can be another PostgreSQL instance !

Example

Here's a basic CSV file containing application logs

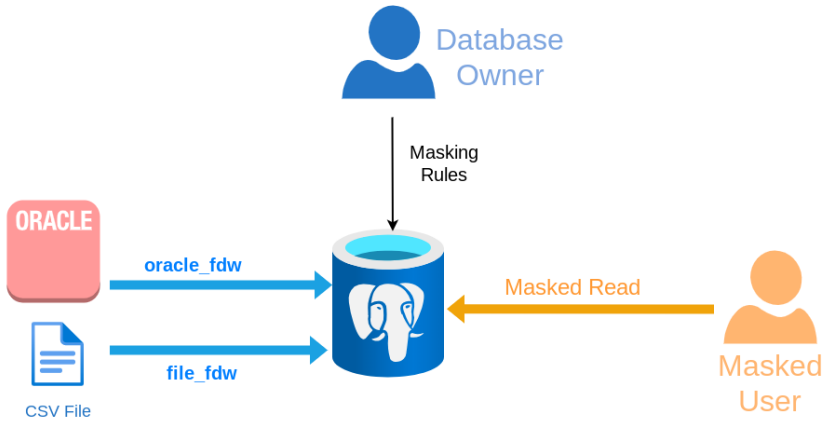


Figure 7: PostgreSQL Masking Data Wrappers

```
$ cat /tmp/app.log
Mon Nov 04 08:25:32 2024        sarah    10.0.0.45       view_dashboard
Mon Nov 04 09:15:00 2024        mike     172.16.0.89     update_profile
Mon Nov 04 09:30:45 2024        emma     192.168.2.200   download_report
[...]
```

Let's create a foreign table based on this file

```
CREATE EXTENSION IF NOT EXISTS file_fdw;

CREATE SERVER external_files FOREIGN DATA WRAPPER file_fdw;

CREATE SCHEMA files;

CREATE FOREIGN TABLE files.app_log
(
    tms   TIMESTAMP,
    login VARCHAR(255),
    ip INET,
    action TEXT
)
  SERVER external_files
  OPTIONS ( filename '/tmp/app.log' )
;
```

We can now declare masking rules on the columns of the foreign table, just like we would do for a regular table.

```
SECURITY LABEL FOR anon ON COLUMN files.app_log.login
  IS 'MASKED WITH VALUE $$CONFIDENTIAL$$';
```

```
SECURITY LABEL FOR anon ON COLUMN files.app_log.ip
  IS 'MASKED WITH FUNCTION anon.dummy_ipv4()';
```

… and that's it ! The masked users will now see the filtered data :

```
SET ROLE some_masked_user;

SELECT * FROM files.app_log LIMIT 1;
           tms            |    login     |      ip      |    action
--------------------------+--------------+--------------+---------------
 Mon Nov 04 08:23:15 2024 | CONFIDENTIAL | 85.249.91.21 | login_success
```

Or export the data to a new CSV file

```
SET ROLE some_masked_user;

COPY files.app_log TO '/tmp/anonymized_app.log'
```

Various Masking Strategies

The extension provides functions to implement 8 main anonymization strategies:

- Destruction
- Adding Noise
- Local Differential Privacy (LDP)
- Randomization
- Faking
- Advanced Faking
- Pseudonymization
- Generic Hashing
- Partial scrambling
- Conditional masking
- Generalization
- Using pg_catalog functions
- Image bluring
- Write your own Masks !

Depending on your data, you may need to use different strategies on different columns :

- For names and other 'direct identifiers' , Faking is often useful
- Shuffling is convenient for foreign keys
- Adding Noise is interesting for numeric values and dates
- Partial Scrambling is perfect for email address and phone numbers
- etc.

Destruction

First of all, the fastest and safest way to anonymize a data is to destroy it :-)

In many cases, the best approach to hide the content of a column is to replace all the values with a single static value.

For instance, you can replace a entire column by the word 'CONFIDENTIAL' like this:

```
SECURITY LABEL FOR anon
  ON COLUMN users.address
  IS 'MASKED WITH VALUE ''CONFIDENTIAL'' ';
```

Adding Noise

This is also called **Variance**. The idea is to "shift" dates and numeric values. For example, by applying a +/- 10% variance to a salary column, the dataset will remain meaningful.

- `anon.noise(original_value,ratio)` where original_value can be an `integer`, a `bigint` or a `double precision`. If the ratio is 0.33, the return value will be the original value randomly shifted with a ratio of +/- 33%
- `anon.dnoise(original_value, interval)` where original_value can be a date, a timestamp, or a time. If interval = '2 days', the return value will be the original value randomly shifted by +/- 2 days
- `anon.noisy_gps_coordinates(origin, radius, is_metric)` returns GPS coordinates located in a radius (default `10.0`) around a point. The radius can be expressed in kimometers (default) or miles.

  ```
  CREATE TABLE customer( .... , gps_location geometry(Point, 4326));

  SECURITY LABEL FOR anon ON COLUMN customer.gps_location
  IS 'MASKED WITH FUNCTION ST_SetSRID(
      noisy_gps_coordinates(gps_location::point, 10.0, true)::geometry,
      4326
  )';
  ```

WARNING : The `noise()` masking functions are vulnerable to a form of repeat attack, especially with Dynamic Masking. A masked user can guess an original value by requesting its masked value multiple times and then simply use the `AVG()` function to get a close approximation. (See `demo/noise_reduction_attack.sql` for more details). In a nutshell, these functions are best fitted for Anonymous Dumps and Static Masking. They should be avoided when using Dynamic Masking.

Local Differential Privacy (LDP)

Local Differential Privacy is a stronger approach to adding noise. Unlike the `noise()` functions above, LDP provides a formal mathematical guarantee: given the output, an observer cannot determine the original value with high confidence, no matter what auxiliary information they have. The strength of this guarantee is controlled by a parameter called **epsilon** – a smaller epsilon means stronger privacy but less accuracy.

This is particularly useful for **survey data** and **categorical values** (e.g. ratings, age brackets, answer choices) where you want to collect aggregate statistics while protecting individual responses.

GRRM (Generalized Randomized Response Mechanism)

The GRRM method works on categorical values represented as integers from 1 to `max_v`. Each value is either kept truthfully or replaced with a uniformly random alternative:

```
-- Perturb a rating (1-5) with epsilon = 1.0
SELECT anon.ldp_grrm(3, 1.0, 5);
```

You can also specify the privacy level using a "kept percentage" (PTTT) instead of epsilon. A kept percentage of 0.65 means roughly 65% of the differential probability is allocated to the true value:

```
SELECT anon.ldp_grrm_pttt(3, 0.65, 5);
```

To use this as a masking rule:

```
SECURITY LABEL FOR anon
  ON COLUMN survey.satisfaction_rating
  IS 'MASKED WITH FUNCTION anon.ldp_grrm(satisfaction_rating, 1.0, 5)';
```

Inspecting the privacy parameters

You can check the probabilities implied by your choice of epsilon:

```
-- Probability of keeping the true value
SELECT anon.ldp_truth_probability(1.0, 5);

-- Probability of each specific lie value
SELECT anon.ldp_lie_probability(1.0, 5);
```

Randomization

The extension provides a large choice of functions to generate purely random data :

Basic Random values

- `anon.random_date()` returns a date
- `anon.random_string(n)` returns a TEXT value containing `n` letters
- `anon.random_zip()` returns a 5-digit code
- `anon.random_phone(p)` returns a 8-digit phone with `p` as a prefix
- `anon.random_hash(seed)` returns a hash of a random string for a given seed
- `anon.random_gps_coordinates()` returns random GPS coordinates
- `anon.random_point_in_box(Box)` returns a random point in Box

Random between

To pick any value inside between two bounds:

- `anon.random_date_between(d1,d2)` returns a date between `d1` and `d2`
- `anon.random_int_between(i1,i2)` returns an integer between `i1` and `i2`
- `anon.random_bigint_between(b1,b2)` returns a bigint between `b1` and `b2`

NOTE: With these functions, the lower and upper bounds are included. For instance `anon.random_int_between(1,3)` returns either 1, 2 or 3.

For more advanced interval descriptions, check out the Random in Range section.

Random in Array

The `random_in` function returns an element a given array

For example:

- `anon.random_in(ARRAY[1,2,3])` returns an int between 1 and 3
- `anon.random_in(ARRAY['red','green','blue'])` returns a text

Random in Enum

This is one especially useful when working with `ENUM` types!

- `anon.random_in_enum(variable_of_an_enum_type)` returns any val

```
CREATE TYPE card AS ENUM ('visa', 'mastercard', 'amex');

SELECT anon.random_in_enum(NULL::CARD);
 random_in_enum
----------------
 mastercard

CREATE TABLE customer (
  id INT,
```

```
  ...
  credit_card CARD
);

SECURITY LABEL FOR anon ON COLUMN customer.creditcard
IS 'MASKED WITH FUNCTION anon.random_in_enum(creditcard)'
```

Random in Range

RANGE types are a powerful way to describe an interval of values, where can define inclusive or excluvive bounds:

https://www.postgresql.org/docs/current/rangetypes.html#RANGETYPES-EXAMPLES

There a function for each subtype of range:

- `anon.random_in_int4range('[5,6)')` returns an INT of value 5
- `anon.random_in_int8range('(6,7]')` returns a BIGINT of value 7
- `anon.random_in_numrange('[0.1,0.9]')` returns a NUMERIC between 0.1 and 0.9
- `anon.random_in_daterange('[2001-01-01, 2001-12-31)')` returns a date in 2001
- `anon.random_in_tsrange('[2022-10-01,2022-10-31]')` returns a TIMESTAMP in october 2022
- `anon.random_in_tstzrange('[2022-10-01,2022-10-31]')` returns a TIMESTAMP WITH TIMEZONE in october 2022

NOTE: It is not possible to get a random value from a RANGE with an infinite bound. For example `anon.random_in_int4range('[2022,)')` returns NULL.

Random Sequence ID

When masking a SERIAL columns it can be useful to general a `UNIQUE` value based on a sequence.

- `anon.random_id()` returns a BIGINT
- `anon.random_id_int()` returns a INT
- `anon.random_id_small_int()` returns a SMALLINT

Each call to these functions will return a incremented value much like the nextval() function.

At any time, you can reset the current sequence value with a new value. For instance:

```
SELECT pg_catalog.setval('anon.random_id_seq', 42);
```

Faking

The idea of **Faking** is to replace sensitive data with **random-but-plausible** values. The goal is to avoid any identification from the data record while remaining suitable for testing, data analysis and data processing.

In order to use the faking functions, you have to `init()` the extension in your database first:

```
SELECT anon.init();
```

The `init()` function will import a default dataset of random data (iban, names, cities, etc.).

> This dataset is in English and very small (1000 values for each category). If you want to use localized data or load a specific dataset, please read the Custom Fake Data section.

Once the fake data is loaded, you have access to these faking functions:

- `anon.fake_address()` returns a complete post address
- `anon.fake_city()` returns an existing city
- `anon.fake_country()` returns a country
- `anon.fake_company()` returns a generic company name
- `anon.fake_email()` returns a valid email address
- `anon.fake_first_name()` returns a generic first name
- `anon.fake_iban()` returns a valid IBAN
- `anon.fake_last_name()` returns a generic last name
- `anon.fake_postcode()` returns a valid zipcode
- `anon.fake_siret()` returns a valid SIRET

For TEXT and VARCHAR columns, you can use the classic Lorem Ipsum generator:

- `anon.lorem_ipsum()` returns 5 paragraphs
- `anon.lorem_ipsum(2)` returns 2 paragraphs
- `anon.lorem_ipsum( paragraphs := 4 )` returns 4 paragraphs
- `anon.lorem_ipsum( words := 20 )` returns 20 words
- `anon.lorem_ipsum( characters := 7 )` returns 7 characters
- `anon.lorem_ipsum( characters := anon.length(table.column) )` returns the same amount of characters as the original string

Advanced Faking

Generating fake data is a complex topic. The `fake_` functions provided above are limited to basic use case. For more advanced faking methods, in particular if you are looking for **localized fake data**, PostgreSQL Anonymizer provides an advanced faking engine with localisation support.

This engine (fake-rs) is available via more than 70 functions with the `dummy_` prefix:

tips:

The fake_* and dummy_* functions achieve the same goal.

The fake_* functions are the first implementation in pl/pgsql. They were introduced in Version 1. It's a rather naïve and limited approach.

The dummy_* functions are a new implementation based on a Rust library. It provides a more advanced fake generator and adds localization. It was introduced in Version 2.

New users should always prefer the dummy_* functions. The fake_* functions are kept for backward compatibility.

- anon.dummy_bic()
- anon.dummy_bs()
- anon.dummy_bs_adj()
- anon.dummy_bs_noun()
- anon.dummy_bs_verb()
- anon.dummy_building_number()
- anon.dummy_buzzword()
- anon.dummy_buzzword_middle()
- anon.dummy_buzzword_tail()
- anon.dummy_catchphrase()
- anon.dummy_cell_number()
- anon.dummy_city_name()
- anon.dummy_city_prefix()
- anon.dummy_city_suffix()
- anon.dummy_color()
- anon.dummy_company_name()
- anon.dummy_company_suffix()
- anon.dummy_country_code()
- anon.dummy_country_name()
- anon.dummy_credit_card_number()
- anon.dummy_currency_code()
- anon.dummy_currency_name()
- anon.dummy_currency_symbol()
- anon.dummy_dir_path()
- anon.dummy_domain_suffix()
- anon.dummy_file_extension()
- anon.dummy_file_name()
- anon.dummy_file_path()
- anon.dummy_first_name()
- anon.dummy_free_email()
- anon.dummy_free_email_provider()
- anon.dummy_health_insurance_code()
- anon.dummy_hex_color()
- anon.dummy_hsl_color()

- anon.dummy_hsla_color()
- anon.dummy_industry()
- anon.dummy_ip()
- anon.dummy_ipv4()
- anon.dummy_ipv6()
- anon.dummy_isbn()
- anon.dummy_isbn13()
- anon.dummy_isin()
- anon.dummy_last_name()
- anon.dummy_latitude()
- anon.dummy_licence_plate()
- anon.dummy_longitude()
- anon.dummy_mac_address()
- anon.dummy_name()
- anon.dummy_name_with_title()
- anon.dummy_phone_number()
- anon.dummy_post_code()
- anon.dummy_profession()
- anon.dummy_rfc_status_code()
- anon.dummy_rgb_color()
- anon.dummy_rgba_color()
- anon.dummy_safe_email()
- anon.dummy_secondary_address()
- anon.dummy_secondary_address_type()
- anon.dummy_state_abbr()
- anon.dummy_state_name()
- anon.dummy_street_name()
- anon.dummy_street_suffix()
- anon.dummy_suffix()
- anon.dummy_timezone()
- anon.dummy_title()
- anon.dummy_user_agent()
- anon.dummy_username()
- anon.dummy_uuidv1()
- anon.dummy_uuidv3()
- anon.dummy_uuidv4()
- anon.dummy_uuidv5()
- anon.dummy_valid_statux_code()
- anon.dummy_word()
- anon.dummy_words(int4range)
- anon.dummy_zip_code()

For each of this function, you can add the `_locale(...)` suffix and specify in which local context you want.

For example:

```
SELECT anon.dummy_last_name();
 dummy_last_name
-------------------------
 Tillman

SELECT anon.dummy_last_name_locale('fr_FR');
 dummy_last_name_locale
-------------------------
 Granier

SELECT anon.dummy_last_name_locale('pt_BR');
 dummy_last_name_locale
-------------------------
 Barreto
```

Currently 7 locales are available: ar_SA, en_US(default), fr_FR, ja_JP, pt_BR, zh_CN, zh_TW.

Not that some `dummy_` functions are not implemented for certain locales. If you wish to contribute or ask for missing fake data, please contact directly the fake-rs project, which is the library that this extension is using under the hood !

Pseudonymization

Pseudonymization is similar to Faking in the sense that it generates realistic values. The main difference is that the pseudonymization is deterministic : the functions always will return the same fake value based on a seed and an optional salt.

In order to use the faking functions, you have to `init()` the extension in your database first:

```
SELECT anon.init();
```

Once the fake data is loaded you have access to 10 pseudo functions:

- `anon.seeded_first_name(seed,locale,salt)` returns a generic first name
- `anon.seeded_last_name(seed,locale,salt)` returns a generic last name
- `anon.seeded_email(seed,locale,salt)` returns a valid email address
- `anon.seeded_city(seed,locale,salt)` returns an existing city
- `anon.seeded_country(seed,locale,salt)` returns a country
- `anon.seeded_company(seed,locale,salt)` returns a generic company name
- `anon.seeded_iban(seed,locale,salt)` returns a valid IBAN
- `anon.seeded_siret(seed,locale,salt)` returns a valid SIRET

The second argument (`local`) and the third (`salt`) are optional. You can call each function with only the seed like this `anon.seeded_city('bob')` or with a given locale `anon.seeded_city('bob','en_US')` or with a locale and a salt anon.seeded_city('bob','en_US', 'xyz')'

The `locale` defines the language of the fake data produced. The default locale is `en_US`. **There is no other preloaded locale**. If you want to add another locale, see the Custom Fake Data chapter. If there's no custom fake data loaded for the given locale, the function will return `NULL`.

The `salt` is here to increase complexity and avoid dictionary and brute force attacks (see warning below). If a specific salt is not given, the value of the `anon.salt` GUC parameter is used instead (see the Generic Hashing section for more details).

The seed can be any information related to the subject. For instance, we can consistently generate the same fake email address for a given person by using her login as the seed :

```
SECURITY LABEL FOR anon
  ON COLUMN users.emailaddress
  IS 'MASKED WITH FUNCTION anon.seeded_email(users.login) ';
```

!!! note "Pseudonymization and Unicity"

```
You may want to produce unique values using a pseudonymization
function. For instance, if you want to mask an `email` column that is declared
as `UNIQUE`. In this case, you will need to initialize the extension with a fake
dataset that is **way bigger** than the numbers of rows of the table. Otherwise you
may see some "collisions" happening, i.e. two different original values producing
the same pseudo value.
```

OBSOLETE: pseudo_* functions

Until versions 3.2, the pseudonymization functions were named `pseudo_address(...)`, `pseudo_city(...)`. They are now replaced by the `seeded_*` functions described above.

The

Pseudonymizing primary keys

It is also possible to pseudonymize a primary key using:

- `anon.pseudo_shift(id)` returns a shifted version of the id
- `anon.pseudo_xor(id)` returns an exclusive OR value of the id

Both `anon.pseudo_shift(BIGINT)` and `anon.pseudo_xor(BIGINT)` functions use a secret value (`anon.shift`) to pseudonymize the primary key. That secret value can be initialized randomly with `anon.set_shift()` or defined with `anon.set_shift(INT)`.

This is very useful to replace `anon.random_id()` when using Backup Masking.

Pseudonymization IS NOT Anonymization

Pseudonymization is often confused with anonymization but in fact they serve 2 different purposes : `pseudonymization` is a way to **protect** the personal information but the pseudonymized data is still "linked" to the real data. The GDPR makes it very clear that personal data which has undergone pseudonymization is still related to a person. (see GDPR Recital 26)

Generic hashing

Hashing is another pseudonymization technique (see **WARNING** above). In practice it is sometimes useful to generate a determinist hash of the original data.

For instance, when a pair of primary key / foreign key is a "natural key", it may contain actual information (like a customer number containing a birth date or something similar).

Hashing such columns allows to keep referential integrity intact even for relatively unusual source data. Therefore, the

- `anon.digest(value,salt,algorithm)` lets you choose a salt, and a hash algorithm from a pre-defined list
- `anon.hash(value)` will return a text hash of the value using a secret salt (defined by the `anon.salt` parameter) and hash algorithm (defined by the `anon.algorithm` parameter). The default value of `anon.algorithm` is `sha256` and possible values are: md5, sha224, sha256, sha384 or sha512. The default value of `anon.salt` is an empty string. You can modify these values with:

  ```
  ALTER DATABASE foo SET anon.salt TO 'xsfnjefnjsnfjsnf';
  ALTER DATABASE foo SET anon.algorithm TO 'sha384';
  ```

Keep in mind that hashing is a form a Pseudonymization. This means that the data can be "de-anonymized" using the hashed value and the masking function. If an attacker gets access to these 2 elements, he or she could re-identify some persons using `brute force` or `dictionary` attacks. Therefore, **the salt and the algorithm used to hash the data must be protected with the same level of security that the original dataset.**

In a nutshell, we recommend that you use the `anon.hash()` function rather than `anon.digest()` because the salt will not appear clearly in the masking rule.

Furthermore: in practice the hash function will return a long string of character like this:

```
SELECT anon.hash('bob');
                                    hash
```

```
95b6accef02c5a725a8c9abf19ab5575f99ca3d9997984181e4b3f81d96cbca4d0977d694ac490350e01d0d21363
```

For some columns, this may be too long and you may have to cut some parts the hash in order to fit into the column. For instance, if you have a foreign key based on a phone number and the column is a VARCHAR(12) you can transform the data like this:

```
SECURITY LABEL FOR anon ON COLUMN people.phone_number
IS 'MASKED WITH FUNCTION anon.left(anon.hash(phone_number),12)';

SECURITY LABEL FOR anon ON COLUMN call_history.fk_phone_number
IS 'MASKED WITH FUNCTION anon.left(anon.hash(fk_phone_number),12)';
```

Of course, cutting the hash value to 12 characters will increase the risk of "collision" (2 different values having the same fake hash). In such case, it's up to you to evaluate this risk.

