UPGRADE

A practical guide

DALIBO

Feb. 2023

Contents

Figure 1: PostgreSQL Anonymizer

Anonymization & Data Masking for PostgreSQL

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

The project relies on a **declarative approach** of anonymization. This means we're using the PostgreSQL Data Definition Language (DDL) in order to specify the anonymization strategy inside the table definition itself.

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

- Anonymous Dumps : Simply export the masked data into an SQL file
- Static Masking : Remove permanently the PII according to the rules
- Dynamic Masking : Hide PII only for the masked users
- Generalization : Reducing the accuracy of dates and numbers

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

Read the Concepts section for more details and NEWS.md for information about the latest version.

Declaring The Masking Rules

The main idea of this extension is to offer **anonymization by design**.

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 masking 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 :

```
=# CREATE EXTENSION IF NOT EXISTS anon CASCADE;

=# SELECT anon.init();

=# CREATE TABLE player( id SERIAL, name TEXT, points INT);

=# 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';
```

Static Masking

You can permanently remove the PII from a database with `anon.anonymize_database()`. This will destroy the original data. Use with care.

```
=# 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

=# SECURITY LABEL FOR anon ON COLUMN customer.full_name
-# IS 'MASKED WITH FUNCTION anon.fake_first_name() || '' '' || anon.fake_last_name()';

=# SECURITY LABEL FOR anon ON COLUMN customer.birth
-# IS 'MASKED WITH FUNCTION anon.random_date_between(''1920-01-01''::DATE,now())';
```

```
=# SECURITY LABEL FOR anon ON COLUMN customer.employer
-# IS 'MASKED WITH FUNCTION anon.fake_company()';

=# SECURITY LABEL FOR anon ON COLUMN customer.zipcode
-# IS 'MASKED WITH FUNCTION anon.random_zip()';

=# SELECT anon.anonymize_database();

=# SELECT * FROM customer;
 id  |     full_name     |   birth    |     employer     | zipcode | fk_shop
-----+-------------------+------------+------------------+---------+---------
 911 | michel Duffus     | 1970-03-24 | Body Expressions | 63824   | 12
 112 | andromach Tulip   | 1921-03-24 | Dot Darcy        | 38199   | 423
```

You can also use `anonymize_table()` and `anonymize_column()` to remove data from a subset of the database.

Dynamic Masking

You can hide the PII from a role by declaring it as a "MASKED". Other roles will still access the original data.

Example:

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

Step 2 : Declare a masked user

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

Step 3 : Declare the masking rules

```
=# SECURITY LABEL FOR anon ON COLUMN people.lastname
-# IS 'MASKED WITH FUNCTION anon.fake_last_name()';

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

Step 4 : Connect with the masked user

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

Anonymous Dumps

Due to the core design of this extension, you cannot use `pg_dump` with a masked user. If you want to export the entire database with the anonymized data, you must use the `pg_dump_anon` command line. For example

```
pg_dump_anon.sh -h localhost -p 5432 -U bob bob_db > dump.sql
```

For more details, read the Anonymous Dumps section.

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.

You can either open an issue or send a message at contact@dalibo.com.

Requirements

This extension works with all supported versions of PostgreSQL.

It requires an extension called pgcrypto which is delivered by the `postgresql-contrib` package of the main linux distributions.

Install

See the INSTALL section

Limitations

- The dynamic masking system only works with one schema (by default `public`). When you start the masking engine with `start_dynamic_masking()`, you can specify the schema that will be masked with. **However** static masking with `anon.anonymize()`and Anonymous Dumps will work fine with multiple schemas.
- The Anonymous Dumps may not be consistent. Use Static Masking combined with `pg_dump` if you can't fence off your database from `DML` or `DDL` commands during the export.

Performance

See docs/performances.md

Anonymous Dumps

EXPERIMENTAL : Transparent Anonymous Dumps

> WARNING: This feature is under development and will not be officially supported until version 2.0 is released. Use with care. For a more stable solution, see the pg_dump_anon section.

To export the anonymized data from a database, follow these 2 steps:

1. Create a masked user

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

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

2. Grant read access to that user

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

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

NOTE: Replace `foo` with any other schema you have inside you database.

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 dump_anon \
        --no-security-labels \
        --extension pgcatalog.plpgsql \
        --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.

- `--extension pgcatalog.plpgsql` will remove the `anon` extension, which useless inside the anonymized dump. This option is only available with `pg_dump 14` and later.
- `--format=custom` is supported

pg_dump_anon

The `pg_dump_anon` command support most of the options of the regular [pg_dump] command. The PostgreSQL environment variables ($PGHOST, PGUSER, etc.) and the .pgpass file are also supported.

Example

A user named `bob` can export an anonymous dump of the `app` database like this:

```
pg_dump_anon -h localhost -U bob --password --file=anonymous_dump.sql app
```

WARNING: The name of the database must be the last parameter.

For more details about the supported options, simply type `pg_dump_anon --help`

Install With Go

```
go install gitlab.com/dalibo/postgresql_anonymizer/pg_dump_anon
```

Install With docker

If you do not want to instal Go on your production servers, you can fetch the binary with:

```
docker run --rm -v "$PWD":/go/bin golang go get gitlab.com/dalibo/postgresql_anonymizer/pg_d
sudo install pg_dump_anon $(pg_config --bindir)
```

Limitations

- The user password is asked automatically. This means you must either add the `--password` option to define it interactively or declare it in the PGPASSWORD variable or put it inside the .pgpass file (however on Windows,the PGPASSFILE variable must be specified explicitly)
- The `plain` format is the only supported format. The other formats (`custom`, `dir` and `tar`) are not supported

Obsolete: pg_dump_anon.sh

Before version 1.0, `pg_dump_anon` was a bash script. This script was nice and simple, however under certain conditions the backup were not consistent. See issue #266 for more details.

This script is now renamed to `pg_dump_anon.sh` and it is still available for backwards compatibility. But it will be deprecated in version 2.0.

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.

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.maskschema

Type	Text
Default value	'mask'
Visible	to all users

The schema (i.e. 'namespace') where the dynamic masking views will be stored.

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.sourceshema

Type	Text
Default value	'public'
Visible	to all users

The schema (i.e. 'namespace') where the tables are masked by the dynamic masking engine.

Change this value before starting dynamic masking.

```
ALTER DATABASE foo SET anon.sourceschema TO 'my_app';
```

Then reconnect so that the change takes effect and start the engine.

```
SELECT start_dynamic_masking();
```

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.

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

Alternative fake data packages

The projet 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 prefered 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

As an example, here's a python script that will generate fake data for you:

https://gitlab.com/dalibo/postgresql_anonymizer/-/blob/master/python/populate.py

To produce 5000 emails in French & German, you'd call the scripts like this:

```
populate.py --table email --locales fr,de --lines 5000
```

This will output the fake data in `CSV` format.

Use `populate.py --help` for more details about the script parameters.

You can load the fake data directly into the extension like this:

```
TRUNCATE anon.email;

COPY anon.email
FROM
PROGRAM 'populate.py --table email --locales fr,de --lines 5000';

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

CLUSTER anon.email;
```

> **IMPORTANT** : This script is provided as an example, it is not officially supported.

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 offer **anonymization by design**.

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:

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

INSERT INTO player VALUES
  ( 1, 'Kareem Abdul-Jabbar', 38387),
```

```
  ( 5, 'Michael Jordan', 32292 );

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';
```

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.pg_masking_rules`:

```
SELECT * FROM anon.pg_masking_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();
```

Limitations

- The maximum length of a masking rule is 1024 characters. If you need more, you should probably write a dedicated masking function.
- 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.

Declaring Rules with COMMENTs is deprecated.

Previous version of the extension allowed users to declare masking rules using the `COMMENT` syntax.

This is not suppported any more. `SECURITY LABELS` are now the only way to declare rules. — 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 usefull 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 dictionnaries

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

For instance, you can see the english identifier dictionary here.