WARNING: The hashing functions will fail when the input contains an unescaped character (especially a single backslash). In most situation, this is the sign of a bug in the application, generally when data input is not sanitized properly. Users who really want to mask unescaped characters with this function should disable the `standard_conforming_strings` parameter. See Issue 539 for more details.

Partial Scrambling

Partial scrambling leaves out some part of the data. For instance : a credit card number can be replaced by '40XX XXXX XXXX XX96'.

2 functions are available:

- `anon.partial('abcdefgh',1,'xxxx',3)` will return 'axxxxfgh';
- `anon.partial_email('daamien@gmail.com')` will become 'da******@gm******.com'

You can also remove items from an array with a regexp:

- `anon.array_remove_regex(disease,'(?i)(hiv|aids)')`

Conditional Masking

In some situations, you may want to apply a masking filter only for some value or for a limited number of lines in the table.

For instance, if you want to "preserve NULL values", i.e. masking only the lines that contains a value, you can use the `anon.ternary` function, which works like a `CASE WHEN x THEN y ELSE z` statement:

```
SECURITY LABEL FOR anon ON COLUMN player.score
  IS 'MASKED WITH FUNCTION anon.ternary(score IS NULL,
```

```
                                        NULL,
                                        anon.random_int_between(0,100));
```

You can also use the `anon.ternary` function to keep a ratio of NULL values in the otherwise anonymized data like in the following example where each line as as 10% chance to be a NULL value:

```
SECURITY LABEL FOR anon ON COLUMN player.score
  IS 'MASKED WITH FUNCTION anon.ternary(pg_catalog.random() <= .1,
                                        NULL,
                                        anon.random_int_between(0,100));
```

You may also want to exclude some lines within the table. Like keeping the password of some users so that they still may be able to connect to a testing deployment of your application:

```
SECURITY LABEL FOR anon ON COLUMN account.password
  IS 'MASKED WITH FUNCTION anon.ternary( id > 1000, NULL::TEXT, password)';
```

WARNING : Conditional masking may create a partially deterministic "connection" between the original data and the masked data. And that connection can be used to retrieve personal information from the masked data. For instance, if NULL values are preserved for a "deceased_date" column, it will reveal which persons are still actually alive… In a nutshell: conditional masking may often produce a dataset that is not fully anonymized and therefore would still technically contain personal information.

Generalization

Generalization is the principle of replacing the original value by a range containing this value. For instance, instead of saying 'Paul is 42 years old', you would say 'Paul is between 40 and 50 years old'.

> The generalization functions are a data type transformation. Therefore it is not possible to use them with the dynamic masking engine. However they are useful to create anonymized views. See example below.

Let's imagine a table containing health information:

```
SELECT * FROM patient;
 id |   name   |  zipcode |   birth    |    disease
----+----------+----------+------------+---------------
  1 | Alice    |    47678 | 1979-12-29 | Heart Disease
  2 | Bob      |    47678 | 1959-03-22 | Heart Disease
  3 | Caroline |    47678 | 1988-07-22 | Heart Disease
  4 | David    |    47905 | 1997-03-04 | Flu
  5 | Eleanor  |    47909 | 1999-12-15 | Heart Disease
  6 | Frank    |    47906 | 1968-07-04 | Cancer
  7 | Geri     |    47605 | 1977-10-30 | Heart Disease
```

```
  8 | Harry    |    47673 | 1978-06-13 | Cancer
  9 | Ingrid   |    47607 | 1991-12-12 | Cancer
```

We can build a view upon this table to suppress some columns (`SSN` and `name`) and generalize the zipcode and the birth date like this:

```
CREATE VIEW anonymized_patient AS
SELECT
    'REDACTED' AS lastname,
    anon.generalize_int4range(zipcode,100) AS zipcode,
    anon.generalize_tsrange(birth,'decade') AS birth
    disease
FROM patients;
```

The anonymized table now looks like that:

```
SELECT * FROM anonymized_patient;
 lastname |   zipcode     |           birth              |    disease
----------+---------------+------------------------------+--------------
 REDACTED | [47600,47700) | ["1970-01-01","1980-01-01") | Heart Disease
 REDACTED | [47600,47700) | ["1950-01-01","1960-01-01") | Heart Disease
 REDACTED | [47600,47700) | ["1980-01-01","1990-01-01") | Heart Disease
 REDACTED | [47900,48000) | ["1990-01-01","2000-01-01") | Flu
 REDACTED | [47900,48000) | ["1990-01-01","2000-01-01") | Heart Disease
 REDACTED | [47900,48000) | ["1960-01-01","1970-01-01") | Cancer
 REDACTED | [47600,47700) | ["1970-01-01","1980-01-01") | Heart Disease
 REDACTED | [47600,47700) | ["1970-01-01","1980-01-01") | Cancer
 REDACTED | [47600,47700) | ["1990-01-01","2000-01-01") | Cancer
```

The generalized values are still useful for statistics because they remain true, but they are less accurate, and therefore reduce the risk of re-identification.

PostgreSQL offers several RANGE data types which are perfect for dates and numeric values.

For numeric values, 3 functions are available:

- `generalize_int4range(value, step)`
- `generalize_int8range(value, step)`
- `generalize_numrange(value, step)`

...where `value` is the data that will be generalized, and `step` is the size of each range.

Using `pg_catalog` functions

Since version 1.3, the `pg_catalog` schema is not trusted by default. This is a security measure designed to prevent users from using sophisticated functions inside masking rules (such as `pg_catalog.query_to_xml`, `pg_catalog.ts_stat`

or the system administration functions) that should not be used as masking functions.

However, the extension allows using some useful and safe functions from the `pg_catalog` schema for your convenience. These are small subset of functions that are declared as `TRUSTED` for anonymization.

The list of TRUSTED pg_catalog functions is available via the `anon.pg_trusted_functions` views :

```
SELECT * FROM anon.pg_trusted_functions;
```

If you need to use a pg_catalog function which is not in this list, you can ask a superuser to trust it with:

```
SECURITY LABEL FOR anon ON FUNCTION pg_catalog.foo IS 'TRUSTED';
```

> Note: Even when multiple masking policies are defined, the functions must be declared as `TRUSTED` in the "anon" policy and they will be trusted for all policies.

Image bluring

Images can show some sensitive data, for example

- A photo concerning personal data.
- A barcode representing personal data.

it is possible to blur this image using

- `anon.image_blur(data,sigma)` returns a bytea
- data type `bytea`: the image data
- sigma type `numeric` (default 10.0): This parameter controls the amount of blurring. A higher sigma value results in a more blurred image, while a lower sigma value results in a less blurred image.

usage :

```
CREATE TABLE images (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    image_data BYTEA NOT NULL
);

SECURITY LABEL FOR anon ON COLUMN images.image_data
IS 'MASKED WITH FUNCTION anon.image_blur(image_data,1.0)';
```

Write your own Masks !

You can also use your own function as a mask. The function must either be destructive (like Partial Scrambling) or insert some randomness in the dataset

(like Faking).

Especially for complex data types, you may have to write your own function. This will be a common use case if you have to hide certain parts of a JSON field.

For example:

```
CREATE TABLE company (
  business_name TEXT,
  info JSONB
)
```

The `info` field contains unstructured data like this:

```
SELECT jsonb_pretty(info) FROM company WHERE business_name = 'Soylent Green';
           jsonb_pretty
---------------------------------
 {
     "employees": [
         {
             "lastName": "Doe",
             "firstName": "John"
         },
         {
             "lastName": "Smith",
             "firstName": "Anna"
         },
         {
             "lastName": "Jones",
             "firstName": "Peter"
         }
     ]
 }
(1 row)
```

NOTE: Function with SQL Standard body bypass anonymization, therefore they cannot be used with transparent dynamic masking expect for those in the the `catalog` and `anon` schemas.

Using the PostgreSQL JSON functions and operators, you can walk through the keys and replace the sensitive values as needed.

```
CREATE SCHEMA custom_masks;

CREATE FUNCTION custom_masks.remove_last_name(j JSONB)
RETURNS JSONB
VOLATILE
LANGUAGE SQL
AS $func$
```

```
SELECT
  json_build_object(
    'employees' ,
    array_agg(
      jsonb_set(e ,'{lastName}', to_jsonb(anon.fake_last_name()))
    )
  )::JSONB
FROM jsonb_array_elements( j->'employees') e
$func$;

-- This step requires superuser privilege
SECURITY LABEL FOR anon ON FUNCTION custom_masks.remove_last_name IS 'TRUSTED';
```

Then check that the function is working correctly:

```
SELECT custom_masks.remove_last_name(info) FROM company;
```

When that's ok you can declare this function as the mask of the `info` field:

```
SECURITY LABEL FOR anon ON COLUMN company.info
IS 'MASKED WITH FUNCTION custom_masks.remove_last_name(info)';
```

And try it out !

```
# SELECT anonymize_table('company');
# SELECT jsonb_pretty(info) FROM company WHERE business_name = 'Soylent Green';
            jsonb_pretty
--------------------------------------
 {                                    +
     "employees": [                   +
         {                            +
             "lastName": "Prawdzik",  +
             "firstName": "John"      +
         },                           +
         {                            +
             "lastName": "Baltazor",  +
             "firstName": "Anna"      +
         },                           +
         {                            +
             "lastName": "Taylan",    +
             "firstName": "Peter"     +
         }                            +
     ]                                +
 }
(1 row)
```

This is just a quick and dirty example. As you can see, manipulating a sophisticated JSON structure with SQL is possible, but it can be tricky at first! There are multiple ways of walking through the keys and updating values. You will

probably have to try different approaches, depending on your real JSON data and the performance you want to reach. — title: masking_views draft: false toc: true —

Masking Views

The principe of a masking view is simply to build dedicated interface upon a table. This is useful when the masking policy needs to modify the database model.

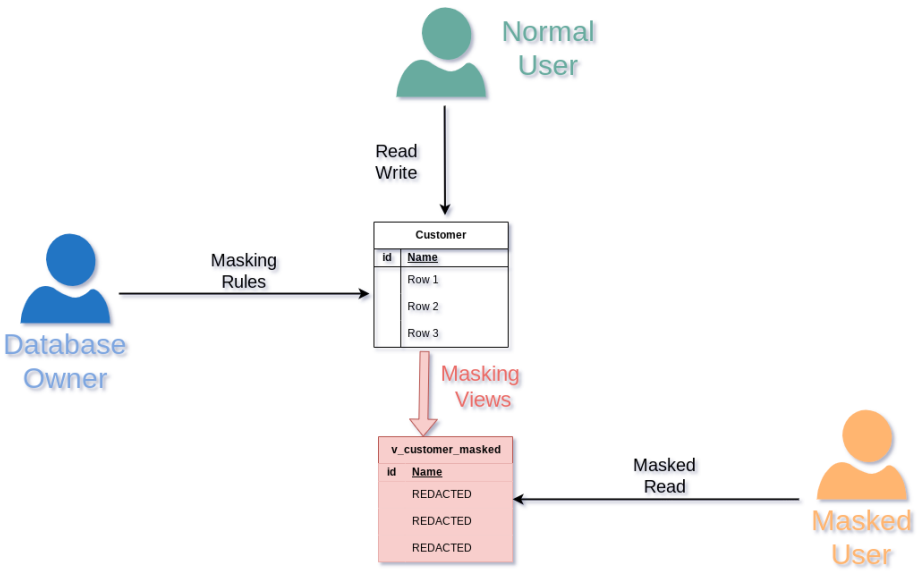


Figure 8: PostgreSQL Masking Views

Generalization

The idea of generalization is to replace data with a broader, less accurate value. For instance, instead of saying "Bob is 28 years old", you can say "Bob is between 20 and 30 years old". This is interesting for analytics because the data remains true while avoiding the risk of re-identification.

Generalization is a way to achieve k-anonymity.

PostgreSQL can handle generalization very easily with the RANGE data types, a very powerful way to store and manipulate a set of values contained between a lower and an upper bound.

Example

Here's a basic table containing medical data:

```
# SELECT * FROM confidential.patient;
     ssn      | firstname | zipcode |   birth    |    disease
--------------+-----------+---------+------------+---------------
 253-51-6170 | Alice     |   47012 | 1989-12-29 | Heart Disease
 091-20-0543 | Bob       |   42678 | 1979-03-22 | Allergy
 565-94-1926 | Caroline  |   42678 | 1971-07-22 | Heart Disease
 510-56-7882 | Eleanor   |   47909 | 1989-12-15 | Acne
 098-24-5548 | David     |   47905 | 1997-03-04 | Flu
 118-49-5228 | Jean      |   47511 | 1993-09-14 | Flu
 263-50-7396 | Tim       |   47900 | 1981-02-25 | Heart Disease
 109-99-6362 | Bernard   |   47168 | 1992-01-03 | Asthma
 287-17-2794 | Sophie    |   42020 | 1972-07-14 | Asthma
 409-28-2014 | Arnold    |   47000 | 1999-11-20 | Diabetes
(10 rows)
```

We want the anonymized data to remain **true** because it will be used for statistics. We can build a view upon this table to remove useless columns and generalize the indirect identifiers :

```
CREATE SCHEMA stats;

CREATE MATERIALIZED VIEW stats.generalized_patient AS
SELECT
  'REDACTED'::TEXT AS firstname,
  anon.generalize_int4range(zipcode,1000) AS zipcode,
  anon.generalize_daterange(birth,'decade') AS birth,
  disease
FROM confidential.patient;
```

This will give us a less accurate view of the data:

```
# SELECT * FROM generalized_patient;
 firstname |    zipcode    |          birth          |    disease
-----------+---------------+-------------------------+---------------
 REDACTED  | [47000,48000) | [1980-01-01,1990-01-01) | Heart Disease
 REDACTED  | [42000,43000) | [1970-01-01,1980-01-01) | Allergy
 REDACTED  | [42000,43000) | [1970-01-01,1980-01-01) | Heart Disease
 REDACTED  | [47000,48000) | [1980-01-01,1990-01-01) | Acne
 REDACTED  | [47000,48000) | [1990-01-01,2000-01-01) | Flu
 REDACTED  | [47000,48000) | [1990-01-01,2000-01-01) | Flu
 REDACTED  | [47000,48000) | [1980-01-01,1990-01-01) | Heart Disease
 REDACTED  | [47000,48000) | [1990-01-01,2000-01-01) | Asthma
 REDACTED  | [42000,43000) | [1970-01-01,1980-01-01) | Asthma
 REDACTED  | [47000,48000) | [1990-01-01,2000-01-01) | Diabetes
```

```
(10 rows)
```

Now we can give read access only to the masking views for a given user:

```
CREATE USER bob;

REVOKE USAGE          ON SCHEMA confidential                FROM bob;
REVOKE ALL PRIVILEGES ON ALL TABLES IN SCHEMA confidential  FROM bob;
GRANT  USAGE          ON SCHEMA stats                       TO bob;
GRANT  SELECT         ON ALL TABLES IN SCHEMA stats         TO bob;
```

Generalization Functions

PostgreSQL Anonymizer provides 6 generalization functions. One for each RANGE type. Generally these functions take the original value as the first parameter, and a second parameter for the length of each step.

For numeric values :

- `anon.generalize_int4range(42,5)` returns the range `[40,45)`
- `anon.generalize_int8range(12345,1000)` returns the range `[12000,13000)`
- `anon.generalize_numrange(42.32378,10)` returns the range `[40,50)`

For time values :

- `anon.generalize_tsrange('1904-11-07','year')` returns `['1904-01-01','1905-01-01')`
- `anon.generalize_tstzrange('1904-11-07','week')` returns `['1904-11-07','1904-11-14')`
- `anon.generalize_daterange('1904-11-07','decade')` returns `[1900-01-01,1910-01-01)`

The possible steps are : microseconds, milliseconds, second, minute, hour, day, week, month, year, decade, century and millennium.

Limitations

Singling out and extreme values

"Singling Out" is the possibility to isolate an individual in a dataset by using extreme value or exceptional values.

For example:

```
# SELECT * FROM employees;

  id  |  name          | job  | salary
------+----------------+------+--------
 1578 | xkjefus3sfzd   | NULL |   1498
 2552 | cksnd2se5dfa   | NULL |   2257
 5301 | fnefckndc2xn   | NULL |  45489
 7114 | npodn5ltyp3d   | NULL |   1821
```

In this table, we can see that a particular employee has a very high salary, very far from the average salary. Therefore this person is probably the CEO of the company.

With generalization, this is important because the size of the range (the "step") must be wide enough to prevent the identification of one single individual.

k-anonymity is a way to assess this risk.

Generalization is not compatible with dynamic masking

By definition, with generalization the data remains true, but the column type is changed.

This means that the transformation is not transparent, and therefore it cannot be used with dynamic masking.

k-anonymity

k-anonymity is an industry-standard term used to describe a property of an anonymized dataset. The k-anonymity principle states that within a given dataset, any anonymized individual cannot be distinguished from at least `k-1` other individuals. In other words, k-anonymity might be described as a "hiding in the crowd" guarantee. A low value of `k` indicates there's a risk of re-identification using linkage with other data sources.

You can evaluate the k-anonymity factor of a table in 2 steps :

Step 1: First define the columns that are indirect identifiers (also known as quasi identifiers) like this:

```
SECURITY LABEL FOR k_anonymity ON COLUMN patient.firstname
IS 'INDIRECT IDENTIFIER';

SECURITY LABEL FOR k_anonymity ON COLUMN patient.zipcode
IS 'INDIRECT IDENTIFIER';

SECURITY LABEL FOR k_anonymity ON COLUMN patient.birth
IS 'INDIRECT IDENTIFIER';
```

Step 2: Once the indirect identifiers are declared :

```
SELECT anon.k_anonymity('generalized_patient')
```

The higher the value, the better…

References

-

How Google Anonymizes Data

title: performances draft: false toc: true —

Performances

Any anonymization process has a price as it will consume CPU time, RAM space and probably a bunch of disk I/O… Here's a a quick overview of the question depending on what strategy you are using….

In a nutshell, the anonymization performances will mainly depend on 2 important factors:

- The size of the database
- The number of masking rules

Static Masking

Basically what static masking does it rewrite entirely the masked tables on disk. This may be slow depending on your environment. And during this process, the tables will be locked.

On large tables with several secondary indexes, you can speed this up by dropping and rebuilding them around the masking, with `anon.static_masking.drop_indexes` (see Dropping indexes).

As an example: Anonymizing a 44GB database with 29 masking rules on an AWS EC2 instance takes approximately 25 minutes (see MR 107 for more details).

> In this case, the cost of anonymization is "paid" by all the users but it is paid **once and for all**.

Dynamic Masking

With dynamic masking, the real data is replaced on-the-fly **every time** a masked user sends a query to the database. This means that the masking users will have slower response time than regular (unmasked) users. This is generally ok because usually masked users are not considered as important as the regular ones.

If you apply 3 or 4 rules to a table, the response time for the masked users should approx. 20% to 30% slower than for the normal users.

As the masking rules are applied for each queries of the masked users, the dynamic masking is appropriate when you have a limited number of masked users that connect only from time to time to the database. For instance, a data analyst connecting once a week to generate a business report.

If there are multiple masked users or if a masked user is very active, you should probably export the masked data once-a-week on a secondary instance and let these users connect to this secondary instance.

> In this case, the cost of anonymization is "paid" only by the masked users.

Anonymous Dumps

Some benchmarks made in march 2022 suggest that the `pg_dump_anon` wrapper is twice as slow as the regular `pg_dump` tool.

If the backup process of your database takes 1 hour with `pg_dump`, then anonymizing and exporting the entire database with `pg_dump_anon` will probably take 2 hours.

> In this case, the cost of anonymization is "paid" by the user asking for the anonymous export. Other users of the database will not be affected.

How to speed things up ?

Prefer `MASKED WITH VALUE` whenever possible

It is always faster to replace the original data with a static value instead of calling a masking function.

Sampling

If you need to anonymize data for testing purpose, chances are that a smaller subset of your database will be enough. In that case, you can easily speed up the anonymization by downsizing the volume of data.

Checkout the Sampling section for more details.

Materialized Views

Dynamic masking is not always required! In some cases, it is more efficient to build Materialized Views instead.

For instance:

```
CREATE MATERIALIZED VIEW masked_customer AS
SELECT
    id,
    anon.random_last_name() AS name,
    anon.random_date_between('1920-01-01'::DATE,now()) AS birth,
    fk_last_order,
    store_id
FROM customer;
```

Materialized Views: https://www.postgresql.org/docs/current/static/sql-creatematerializedview.html

title: privacy_by_default draft: false toc: true —

Privacy By Default

Principle

The GDPR regulation (and other privacy laws) introduces the concept of data protection by default. In a nutshell, it means that **by default**, organisations should ensure that data is processed with the highest privacy protection so that by default personal data isn't made accessible to an indefinite number of persons.

By applying this principle to anonymization, we end up with the idea of **privacy by default** which basically means that all columns of all tables should be masked by default, without having to declare a masking rule for each of them.

To enable this feature, simply set the option `anon.privacy_by_default` to `on`.