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.

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 a "MASKED" one. Other roles will still access the original data.

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

Step 2 : Declare a masked user

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

Step 3 : Declare the masking rules

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

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

Step 4 : Connect with the masked user

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

How to change the type of a masked column

When dynamic masking is activated, you are not allowed to change the datatype of a column if there's a mask upon it.

To modify a masked column, you need to switch of temporarily the masking engine like this:

```
BEGIN;
SELECT anon.stop_dynamic_masking();
ALTER TABLE people ALTER COLUMN phone TYPE VARCHAR(255);
```

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

How to drop a masked table

The dynamic masking engine will build *masking views* upon the masked tables. This means that it is not possible to drop a masked table directly. You will get an error like this :

```
# DROP TABLE people;
psql: ERROR:  cannot drop table people because other objects depend on it
DETAIL:  view mask.company depends on table people
```

To effectively remove the table, it is necessary to add the `CASCADE` option, so that the masking view will be dropped too:

```
DROP TABLE people CASCADE;
```

How to unmask a role

Simply remove the security label like this:

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

To unmask all masked roles at once you can type:

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

Limitations

Listing the tables

Due to how the dynamic masking engine works, when a masked role will try to display the tables in psql with the `\dt` command, then psql will not show any tables.

This is because the `search_path` of the masked role is rigged.

You can try adding explicit schema you want to search, for instance:

```
\dt *.*
\dt public.*
```

Only one schema

The dynamic masking system only works with one schema (by default `public`). When you start the masking engine with `start_dynamic_masking()`, you can specify the schema that will be masked with:

```
ALTER DATABASE foo SET anon.sourceschema TO 'sales';
```

Then open a new session to the database and type:

```
SELECT start_dynamic_masking();
```

However static masking with `anon.anonymize()`and anonymous export with `anon.dump()` will work fine with multiple schemas.

Performances

Dynamic Masking is known to be very slow with some queries, especially if you try to join 2 tables on a masked key using hashing or pseudonymization.

Graphic Tools

When you are using a masked role with a graphic interface such as DBeaver or pgAdmin, the "data" panel may produce the following error when trying to display the content of a masked table called `foo`:

```
SQL Error [42501]: ERROR: permission denied for table foo
```

This is because most of these tools will directly query the `public.foo` table instead of being "redirected" by the masking engine toward the `mask.foo` view.

In order the view the masked data with a graphic tool, you can either:

1- Open the SQL query panel and type `SELECT * FROM foo`

2- Navigate to `Database > Schemas > mask > Views > foo`

Generalization

Reducing the accuracy of sensitive data

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 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 MATERIALIZED VIEW generalized_patient AS
SELECT
  'REDACTED'::TEXT AS firstname,
  anon.generalize_int4range(zipcode,1000) AS zipcode,
  anon.generalize_daterange(birth,'decade') AS birth,
  disease
FROM 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)
```

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: 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 personnal 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

`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 differents users:

```
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":

```
CREATE DATABASE boutique OWNER paul;
```

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

```
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

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;
```

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 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:

```
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

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

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.

Exercices

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:

```
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

```
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

```
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;
```

S104

```
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:

```
SELECT anon.shuffle_column('payout','fk_customer_id','id');
```

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);
```

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 personnal 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 personnal 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;
```

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;
```

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:

```
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:

```
SELECT * FROM supplier;
```

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

```
SELECT * FROM supplier;
```

Exercices

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 everytime 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

```
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:

```
SELECT * FROM company;
```

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

```
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:

```
SELECT * FROM company;
```

```
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 personnal 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
);
```

```
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 personnal 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
```

Exercices

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 adresses. But an IP adress 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

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:

```
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 anon ON COLUMN v_asthma_eyes.eyes
IS 'INDIRECT IDENTIFIER';

SECURITY LABEL FOR anon 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.

Exercices

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 !

```
DROP EXTENSION anon CASCADE;
```

```
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;
```

```
DROP DATABASE IF EXISTS boutique;
```

```
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?