Example

Imagine a database named `foo` with a basic table containing HTTP logs:

```
# SELECT * FROM access_logs LIMIT 1;
      date_open      |     ip_addr     |    url     |        browser_agent
---------------------+-----------------+------------+-----------------------------
 2009-01-08 00:00:00 | 192.168.100.128 | /home.html | Mozilla/5.0 (Windows; en_US)
(1 row)
```

Now let's activate privacy by default:

```
ALTER DATABASE foo SET anon.privacy_by_default = True;
```

> The setting will be applied for the next sessions, i.e. **You need to reconnect to the database for the change to visible**

We can now anonymize the table without writing any masking rule.

```
# SELECT anon.anonymize_database();
 anonymize_database
--------------------
 t

# SELECT * FROM access_logs LIMIT 1;
 date_open | ip_addr | url | browser_agent
-----------+---------+-----+---------------
           |         |     | unknown
```

Unmasking columns

As we can see, when the `anon.privacy_by_default` is defined all the values will be replaced by the column's default value or NULL. The entire dataset is destroyed.

Now instead of writing rules to mask the sensible columns, we will write rules to **unmask** the ones we want to allow.

For instance, let's say that we want to keep the authentic value of the `url` field, we can simply "unmask" the column like this:

```
SECURITY LABEL FOR anon ON COLUMN access_logs.url
IS 'NOT MASKED';
```

This can also be achieved by a masking rule that will replace the value with itself:

```
SECURITY LABEL FOR anon ON COLUMN access_logs.url
IS 'MASKED WITH VALUE url';
```

Now we'd like to unmask the `date_open` field in the anonymized dataset but we need to generalize the dates to keep only the year:

```
SECURITY LABEL FOR anon ON COLUMN access_logs.date_open
IS 'MASKED WITH FUNCTION make_date(EXTRACT(year FROM date_open)::INT,1,1)';
```

Caveat: Add a DEFAULT to the NOT NULL columns

It is a bit ironic that the `anon.privacy_by_default` parameter **is not** enabled by default. This reason is simple: activating this option **may or may not** lead to constraint violations depending on the columns constraints placed in the database model.

Let's say we want to add a `NOT NULL` constraint on the `date_open` column:

```
ALTER TABLE public.access_logs
  ALTER COLUMN date_open
  SET NOT NULL;
```

Now if we try to anonymize the table, we get the following violation:

```
SELECT anon.anonymize_table('public.access_logs') as test4;
ERROR:  Cannot mask a "NOT NULL" column with a NULL value
HINT:  If privacy_by_design is enabled, add a default value to the column
```

The solution here is simply to define a default value and this value will be used for the `privacy_by_default` mechanism.

```
ALTER TABLE public.access_logs
  ALTER COLUMN date_open
  SET DEFAULT now();
```

Other constraints (foreign keys, UNIQUE, CHECK, etc.) should work fine without a DEFAULT value. — title: replica_masking draft: false toc: true —

Anonymous Replica

Principle

In some situations, you may want to have an anonymized copy of your production database on another instance like with Backup Masking (aka "Anonymized Dumps") but you also would like this copy to be up-to-date with the original data like with Dynamic Masking…

With the Replica Masking feature, you can use PostgreSQL logical replication to create an anonymized clone of your production database.

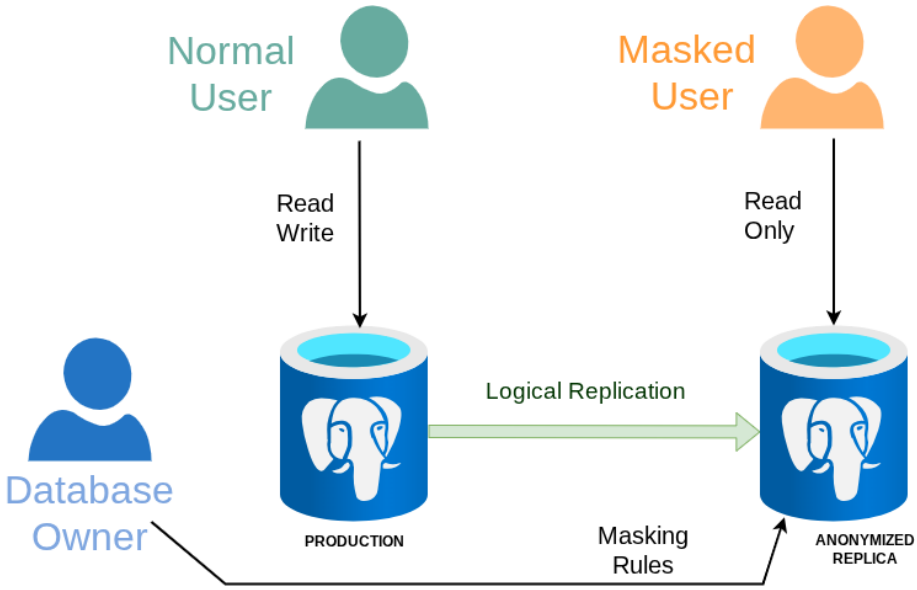


Figure 9: PostgreSQL Replica Masking

Preamble: Learn about logical replication !

PostgreSQL logical replication is a powerful mechanism. Before setting up a anonymous replica, be sure that you are able to configure standard logical replication correctly.

There are many tutorials available for that and we also recommend reading the PostgreSQL manual:

https://www.postgresql.org/docs/current/logical-replication.html

Quick Setup

Example

Let's say we want to anonymize a table `person` in a database `foo` like this:

```
CREATE TABLE person (
  id SERIAL PRIMARY KEY,
  name TEXT,
  company TEXT
);


INSERT INTO person VALUES (1, 'Alice', 'CompanyA');
INSERT INTO person VALUES (2, 'Bob', 'CompanyB');
INSERT INTO person VALUES (3, 'Charlie', 'CompanyC');
INSERT INTO person VALUES (4, 'David', 'CompanyD');
INSERT INTO person VALUES (5, 'Eve', 'CompanyE');
```

A- On the publisher database

A1- Create a replication role:

```
CREATE ROLE anon_replicator LOGIN REPLICATION PASSWORD 'CHANGE-ME-3747';
GRANT USAGE  ON SCHEMA public TO anon_replicator;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO anon_replicator;

-- Schema used to host the function used by the replica triggers
CREATE SCHEMA anon_replica;
GRANT SELECT ON ALL TABLES IN SCHEMA anon_replica TO anon_replicator;
```

Be sure to configure your `pg_hba.conf` file to allow `anon_replicator` to connect from the subscriber database.

A2- Create a publication:

```
CREATE PUBLICATION pub FOR TABLE person;
```

All of this is pretty standard. There's nothing special regarding anonymization on the publisher database. In fact, the publisher database "does not know" that the data will be masked on the subscriber.

B- On the subscriber database

B1- Create the table (DDL commands are NOT replicated):

```
CREATE TABLE person (
  id SERIAL PRIMARY KEY,
  name TEXT,
  company TEXT
);
```

B2- Enable replica masking:

```
ALTER DATABASE foo SET anon.replica_masking TO on;
ALTER DATABASE foo SET anon.replica_masking.schema TO 'anon_replica';
```

B3- Reconnect to the database so that the configuration is applied.

B4- Define the masking rules:

```
SECURITY LABEL FOR anon ON COLUMN person.company
  IS 'MASKED WITH FUNCTION pg_catalog.md5(company)';

SECURITY LABEL FOR anon ON COLUMN person.name
  IS 'MASKED WITH FUNCTION anon.dummy_first_name()';
```

B5- start the replica masking engine:

```
SELECT anon.start_replica_masking();
```

B6- Create the subscription:

```
CREATE SUBSCRIPTION anon_sub
CONNECTION 'host=prod_srv user=anon_replicator password=CHANGE-ME-3747 dbname=foo'
PUBLICATION pub;
```

Wait for a few milliseconds while the data is being synchronized and masked...

Et voilà !

```
SELECT * FROM person;
```

```
 id |   name    |             company
----+-----------+----------------------------------
  1 | Christine | a1e551387ba94e882ccc5356948d6462
  2 | Percival  | 75b4e152a05dae2f1d7991182e707fad
  3 | Ignatius  | e2a211f97064ee5a86853ae61e1bb2b9
  4 | Karley    | 8d543957c23828bb0d888cf7da59a817
  5 | Alfredo   | 566ca1969819cbf2098202255914bf23
```

Changing the masking rules

Anytime you add or remove a masking rule, you need to update the replica masking engine.

```
SELECT anon.refresh_replica_masking();
```

Anonymized Standby

In complement to Replica Masking, it is possible to use Hot Standby replication to build a distant clone of the Anonymized Replica. This is useful to export the database to a remote datacenter because the Anonymized Replica will operate as

a masking proxy, “cleaning” the personal information before it gets transferred to the Standby instance.

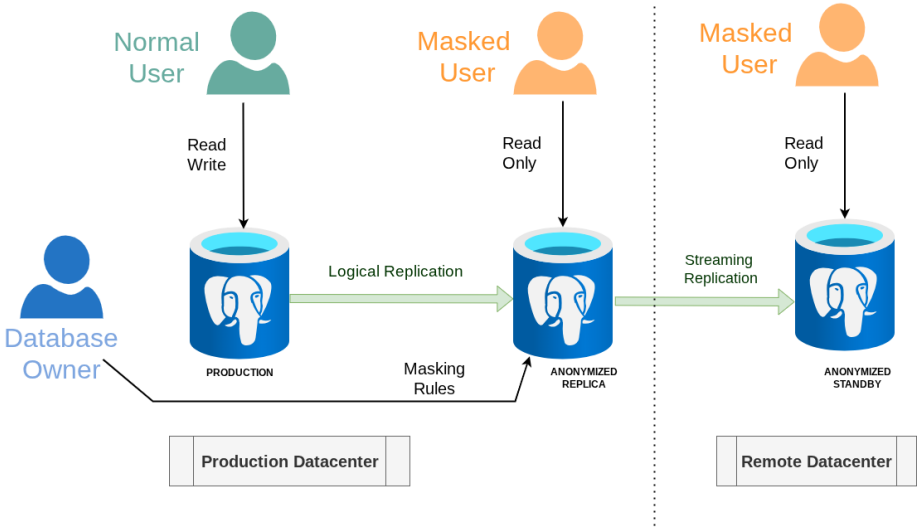


Figure 10: PostgreSQL Standby Masking

Security

Keep in mind that the masking rules are applied on-the-fly in the subscriber database, which means:

- The original data is transferred through the connection between the publisher and the subscriber. Therefore this connection should be protected like in a regular logical replication setup.
- The superuser of the subscriber instance and the owner of the subscriber database can disable Replica Masking at anytime. They can both access the original, just like the superuser and the owner of the publisher database. Therefore, a third role should be created on the subscriber database to provide unprivileged and read-only access to the data.
- The replication role is also able to access the original data at any time.
- The logs of the subscriber database may contain unmasked data.

Limitations

- Anonymous replication is based on logical replication, therefore it has the same restrictions, in particular: DDL commands, sequences, Large Objects are NOT replicated.

- The `REPLICA IDENTITY FULL` method is NOT supported. This means that all replicated tables MUST have a primary key.
- The primary key of a table should not be masked.

But I want to anonymize a primary key!

If you need to anonymize a primary key in a table, this means that it is a natural key (as opposed to a surrogate key).

Natural keys are problematic for many reasons:

- they can change over time (like email addresses or product codes), forcing cascading updates throughout related tables
- they're often not truly unique in practice, even seemingly unique values like SSNs can have duplicates or exceptions
- they tend to be longer and more complex than simple integers
- they make joins slower and indexes larger
- they can contain sensitive information that you might not want exposed in URLs or logs.
- they may change whenever business rules evolve, requiring database restructuring.

Surrogate keys (i.e. auto-incrementing integers) avoid these issues by providing stable, meaningless identifiers that never need to change.

In particular for anonymization: surrogate keys make your life easier since you don't have to mask them. In the other hand, natural keys are often a nightmare: in most situations they will force you to use complex pseudonymization techniques, and keep in mind that that Pseudonymization Is Not Anonymization !

Pseudonymization Is Not Anonymization: masking_functions.md#pseudonymizat

title: runbooks/0-intro draft: false toc: true —

---

run-sql:

- dbname: boutique
- user: paul
- parse_query: False …

Welcome to Paul's Boutique !

This is a 5 hours workshop that demonstrates various anonymization techniques using the PostgreSQL Anonymizer extension.

The Story

Paul's boutique

Paul's boutique has a lot of customers. Paul asks his friend Pierre, a Data Scientist, to make some statistics about his clients : average age, etc…

Pierre wants a direct access to the database in order to write SQL queries.

Jack is an employee of Paul. He's in charge of relationship with the various suppliers of the shop.

Paul respects his suppliers privacy. He needs to hide the personal information to Pierre, but Jack needs read and write access the real data.

Objectives

Using the simple example above, we will learn:

- How to write masking rules
- The difference between static and dynamic masking
- Implementing advanced masking techniques

About GDPR

This tutorial **does not** go into the details of the GPDR act and the general concepts of anonymization.

For more information about it, please refer to the talk below:

- Anonymisation, Au-delà du RGPD (Video / French)
- Anonymization, Beyond GDPR (PDF / english)

Requirements

In order to make this workshop, you will need:

- A Linux VM (preferably `Debian 12 bookworm` or `Ubuntu 24.04`)
- A PostgreSQL instance (preferably `PostgreSQL 17`)
- The PostgreSQL Anonymizer (anon) extension, installed and initialized by a superuser
- A database named "boutique" owned by a **superuser** called "paul"
- A role "pierre" and a role "jack", both allowed to connect to the database "boutique"

Check out the INSTALL section to learn how to install the PostgreSQL Anonymizer extension:

A simple way to deploy a workshop environment is to install Docker Desktop and download the image below:

```
ANON_IMG=registry.gitlab.com/dalibo/postgresql_anonymizer:stable
docker pull $ANON_IMG
```

And you can then launch it with:

```
docker run --name anon_tuto --detach -e POSTGRES_PASSWORD=x $ANON_IMG
docker exec -it anon_tuto psql -U postgres
```

!!! tip

```
Check out the [INSTALL section](https://postgresql-anonymizer.readthedocs.io/en/stable/INSTA
in the [documentation](https://postgresql-anonymizer.readthedocs.io/en/stable/)
to learn how to install the extension in your PostgreSQL instance.
```

The Roles

We will with 3 different users:

“‘text { .run-postgres user=postgres dbname=postgres show_result=false } CREATE ROLE paul LOGIN SUPERUSER PASSWORD ‘CHANGEME’;

CREATE ROLE pierre LOGIN PASSWORD ‘CHANGEME’;

CREATE ROLE jack LOGIN PASSWORD ‘CHANGEME’; GRANT pg_read_all_data TO jack; GRANT pg_write_all_data TO jack;

```
Unless stated otherwise, all commands must be executed with the role `paul`.
```

```
!!! tip

    Setup a `.pgpass` file to simplify the connections !
```

```
```console
cat > ~/.pgpass << EOL
*:*:boutique:paul:CHANGEME
*:*:boutique:pierre:CHANGEME
*:*:boutique:jack:CHANGEME
EOL
chmod 0600 ~/.pgpass
```

The Sample database

We will work on a database called “boutique”:

```
text { .run-postgres user=postgres dbname=postgres } CREATE
DATABASE boutique OWNER paul;
```

We need to activate the `anon` library inside that database:
```

`text { .run-postgres user=postgres dbname=postgres } ALTER DATABASE boutique SET session_preload_libraries = 'anon';` — title: runbooks/1-static_masking draft: false toc: true —

---

run-sql:

- dbname: boutique
- user: paul
- parse_query: False …

1- Static Masking

Static Masking is the simplest way to hide personal information! This idea is simply to destroy the original data or replace it with an artificial one.

Requirements

Please check out the intro of this tutorial if you haven't read it yet

The story

Over the years, Paul has collected data about his customers and their purchases in a simple database. He recently installed a brand new sales application and the old database is now obsolete. He wants to save it and he would like to remove all personal information before archiving it.

How it works

Anon Static image

Learning Objective

In this section, we will learn:

- How to write simple masking rules
- The advantage and limitations of static masking
- The concept of "Singling Out" a person

The "customer" table

"'text { .run-postgres parse_query=False } DROP TABLE IF EXISTS customer CASCADE;

DROP TABLE IF EXISTS payout CASCADE;

CREATE TABLE customer (id SERIAL PRIMARY KEY, firstname TEXT, lastname TEXT, phone TEXT, birth DATE, postcode TEXT);

Insert a few persons:

```text { .run-postgres  }
INSERT INTO customer
VALUES
(107,'Sarah','Conor','060-911-0911', '1965-10-10', '90016'),
(258,'Luke', 'Skywalker', NULL, '1951-09-25', '90120'),
(341,'Don', 'Draper','347-515-3423', '1926-06-01', '04520')
;
```

```text { .run-postgres  } SELECT * FROM customer;
```

The "payout" table

Sales are tracked in a simple table:

```text { .run-postgres  parse_query=False  } CREATE TABLE payout
(     id SERIAL PRIMARY KEY,     fk_customer_id INT REFERENCES
customer(id),     order_date DATE,     payment_date DATE,     amount
INT );
```

Let's add some orders:

```text { .run-postgres  } INSERT INTO payout VALUES (1,107,'2021-10-01','2021-10-01',
'7'), (2,258,'2021-10-02','2021-10-03', '20'), (3,341,'2021-10-02','2021-10-02',
'543'), (4,258,'2021-10-05','2021-10-05', '12'), (5,258,'2021-10-06','2021-10-06',
'92') ;
```

Activate the extension

```text { .run-postgres  } CREATE EXTENSION IF NOT EXISTS anon;
```

Declare the masking rules

Paul wants to hide the last name and the phone numbers of his clients. He will use the `dummy_last_name()` and `partial()` functions for that:

"'text { .run-postgres parse_query=False } SECURITY LABEL FOR anon ON COLUMN customer.lastname IS 'MASKED WITH FUNCTION anon.dummy_last_name()';

SECURITY LABEL FOR anon ON COLUMN customer.phone IS 'MASKED WITH FUNCTION anon.partial(phone,2,

$$X - XXX - XX$$

,2)';

```## Apply the rules permanently
```

```text { .run-postgres }
SELECT anon.anonymize_table('customer');
```

```
text { .run-postgres } SELECT id, firstname, lastname, phone FROM
customer;
```

> This is called `Static Masking` because the **real data has been permanently replaced**. We'll see later how we can use dynamic anonymization or anonymous exports.

Exercises

E101 - Mask the client's first names

Declare a new masking rule and run the static anonymization function again.

E102 - Hide the last 3 digits of the postcode

Paul realizes that the postcode gives a clear indication of where his customers live. However he would like to have statistics based on their postcode area.

Add a new masking rule to replace the last 3 digits by 'x'.

E103 - Count how many clients live in each postcode area?

Aggregate the customers based on their anonymized postcode.

E104 - Keep only the year of each birth date

Paul wants age-based statistic. But he also wants to hide the real birth date of the customers.

Replace all the birth dates by January 1rst, while keeping the real year.

You can use the make_date or date_trunc functions !

See https://www.postgresql.org/docs/current/functions-datetime.html#FUNCTIONS-DATETIME-TABLE

E105 - Singling out a customer

Even if the "customer" is properly anonymized, we can still isolate a given individual based on data stored outside of the table. For instance, we can identify the best client of Paul's boutique with a query like this:

```
text { .run-postgres  } WITH best_client AS (     SELECT SUM(amount),
fk_customer_id     FROM payout     GROUP BY fk_customer_id     ORDER
BY 1 DESC     LIMIT 1 ) SELECT c.* FROM customer c JOIN best_client
b ON (c.id = b.fk_customer_id)
```

This is called **Singling Out a person.**

We need to anonymize even further by removing the link between a person and its company. In the `payout` table, this link is materialized by a foreign key on the field `fk_customer_id`. However we can't remove values from this column or insert fake identifiers because if would break the foreign key constraint.

How can we separate the customers from their payouts while respecting the integrity of the data?

Find a function that will shuffle the column `fk_customer_id` of the `payout` table

Check out the shuffling section of the documentation.

Solutions

S101

"'text { .run-postgres } SECURITY LABEL FOR anon ON COLUMN customer.firstname IS 'MASKED WITH FUNCTION anon.dummy_first_name()';

SELECT anon.anonymize_table('customer');

SELECT id, firstname, lastname FROM customer;

S102

```text { .run-postgres  }
SECURITY LABEL FOR anon ON COLUMN customer.postcode
IS 'MASKED WITH FUNCTION anon.partial(postcode,2,$$xxx$$,0)';

SELECT anon.anonymize_table('customer');

SELECT id, firstname, lastname, postcode
FROM customer;
```

S103

```text { .run-postgres  } SELECT postcode, COUNT(id) FROM customer
GROUP BY postcode;
```

S104

"'text { .run-postgres parse_query=False } SECURITY LABEL FOR anon ON FUNCTION pg_catalog.date_trunc(text,interval) IS 'TRUSTED';

SECURITY LABEL FOR anon ON COLUMN customer.birth IS

$$MASKED WITH FUNCTION pg_catalog.date_trunc('year', birth)$$

;

SELECT anon.anonymize_table('customer');

SELECT id, firstname, lastname, birth FROM customer;

```
### S105

Let's mix up the values of the `fk_customer_id`:

```text { .run-postgres  }
SELECT anon.shuffle_column('payout','fk_customer_id','id');
```

Now let's try to single out the best client again :

```
text { .run-postgres  } WITH best_client AS (     SELECT SUM(amount),
fk_customer_id     FROM payout     GROUP BY fk_customer_id     ORDER
BY 1 DESC     LIMIT 1 ) SELECT c.* FROM customer c JOIN best_client
b ON (c.id = b.fk_customer_id);
```

WARNING: Note that the link between a `customer` and its `payout` is now completely false. For instance, if a customer A had 2 payouts. One of these payout may be linked to a customer B, while the second one is linked to a customer C.

In other words, this shuffling method with respect the foreign key constraint (aka the referential integrity) but it will break the data integrity. For some use case, this may be a problem.

In this case, Pierre will not be able to produce a BI report with the shuffle data, because the links between the customers and their payments are fake. — title: runbooks/2-dynamic_masking draft: false toc: true —

2- Dynamic Masking

With Dynamic Masking, the database owner can hide personal data for some users, while other users are still allowed to read and write the authentic data.

Requirements

Please check out the intro of this tutorial if you haven't read it yet

The Story

Paul has 2 employees:

- Jack is operating the new sales application, he needs access to the real data. He is what the GPDR would call a **"data processor"**.
```

- Pierre is a data analyst who runs statistic queries on the database. He should not have access to any personal data.

How it works

Anon Dynamic image

Objectives

In this section, we will learn:

- How to write simple masking rules
- The advantage and limitations of dynamic masking
- The concept of "Linkability" of a person

The `company` table

“‘text { .run-postgres parse_query=False }

DROP TABLE IF EXISTS supplier CASCADE;

DROP TABLE IF EXISTS company CASCADE;

CREATE TABLE company (id SERIAL PRIMARY KEY, name TEXT, vat_id TEXT UNIQUE);

```` run-postgres
INSERT INTO company
VALUES
(952,'Shadrach', 'FR62684255667'),
(194,E'Johnny\'s Shoe Store','CHE670945644'),
(346,'Capitol Records','GB663829617823')
;

SELECT * FROM company;
````

The `supplier` table

```
text { .run-postgres parse_query=False } CREATE TABLE supplier
(     id SERIAL PRIMARY KEY,     fk_company_id INT REFERENCES
company(id),     contact TEXT,     phone TEXT,     job_title TEXT
);

INSERT INTO supplier
VALUES
(299,194,'Johnny Ryall','597-500-569','CEO'),
(157,346,'George Clinton', '131-002-530','Sales manager')
;

SELECT * FROM supplier;
```

Activate the extension

```
ALTER DATABASE boutique
  SET session_preload_libraries TO 'anon';

CREATE EXTENSION IF NOT EXISTS anon;

SELECT anon.init();
```

Dynamic Masking

Activate the masking engine

```
text { .run-postgres parse_query=False } ALTER DATABASE boutique
SET anon.transparent_dynamic_masking TO true;
```

Masking a role

```
SECURITY LABEL FOR anon ON ROLE pierre IS 'MASKED';

GRANT pg_read_all_data to pierre;
```

Now connect as Pierre and try to read the supplier table:

```
text { .run-postgres user=pierre} SELECT * FROM supplier;
```

For the moment, there is no masking rule so Pierre can see the original data in each table.

Masking the supplier names

Connect as Paul and define a masking rule on the supplier table:

```
SECURITY LABEL FOR anon ON COLUMN supplier.contact
  IS 'MASKED WITH VALUE $$CONFIDENTIAL$$';
```

Now connect as Pierre and try to read the supplier table again:

```
text { .run-postgres user=pierre} SELECT * FROM supplier;
```

Now connect as Jack and try to read the real data:

```
text { .run-postgres user=jack } SELECT * FROM supplier;
```

Exercises

E201 - Guess who is the CEO of "Johnny's Shoe Store"

Masking the supplier contact is clearly not enough to provide anonymity.

Connect as Pierre and write a simple SQL query that joins the `supplier` and the `company` tables. See how that could reindentify some suppliers based on their job and their company.

With this request we managed to link a person to a company and we know it's job title. Since company names and job positions are available in many public datasets: a simple search on Linkedin or Google would give us the real names of many of the employees of these companies…

This is called **Linkability**: the ability to connect multiple records concerning the same data subject.

E202 - Anonymize the companies

We need to anonymize the `company` table, too. Even if they don't contain personal information, some fields can be used to **infer** the identity of their employees…

Connect as Paul and write 2 masking rules (security labels) for the company table.

- The first one will replace the `name` field with a fake name.
- The second rule will replace the `vat_id` with a random sequence of 10 characters

Go to thedocumentation and look at the faking functions and the random functions !

Connect as Pierre and check that he cannot view the real company info.

Connect as Jack and check that he can view the real values.

E203 - Pseudonymize the company name

Because of dynamic masking, the fake values will be different every time Pierre tries to read the table.

Pierre would like to have always the same fake values for a given company.

This is called pseudonymization.

Connect as Paul and write a new masking rule over the `vat_id` field by generating a hash of 10 characters using the `anon.digest()` function.

Write a new masking rule over the `name` field by using a pseudonymizing function.

Solutions

S201

```
text { .run-postgres user=pierre } SELECT s.id, s.contact, s.job_title,
c.name FROM supplier s JOIN company c ON s.fk_company_id = c.id;
```

S202

```
SECURITY LABEL FOR anon ON COLUMN company.name
  IS 'MASKED WITH FUNCTION anon.dummy_company_name()';

SECURITY LABEL FOR anon ON COLUMN company.vat_id
IS 'MASKED WITH FUNCTION anon.random_string(10)';
```

Now connect as Pierre and read the table again:

```
text { .run-postgres user=pierre } SELECT * FROM company;
```

Pierre will see different "fake data" every time he reads the table:

```
text { .run-postgres user=pierre } SELECT * FROM company;
```

Jack still sees the real data

```
text { .run-postgres user=jack } SELECT * FROM company;
```

S203

```
SECURITY LABEL FOR anon ON COLUMN company.vat_id
IS $$ MASKED WITH FUNCTION anon.left(anon.digest(vat_id, 'xxx', 'md5'),10) $$;

SECURITY LABEL FOR  anon ON COLUMN company.name
  IS 'MASKED WITH FUNCTION anon.pseudo_company(id)';
```

Connect as Pierre and read the table multiple times:

```
text { .run-postgres user=pierre } SELECT * FROM company;

text { .run-postgres user=pierre } SELECT * FROM company;
```

Now the fake company name is always the same.

title: runbooks/3-anonymous_dumps draft: false toc: true —

3- Anonymous Dumps

In many situation, what we want is basically to export the anonymized data into another database (for testing or to produce statistics). We will simply use pg_dump for that !

The Story

Paul has a website and a comment section where customers can express their views.

He hired a web agency to develop a new design for his website. The agency asked for a SQL export (dump) of the current website database. Paul wants to `clean` the database export and remove any personal information contained in the comment section.

How it works

Anon Dump image

Learning Objective

- Extract the anonymized data from the database
- Write a custom masking function to handle a JSON field.

Load the data

```
DROP TABLE IF EXISTS website_comment CASCADE;

CREATE TABLE website_comment (
  id SERIAL PRIMARY KEY,
  message JSONB
);

INSERT INTO website_comment
VALUES
  (1, json_build_object(
        'meta', json_build_object(
          'name', 'Lee Perry',
          'ip_addr','40.87.29.113'),
        'content', 'Hello Nasty!')),
  (2, json_build_object(
        'meta', json_build_object(
          'name', '',
          'email', 'biz@bizmarkie.com'),
        'content', 'Great Shop')),
  (3,json_build_object(
        'meta', json_build_object(
          'name','Jimmy'),
        'content','Hi ! This is me, Jimmy James'));
```

Check the content of the website comments:

```
SELECT
```

```
  message->'meta'->'name' AS name,
  message->'content' AS content
FROM website_comment
ORDER BY id ASC;
```

Activate the extension

```
CREATE EXTENSION IF NOT EXISTS anon;
```

Masking a JSON column

The `comment` field is filled with personal information and the fact the field does not have a standard schema makes our tasks harder.

In general, unstructured data are difficult to mask.

As we can see, web visitors can write any kind of information in the comment section. Our best option is to remove this key entirely because there's no way to extract personal data properly.

We can *clean* the comment column simply by removing the `content` key in the `message` column !

```
SELECT message - ARRAY['content'] AS message_without_content
FROM website_comment
WHERE id=1;
```

First let's create a dedicated schema and declare it as trusted. This means the `anon` extension will accept the functions located in this schema as valid masking functions. Only a superuser should be able to add functions in this schema.

```
CREATE SCHEMA IF NOT EXISTS my_masks;

SECURITY LABEL FOR anon ON SCHEMA my_masks IS 'TRUSTED';
```

Now we can write a function that remove the message content:

```
CREATE OR REPLACE FUNCTION my_masks.remove_content(j JSONB)
RETURNS JSONB
AS $func$
  SELECT j - ARRAY['content']
$func$
LANGUAGE SQL
;
```

Let's try it!

```
SELECT my_masks.remove_content(message)
FROM website_comment;
```

And now we can use it in a masking rule:

```
SECURITY LABEL FOR anon ON COLUMN website_comment.message
IS 'MASKED WITH FUNCTION my_masks.remove_content(message)';
```

Then we need to create a dedicated role to export the masked data. We will call this role `anon_dumper` (the name does not matter) and declare that this role is masked.

```
DROP ROLE IF EXISTS anon_dumper;

CREATE ROLE anon_dumper LOGIN PASSWORD 'CHANGEME';

ALTER ROLE anon_dumper SET anon.transparent_dynamic_masking TO TRUE;

SECURITY LABEL FOR anon ON ROLE anon_dumper IS 'MASKED';

GRANT pg_read_all_data TO anon_dumper;
```

For convenience, add a new entry in the `.pgpass` file.

```
cat > ~/.pgpass << EOL
*:*:boutique:anon_dumper:CHANGEME
EOL
```

Finally we can export an **anonymous dump** of the table with `pg_dump`:

```
export PATH=$PATH:$(pg_config --bindir)
export PGHOST=localhost
pg_dump -U anon_dumper boutique --table=website_comment > /tmp/dump.sql
```

Exercises

E301 - Dump the anonymized data into a new database

Create a database named `boutique_anon` and transfer the entire database into it.

E302 - Remove the email address

Replace the `remove_content` function with a better one called `remove_content_and_ip` that will nullify the `email` key.

HINT: you can use `jsonb_set(message, '{meta, email}', '{}')` to remove the email value.

E303 - Pseudonymize the IP address

Pierre plans to extract general information from the metadata. For instance, he wants to calculate the number of unique visitors based on the different IP addresses.

But an IP address is an **indirect identifier**, so Paul needs to anonymize this field while maintaining the fact that some values appear multiple times.

HINT: First you can create a new `meta` object using `jsonb_build_object()` and then use function `jsonb_set` replace the `meta` key

Solutions

S301

```
export PATH=$PATH:$(pg_config --bindir)
export PGHOST=localhost
dropdb -U paul --if-exists boutique_anon
createdb -U paul boutique_anon --owner paul
pg_dump -U anon_dumper boutique | psql -U paul --quiet boutique_anon
```

```
export PGHOST=localhost
psql -U paul boutique_anon -c 'SELECT COUNT(*) FROM company'
```

S302

```
CREATE OR REPLACE FUNCTION my_masks.remove_content_and_ip(message JSONB)
RETURNS JSONB
VOLATILE
LANGUAGE SQL
AS $func$
SELECT
  jsonb_set(message, '{meta, email}', '{}')
  - ARRAY['content'];
$func$;
```

```
SELECT my_masks.remove_content_and_ip(message)
FROM website_comment;
```

```
SECURITY LABEL FOR anon ON COLUMN website_comment.message
IS 'MASKED WITH FUNCTION my_masks.remove_content_and_ip(message)';
```

S303

```
CREATE OR REPLACE FUNCTION my_masks.clean_comment(message JSONB)
RETURNS JSONB
VOLATILE
LANGUAGE SQL
AS $func$
SELECT
  jsonb_set(
    message,
    ARRAY['meta'],
    jsonb_build_object(
        'name',anon.fake_last_name(),
        'ip_address', md5((message->'meta'->'ip_addr')::TEXT),
        'email', NULL
    )
  ) - ARRAY['content'];
$func$;
```

```
SELECT my_masks.clean_comment(message)
FROM website_comment;
```

```
SECURITY LABEL FOR anon ON COLUMN website_comment.message
IS 'MASKED WITH FUNCTION my_masks.clean_comment(message)';
```

4- Generalization

The main idea of generalization is to `blur` the original data. For example, instead of saying `Mister X was born on July 25, 1989`, we can say `Mister X was born is the 80's`. The information is still true, but it is less precise and it can't be used to reidentify the subject.

The Story

Paul hired dozens of employees over the years. He kept a record of their hair color, size and medical condition.

Paul wants to extract weird stats from these details. He provides generalized views to Pierre.

How it works

Anon Views image

Learning Objective

In this section, we will learn:

- The difference between masking and generalization
- The concept of `K-anonymity`

The `employee` table

“‘text { .run-postgres parse_query=False } DROP TABLE IF EXISTS employee CASCADE;

CREATE TABLE employee (id INT PRIMARY KEY, full_name TEXT, first_day DATE, last_day DATE, height INT, hair TEXT, eyes TEXT, size TEXT, asthma BOOLEAN, CHECK(hair = ANY(ARRAY[‘bald’,‘blond’,‘dark’,‘red’])), CHECK(eyes = ANY(ARRAY[‘blue’,‘green’,‘brown’])) , CHECK(size = ANY(ARRAY[‘S’,‘M’,‘L’,‘XL’,‘XXL’])));

```
This is awkward and illegal.


Loading the data:


``` run-postgres
INSERT INTO employee
  VALUES
(1,'Luna Dickens','2018-07-22','2018-12-15',180,'blond','blue','L',True),
(2,'Paul Wolf','2020-01-15',NULL,177,'bald','brown','M',False),
(3,'Rowan Hoeger','2018-12-01','2018-12-15',202,'dark','blue','XXL',True)
;

SELECT count(*) FROM employee;

SELECT full_name,first_day, hair, size, asthma
FROM employee
LIMIT 3;
```

Data suppression

Paul wants to find if there’s a correlation between asthma and the eyes color.

He provides the following view to Pierre.

```
DROP MATERIALIZED VIEW IF EXISTS v_asthma_eyes;

CREATE MATERIALIZED VIEW v_asthma_eyes AS
SELECT eyes, asthma
FROM employee;

SELECT *
FROM v_asthma_eyes
LIMIT 3;
```
```

Pierre can now write queries over this view.

```
SELECT
  eyes,
  100*COUNT(1) FILTER (WHERE asthma) / COUNT(1) AS asthma_rate
FROM v_asthma_eyes
GROUP BY eyes;
```

Pierre just proved that asthma is caused by blue eyes ;-)

K-Anonymity

The `asthma` and `eyes` columns are considered as indirect identifiers.

Indirect personal identifiers (or "quasi-identifiers") are pieces of information that, when combined with other data can identify an individual. Examples of indirect identifiers include: Date of birth, Gender, Zip code, etc.

With PostgreSQL Anonymizer, we can declare that a column is an indirect identifiers, like this:

```
SECURITY LABEL FOR k_anonymity
  ON COLUMN v_asthma_eyes.eyes
  IS 'INDIRECT IDENTIFIER';

SECURITY LABEL FOR k_anonymity
  ON COLUMN v_asthma_eyes.asthma
  IS 'INDIRECT IDENTIFIER';
```

```
SELECT anon.k_anonymity('v_asthma_eyes');
```

The v_asthma_eyes has '2-anonymity'. This means that each quasi-identifier combination (the 'eyes-asthma' tuples) occurs in at least 2 records for a dataset.

In other words, it means that each individual in the view cannot be distinguished from at least 1 (k-1) other individual.

Range and Generalization functions

Now let's add another view over the `employee` table.

We will generalize the dates of to keep only the month and year.

```
DROP MATERIALIZED VIEW IF EXISTS v_staff_per_month;
CREATE MATERIALIZED VIEW v_staff_per_month AS
SELECT
    anon.generalize_daterange(first_day,'month') AS first_day,
    anon.generalize_daterange(last_day,'month') AS last_day
FROM employee;
```

```
SELECT *
```

```
FROM v_staff_per_month
LIMIT 3;
```

Pierre can write a query to find how many employees were hired in november 2021.

```
SELECT COUNT(1)
       FILTER (
            WHERE make_date(2019,11,1)
            BETWEEN lower(first_day)
            AND COALESCE(upper(last_day),now())
       )
FROM v_staff_per_month;
```

Declaring the indirect identifiers

Now let's check the k-anonymity of this view by declaring which columns are indirect identifiers :

```
SECURITY LABEL FOR k_anonymity
  ON COLUMN v_staff_per_month.first_day
  IS 'INDIRECT IDENTIFIER';

SECURITY LABEL FOR k_anonymity
  ON COLUMN v_staff_per_month.last_day
  IS 'INDIRECT IDENTIFIER';

SELECT anon.k_anonymity('v_staff_per_month');
```

In this case, the k factor is 1 which means that there is at least one unique individual who be identified directly by his/her first and last dates.

Exercises

E401 - Simplify `v_staff_per_month` and decrease granularity

Generalizing dates per month is not enough. Write another view called `v_staff_per_year` that will generalize dates per year.

Also simplify the view by using a range of int to store the years instead of a date range.

E402 - Staff progression over the years

How many people worked for Paul for each year between 2018 and 2021?

E403 - Reaching 2-anonymity for the `v_staff_per_year` view

What is the k-anonymity of `v_staff_per_month_years`?

Solutions

S401

```
DROP MATERIALIZED VIEW IF EXISTS v_staff_per_year;

CREATE MATERIALIZED VIEW v_staff_per_year AS
SELECT
  int4range(
    extract(year from first_day)::INT,
    extract(year from last_day)::INT,
    '[]'
  ) AS period
FROM employee;
```

`'[]'` will include the upper bound

```
SELECT *
FROM v_staff_per_year
LIMIT 3;
```

S402

```
SELECT
  year,
  COUNT(1) FILTER (
      WHERE year <@ period
  )
FROM
    generate_series(2018,2021) year,
    v_staff_per_year
GROUP BY year
ORDER BY year ASC;
```

S403

```
SECURITY LABEL FOR k_anonymity
  ON COLUMN v_staff_per_year.period
  IS 'INDIRECT IDENTIFIER';

SELECT anon.k_anonymity('v_staff_per_year');
```

5- Anonymized Replica

With logical replication, you can maintain a replica that is always up to date and always anonymized. This feature was introduced in version 2.3 and became

generally available in version 3.0.

For more details, check out the Replica Masking section of the documentation.

The Story

Pierre needs permanent access to the boutique data to run his analytics. With dynamic masking, he was given a masked view of the production database. But Paul would now prefer to keep Pierre away from the production server entirely.

The solution is a dedicated, anonymized replica: Pierre gets a read-only, always-fresh, fully masked copy of the data, and never touches production.

How it works

Anon Anonymized Replica image

Learning Objective

In this section, we will learn:

- How to configure the publication on the source database
- How to configure the anonymized replica
- How to verify the anonymization in real time

A note on this setup

In production, the anonymized replica is a **separate PostgreSQL server**.

To keep this tutorial self-contained, the source and the replica are two databases of the **same instance**: `boutique` (the source) and `boutique_replica` (the anonymized replica). Logical replication works just as well between two databases of the same cluster.

This chapter creates cluster-wide objects (a replication role, a publication, a replication slot and a second database). If you want to replay it from scratch, clean them up first (see the Conclusion).

Prerequisites

Logical replication requires `wal_level = logical`. The default value is `replica`; raising it requires a restart of the instance (check it with `SHOW wal_level;`).

The `anon` extension must also be loaded in every session that declares masking rules. The simplest way to guarantee this is to preload it globally with `shared_preload_libraries = 'anon'`.

The source database

We use the `company` and `supplier` tables, filled with real data:

“‘text { .run-postgres parse_query=False show_result=false } DROP TABLE IF EXISTS supplier CASCADE; DROP TABLE IF EXISTS company CASCADE;

CREATE TABLE company (id INT PRIMARY KEY, name TEXT, vat_id TEXT UNIQUE);

INSERT INTO company VALUES (952,‘Shadrach’,‘FR62684255667’), (194,E’Johnny’s Shoe Store’,‘CHE670945644’), (346,‘Capitol Records’,‘GB663829617823’);

CREATE TABLE supplier (id INT PRIMARY KEY, fk_company_id INT REFERENCES company(id), contact TEXT, phone TEXT, job_title TEXT);

INSERT INTO supplier VALUES (299,194,‘Johnny Ryall’,‘597-500-569’,‘CEO’), (157,346,‘George Clinton’,‘131-002-530’,‘Sales manager’);

```text { .run-postgres parse_query=False }
SELECT * FROM company ORDER BY id;
```

With replica masking, the source database needs no masking rules: the anonymization happens on the replica, as the data is received.