:::

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: index draft: false toc: true —

Figure 2: PostgreSQL Anonymizer

Anonymization & Data Masking for PostgreSQL

`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 3 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

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

Beyond masking, it is also possible to use a fourth approach called Generalization which is perfect for statistics and data analytics.

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

Example

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

Step 1 : Activate the dynamic masking engine

```
=# CREATE EXTENSION IF NOT EXISTS anon CASCADE;
=# SELECT anon.start_dynamic_masking();
```

Step 2 : Declare a masked user

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

Step 3 : Declare the masking rules

```
=# SECURITY LABEL FOR anon ON COLUMN people.lastname
-# IS 'MASKED WITH FUNCTION anon.fake_last_name()';

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

Step 4 : Connect with the masked user

```
=# \connect - skynet
=> SELECT * FROM people;
 id | firstname | lastname  |   phone
----+-----------+-----------+------------
 T1 | 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.

You can either open an issue or send a message at contact@dalibo.com.

Figure 3: PostgreSQL Anonymizer

Anonymization & Data Masking for PostgreSQL

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

The project relies on a **declarative approach** of anonymization. This means we're using the PostgreSQL Data Definition Language (DDL) in order to specify the anonymization strategy inside the table definition itself.

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

- Anonymous Dumps : Simply export the masked data into an SQL file
- Static Masking : Remove permanently the PII according to the rules
- Dynamic Masking : Hide PII only for the masked users
- Generalization : Reducing the accuracy of dates and numbers

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

Read the Concepts section for more details and NEWS.md for information about the latest version.

Declaring The Masking Rules

The main idea of this extension is to offer **anonymization by design**.

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 masking 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 :

```
=# CREATE EXTENSION IF NOT EXISTS anon CASCADE;

=# SELECT anon.init();

=# CREATE TABLE player( id SERIAL, name TEXT, points INT);

=# 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';
```

Static Masking

You can permanently remove the PII from a database with `anon.anonymize_database()`. This will destroy the original data. Use with care.

```
=# 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

=# SECURITY LABEL FOR anon ON COLUMN customer.full_name
```

```
-# IS 'MASKED WITH FUNCTION anon.fake_first_name() || '' '' || anon.fake_last_name()';

=# SECURITY LABEL FOR anon ON COLUMN customer.birth
-# IS 'MASKED WITH FUNCTION anon.random_date_between(''1920-01-01''::DATE,now())';

=# SECURITY LABEL FOR anon ON COLUMN customer.employer
-# IS 'MASKED WITH FUNCTION anon.fake_company()';

=# SECURITY LABEL FOR anon ON COLUMN customer.zipcode
-# IS 'MASKED WITH FUNCTION anon.random_zip()';

=# SELECT anon.anonymize_database();

=# SELECT * FROM customer;
 id  |     full_name      |   birth    |     employer      | zipcode | fk_shop
-----+--------------------+------------+-------------------+---------+---------
 911 | michel Duffus      | 1970-03-24 | Body Expressions | 63824   | 12
 112 | andromach Tulip    | 1921-03-24 | Dot Darcy         | 38199   | 423
```

You can also use `anonymize_table()` and `anonymize_column()` to remove data from a subset of the database.

Dynamic Masking

You can hide the PII from a role by declaring it as a "MASKED". Other roles will still access the original data.

Example:

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

Step 2 : Declare a masked user

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

Step 3 : Declare the masking rules

```
=# SECURITY LABEL FOR anon ON COLUMN people.lastname
-# IS 'MASKED WITH FUNCTION anon.fake_last_name()';
```

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

Step 4 : Connect with the masked user

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

Anonymous Dumps

Due to the core design of this extension, you cannot use `pg_dump` with a masked user. If you want to export the entire database with the anonymized data, you must use the `pg_dump_anon` command line. For example

```
pg_dump_anon.sh -h localhost -p 5432 -U bob bob_db > dump.sql
```

For more details, read the Anonymous Dumps section.

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.

You can either open an issue or send a message at contact@dalibo.com.

Requirements

This extension works with all supported versions of PostgreSQL.

It requires an extension called pgcrypto which is delivered by the `postgresql-contrib` package of the main linux distributions.

Install

See the INSTALL section

Limitations

- The dynamic masking system only works with one schema (by default `public`). When you start the masking engine with `start_dynamic_masking()`, you can specify the schema that will be masked with. **However** static masking with `anon.anonymize()`and Anonymous Dumps will work fine with multiple schemas.
- The Anonymous Dumps may not be consistent. Use Static Masking combined with `pg_dump` if you can't fence off your database from `DML` or `DDL` commands during the export.

Performance

See docs/performances.md

INSTALL

The installation process is composed of 4 basic steps:

- Step 1: **Deploy** the extension into the host server
- Step 2: **Load** the extension in the PostgreSQL instance
- Step 3: **Create** the extension inside the database
- Step 4: **Initialize** the extension internal data

There are multiple ways to install the extension :

- Install on RedHat / CentOS
- Install with PGXN
- Install from source
- Install with docker
- Install as a black box
- Install on MacOS
- Install on Windows
- Install in the cloud
- Uninstall

In the examples below, we load the extension (step2) using a parameter called `session_preload_libraries` but there are other ways to load it. See Load the extension for more details.

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

Choose your version : `Stable` or `Latest` ?

This extension is available in two versions :

- `stable` is recommended for production
- `latest` is useful if you want to test new features

Install on RedHat / CentOS

> This is the recommended way to install the `stable` extension This method works for RHEL/CentOS 7 and 8. If you're running RHEL/CentOS 6, consider upgrading or read the Install With PGXN section.

Step 0: Add the PostgreSQL Official RPM Repo to your system. It should be something like:

```
sudo yum install https://.../pgdg-redhat-repo-latest.noarch.rpm
```

Step 1: Deploy

```
sudo yum install postgresql_anonymizer_14
```

(Replace `14` with the major version of your PostgreSQL instance.)

Step 2: Load the extension.

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

(If you're already loading extensions that way, just add `anon` the current list)

Step 3: Close your session and open a new one. Create the extension.

```
CREATE EXTENSION anon CASCADE;
```

Step 4: Initialize the extension

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

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

Install With PGXN :

This method will install the `stable` extension

Step 1: Deploy the extension into the host server with:

```
sudo apt install pgxnclient postgresql-server-dev-12
sudo pgxn install postgresql_anonymizer
```

(Replace `12` with the major version of your PostgreSQL instance.)

Step 2: Load the extension.

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

(If you're already loading extensions that way, just add `anon` the current list)

Step 3: Close your session and open a new one. Create the extension.

```
CREATE EXTENSION anon CASCADE;
```

Step 4: Initialize the extension

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

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

Additional notes:

- PGXN can also be installed with `pip install pgxn`
- If you have several versions of PostgreSQL installed on your system, you may have to point to the right version with the `--pg_config` parameter. See Issue #93 for more details.
- Check out the pgxn install documentation for more information.

Install From source

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

Step 0: First you need to install the postgresql development libraries. On most distributions, this is available through a package called `postgresql-devel` or `postgresql-server-dev`.

Step 1: Download the source from the official repository on Gitlab, either the archive of the latest release, or the latest version from the `master` branch:

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

Step 2: Build the project like any other PostgreSQL extension:

```
make extension
sudo make install
```

NOTE: If you have multiple versions of PostgreSQL on the server, you may need to specify which version is your target by defining the `PG_CONFIG` env variable like this:

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

Step 3: Load the extension:

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

(If you're already loading extensions that way, just add `anon` the current list)

Step 4: Close your session and open a new one. Create the extension.

```
CREATE EXTENSION anon CASCADE;
```

Step 5: Initialize the extension:

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

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

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:

```
PG_MAJOR_VERSION=11 make docker_image
```

Install as a "Black Box"

You can also treat the docker image as an "anonymizing black box" by using a specific entrypoint script called `/anon.sh`. You pass the original data and the masking rules to the `/anon.sh` script and it will return a anonymized dump.