Publication side: the replication role

On the source, create a dedicated replication role:

“‘text { .run-postgres parse_query=False show_result=false } CREATE ROLE anon_replicator LOGIN REPLICATION PASSWORD ‘CHANGEME’;

GRANT USAGE ON SCHEMA public TO anon_replicator; GRANT SELECT ON ALL TABLES IN SCHEMA public TO anon_replicator;

```
## Publication side: create the publication

We only publish the tables that Pierre needs to read. The `customer` and
`payout` tables are not published here.

```text { .run-postgres parse_query=False show_result=false }
CREATE PUBLICATION pub_boutique
  FOR TABLE company, supplier;
```

Check the publication:

```
text { .run-postgres parse_query=False } SELECT pubname, schemaname,
tablename FROM pg_publication_tables;
```

When the replica is **another database of the same cluster**, the replication slot must be created beforehand:
```

```
text { .run-postgres parse_query=False } SELECT slot_name FROM
pg_create_logical_replication_slot('boutique_slot','pgoutput');
```

!!! warning

```
This is a PostgreSQL restriction, see [CREATE SUBSCRIPTION] for more details.
When the replica is a **separate server** (the production setup), this
restriction does not apply and the slot can be created by
`CREATE SUBSCRIPTION` itself.
```

Replica side: create the database

```
text { .run-postgres user=postgres dbname=postgres parse_query=False
show_result=false } CREATE DATABASE boutique_replica OWNER paul;
```

The `anon.replica_masking` parameter allows the masking triggers to be installed on this database. It must be enabled before calling `anon.start_replica_masking()`, and therefore before the subscription is created:

```
text { .run-postgres user=postgres dbname=postgres parse_query=False
show_result=false } ALTER DATABASE boutique_replica SET anon.replica_masking
TO on;
```

If you use a `.pgpass` file, add entries for the new database:

```
cat >> ~/.pgpass << EOL
*:*:boutique_replica:paul:CHANGEME
*:*:boutique_replica:pierre:CHANGEME
EOL
```

Replica side: initialize the extension and the tables

`anon.init()` is required on the replica: the faking and pseudonymization functions rely on the fake datasets it loads.

The tables must also exist **before** the subscription is created, since logical replication does not replicate DDL:

“‘text { .run-postgres user=paul dbname=boutique_replica parse_query=False show_result=false } CREATE EXTENSION IF NOT EXISTS anon; SELECT anon.init();

CREATE TABLE company (id INT PRIMARY KEY, name TEXT, vat_id TEXT UNIQUE);

CREATE TABLE supplier (id INT PRIMARY KEY, fk_company_id INT REFERENCES company(id), contact TEXT, phone TEXT, job_title TEXT);

```
## Replica side: masking rules
```

```
The replica applies its rules when the data is received.
Declare rules similar to the ones of the Dynamic Masking tutorial:

```text { .run-postgres user=paul dbname=boutique_replica parse_query=False show_result=fals
SECURITY LABEL FOR anon ON COLUMN company.name
  IS 'MASKED WITH FUNCTION anon.pseudo_company(id)';

SECURITY LABEL FOR anon ON COLUMN company.vat_id
  IS 'MASKED WITH FUNCTION anon.random_string(10)';

SECURITY LABEL FOR anon ON COLUMN supplier.contact
  IS 'MASKED WITH VALUE $$CONFIDENTIAL$$';

SECURITY LABEL FOR anon ON COLUMN supplier.phone
  IS 'MASKED WITH FUNCTION anon.partial(phone,0,$$XX-XXX-XXX$$,0)';
```

Pseudonymizing `company.name` guarantees that Pierre always sees the same fake name for a given company, which lets him write consistent joins between `company` and `supplier`.

Declare all the masking rules **before** creating the subscription: a rule added later only masks the changes that arrive afterwards (see exercise E501).

Replica side: enable and subscribe

`anon.start_replica_masking()` installs, on each masked table, a trigger declared `ENABLE REPLICA`: it only fires during the replication apply process — which is also why writes performed locally on the replica are not masked.

```
text { .run-postgres user=paul dbname=boutique_replica parse_query=False
show_result=false } SELECT anon.start_replica_masking();
```

The subscription reuses the slot we created earlier, hence `create_slot=false`:

```
text { .run-postgres user=paul dbname=boutique_replica parse_query=False
show_result=false } CREATE SUBSCRIPTION sub_boutique   CONNECTION
'host=localhost port=65432 user=anon_replicator password=CHANGEME
dbname=boutique'   PUBLICATION pub_boutique   WITH (slot_name='boutique_slot',
create_slot=false);
```

!!! note

```
In this tutorial environment, the source database listens on port
`65432`. In production, the connection string points to your **source
server** (usually port `5432`). For a two-server setup, also remember
that the `pg_hba.conf` of the source must allow replication
connections for `anon_replicator`, that raising `wal_level` requires
a restart of the source instance, and that the reader roles (such as
`pierre`) must be created on the replica instance.
```
```

Replication is asynchronous; we wait a moment for the initial copy to complete:

```
text { .run-postgres user=paul dbname=boutique_replica parse_query=False
show_result=false } SELECT pg_sleep(3);
```

Check the subscription status:

```
text { .run-postgres user=paul dbname=boutique_replica parse_query=False
} SELECT subname, pid, received_lsn, latest_end_lsn FROM pg_stat_subscription;
```

Verification

On the replica, the data is anonymized, even for Paul:

```
text { .run-postgres user=paul dbname=boutique_replica parse_query=False
} SELECT * FROM company ORDER BY id;
```

Unlike with dynamic masking, **all users** see the anonymized data on the replica: as long as the masking rules are declared before the data arrives, the original data is never stored there. (Exercise E501 shows what happens when a rule comes too late.)

For comparison, on the source the data is still real:

```
text { .run-postgres parse_query=False } SELECT * FROM company
ORDER BY id;
```

A new row inserted on the source is replicated and anonymized automatically:

```
text { .run-postgres parse_query=False show_result=false } INSERT
INTO company VALUES (501,'Acme Corp','US123456789');
```

```
text { .run-postgres parse_query=False show_result=false } SELECT
pg_sleep(2);
```

```
text { .run-postgres user=paul dbname=boutique_replica parse_query=False
} SELECT * FROM company WHERE id = 501;
```

Beware of primary keys

Logical replication identifies rows by their replica identity — by default, their primary key:

- With PostgreSQL Anonymizer, every replicated table must have a primary key (`REPLICA IDENTITY FULL` is not supported).
- **Do not mask the columns of the primary key!**
- Otherwise `UPDATE` and `DELETE` statements will no longer propagate.

Technical primary keys (sequences, UUID) are usually not personal data and do not need to be masked.

For more details on masking columns involved in referential integrity, see the Masking Foreign Keys section.

Exercises

As a reminder, unless stated otherwise, all commands are to be run as **paul** (the owner).

E501 - Add a masking rule after the fact

All our masking rules were declared **before** the subscription was created. Declare a new rule on `supplier.job_title` while the replication is running.

What happens to the rows that are already on the replica? How can you fix them?

E502 - A masked role on the replica?

Connect as Pierre and check what he sees on the replica.

Does Pierre need to be declared as a `MASKED` role on `boutique_replica`, like he was on the source? Why or why not?

E503 - Verify the anonymized replication

On the source, insert a new supplier; on the replica, check that it appears and is anonymized.

Solutions

S501

Declare the rule, then refresh the masking triggers with `anon.refresh_replica_masking()`:

“‘text { .run-postgres user=paul dbname=boutique_replica parse_query=False show_result=false } SECURITY LABEL FOR anon ON COLUMN supplier.job_title IS ‘MASKED WITH VALUE

CONFIDENTIAL

’;

SELECT anon.refresh_replica_masking();

```
  The triggers only mask **incoming** rows: the suppliers that were
replicated before the rule was declared still show their real job titles
on the replica!
```

```text { .run-postgres user=paul dbname=boutique_replica parse_query=False }
SELECT id, job_title FROM supplier ORDER BY id;
```

To fix the existing rows, apply the rules once with static masking (see the Static Masking tutorial). From then on, every incoming row is masked:

```
text { .run-postgres user=paul dbname=boutique_replica parse_query=False
show_result=false } SELECT anon.anonymize_table('supplier');
```

```
text { .run-postgres user=paul dbname=boutique_replica parse_query=False
} SELECT id, job_title FROM supplier ORDER BY id;
```

This is why the masking rules should be declared **before** the subscription is created.

S502

Give Pierre read access to the replica database:

```
text { .run-postgres user=paul dbname=boutique_replica parse_query=False
show_result=false } GRANT USAGE ON SCHEMA public TO pierre; GRANT
SELECT ON ALL TABLES IN SCHEMA public TO pierre;
```

If Pierre already holds the cluster-wide `pg_read_all_data` privilege (granted in the Dynamic Masking tutorial), these grants are redundant. On a standalone replica server, they are required.

```
text { .run-postgres user=pierre dbname=boutique_replica parse_query=False
} SELECT s.contact, s.phone, c.name FROM supplier s JOIN company
c ON s.fk_company_id = c.id;
```

There is no need to declare Pierre as a `MASKED` role on the replica: the anonymization happened at replication time, not at query time. The `MASKED` label that Pierre carries from the Dynamic Masking tutorial has no effect here, because `anon.transparent_dynamic_masking` is not enabled on `boutique_replica`.

S503

```
text { .run-postgres parse_query=False show_result=false } INSERT
INTO supplier VALUES (42,501,'Wile E. Coyote','555-0100','Buyer');
```

```
text { .run-postgres parse_query=False show_result=false } SELECT
pg_sleep(2);
```

```
text { .run-postgres user=paul dbname=boutique_replica parse_query=False
} SELECT s.contact, s.phone, s.job_title, c.name FROM supplier s
JOIN company c ON s.fk_company_id = c.id WHERE s.id = 42;
```

The new supplier has been replicated and anonymized automatically: the contact and the job title are hidden (the latter by the rule added in E501), the phone is masked and the company name is the same pseudonym as in the previous sections. — title: runbooks/9-conclusion draft: false toc: true —

Conclusion

Clean up !

The Anonymized Replica tutorial created a second database with a subscription that holds a replication slot on `boutique`. We drop the subscription first, otherwise `boutique` could not be dropped. (If you skipped that tutorial, skip the next two blocks.)

```
DROP SUBSCRIPTION IF EXISTS sub_boutique;

DROP DATABASE IF EXISTS boutique_replica WITH (FORCE);

DROP DATABASE IF EXISTS boutique WITH (FORCE);

REASSIGN OWNED BY jack TO postgres;

REASSIGN OWNED BY paul TO postgres;

REASSIGN OWNED BY pierre TO postgres;

DROP ROLE IF EXISTS jack;
DROP ROLE IF EXISTS paul;
DROP ROLE IF EXISTS pierre;
DROP ROLE IF EXISTS dump_anon;
DROP ROLE IF EXISTS anon_replicator;
```

Also...

Other projects you may like

- pg_sample : extract a small dataset from a larger PostgreSQL database

Help Wanted!

This is a free and open project!

labs.dalibo.com/postgresql_anonymizer

Please send us feedback on how you use it, how it fits your needs (or not), etc.
— title: sampling draft: false toc: true —

Sampling

Principle

The GDPR introduces the concept of "data minimization" which means that the collection of personal information must be limited to what is directly relevant and necessary to accomplish a specified purpose.

If you're writing an anonymization policy for a dataset, chances are that you don't need to anonymize **the entire database**. In most cases, extract a subset

of the table is sufficient. For example, if you want to export an anonymous dumps of the data for testing purpose in a CI workflow, extracting and masking only 10% of the database may be enough.

Furthermore, anonymizing a smaller portion (i.e a "sample") of the dataset will be way faster.

With PostgreSQL Anonymizer, you can use 2 different sampling methods :

- Sampling with TABLESAMPLE
- Sampling with RLS Policies

You can also Truncate Tables for the masked users !

Sampling with TABLESAMPLE

Let's say you have a huge amounts of http logs stored in a table. You want to remove the ip addresses and extract only 10% of the table:

```
CREATE TABLE http_logs (
  id integer NOT NULL,
  date_opened DATE,
  ip_address INET,
  url TEXT
);

SECURITY LABEL FOR anon ON COLUMN http_logs.ip_address
IS 'MASKED WITH VALUE NULL';

SECURITY LABEL FOR anon ON TABLE http_logs
IS 'TABLESAMPLE BERNOULLI(10)';
```

Now you can either do static masking, dynamic masking or an anonymous dumps. The mask data will represent a 10% portion of the real data.

The syntax is exactly the same as the TABLESAMPLE clause which can be placed at the end of a SELECT statement.

You can also defined a sampling ratio at the database-level and it will be applied to all the tables that don't have their own `TABLESAMPLE` rule.

```
SECURITY LABEL FOR anon ON DATABASE app
IS 'TABLESAMPLE SYSTEM(33)';
```

Sampling with RLS policies

Another approach for sampling is to use Row Level Security Policies, also known as `RLS` or `Row Security Policies`.

Let's use the same example as a above, this time we want to define a limit so the mask users can only see the logs of the last 6 months.

```
CREATE TABLE http_logs (
  id integer NOT NULL,
  date_opened DATE,
  ip_address INET,
  url TEXT
);

SECURITY LABEL FOR anon ON COLUMN http_logs.ip_address
  IS 'MASKED WITH VALUE NULL';

ALTER TABLE http_logs ENABLE ROW LEVEL SECURITY;

CREATE POLICY http_logs_sampling_for_masked_users
  ON http_logs
  USING (
    NOT anon.hasmask(CURRENT_USER::REGROLE)
    OR date_opened >= now() - '6 months'::INTERVAL
  );
```

This RLS policy is based on 2 conditions:

- if the current user is not masked, the first condition is true and he/she can read all the lines
- if the current user is masked, the first condition is false and he/she can only read the lines that satisfy the second condition

Sampling with RLS policies is more powerful than the TABLESAMPLE method, however maintaining a set of RLS policies is known to be difficult in the long run. The benefits from Postgres RLS can dissipate when the size of the organization, the amount of data collected, and the number of restrictions grow in size and complexity.

In order to run static masking with RLS, the user has to be the owner of the tables (with `NO FORCE ROW LEVEL SECURITY` (default)) or have the `BYPASSRLS` attribute.

Maintaining Referential Integrity

!!! note

`The sampling methods described above **MAY FAIL** if you have foreign keys pointing at the table you want to sample.`

Extracting a subset of a database while maintaining referential integrity is tricky and it is not supported by this extension.

If you really need to keep referential integrity in an anonymized dataset, you need to do it in 2 steps:

- First, extract a sample with pg_sample
- Second, anonymize that sample

There may be other sampling tools for PostgreSQL but pg_sample is probably the best one.

Truncate Tables for the masked users

In certain situations, you can also erase completely a table instead of just masking some of the columns.

For instance, let's say that masked users should not see anything in the `http_logs` table below

```
CREATE TABLE http_logs (
  id integer NOT NULL,
  date_opened DATE,
  ip_address INET,
  url TEXT
);
```

Using the TABLESAMPLE clause, you can simply set the sampling ratio to 0

```
SECURITY LABEL FOR anon ON TABLE http_logs IS ' TABLESAMPLE SYSTEM (0)';
```

Now the table will be erased for the masked users !

```
SET ROLE the_database_owner;

SELECT count(*) FROM http_logs;
  count
---------
 156706

SET ROLE a_masked_user;

SELECT count(*) FROM http_logs;
  count
---------
     0
```

Security

Permissions

Here's an overview of what users can do depending on the privileges they have:

Action	Superuser	Owner	Masked Role
Create the extension	Yes		
Drop the extension	Yes		
Init the extension	Yes		
Reset the extension	Yes		
Configure the extension	Yes		
Put a mask upon a role	Yes		
Enable dynamic masking	Yes		
Declare a trusted schema	Yes		
Declare a trusted function	Yes		
Create a table	Yes	Yes	
Declare a masking rule	Yes	Yes	
Insert, delete, update a row	Yes	Yes	
Static Masking	Yes	Yes	
Select the real data	Yes	Yes	
Regular Dump	Yes	Yes	
Anonymous Dump	Yes	Yes	
Use the masking functions	Yes	Yes	Yes (1)
Select the masked data	Yes	Yes	Yes
View the masking rules	Yes	Yes	Yes

(1) Except when transparent dynamic masking is enabled: a masked role is then denied any direct call to the `RESTRICTED` functions, namely `anon.pseudo_*()`, `anon.hash()`, `anon.digest()` and `anon.custom_value()`. These functions are deterministic, so a masked role able to call them could run them over candidate values and match the result against the masked data it reads, thus recovering the original values.

Limit masking filters only to trusted schemas

By default, the database owner can only write masking rules with functions that are located in the trusted schemas which are controlled by the superusers.

Out of the box, only the `anon` schema is declared as trusted. This means that by default the functions from the `pg_catalog` cannot be used in masking rules.

For more details, read the Using pg_catalog functions section.

Timing attacks in LDP functions

> This section is intended for maintainers of this extension, not end users.

The GRRM perturbation function (`anon.ldp_grrm`) decides at runtime whether to keep the original value or replace it with a random lie. A naive implementation using an `if/else` branch can leak which path was taken through differences

in execution time. An attacker who can measure response times precisely enough could use this to figure out whether a particular response is the true value or a perturbed one, which defeats the whole purpose of the differential privacy guarantee.

To prevent this, the function uses **branchless bitwise selection**: both the true value and the lie value are always computed, and a bitmask is used to pick one of the two without any conditional jump. This makes the execution time constant regardless of which path is taken.

If you modify this function or add new LDP perturbation methods, keep this pattern in mind. Avoid `if/else` or `match` on any value that depends on the random keep-or-lie decision. Instead compute both outcomes and select with bitwise operations.

Security context of the functions

Most of the functions of this extension are declared with the `SECURITY INVOKER` tag. This means that these functions are executed with the privileges of the user that calls them. This is an important restriction.

This extension contains another few functions declared with the tag `SECURITY DEFINER`.

Hardening principles

You should follow the *Principle of Least Privilege* when designing your masking policy. To achieve this the user accounts used for static masking should operate with only the minimum permissions necessary to perform data masking.

The `TRUSTED` schema and function mechanism is trying to do what PostgreSQL is doing at runtime during the declaration phase… with less tools. It's more reliable and sustainable to rely on PostgreSQL's ACL to ensure the safety of your data.

The `anon.nosuperuser` guc is intended to help in the implementation of the *Principle of Least Privilege*. When it's set to `on`, users with the `SUPERUSER` attribute cannot run:

- static masking;
- parallel static masking;
- dynamic masking;
- replica masking.

You should use the guc `anon.nouperuser` for your deployments, as the TRUSTED mechanism is likely be removed in future versions.

Static masking good practices

PostgreSQL keeps a small subset of the data in its system catalogs for both univariate and multivariate statistics such as for most-common-values and histograms.

This data can still be visible after static anonymization has been performed. Usually a substantial amount of data is modified and an autoanalyze will run on the tables. Whether and when this happens depends on the cluster configuration and table size. Furthermore, there is no way to predict when autoanalyze will run.

If a `pg_dump --statistics` (v18) runs immediately after static anonymization, it could export unmasked data.

Therefore every static masking function attempts to run an `ANALYZE` command on the target relation at the end of the process. This command requires ownership of the relevant object, the `MAINTAIN` privilege on it (v17) or membership in the `pg_maintain` role (v17). If the user lacks the required privilege a warning will be issued and the execution will continue without error.

Example:

```
-- Before PostgreSQL 17
WARNING:  skipping "hundred" --- only table or database owner can analyze it

-- After PostgreSQL 17
WARNING:  permission denied to analyze "player", skipping it
```

Unfortunately, there is no easy way to intercept the aforementioned warning from the extension. If the version you are running allows it, grant the appropriate privilege to the anonymization user, otherwise run the `ANALYZE` command separately with a privileged user.

More generally, it is a good idea to check for table bloat and consider running `VACUUM FULL` or `REPACK` where appropriate. For tables that are not rebuilt, running `VACUUM FREEZE` may help prevent transaction ID wraparound issues and the resulting need for aggressive autovacuum activity later on.

Function with SQL standard body

Functions with SQL-standard bodies undergo parse analysis only at creation time. As a result, transparent dynamic masking would completely miss them. Users subject to transparent dynamic masking are therefore forbidden from executing such functions.

This includes:

- functions in contrib extensions that use this form (`citext`, `earthdistance`, `lo`, `pageinspect`, `pg_freespacemap`, `xml2` in 18.6),

- masking rules such as: `MASKED WITH FUNCTION mask.stars(num)`, where `mask.stars()` has a `RETURN` body, whether the function is used directly or through `pg_dump`.

Selective Masking (BETA)

Principle

In some context, it is relevant to mask only certain row of a table.

The selective masking syntax allows to filter based on a condition.

This feature is currently under heavy development. This implementation of Selective Masking is provided for testing purpose only. Major breaking changes may be introduced at any time and we may even remove this feature entirely if we feel it does not reach our standard of quality and stability

Example

Imagine a `users` table containing all the users of an application.

```
SELECT * FROM users;
 id | login |            password             | admin
----+-------+----------------------------------+-------
  1 | alice | adfsqfcksqhdqijsdizjdfiqqlq<iqq | f
  2 | bob   | a_very_bad_password              | t
  3 | carol | 1234                             | f
```

We want to anonymize the login and password columns with the 2 masking rules below:

```
SECURITY LABEL FOR anon ON COLUMN users.login    IS 'MASKED WITH VALUE NULL';
SECURITY LABEL FOR anon ON COLUMN users.password IS 'MASKED WITH VALUE NULL';
```

We may want to anonymize most of the users in the table while keeping the real data of the administrators so that they can still use the application in anonymized environments.

We can add a rule on table to filter out all the admin users:

```
SECURITY LABEL FOR anon ON TABLE users IS 'MASKED WHEN admin IS FALSE';
```

Now let's anonymize the table:

```
SELECT anon.anonymize_table('users');

SELECT * FROM users;
 id | login |            password             | admin
----+-------+----------------------------------+-------
```

```
1 |       |                                  | f
2 | bob   | a_very_bad_password              | t
3 |       |                                  | f
```

NOTE: The Selective Masking is incompatible with Sampling.

Sampling: sampling.md

title: static_masking draft: false toc: true —

Permanently remove sensitive data

Sometimes, it is useful to transform directly the original dataset. You can do that with different methods:

- Applying masking rules
- Shuffling a column
- Adding noise to a column

Anonymization also tries to recompute the statistics on the objects it operates on. Please refer to the security page for information on this topic and more general security consideration regarding static anonymization.

These methods will destroy the original data. Use with care.

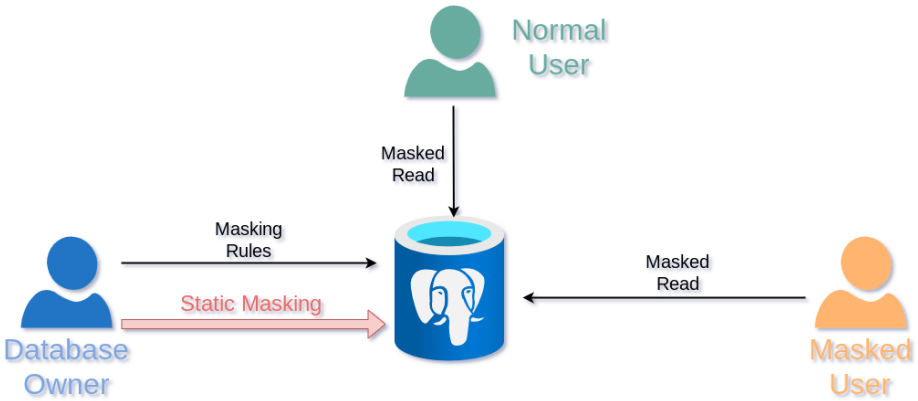


Figure 11: PostgreSQL Static Masking

Applying masking rules

You can permanently apply the masking rules of a database with `anon.anonymize_database()`.

Let's use a basic example :

```
CREATE TABLE customer(
  id SERIAL,
  full_name TEXT,
  birth DATE,
  employer TEXT,
  zipcode TEXT,
  fk_shop INTEGER
);

INSERT INTO customer
VALUES
(911,'Chuck Norris','1940-03-10','Texas Rangers', '75001',12),
(312,'David Hasselhoff','1952-07-17','Baywatch', '90001',423)
;

SELECT * FROM customer;
```

```
 id  |   full_name      |   birth    |    employer    | zipcode | fk_shop
-----+------------------+------------+----------------+---------+---------
 911 | Chuck Norris     | 1940-03-10 | Texas Rangers | 75001   | 12
 112 | David Hasselhoff | 1952-07-17 | Baywatch      | 90001   | 423
```

Step 1: Load the extension :

```
CREATE EXTENSION IF NOT EXISTS anon CASCADE;
SELECT anon.init();
```

Step 2: Declare the masking rules

```
SECURITY LABEL FOR anon ON COLUMN customer.full_name
IS 'MASKED WITH FUNCTION anon.dummy_name()';

SECURITY LABEL FOR anon ON COLUMN customer.employer
IS 'MASKED WITH FUNCTION anon.dummy_company_name()';

SECURITY LABEL FOR anon ON COLUMN customer.zipcode
IS 'MASKED WITH FUNCTION anon.random_zip()';
```

Step 3: Replace authentic data in the masked columns :

```
SELECT anon.anonymize_database();
ANALYZE;

SELECT * FROM customer;
```

```
 id  |  full_name  |   birth    |      employer           | zipcode | fk_shop
-----+-------------+------------+-------------------------+---------+---------
```

```
 911 | jesse Kosel | 1940-03-10 | Marigold Properties LLC | 62172   |      12
 312 | leolin Bose | 1952-07-17 | Inventure Inc           | 20026   |     423
```

You can also use `anonymize_schema()`, `anonymize_table()` and `anonymize_column()` to remove data from a subset of the database :

```
SELECT anon.anonymize_schema('public');
SELECT anon.anonymize_table('customer');
SELECT anon.anonymize_column('customer','zipcode');
```

`anonymize_schema()` masks only the tables of the given schema. Like the other functions, it accepts a masking policy as its last argument (default `'anon'`).

!!! info

```
Static masking does NOT apply masking ruled places upon views, even
if they are updatable!
```

!!! warning **Static masking is a slow process**.

```
The principle of static masking is to update all lines of all tables
containing at least one masked column. This basically means that PostgreSQL
will rewrite all the data on disk. Depending on the database size, the
hardware and the instance config, it may be faster to export the anonymized data
(See [Anonymous Dumps] ) and reload it into the database.
```

Disabling Static Masking

You may be scared that someone could accidentally run `anon.anonymize_database()` and wipe out all the data.

If so, you can disable this feature globally with:

```
ALTER SYSTEM SET anon.static_masking TO off
```

Or disable it for a single user :

```
ALTER ROLE bob SET anon.static_masking TO off;
```

Or disable it everyone except one user

```
ALTER DATABASE mydb SET anon.static_masking = FALSE;
ALTER ROLE daniel SET anon.static_masking = TRUE;
```

Static Masking and Multiple Masking Policies

When using multiple masking policies, you can simply add the policy name at the end of the static masking functions.

For instance, if you defined a masking policy named “rgpd”, you can apply it with

```
SELECT anon.anonymize_table('customer','rgpd');
SELECT anon.anonymize_column('customer','zipcode','rgpd');
ANALYZE;
```

By default, there's a single masking policy named "anon".

Parallel Static Masking

For large databases, static masking can be a time-consuming operation. To improve performance, the extension supports parallel static masking using PostgreSQL background workers.

Instead of using `anon.anonymize_database()`, you can use the parallel version:

```
SELECT anon.anonymize_database_parallel(4);
ANALYZE;
```

The parameter specifies the number of parallel workers to use. The function will:

1. Analyze the foreign key relationships between tables
2. Group tables to avoid constraint violations
3. Distribute the work across multiple background workers
4. Process tables in parallel where possible

To restrict the parallel run to a single schema, use `anon.anonymize_schema_parallel('public', 4)`; it masks only the masked tables of that schema.

The maximum number of background workers can be configured using the GUC parameter `anon.max_bg_workers`:

```
-- Set the maximum number of background workers (default: 4)
SET anon.max_bg_workers = 8;

-- Or configure it system-wide
ALTER SYSTEM SET anon.max_bg_workers = 8;
SELECT pg_reload_conf();
```

The `anon.max_bg_workers` parameter accepts values between 1 and 64. On servers with many CPUs, you may want to increase this value to improve performance. The value is capped by PostgreSQL's guc: `max_worker_processes` (default: 8).

Important considerations:

- **Foreign Keys**: Tables with foreign key relationships are grouped together and processed sequentially to maintain referential integrity.
- **Independent Tables**: Tables without foreign key relationships can be processed in parallel.
- **Resource Usage**: More workers mean more CPU and I/O usage. Monitor your system resources when increasing this value.

- **Worker Availability**: The actual number of workers used may be limited by PostgreSQL's `max_worker_processes` configuration.
- **Logs**: In this first version of parallel static masking, dynamic background workers are used. So there is no communication between main and background process. That's why errors are logged into the PostgreSQL log file, but not in stdout. If anonymize_database_parallel response is False, then check pg logs.
- **atomic function**: The anon.anonymize_database_parallel() is NOT an atomic function. If a worker fails, the others will not rollback, leaving the database in an intermediate state: some tables will be anonymized, while others will remain unchanged. In such situation, check the logs to find which tables were not anonymized, and retry to apply the masking rules with anon.anonymize_table().

Faster masking by dropping indexes

!!! warning "Beta"

`This option is still considered **beta** for now: please consider opening an issue if you run into any bug.`

Static masking keeps every secondary index up to date while it rewrites the masked rows, which is often the slowest part on a large, heavily-indexed table. When `anon.static_masking.drop_indexes` is enabled, `anonymize_table()` (and therefore `anonymize_database()`) drops a table's plain secondary indexes before masking it and rebuilds them afterwards, which is usually much faster than maintaining them row by row:

```
SET anon.static_masking.drop_indexes = on;
SELECT anon.anonymize_database();
ANALYZE;
```

The parameter is `off` by default and only affects performance, never the masking result. Enable it for large tables carrying several secondary indexes; on a primary-key-only table there is nothing to drop and it makes no difference.

Important considerations:

- **Atomicity**: the drop, the masking and the rebuild run in the same transaction, so a failure restores the indexes along with the data.
- **Indexes kept in place**: unique, constraint (primary key, unique, exclusion), replica-identity, clustered, non-default-tablespace, invalid and partitioned indexes are never dropped, along with indexes carrying a custom per-column statistics target. A partitioned table's indexes cannot be dropped individually, so the option does nothing for it directly. Only standalone indexes on the leaf partitions you mask benefit.
- **Tablespace**: a dropped index is rebuilt where it was, on the database's default tablespace, so a non-empty `default_tablespace` never silently moves it (indexes on a non-default tablespace are kept, as noted above).

- **Deferrable constraints**: a table involved in a deferrable constraint, on either side, is skipped entirely with a `NOTICE`. Masking emits `SET CONSTRAINTS ALL DEFERRED`, which forbids `CREATE INDEX` on that table in the same transaction.
- **Locking**: this option raises the table lock for the masking from `ROW EXCLUSIVE` to `ACCESS EXCLUSIVE`, so it also blocks concurrent reads.
- **Parallel masking**: a session-level `SET` is not seen by the background workers of `anonymize_database_parallel()`; set it with `ALTER DATABASE` or `ALTER SYSTEM` for those. Those workers disable nested parallelism, so the index rebuild runs single-threaded there; keep `max_worker_processes` above the number of `jobs`. `anonymize_column()` is not affected.
- **Statistics**: rebuilding an index does not restore the statistics `ANALYZE` may have gathered on an expression index. Run `ANALYZE` after masking with this option (good practice anyway once the data has changed).

Shuffling

Shuffling mixes values within the same columns.

- `anon.shuffle_column(shuffle_table, shuffle_column, primary_key)` will rearrange all values in a given column. You need to provide a primary key of the table.

This is useful for foreign keys because referential integrity will be kept.

IMPORTANT: `shuffle_column()` is not a masking function because it works "vertically" : it will modify all the values of a column at once.

Adding noise to a column

There are also some functions that can add noise on an entire column:

- `anon.add_noise_on_numeric_column(table, column, ratio)` if ratio = 0.33, all values of the column will be randomly shifted with a ratio of +/- 33%
- `anon.add_noise_on_datetime_column(table, column, interval)` if interval = '2 days', all values of the column will be randomly shifted by +/- 2 days

IMPORTANT : These noise functions are vulnerable to a form of repeat attack. See `demo/noise_reduction_attack.sql` for more details. — title: support draft: false toc: true —

Support

This page lists the PostgreSQL versions and platforms that PostgreSQL Anonymizer officially supports. To install it, see Install.

Supported versions

This extension is available in two versions :

- `stable` is recommended for production
- `latest` is useful if you want to test new features

Compatibility

PostgreSQL Anonymizer is designed to work on the most current setups. As we are trying to find the right balance between innovation and backward compatibility, we define a comprehensive list of platforms and software that we officially support for each major version.

Version	Released	EOL	Postgres	OS
4.x (WIP)	jan. 2027	jan. 2028	15 to 19	RHEL 9 & 10, Debian 13 & 14, Ubuntu 26.04
3.x	jan. 2026	jan. 2027	14 to 18	RHEL 8, 9 & 10, Debian 12 & 13, Ubuntu 24.04
2.x	dec. 2024	jan. 2026	13 to 17	RHEL 8 & 9, Debian 11 & 12, Ubuntu 24.04
1.3	mar. 2024	dec. 2024	12 to 16	RHEL 8 & 9
1.2	jan. 2024	mar. 2024	12 to 16	RHEL 8 & 9
1.1	sept. 2022	jan. 2024	11 to 15	RHEL 7 & 8

The extension may work on other distributions than the ones above, however we provide packages only for these versions and we do not guarantee free community support for other OS.

Support policy

!!! Note

```
Some cloud providers, editors and distributions don't follow our release
cycle and may offer an outdated version. For our part, we only provide
community support for the `stable` and `latest` versions. Whatever the
platform, always refer to its own documentation for the exact install
procedure. Extended Long Term Support is available via commercial support -

contact our [sales team](mailto:contact@dalibo.com) for more details.
```

Welcome to Paul's Boutique !

This is a 5 hours workshop that demonstrates various anonymization techniques using the PostgreSQL Anonymizer extension.

The Story

Paul's boutique

Paul's boutique has a lot of customers. Paul asks his friend Pierre, a Data Scientist, to make some statistics about his clients : average age, etc…

Pierre wants a direct access to the database in order to write SQL queries.

Jack is an employee of Paul. He's in charge of relationship with the various suppliers of the shop.

Paul respects his suppliers privacy. He needs to hide the personal information to Pierre, but Jack needs read and write access the real data.

Objectives

Using the simple example above, we will learn:

- How to write masking rules
- The difference between static and dynamic masking
- Implementing advanced masking techniques

About GDPR

This tutorial **does not** go into the details of the GPDR act and the general concepts of anonymization.

For more information about it, please refer to the talk below:

- Anonymisation, Au-delà du RGPD (Video / French)
- Anonymization, Beyond GDPR (PDF / english)

Requirements

In order to make this workshop, you will need:

- A Linux VM (preferably `Debian 12 bookworm` or `Ubuntu 24.04`)
- A PostgreSQL instance (preferably `PostgreSQL 17`)
- The PostgreSQL Anonymizer (anon) extension, installed and initialized by a superuser
- A database named "boutique" owned by a **superuser** called "paul"
- A role "pierre" and a role "jack", both allowed to connect to the database "boutique"

Check out the INSTALL section to learn how to install the PostgreSQL Anonymizer extension:

!!! tip

`A simple way to deploy a workshop environment is to install [Docker Desktop] and download the image below:`

```
ANON_IMG=registry.gitlab.com/dalibo/postgresql_anonymizer:stable
docker pull $ANON_IMG
```

And you can then launch it with:

```
docker run --name anon_tuto --detach -e POSTGRES_PASSWORD=x $ANON_IMG
docker exec -it anon_tuto psql -U postgres
```

!!! tip

```
Check out the [INSTALL section](https://postgresql-anonymizer.readthedocs.io/en/stable/INSTA
in the [documentation](https://postgresql-anonymizer.readthedocs.io/en/stable/)
to learn how to install the extension in your PostgreSQL instance.
```

The Roles

We will with 3 different users:

```
CREATE ROLE paul LOGIN SUPERUSER PASSWORD 'CHANGEME';

CREATE ROLE pierre LOGIN PASSWORD 'CHANGEME';

CREATE ROLE jack LOGIN PASSWORD 'CHANGEME';
GRANT pg_read_all_data TO jack;
GRANT pg_write_all_data TO jack;
```

Unless stated otherwise, all commands must be executed with the role `paul`.

!!! tip

```
Setup a `.pgpass` file to simplify the connections !
```

```
cat > ~/.pgpass << EOL
*:*:boutique:paul:CHANGEME
*:*:boutique:pierre:CHANGEME
*:*:boutique:jack:CHANGEME
EOL
chmod 0600 ~/.pgpass
```

The Sample database

We will work on a database called “boutique”:

```
CREATE DATABASE boutique OWNER paul;
```

We need to activate the `anon` library inside that database:

```
ALTER DATABASE boutique
  SET session_preload_libraries = 'anon';
```

1- Static Masking

Static Masking is the simplest way to hide personal information! This idea is simply to destroy the original data or replace it with an artificial one.

Requirements

Please check out the intro of this tutorial if you haven't read it yet

The story

Over the years, Paul has collected data about his customers and their purchases in a simple database. He recently installed a brand new sales application and the old database is now obsolete. He wants to save it and he would like to remove all personal information before archiving it.

How it works

Learning Objective

In this section, we will learn:

- How to write simple masking rules
- The advantage and limitations of static masking
- The concept of "Singling Out" a person

The "customer" table

```
DROP TABLE IF EXISTS customer CASCADE;

DROP TABLE IF EXISTS payout CASCADE;

CREATE TABLE customer (
    id SERIAL PRIMARY KEY,
    firstname TEXT,
    lastname TEXT,
    phone TEXT,
    birth DATE,
    postcode TEXT
);
```

Insert a few persons:

```
INSERT INTO customer
VALUES
(107,'Sarah','Conor','060-911-0911', '1965-10-10', '90016'),
(258,'Luke', 'Skywalker', NULL, '1951-09-25', '90120'),
```

```
(341,'Don', 'Draper','347-515-3423', '1926-06-01', '04520')
;

SELECT * FROM customer;
```

id	firstname	lastname	phone	birth	postcode
107	Sarah	Conor	060-911-0911	1965-10-10	90016
258	Luke	Skywalker	None	1951-09-25	90120
341	Don	Draper	347-515-3423	1926-06-01	04520

The "payout" table

Sales are tracked in a simple table:

```
CREATE TABLE payout (
    id SERIAL PRIMARY KEY,
    fk_customer_id INT REFERENCES customer(id),
    order_date DATE,
    payment_date DATE,
    amount INT
);
```

Let's add some orders:

```
INSERT INTO payout
VALUES
(1,107,'2021-10-01','2021-10-01', '7'),
(2,258,'2021-10-02','2021-10-03', '20'),
(3,341,'2021-10-02','2021-10-02', '543'),
(4,258,'2021-10-05','2021-10-05', '12'),
(5,258,'2021-10-06','2021-10-06', '92')
;
```

Activate the extension

```
CREATE EXTENSION IF NOT EXISTS anon;
```

Declare the masking rules

Paul wants to hide the last name and the phone numbers of his clients. He will use the `dummy_last_name()` and `partial()` functions for that:

```
SECURITY LABEL FOR anon ON COLUMN customer.lastname
  IS 'MASKED WITH FUNCTION anon.dummy_last_name()';

SECURITY LABEL FOR anon ON COLUMN customer.phone
  IS 'MASKED WITH FUNCTION anon.partial(phone,2,$$X-XXX-XX$$,2)';
```

Apply the rules permanently

```
SELECT anon.anonymize_table('customer');
```

anonymize_table
True

```
SELECT id, firstname, lastname, phone
FROM customer;
```

id	firstname	lastname	phone
107	Sarah	Greenholt	06X-XXX-XX11
258	Luke	Jacobi	None
341	Don	Howell	34X-XXX-XX23

> This is called `Static Masking` because the **real data has been permanently replaced**. We'll see later how we can use dynamic anonymization or anonymous exports.

Exercises

E101 - Mask the client's first names

Declare a new masking rule and run the static anonymization function again.

E102 - Hide the last 3 digits of the postcode

Paul realizes that the postcode gives a clear indication of where his customers live. However he would like to have statistics based on their postcode area.

Add a new masking rule to replace the last 3 digits by 'x'.

E103 - Count how many clients live in each postcode area?

Aggregate the customers based on their anonymized postcode.

E104 - Keep only the year of each birth date

Paul wants age-based statistic. But he also wants to hide the real birth date of the customers.

Replace all the birth dates by January 1rst, while keeping the real year.

> You can use the make_date or date_trunc functions !

See https://www.postgresql.org/docs/current/functions-datetime.html#FUNCTIONS-DATETIME-TABLE

E105 - Singling out a customer

Even if the "customer" is properly anonymized, we can still isolate a given individual based on data stored outside of the table. For instance, we can identify the best client of Paul's boutique with a query like this:

```
WITH best_client AS (
    SELECT SUM(amount), fk_customer_id
    FROM payout
    GROUP BY fk_customer_id
    ORDER BY 1 DESC
    LIMIT 1
)
SELECT c.*
FROM customer c
JOIN best_client b ON (c.id = b.fk_customer_id)
```

id	firstname	lastname	phone	birth	postcode
341	Don	Howell	34X-XXX-XX23	1926-06-01	04520

This is called **Singling Out a person.**

We need to anonymize even further by removing the link between a person and its company. In the `payout` table, this link is materialized by a foreign key on the field `fk_customer_id`. However we can't remove values from this column or insert fake identifiers because if would break the foreign key constraint.

How can we separate the customers from their payouts while respecting the integrity of the data?

Find a function that will shuffle the column `fk_customer_id` of the `payout` table

Check out the shuffling section of the documentation.