Here's an example in 4 steps:

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

```
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`)

```
SELECT pg_catalog.set_config('search_path', 'public', false);

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

SECURITY LABEL FOR anon ON COLUMN people.lastname
IS 'MASKED WITH FUNCTION anon.fake_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
ANON="docker run --rm -i $IMG /anon.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
```

Install on MacOS

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

However it should be possible to build the extension with the following lines:

```
export C_INCLUDE_PATH="$(xcrun --show-sdk-path)/usr/include"
make extension
make install
```

Install on Windows

WE DO NOT PROVIDE COMMUNITY SUPPORT FOR THIS EXTENSION ON WINDOWS.

However it is possible to compile it using Visual Studio and the `build.bat` file.

We provide Windows binaries and install files as part of our commercial support.

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 (March 2024), the following platforms provide PostgreSQL Anonymizer:

- Crunchy Bridge
- Google Cloud SQL
- Neon
- Postgres.ai
- Tembo

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 some major actors in this domain, we DO NOT have internal knowledge on wether 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: Alternative way to load the extension

It is recommended to load the extension like this:

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

It has several benefits:

- First, it will be dumped by `pg_dump` with the `-C` option, so the database dump will be self efficient.
- Second, it is propagated to a standby instance by streaming replication. Which means you can use the anonymization functions on a read-only clone of the database (provided the extension is installed on the standby instance)

However, you can load the extension globally in the instance using the `shared_preload_libraries` parameter :

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

Then restart the PostgreSQL instance.

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 local_preload_libraries;
SHOW session_preload_libraries;
SHOW shared_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 4.

Uninstall

Step 1: Remove all rules

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

THIS IS NOT MANDATORY ! It is possible to keep the masking rules inside the database schema even if the anon extension is removed !

Step 2: Drop the extension

```
DROP EXTENSION anon CASCADE;
```

The `anon` extension also installs pgcrypto as a dependency, if you don't need it, you can remove it too:

```
DROP EXTENSION pgcrypto;
```

Step 3: Unload the extension

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

Step 4: Uninstall the extension

For Redhat / CentOS / Rocky:

```
sudo yum remove postgresql_anonymizer_14
```

Replace 14 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
- Chinese: https://www.youtube.com/watch?v=n9atI31FcSM

Similar technologies

- greenmask Anonymous dump utility written in Golang
- pganonymize A commandline tool for anonymizing PostgreSQL databases
- pgantomizer Anonymous dumps based on masking rules written in a YAML file
- pgsodium and postgresql-anonymizer Pseudononymous Access To Encrypted Table
- pg_diffix PostgreSQL extension implementing differential privacy (inactive)
- pg_anonymize PostgreSQL extension implementing dynamic data anonymization
- pg-anonymizer Dump anonymized PostgreSQL database with a NodeJS CLI

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: masking_functions draft: false toc: true —

Various Masking Strategies

The extension provides functions to implement 8 main anonymization strategies:

- Destruction
- Adding Noise
- Randomization
- Faking
- Advanced Faking
- Pseudonymization
- Generic Hashing
- Partial scrambling
- Conditional masking
- Generalization
- Using pg_catalog functions
- 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

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.

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

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 powerfull 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.

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 functions provided here are limited to basic use case. For more advanced faking methods, in particular if you are looking for **localized fake data**, take a look at PostgreSQL Faker, an extension based upon the well-known Faker python library.

This extension provides an advanced faking engine with localisation support.

For example:

```
CREATE SCHEMA faker;
CREATE EXTENSION faker SCHEMA faker;
SELECT faker.faker('de_DE');
SELECT faker.first_name_female();
 first_name_female
-------------------
 Mirja
```

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.pseudo_first_name(seed,salt)` returns a generic first name
- `anon.pseudo_last_name(seed,salt)` returns a generic last name
- `anon.pseudo_email(seed,salt)` returns a valid email address
- `anon.pseudo_city(seed,salt)` returns an existing city
- `anon.pseudo_country(seed,salt)` returns a country
- `anon.pseudo_company(seed,salt)` returns a generic company name
- `anon.pseudo_iban(seed,salt)` returns a valid IBAN
- `anon.pseudo_siret(seed,salt)` returns a valid SIRET

The second argument (`salt`) is optional. You can call each function with only the seed like this `anon.pseudo_city('bob')`. 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.pseudo_email(users.login) ';
```

NOTE : 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.