Solutions

S101

```
SECURITY LABEL FOR anon ON COLUMN customer.firstname
IS 'MASKED WITH FUNCTION anon.dummy_first_name()';

SELECT anon.anonymize_table('customer');
```

```
SELECT id, firstname, lastname
FROM customer;
```

S102

```
SECURITY LABEL FOR anon ON COLUMN customer.postcode
IS 'MASKED WITH FUNCTION anon.partial(postcode,2,$$xxx$$,0)';

SELECT anon.anonymize_table('customer');

SELECT id, firstname, lastname, postcode
FROM customer;
```

S103

```
SELECT postcode, COUNT(id)
FROM customer
GROUP BY postcode;
```

postcode	count
90xxx	2
04xxx	1

S104

```
SECURITY LABEL FOR anon ON FUNCTION pg_catalog.date_trunc(text,interval)
  IS 'TRUSTED';

SECURITY LABEL FOR anon ON COLUMN customer.birth
  IS $$ MASKED WITH FUNCTION pg_catalog.date_trunc('year',birth) $$;

SELECT anon.anonymize_table('customer');

SELECT id, firstname, lastname, birth
FROM customer;
```

S105

Let's mix up the values of the `fk_customer_id`:

```
SELECT anon.shuffle_column('payout','fk_customer_id','id');
```

shuffle_column
True

Now let's try to single out the best client again :

```
WITH best_client AS (
    SELECT SUM(amount), fk_customer_id
    FROM payout
    GROUP BY fk_customer_id
    ORDER BY 1 DESC
    LIMIT 1
)
SELECT c.*
FROM customer c
JOIN best_client b ON (c.id = b.fk_customer_id);
```

id	firstname	lastname	phone	birth	postcode
258	Jordyn	Simonis	None	1951-01-01	90xxx

WARNING

Note that the link between a `customer` and its `payout` is now completely false. For instance, if a customer A had 2 payouts. One of these payout may be linked to a customer B, while the second one is linked to a customer C.

In other words, this shuffling method with respect the foreign key constraint (aka the referential integrity) but it will break the data integrity. For some use case, this may be a problem.

In this case, Pierre will not be able to produce a BI report with the shuffle data, because the links between the customers and their payments are fake. — title: tutorials/2-dynamic_masking draft: false toc: true —

2- Dynamic Masking

With Dynamic Masking, the database owner can hide personal data for some users, while other users are still allowed to read and write the authentic data.

Requirements

Please check out the intro of this tutorial if you haven't read it yet

The Story

Paul has 2 employees:

- Jack is operating the new sales application, he needs access to the real data. He is what the GPDR would call a **"data processor"**.
- Pierre is a data analyst who runs statistic queries on the database. He should not have access to any personal data.

How it works

Objectives

In this section, we will learn:

- How to write simple masking rules
- The advantage and limitations of dynamic masking
- The concept of "Linkability" of a person

The `company` table

```
DROP TABLE IF EXISTS supplier CASCADE;

DROP TABLE IF EXISTS company CASCADE;

CREATE TABLE company (
    id SERIAL PRIMARY KEY,
    name TEXT,
    vat_id TEXT UNIQUE
);
```

```
INSERT INTO company
VALUES
(952,'Shadrach', 'FR62684255667'),
(194,E'Johnny\'s Shoe Store','CHE670945644'),
(346,'Capitol Records','GB663829617823')
;
```

```
SELECT * FROM company;
```

id	name	vat_id
952	Shadrach	FR62684255667
194	Johnny's Shoe Store	CHE670945644
346	Capitol Records	GB663829617823

The supplier table

```
CREATE TABLE supplier (
    id SERIAL PRIMARY KEY,
    fk_company_id INT REFERENCES company(id),
    contact TEXT,
    phone TEXT,
    job_title TEXT
);
```

```
INSERT INTO supplier
VALUES
(299,194,'Johnny Ryall','597-500-569','CEO'),
(157,346,'George Clinton', '131-002-530','Sales manager')
;
```

```
SELECT * FROM supplier;
```

id	fk_company_id	contact	phone	job_title
299	194	Johnny Ryall	597-500-569	CEO
157	346	George Clinton	131-002-530	Sales manager

Activate the extension

```
ALTER DATABASE boutique
  SET session_preload_libraries TO 'anon';

CREATE EXTENSION IF NOT EXISTS anon;

SELECT anon.init();
```

Dynamic Masking

Activate the masking engine

```
ALTER DATABASE boutique
  SET anon.transparent_dynamic_masking TO true;
```

Masking a role

```
SECURITY LABEL FOR anon ON ROLE pierre IS 'MASKED';

GRANT pg_read_all_data to pierre;
```

Now connect as Pierre and try to read the supplier table:

```
SELECT * FROM supplier;
```

id	fk_company_id	contact	phone	job_title
299	194	Johnny Ryall	597-500-569	CEO
157	346	George Clinton	131-002-530	Sales manager

For the moment, there is no masking rule so Pierre can see the original data in each table.

Masking the supplier names

Connect as Paul and define a masking rule on the supplier table:

```
SECURITY LABEL FOR anon ON COLUMN supplier.contact
  IS 'MASKED WITH VALUE $$CONFIDENTIAL$$';
```

Now connect as Pierre and try to read the supplier table again:

```
SELECT * FROM supplier;
```

id	fk_company_id	contact	phone	job_title
299	194	CONFIDENTIAL	597-500-569	CEO
157	346	CONFIDENTIAL	131-002-530	Sales manager

Now connect as Jack and try to read the real data:

```
SELECT * FROM supplier;
```

id	fk_company_id	contact	phone	job_title
299	194	Johnny Ryall	597-500-569	CEO
157	346	George Clinton	131-002-530	Sales manager

Exercises

E201 - Guess who is the CEO of "Johnny's Shoe Store"

Masking the supplier contact is clearly not enough to provide anonymity.

Connect as Pierre and write a simple SQL query that joins the `supplier` and the `company` tables. See how that could reindentify some suppliers based on their job and their company.

With this request we managed to link a person to a company and we know it's job title. Since company names and job positions are available in many public

datasets: a simple search on Linkedin or Google would give us the real names of many of the employees of these companies…

This is called **Linkability**: the ability to connect multiple records concerning the same data subject.

E202 - Anonymize the companies

We need to anonymize the `company` table, too. Even if they don't contain personal information, some fields can be used to **infer** the identity of their employees…

Connect as Paul and write 2 masking rules (security labels) for the company table.

- The first one will replace the `name` field with a fake name.
- The second rule will replace the `vat_id` with a random sequence of 10 characters

Go to thedocumentation and look at the faking functions and the random functions !

Connect as Pierre and check that he cannot view the real company info.

Connect as Jack and check that he can view the real values.

E203 - Pseudonymize the company name

Because of dynamic masking, the fake values will be different every time Pierre tries to read the table.

Pierre would like to have always the same fake values for a given company.

This is called pseudonymization.

Connect as Paul and write a new masking rule over the `vat_id` field by generating a hash of 10 characters using the `anon.digest()` function.

Write a new masking rule over the `name` field by using a pseudonymizing function.

Solutions

S201

```
SELECT s.id, s.contact, s.job_title, c.name
FROM supplier s
JOIN company c ON s.fk_company_id = c.id;
```

id	contact	job_title	name
299	CONFIDENTIAL	CEO	Johnny's Shoe Store

id	contact	job_title	name
157	CONFIDENTIAL	Sales manager	Capitol Records

S202

```
SECURITY LABEL FOR anon ON COLUMN company.name
  IS 'MASKED WITH FUNCTION anon.dummy_company_name()';

SECURITY LABEL FOR anon ON COLUMN company.vat_id
IS 'MASKED WITH FUNCTION anon.random_string(10)';
```

Now connect as Pierre and read the table again:

```
SELECT * FROM company;
```

id	name	vat_id
952	Quigley and Feest and Sons	IQz8nNJzNN
194	McClure and Sons	wxpIxYMWUw
346	Maggio and Heaney and Sons	eDlESuTcWE

Pierre will see different "fake data" every time he reads the table:

```
SELECT * FROM company;
```

id	name	vat_id
952	Huel and Rath Group	p3bxk1BLGZ
194	Ankunding Inc	cDp1XFJRtJ
346	Thompson and Schultz Group	JcPoPogE6U

Jack still sees the real data

```
SELECT * FROM company;
```

id	name	vat_id
952	Shadrach	FR62684255667
194	Johnny's Shoe Store	CHE670945644
346	Capitol Records	GB663829617823

S203

```
SECURITY LABEL FOR anon ON COLUMN company.vat_id
IS $$ MASKED WITH FUNCTION anon.left(anon.digest(vat_id, 'xxx', 'md5'),10) $$;
```

```
SECURITY LABEL FOR  anon ON COLUMN company.name
  IS 'MASKED WITH FUNCTION anon.pseudo_company(id)';
```

Connect as Pierre and read the table multiple times:

```
SELECT * FROM company;
```

id	name	vat_id
952	Wilkinson LLC	2db762afa4
194	Johnson PLC	61fddf8d83
346	Young-Carpenter	86fe3f164c

```
SELECT * FROM company;
```

id	name	vat_id
952	Wilkinson LLC	2db762afa4
194	Johnson PLC	61fddf8d83
346	Young-Carpenter	86fe3f164c

Now the fake company name is always the same.

title: tutorials/3-anonymous_dumps draft: false toc: true —

3- Anonymous Dumps

In many situation, what we want is basically to export the anonymized data into another database (for testing or to produce statistics). We will simply use pg_dump for that !

The Story

Paul has a website and a comment section where customers can express their views.

He hired a web agency to develop a new design for his website. The agency asked for a SQL export (dump) of the current website database. Paul wants to `clean` the database export and remove any personal information contained in the comment section.

How it works

Learning Objective

- Extract the anonymized data from the database
- Write a custom masking function to handle a JSON field.

Load the data

```
DROP TABLE IF EXISTS website_comment CASCADE;

CREATE TABLE website_comment (
  id SERIAL PRIMARY KEY,
  message JSONB
);

INSERT INTO website_comment
VALUES
  (1, json_build_object(
        'meta', json_build_object(
          'name', 'Lee Perry',
          'ip_addr','40.87.29.113'),
        'content', 'Hello Nasty!')),
  (2, json_build_object(
        'meta', json_build_object(
          'name', '',
          'email', 'biz@bizmarkie.com'),
        'content', 'Great Shop')),
  (3,json_build_object(
        'meta', json_build_object(
          'name','Jimmy'),
        'content','Hi ! This is me, Jimmy James'));
```

Check the content of the website comments:

```
SELECT
  message->'meta'->'name' AS name,
  message->'content' AS content
FROM website_comment
ORDER BY id ASC;
```

name	content
Lee Perry	Hello Nasty!
	Great Shop
Jimmy	Hi ! This is me, Jimmy James

Activate the extension

```
CREATE EXTENSION IF NOT EXISTS anon;
```

Masking a JSON column

The `comment` field is filled with personal information and the fact the field does not have a standard schema makes our tasks harder.

In general, unstructured data are difficult to mask.

As we can see, web visitors can write any kind of information in the comment section. Our best option is to remove this key entirely because there's no way to extract personal data properly.

We can *clean* the comment column simply by removing the `content` key in the `message` column !

```
SELECT message - ARRAY['content'] AS message_without_content
FROM website_comment
WHERE id=1;
```

message_without_content
{'meta': {'name': 'Lee Perry', 'ip_addr': '40.87.29.113'}}

First let's create a dedicated schema and declare it as trusted. This means the `anon` extension will accept the functions located in this schema as valid masking functions. Only a superuser should be able to add functions in this schema.

```
CREATE SCHEMA IF NOT EXISTS my_masks;

SECURITY LABEL FOR anon ON SCHEMA my_masks IS 'TRUSTED';
```

Now we can write a function that remove the message content:

```
CREATE OR REPLACE FUNCTION my_masks.remove_content(j JSONB)
RETURNS JSONB
AS $func$
  SELECT j - ARRAY['content']
$func$
LANGUAGE SQL
;
```

Let's try it!

```
SELECT my_masks.remove_content(message)
FROM website_comment;
```

remove_content
{'meta': {'name': 'Lee Perry', 'ip_addr': '40.87.29.113'}}
{'meta': {'name': '', 'email': 'biz@bizmarkie.com'}}

remove_content
{'meta': {'name': 'Jimmy'}}

And now we can use it in a masking rule:

```
SECURITY LABEL FOR anon ON COLUMN website_comment.message
IS 'MASKED WITH FUNCTION my_masks.remove_content(message)';
```

Then we need to create a dedicated role to export the masked data. We will call this role `anon_dumper` (the name does not matter) and declare that this role is masked.

```
DROP ROLE IF EXISTS anon_dumper;

CREATE ROLE anon_dumper LOGIN PASSWORD 'CHANGEME';

ALTER ROLE anon_dumper SET anon.transparent_dynamic_masking TO TRUE;

SECURITY LABEL FOR anon ON ROLE anon_dumper IS 'MASKED';

GRANT pg_read_all_data TO anon_dumper;
```

For convenience, add a new entry in the `.pgpass` file.

```
cat > ~/.pgpass << EOL
*:*:boutique:anon_dumper:CHANGEME
EOL
```

Finally we can export an **anonymous dump** of the table with `pg_dump`:

```
export PATH=$PATH:$(pg_config --bindir)
export PGHOST=localhost
pg_dump -U anon_dumper boutique --table=website_comment > /tmp/dump.sql
```

Exercises

E301 - Dump the anonymized data into a new database

Create a database named `boutique_anon` and transfer the entire database into it.

E302 - Remove the email address

Replace the `remove_content` function with a better one called `remove_content_and_ip` that will nullify the `email` key.

HINT: you can use `jsonb_set(message, '{meta, email}', '{}')` to remove the email value.

E303 - Pseudonymize the IP address

Pierre plans to extract general information from the metadata. For instance, he wants to calculate the number of unique visitors based on the different IP addresses.

But an IP address is an **indirect identifier**, so Paul needs to anonymize this field while maintaining the fact that some values appear multiple times.

HINT: First you can create a new `meta` object using `jsonb_build_object()` and then use function `jsonb_set` replace the `meta` key

Solutions

S301

```
export PATH=$PATH:$(pg_config --bindir)
export PGHOST=localhost
dropdb -U paul --if-exists boutique_anon
createdb -U paul boutique_anon --owner paul
pg_dump -U anon_dumper boutique | psql -U paul --quiet boutique_anon
```

```
export PGHOST=localhost
psql -U paul boutique_anon -c 'SELECT COUNT(*) FROM company'
```

S302

```
CREATE OR REPLACE FUNCTION my_masks.remove_content_and_ip(message JSONB)
RETURNS JSONB
VOLATILE
LANGUAGE SQL
AS $func$
SELECT
  jsonb_set(message, '{meta, email}', '{}')
  - ARRAY['content'];
$func$;
```

```
SELECT my_masks.remove_content_and_ip(message)
FROM website_comment;
```

remove_content_and_ip
{'meta': {'name': 'Lee Perry', 'email': {}, 'ip_addr': '40.87.29.113'}}
{'meta': {'name': '', 'email': {}}}
{'meta': {'name': 'Jimmy', 'email': {}}}

```
SECURITY LABEL FOR anon ON COLUMN website_comment.message
IS 'MASKED WITH FUNCTION my_masks.remove_content_and_ip(message)';
```

S303

```
CREATE OR REPLACE FUNCTION my_masks.clean_comment(message JSONB)
RETURNS JSONB
VOLATILE
LANGUAGE SQL
AS $func$
SELECT
  jsonb_set(
    message,
    ARRAY['meta'],
    jsonb_build_object(
        'name',anon.fake_last_name(),
        'ip_address', md5((message->'meta'->'ip_addr')::TEXT),
        'email', NULL
    )
  ) - ARRAY['content'];
$func$;
```

```
SELECT my_masks.clean_comment(message)
FROM website_comment;
```

clean_comment
{'meta': {'name': 'Rollins', 'email': None, 'ip_address': '1d8cbcdef988d55982af1536922ddcd1'}}
{'meta': {'name': 'Harper', 'email': None, 'ip_address': None}}
{'meta': {'name': 'Ingram', 'email': None, 'ip_address': None}}

```
SECURITY LABEL FOR anon ON COLUMN website_comment.message
IS 'MASKED WITH FUNCTION my_masks.clean_comment(message)';
```

4- Generalization

The main idea of generalization is to `blur` the original data. For example, instead of saying `Mister X was born on July 25, 1989`, we can say `Mister X was born is the 80's`. The information is still true, but it is less precise and it can't be used to reidentify the subject.

The Story

Paul hired dozens of employees over the years. He kept a record of their hair color, size and medical condition.

Paul wants to extract weird stats from these details. He provides generalized views to Pierre.

How it works

Learning Objective

In this section, we will learn:

- The difference between masking and generalization
- The concept of `K-anonymity`

The `employee` table

```
DROP TABLE IF EXISTS employee CASCADE;

CREATE TABLE employee (
  id INT PRIMARY KEY,
  full_name  TEXT,
  first_day DATE, last_day DATE,
  height INT,
  hair TEXT, eyes TEXT, size TEXT,
  asthma BOOLEAN,
  CHECK(hair = ANY(ARRAY['bald','blond','dark','red'])),
  CHECK(eyes = ANY(ARRAY['blue','green','brown'])) ,
  CHECK(size = ANY(ARRAY['S','M','L','XL','XXL']))
);
```

This is awkward and illegal.

Loading the data:

```
INSERT INTO employee
  VALUES
(1,'Luna Dickens','2018-07-22','2018-12-15',180,'blond','blue','L',True),
(2,'Paul Wolf','2020-01-15',NULL,177,'bald','brown','M',False),
(3,'Rowan Hoeger','2018-12-01','2018-12-15',202,'dark','blue','XXL',True)
;
```

```
SELECT count(*) FROM employee;
```

count
3

```
SELECT full_name,first_day, hair, size, asthma
FROM employee
LIMIT 3;
```

full_name	first_day	hair	size	asthma
Luna Dickens	2018-07-22	blond	L	True
Paul Wolf	2020-01-15	bald	M	False

full_name	first_day	hair	size	asthma
Rowan Hoeger	2018-12-01	dark	XXL	True

Data suppression

Paul wants to find if there's a correlation between asthma and the eyes color.

He provides the following view to Pierre.

```
DROP MATERIALIZED VIEW IF EXISTS v_asthma_eyes;

CREATE MATERIALIZED VIEW v_asthma_eyes AS
SELECT eyes, asthma
FROM employee;

SELECT *
FROM v_asthma_eyes
LIMIT 3;
```

eyes	asthma
blue	True
brown	False
blue	True

Pierre can now write queries over this view.

```
SELECT
  eyes,
  100*COUNT(1) FILTER (WHERE asthma) / COUNT(1) AS asthma_rate
FROM v_asthma_eyes
GROUP BY eyes;
```

eyes	asthma_rate
brown	0
blue	100

Pierre just proved that asthma is caused by blue eyes ;-)

K-Anonymity

The `asthma` and `eyes` columns are considered as indirect identifiers.

Indirect personal identifiers (or "quasi-identifiers") are pieces of information that, when combined with other data can identify an individual. Examples of

indirect identifiers include: Date of birth, Gender, Zip code, etc.

With PostgreSQL Anonymizer, we can declare that a column is an indirect identifiers, like this:

```
SECURITY LABEL FOR k_anonymity
  ON COLUMN v_asthma_eyes.eyes
  IS 'INDIRECT IDENTIFIER';

SECURITY LABEL FOR k_anonymity
  ON COLUMN v_asthma_eyes.asthma
  IS 'INDIRECT IDENTIFIER';

SELECT anon.k_anonymity('v_asthma_eyes');
```

k_anonymity
1

The v_asthma_eyes has '2-anonymity'. This means that each quasi-identifier combination (the 'eyes-asthma' tuples) occurs in at least 2 records for a dataset.

In other words, it means that each individual in the view cannot be distinguished from at least 1 (k-1) other individual.

Range and Generalization functions

Now let's add another view over the `employee` table.

We will generalize the dates of to keep only the month and year.

```
DROP MATERIALIZED VIEW IF EXISTS v_staff_per_month;
CREATE MATERIALIZED VIEW v_staff_per_month AS
SELECT
    anon.generalize_daterange(first_day,'month') AS first_day,
    anon.generalize_daterange(last_day,'month') AS last_day
FROM employee;

SELECT *
FROM v_staff_per_month
LIMIT 3;
```

first_day	last_day
[2018-07-01, 2018-08-01)	[2018-12-01, 2019-01-01)
[2020-01-01, 2020-02-01)	(None, None)
[2018-12-01, 2019-01-01)	[2018-12-01, 2019-01-01)

Pierre can write a query to find how many employees were hired in november 2021.

```
SELECT COUNT(1)
       FILTER (
            WHERE make_date(2019,11,1)
            BETWEEN lower(first_day)
            AND COALESCE(upper(last_day),now())
       )
FROM v_staff_per_month;
```

count
0

Declaring the indirect identifiers

Now let's check the k-anonymity of this view by declaring which columns are indirect identifiers :

```
SECURITY LABEL FOR k_anonymity
  ON COLUMN v_staff_per_month.first_day
  IS 'INDIRECT IDENTIFIER';

SECURITY LABEL FOR k_anonymity
  ON COLUMN v_staff_per_month.last_day
  IS 'INDIRECT IDENTIFIER';

SELECT anon.k_anonymity('v_staff_per_month');
```

In this case, the k factor is 1 which means that there is at least one unique individual who be identified directly by his/her first and last dates.