WARNING : 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

In theory, hashing is not a valid anonymization technique, however in practice it is sometimes necessary 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, sha1, 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.

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'

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 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 in 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 provides bindings to some useful and safe functions from the `pg_catalog` schema for your convenience:

- anon.concat(TEXT,TEXT)
- anon.concat(TEXT,TEXT, TEXT)
- anon.date_add(TIMESTAMP WITH TIME ZONE,INTERVAL)
- anon.date_part(TEXT,TIMESTAMP)
- anon.date_part(TEXT,INTERVAL)

- anon.date_subtract(TIMESTAMP WITH TIME ZONE, INTERVAL)
- anon.date_trunc(TEXT,TIMESTAMP)
- anon.date_trunc(TEXT,TIMESTAMP WITH TIME ZONE,TEXT)
- anon.date_trunc(TEXT,INTERVAL)
- anon.left(TEXT,INTEGER)
- anon.length(TEXT)
- anon.lower(TEXT)
- anon.make_date(INT,INT,INT)
- anon.make_time(INT,INT,DOUBLE PRECISION)
- anon.md5(TEXT)
- anon.random()
- anon.replace(TEXT,TEXT,TEXT)
- anon.regexp_replace(TEXT,TEXT,TEXT)
- anon.regexp_replace(TEXT,TEXT,TEXT,TEXT)
- anon.right(TEXT,INTEGER)
- anon.substr(TEXT,INTEGER)
- anon.substr(TEXT,INTEGER,INTEGER)
- anon.upper(TEXT)

If you need more bindings, you can either

- Write your own mapping function in a trusted schema (see below)
- Set the `pg_catalog` schema as `TRUSTED` (not recommended)
- open an issue

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)
```

Using the PostgreSQL JSON functions and operators, you can walk through the keys and replace the sensitive values as needed.

```
CREATE SCHEMA custom_masks;

-- This step requires superuser privilege
SECURITY LABEL FOR anon ON SCHEMA custom_masks IS 'TRUSTED';

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
$func$;
```

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.

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.

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

Disclaimer

This feature is considered in beta and not ready for production until version 2.0 is published.

Use with care.

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 and 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
-----------+---------+-----+---------------
           |         |     | unkown
```

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 contraint 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.

Sampling

Principle

The GDPR introduces the concept of principle of "[data minimisation]" 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.

Example

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.

Syntax

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)';
```

Maintaining Referential Integrity

> **NOTE** :The sampling method describe above **WILL 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.

Security

Permissions

Here's an overview of what users can do depending on the priviledge 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		
Start dynamic masking	Yes		
Stop dynamic masking	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
Select the masked data	Yes	Yes	Yes
View the masking rules	Yes	Yes	Yes

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 defautt the functions from the `pg_catalog` cannot be used in masking rules.

For more details, read the Using pg_catalog functions section.

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`.

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

These methods will destroy the original data. Use with care.

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.fake_first_name() || '' '' || anon.fake_last_name()';
```

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

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();

SELECT * FROM customer;
```

```
 id  |  full_name  |   birth    |      employer       | zipcode | fk_shop
-----+-------------+------------+---------------------+---------+---------
 911 | jesse Kosel | 1940-03-10 | Marigold Properties | 62172   |      12
 312 | leolin Bose | 1952-07-17 | Inventure           | 20026   |     423
```

You can also use `anonymize_table()` and `anonymize_column()` to remove data from a subset of the database :

```
SELECT anon.anonymize_table('customer');
SELECT anon.anonymize_column('customer','zipcode');
```

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.

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 "verticaly" : 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.

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 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...