Exercises

E401 - Simplify `v_staff_per_month` and decrease granularity

Generalizing dates per month is not enough. Write another view called `v_staff_per_year` that will generalize dates per year.

Also simplify the view by using a range of int to store the years instead of a date range.

E402 - Staff progression over the years

How many people worked for Paul for each year between 2018 and 2021?

E403 - Reaching 2-anonymity for the `v_staff_per_year` view

What is the k-anonymity of `v_staff_per_month_years`?

Solutions

S401

```
DROP MATERIALIZED VIEW IF EXISTS v_staff_per_year;

CREATE MATERIALIZED VIEW v_staff_per_year AS
SELECT
  int4range(
    extract(year from first_day)::INT,
    extract(year from last_day)::INT,
    '[]'
  ) AS period
FROM employee;
```

`'[]'` will include the upper bound

```
SELECT *
FROM v_staff_per_year
LIMIT 3;
```

period
[2018, 2019)
[2020, None)
[2018, 2019)

S402

```
SELECT
  year,
  COUNT(1) FILTER (
      WHERE year <@ period
  )
FROM
    generate_series(2018,2021) year,
    v_staff_per_year
GROUP BY year
ORDER BY year ASC;
```

year	count
2018	2
2019	0
2020	1
2021	1

S403

```
SECURITY LABEL FOR k_anonymity
  ON COLUMN v_staff_per_year.period
  IS 'INDIRECT IDENTIFIER';

SELECT anon.k_anonymity('v_staff_per_year');
```

5- Anonymized Replica

With logical replication, you can maintain a replica that is always up to date and always anonymized. This feature was introduced in version 2.3 and became generally available in version 3.0.

For more details, check out the Replica Masking section of the documentation.

The Story

Pierre needs permanent access to the boutique data to run his analytics. With dynamic masking, he was given a masked view of the production database. But Paul would now prefer to keep Pierre away from the production server entirely.

The solution is a dedicated, anonymized replica: Pierre gets a read-only, always-fresh, fully masked copy of the data, and never touches production.

How it works

Anon Anonymized Replica image

Learning Objective

In this section, we will learn:

- How to configure the publication on the source database
- How to configure the anonymized replica
- How to verify the anonymization in real time

A note on this setup

In production, the anonymized replica is a **separate PostgreSQL server**.

To keep this tutorial self-contained, the source and the replica are two databases of the **same instance**: `boutique` (the source) and `boutique_replica` (the anonymized replica). Logical replication works just as well between two databases of the same cluster.

This chapter creates cluster-wide objects (a replication role, a publication, a replication slot and a second database). If you want to replay it from scratch, clean them up first (see the Conclusion).

Prerequisites

Logical replication requires `wal_level = logical`. The default value is `replica`; raising it requires a restart of the instance (check it with `SHOW wal_level;`).

The `anon` extension must also be loaded in every session that declares masking rules. The simplest way to guarantee this is to preload it globally with `shared_preload_libraries = 'anon'`.

The source database

We use the `company` and `supplier` tables, filled with real data:

```
DROP TABLE IF EXISTS supplier CASCADE;
DROP TABLE IF EXISTS company CASCADE;

CREATE TABLE company (
    id INT PRIMARY KEY,
    name TEXT,
    vat_id TEXT UNIQUE
);

INSERT INTO company VALUES
(952,'Shadrach','FR62684255667'),
(194,E'Johnny\'s Shoe Store','CHE670945644'),
(346,'Capitol Records','GB663829617823');

CREATE TABLE supplier (
    id INT PRIMARY KEY,
    fk_company_id INT REFERENCES company(id),
    contact TEXT,
    phone TEXT,
    job_title TEXT
);
```

```
INSERT INTO supplier VALUES
(299,194,'Johnny Ryall','597-500-569','CEO'),
(157,346,'George Clinton','131-002-530','Sales manager');

SELECT * FROM company ORDER BY id;
```

id	name	vat_id
194	Johnny's Shoe Store	CHE670945644
346	Capitol Records	GB663829617823
952	Shadrach	FR62684255667

With replica masking, the source database needs no masking rules: the anonymization happens on the replica, as the data is received.

Publication side: the replication role

On the source, create a dedicated replication role:

```
CREATE ROLE anon_replicator LOGIN REPLICATION PASSWORD 'CHANGEME';

GRANT USAGE  ON SCHEMA public TO anon_replicator;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO anon_replicator;
```

Publication side: create the publication

We only publish the tables that Pierre needs to read. The `customer` and `payout` tables are not published here.

```
CREATE PUBLICATION pub_boutique
  FOR TABLE company, supplier;
```

Check the publication:

```
SELECT pubname, schemaname, tablename FROM pg_publication_tables;
```

pubname	schemaname	tablename
pub_boutique	public	company
pub_boutique	public	supplier

When the replica is **another database of the same cluster**, the replication slot must be created beforehand:

```
SELECT slot_name
FROM pg_create_logical_replication_slot('boutique_slot','pgoutput');
```

slot_name
boutique_slot

!!! warning

```
This is a PostgreSQL restriction, see [CREATE SUBSCRIPTION] for more details.
When the replica is a **separate server** (the production setup), this
restriction does not apply and the slot can be created by
`CREATE SUBSCRIPTION` itself.
```

Replica side: create the database

```
CREATE DATABASE boutique_replica OWNER paul;
```

The `anon.replica_masking` parameter allows the masking triggers to be installed on this database. It must be enabled before calling `anon.start_replica_masking()`, and therefore before the subscription is created:

```
ALTER DATABASE boutique_replica SET anon.replica_masking TO on;
```

If you use a `.pgpass` file, add entries for the new database:

```
cat >> ~/.pgpass << EOL
*:*:boutique_replica:paul:CHANGEME
*:*:boutique_replica:pierre:CHANGEME
EOL
```

Replica side: initialize the extension and the tables

`anon.init()` is required on the replica: the faking and pseudonymization functions rely on the fake datasets it loads.

The tables must also exist **before** the subscription is created, since logical replication does not replicate DDL:

```
CREATE EXTENSION IF NOT EXISTS anon;
SELECT anon.init();

CREATE TABLE company (
    id INT PRIMARY KEY,
    name TEXT,
    vat_id TEXT UNIQUE
);

CREATE TABLE supplier (
    id INT PRIMARY KEY,
    fk_company_id INT REFERENCES company(id),
```

```
    contact TEXT,
    phone TEXT,
    job_title TEXT
);
```

Replica side: masking rules

The rules declared on the source only serve dynamic masking there: it is the replica that applies its **own** rules when the data is received. Declare rules similar to the ones of the Dynamic Masking tutorial:

```
SECURITY LABEL FOR anon ON COLUMN company.name
  IS 'MASKED WITH FUNCTION anon.pseudo_company(id)';

SECURITY LABEL FOR anon ON COLUMN company.vat_id
  IS 'MASKED WITH FUNCTION anon.random_string(10)';

SECURITY LABEL FOR anon ON COLUMN supplier.contact
  IS 'MASKED WITH VALUE $$CONFIDENTIAL$$';

SECURITY LABEL FOR anon ON COLUMN supplier.phone
  IS 'MASKED WITH FUNCTION anon.partial(phone,0,$$XX-XXX-XXX$$,0)';
```

Pseudonymizing `company.name` guarantees that Pierre always sees the same fake name for a given company, which lets him write consistent joins between `company` and `supplier`.

Declare all the masking rules **before** creating the subscription: a rule added later only masks the changes that arrive afterwards (see exercise E501).

Replica side: enable and subscribe

`anon.start_replica_masking()` installs, on each masked table, a trigger declared `ENABLE REPLICA`: it only fires during the replication apply process — which is also why writes performed locally on the replica are not masked.

```
SELECT anon.start_replica_masking();
```

The subscription reuses the slot we created earlier, hence `create_slot=false`:

```
CREATE SUBSCRIPTION sub_boutique
  CONNECTION 'host=localhost port=65432 user=anon_replicator password=CHANGEME dbname=boutic
  PUBLICATION pub_boutique
  WITH (slot_name='boutique_slot', create_slot=false);
```

!!! note

```
In this tutorial environment, the source database listens on port
`65432`. In production, the connection string points to your **source
server** (usually port `5432`). For a two-server setup, also remember
```

that the `pg_hba.conf` of the source must allow replication connections for `anon_replicator`, that raising `wal_level` requires a restart of the source instance, and that the reader roles (such as `pierre`) must be created on the replica instance.

Replication is asynchronous; we wait a moment for the initial copy to complete:

```
SELECT pg_sleep(3);
```

Check the subscription status:

```
SELECT subname, pid, received_lsn, latest_end_lsn
FROM pg_stat_subscription;
```

subname	pid	received_lsn	latest_end_lsn
sub_boutique	124	0/2737630	0/2737630

Verification

On the replica, the data is anonymized, even for Paul:

```
SELECT * FROM company ORDER BY id;
```

id	name	vat_id
194	Johnson PLC	7Amz57dWEa
346	Young-Carpenter	beEQFttDdh
952	Wilkinson LLC	y01TvEu6Ax

Unlike with dynamic masking, **all users** see the anonymized data on the replica: as long as the masking rules are declared before the data arrives, the original data is never stored there. (Exercise E501 shows what happens when a rule comes too late.)

For comparison, on the source the data is still real:

```
SELECT * FROM company ORDER BY id;
```

id	name	vat_id
194	Johnny's Shoe Store	CHE670945644
346	Capitol Records	GB663829617823
952	Shadrach	FR62684255667

A new row inserted on the source is replicated and anonymized automatically:

```
INSERT INTO company VALUES (501,'Acme Corp','US123456789');
```

```
SELECT pg_sleep(2);

SELECT * FROM company WHERE id = 501;
```

id	name	vat_id
501	Knight, Hurley and Clayton	VGr2HQ0av9

Beware of primary keys

Logical replication identifies rows by their replica identity — by default, their primary key:

- With PostgreSQL Anonymizer, every replicated table must have a primary key (`REPLICA IDENTITY FULL` is not supported).
- **Do not mask the columns of the primary key!**
- Otherwise `UPDATE` and `DELETE` statements will no longer propagate.

Technical primary keys (sequences, UUID) are usually not personal data and do not need to be masked.

For more details on masking columns involved in referential integrity, see the Masking Foreign Keys section.

Exercises

As a reminder, unless stated otherwise, all commands are to be run as **paul** (the owner).

E501 - Add a masking rule after the fact

All our masking rules were declared **before** the subscription was created. Declare a new rule on `supplier.job_title` while the replication is running.

What happens to the rows that are already on the replica? How can you fix them?

E502 - A masked role on the replica?

Connect as Pierre and check what he sees on the replica.

Does Pierre need to be declared as a `MASKED` role on `boutique_replica`, like he was on the source? Why or why not?

E503 - Verify the anonymized replication

On the source, insert a new supplier; on the replica, check that it appears and is anonymized.

Solutions

S501

Declare the rule, then refresh the masking triggers with `anon.refresh_replica_masking()`:

```
SECURITY LABEL FOR anon ON COLUMN supplier.job_title
  IS 'MASKED WITH VALUE $$CONFIDENTIAL$$';

SELECT anon.refresh_replica_masking();
```

The triggers only mask **incoming** rows: the suppliers that were replicated before the rule was declared still show their real job titles on the replica!

```
SELECT id, job_title FROM supplier ORDER BY id;
```

id	job_title
157	Sales manager
299	CEO

To fix the existing rows, apply the rules once with static masking (see the Static Masking tutorial). From then on, every incoming row is masked:

```
SELECT anon.anonymize_table('supplier');

SELECT id, job_title FROM supplier ORDER BY id;
```

id	job_title
157	CONFIDENTIAL
299	CONFIDENTIAL

This is why the masking rules should be declared **before** the subscription is created.

S502

Give Pierre read access to the replica database:

```
GRANT USAGE ON SCHEMA public TO pierre;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO pierre;
```

If Pierre already holds the cluster-wide `pg_read_all_data` privilege (granted in the Dynamic Masking tutorial), these grants are redundant. On a standalone replica server, they are required.

```
SELECT s.contact, s.phone, c.name
FROM supplier s
JOIN company c ON s.fk_company_id = c.id;
```

contact	phone	name
CONFIDENTIAL	XX-XXX-XXX	Johnson PLC
CONFIDENTIAL	XX-XXX-XXX	Young-Carpenter

There is no need to declare Pierre as a `MASKED` role on the replica: the anonymization happened at replication time, not at query time. The `MASKED` label that Pierre carries from the Dynamic Masking tutorial has no effect here, because `anon.transparent_dynamic_masking` is not enabled on `boutique_replica`.

S503

```
INSERT INTO supplier VALUES (42,501,'Wile E. Coyote','555-0100','Buyer');
```

```
SELECT pg_sleep(2);
```

```
SELECT s.contact, s.phone, s.job_title, c.name
FROM supplier s
JOIN company c ON s.fk_company_id = c.id
WHERE s.id = 42;
```

contact	phone	job_title	name
CONFIDENTIAL	XX-XXX-XXX	CONFIDENTIAL	Knight, Hurley and Clayton

The new supplier has been replicated and anonymized automatically: the contact and the job title are hidden (the latter by the rule added in E501), the phone is masked and the company name is the same pseudonym as in the previous sections. — title: tutorials/9-conclusion draft: false toc: true —

Conclusion

Clean up !

The Anonymized Replica tutorial created a second database with a subscription that holds a replication slot on `boutique`. We drop the subscription first, otherwise `boutique` could not be dropped. (If you skipped that tutorial, skip the next two blocks.)

```
DROP SUBSCRIPTION IF EXISTS sub_boutique;
```

```
DROP DATABASE IF EXISTS boutique_replica WITH (FORCE);
```

```
DROP DATABASE IF EXISTS boutique WITH (FORCE);
```

```
REASSIGN OWNED BY jack TO postgres;

REASSIGN OWNED BY paul TO postgres;

REASSIGN OWNED BY pierre TO postgres;

DROP ROLE IF EXISTS jack;
DROP ROLE IF EXISTS paul;
DROP ROLE IF EXISTS pierre;
DROP ROLE IF EXISTS dump_anon;
DROP ROLE IF EXISTS anon_replicator;
```

Also...

Other projects you may like

- pg_sample : extract a small dataset from a larger PostgreSQL database

Help Wanted!

This is a free and open project!

labs.dalibo.com/postgresql_anonymizer

Please send us feedback on how you use it, how it fits your needs (or not), etc. — title: tutorials/**DO_NOT_MODIFY_THESE_FILES** draft: false toc: true —

DO NOT MODIFY THESE FILES

The files in the `docs/tutorial` folder are artifacts generated based on the source files in `docs/runbooks`.

If you want to improve the tutorial, edit the `docs/runbooks/*.md` files.

And then run `make tutorial` to update the artifacts.

Upgrade

Currently there's no way to upgrade easily from a version to another. The operation `ALTER EXTENSION ... UPDATE ...` is not supported.

You need to drop and recreate the extension after every upgrade.

Upgrade to version 3.3 and further versions

Salt Rotation

Prior versions contained a bug that would allow malicious masked users to access the secret salt under certain circumstances. Using that salt they may be able to unmask personal information using a brute-force method. It is strongly advise to rotate it, since pseudonymized values may be reversible or guessable.

Subqueries in masking rules

In previous versions, using a subquery inside a maskig rule was allowed but undocumented. It is now forbidden. To write complex masking rules, it is recommended to create custom masking functions.

Inlining of functions

Inlining of functions is disabled for transparent dynamic masking because such function can bypass Pg Anonymizer's protections. This can have a negative impact on performance if your plans rely on inlining.

Functions with SQL standard body and transparent dynamic masking

Functions with SQL-standard bodies undergo parse analysis only at creation time. As a result, transparent dynamic masking would completely miss them. Users subject to transparent dynamic masking are therefore forbidden from executing such functions.

This includes:

- functions in contrib extensions that use this form (`citext`, `earthdistance`, `lo`, `pageinspect`, `pg_freespacemap`, `xml2` in 18.6),
- masking rules such as: `MASKED WITH FUNCTION mask.stars(num)`, where `mask.stars()` has a `RETURN` body, whether the function is used directly or through `pg_dump`.

Upgrade to version 3.2 and further versions

Security Barrier

For security reason, superusers are not forbidden from running any kind of masking. Following the principle of "Least Privilege", the extension now refuses to run any masking function on behalf of a superuser.

Especially if you are currently using static masking, replica masking or backup masking with a superuser, we advise you to create a dedicated role for that.

The previous behaviour can be restored by setting `anon.nosuperuser` to `FALSE`, in which case the security barrier will be removed and superuser will be able to execute masking rules at their own risk.

New import / export format

The JSON schema used for imports / exports has changed between version 3.1 and 3.2. It is recommended to re-export the rules after the upgrade to 3.2

The migration path for export files is:

- Import the rules in anon version N;
- Upgrade anon from version N to N+1;
- Export the rules using the new schema.

Upgrade to version 3.0 and further versions

PostgreSQL 13 is not supported anymore

PostgreSQL 13 is now EOL. If you're running PostgreSQL Anonymizer on an obsolete PostgreSQL version, please upgrade your instance first.

Legacy Dynamic Masking is fully removed

The "Legacy Dynamic Masking" was the dynamic masking method used in version 1.x. It is now completely removed and replaced by "Transparent Dynamic Masking".

If you are still using Legacy Dynamic Masking in version 2.x, you must disable it **BEFORE** upgrading the extension with:

```
SELECT anon.stop_legacy_dynamic_masking();
```

Breaking changes in internal catalogs

If you wrote SQL requests using the `anon.pg_masking_rules` view, you must replace them with `anon.user_rules` view and adapt accordingly.

Upgrade to version 2.0 and further versions

With version 2, the entire core library was rewritten in Rust. This is a major change that brings new features, better performances and improved stability.

However the changes are mostly internal and for the most part the public interface of the extension does not change. A masking policy written with version 1.3 should work with version 2.0 !

!!! warning New RPM repository !

```
Version 2.0 is not available on the PGDG RPM repository.
If you installed PostgreSQL Anonymizer 1.x using the RPM package, you need
to install the Dalibo Labs repository with the following command:
`dnf install https://yum.dalibo.org/labs/dalibo-labs-4-1.noarch.rpm`
```

Upgrade to version 1.3 and further versions

Starting with version 1.3, the extension enforces a series of security checks and it will refuse some masking rules that were previously accepted.

Here's a few example of the changes you may need to make to your masking policy

Using custom masking functions

If you have developed custom masking functions, you now need to place them inside a dedicated schema and declare that this schema is `trusted`

For example, let's say you have a function `remove_phone` that delete phone numbers from a `JSONB` field

First create a schema:

```
CREATE SCHEMA IF NOT EXISTS my_masks;
```

Then a superuser must declare it as trusted:

```
SECURITY LABEL FOR anon ON SCHEMA my_masks IS 'TRUSTED';
```

Now you can write the function:

```
CREATE OR REPLACE FUNCTION my_masks.remove_phone(j JSONB)
RETURNS JSONB
AS $$
  SELECT j - ARRAY['phone']
$$
LANGUAGE SQL ;
```

And finally use it in a masking rule:

```
SECURITY LABEL FOR anon ON COLUMN player.personal_details
  IS 'MASKED WITH FUNCTION my_masks.remove_phone(personal_details)';
```

See the Write your own Masks ! section of the doc for more details…

Using `pg_catalog` functions

With version 1.3 and later, the `pg_catalog` schema is not longer trusted because it contains system administration functions that should not be used as masking functions.

However the extension provides bindings to some useful and safe commodity functions from the `pg_catalog` schema.

For instance, the following rule

```
SECURITY LABEL FOR anon ON COLUMN employee.phone
  IS 'MASKED WITH FUNCTION md5(phone)'
```

```
SECURITY LABEL FOR anon ON COLUMN employee.phone
  IS 'MASKED WITH FUNCTION anon.md5(phone)'
```

See the Using `pg_catalog` functions section of the doc for more details…

Operators

The `MASKED WITH FUNCTION` syntax is now more strict and in particular operators are not allowed as a masking value.

For instance, until version 1.3

```
SECURITY LABEL FOR anon ON COLUMN player.name
  IS 'MASKED WITH FUNCTION anon.fake_first_name() || anon.fake_last_name()';
```

Now operators must be replaced by an actual function. For instance, the `||` operator would be replaced by `anon.concat`

```
SECURITY LABEL FOR anon ON COLUMN player.name
  IS 'MASKED WITH FUNCTION anon.concat(anon.fake_first_name(),anon.fake_last_name())';
```

Conditional masking rules

The `MASKED WITH VALUE CASE WHEN ...` was never an intended feature but it work by accident.

Until version 1.3, the syntax below was accepted:

```
SECURITY LABEL FOR anon ON COLUMN player.score
  IS 'MASKED WITH VALUE CASE WHEN score IS NULL
                             THEN NULL
                             ELSE anon.random_int_between(0,100)
                             END';
```

The `CASE` syntax is now rejected and can be replaced by the `anon.ternary()` function:

```
SECURITY LABEL FOR anon ON COLUMN player.score
  IS 'MASKED WITH FUNCTION anon.ternary(score IS NULL,
                                        NULL,
                                        anon.random_int_between(0,100)
  )';
```

See the Conditional Masking section of the doc for more details…