UPGRADE

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

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.

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.

Exemple

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
```

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 avalaible 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, `pg_catalog` and `anon` are 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 instance, the rules below would fail because the schema of the lower function is not declared.

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

The correct way to declare it would be :

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

This parameter is kept to `off` in the current version to maintain backward compatibility but we highly encourage users to switch to `on` when possible. In the forthcoming version, we may define `on` as the default behaviour.

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!

Localized fake data

We provide a python script that will generate fake data for you. This script is located in the anon extension directory, usually something like:

```
/usr/share/postgresql/13/extension/anon/populate.py
```

If you want to produce 5000 emails in French & German, you call the scripts like this:

```
$ python3 $(pg_config --sharedir)/extension/anon/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 'python3 [...]/populate.py --table email --locales fr,de --lines 5000';

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

CLUSTER anon.email;
```

Load your own fake data

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

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

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

Using the PostgreSQL Faker extension

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

Advanced Faking: masking_functions.md#advanced-faking

title: datamodel draft: false toc: true —

```
classDiagram

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

  class field_name{
        TEXT attname
        TEXT lang
        INTEGER fk_identifiers_category
  }
  field_name "1..N" --> "1" identifier_category
```

Put on your Masks !

The main idea of this extension is to 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 $$CONFIDENTIAL$$';
```

Using Expressions

You can use more advanced expressions with the `MASKED WITH VALUE` syntax:

```
SECURITY LABEL FOR anon ON COLUMN player.name
  IS 'MASKED WITH VALUE CASE WHEN name IS NULL
                             THEN $$John$$
```

```
                          ELSE anon.random_string(LENGTH(name))
                          END';
```

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 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.

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:

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

Then open a new session to the database and type:

```sql
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:

```sql
SECURITY LABEL FOR anon ON COLUMN patient.firstname IS 'INDIRECT IDENTIFIER';
SECURITY LABEL FOR anon ON COLUMN patient.zipcode IS 'INDIRECT IDENTIFIER';
SECURITY LABEL FOR anon ON COLUMN patient.birth IS 'INDIRECT IDENTIFIER';
```

Step 2: Once the indirect identifiers are declared :

```sql
SELECT anon.k_anonymity('generalized_patient')
```

The higher the value, the better...

References

-

  ### How Google Anonymizes Data

  title: how-to/README draft: false toc: true —

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 already loading extensions that way, just add `anon` the current list)

Step 3: Create the extension and load the anonymization data

```
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 already loading extensions that way, just add `anon` the current list)

Step 3: 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:

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

(If some extensions are already loaded that way, just add a comma and `anon` to the current list.)

Step 4: Create the extension:

```sql
CREATE EXTENSION anon CASCADE;
```

Step 5: Initialize the extension:

```sql
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

> **WARNING** In previous versions, this extension could be installed on various Database As A Service platforms (such as Amazon RDS). Starting with version 0.9, this is not possible anymore. We do not support the former `standalone` method. If privacy and anonymity are a concern to you, we encourage you to contact the customer services of these platforms and ask them if they plan to add this extension to their catalog.

At the time we are writing this (February 2022), a few cloud operators embed PostgreSQL Anonymizer in their offering. Here's a non-exhaustive list:

- Postgres.ai

Addendum: Alternative ways to load the extension

Here's some additional notes about how you can load the extension:

1- Load only for one database

You can load the extension exclusively into a specific database like this:

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

Then quit your current session and open a new one.

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)

2- Load for the instance

You can load the extension with the `shared_preload_libraries` parameter.

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

Then restart the PostgreSQL instance.

3- Load on the fly

For a one-time usage, You can the LOAD command

```
LOAD '/usr/lib/postgresql/12/lib/anon.so';
```

You can read the Shared Library Preloading section of the PostgreSQL documentation for more details.

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;
```

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

Other technologies

- pgsodium and postgresql-anonymizer Pseudononymous Access To Encrypted Table
- pganonymize A commandline tool for anonymizing PostgreSQL databases
- 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
- Generalization

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:

```sql
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 :

- `anon.random_date()` returns a date
- `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`
- `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_in(ARRAY[1,2,3])` returns an element of an INT array
- `anon.random_in(ARRAY['a','b','c'])` returns an element of a TEXT array

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:

```sql
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 := 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 salt is not given, a random secret salt 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 have undergone pseudonymization are 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.hash(value)` will return a text hash of the value using a secret salt and hash algorithm (see below)
- `anon.digest(value,salt,algorithm)` lets you choose a salt, and a hash algorithm from a pre-defined list

By default, a random secret salt is generated when the extension is initialized, and the default hash algorithm is `sha512`. You can change these for the entire database with two functions:

- `anon.set_secret_salt(value)` to define you own salt
- `anon.set_algorithm(value)` to select another hash function. Possible values are: md5, sha1, sha224, sha256, sha384 or sha512

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 pg_catalog.left(anon.hash(phone_number),12)';
```

```
SECURITY LABEL FOR anon ON COLUMN call_history.fk_phone_number
IS 'MASKED WITH FUNCTION pg_catalog.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’

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.

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).

For instance if you wrote a function `foo()` inside the schema `bar`, then you can apply it like this:

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

SECURITY LABEL FOR anon ON COLUMN player.score
IS 'MASKED WITH FUNCTION bar.foo()';
```

> NOTE: The `bar` schema must be declared as `TRUSTED` by a superuser.

Example: Writing a masking function for a JSONB column

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.

```
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 thant 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. There are multiple ways to extract a sample of database:

- TABLESAMPLE
- pg_sample

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.O 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 principe 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 write 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 date 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.

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

The database owner is allowed to declare masking rules. He or She can also create a function containing arbitrary code and use this function inside a masking rule. In certain circumstances, the database owner can "trick" a superuser into querying a masked table and thus executing the arbitrary code.

To prevent this, the superusers can configure the parameters below :

```
anon.restrict_to_trusted_schemas = on
```

With this setting, the database owner can only write masking rules with functions that are located in the trusted schemas which are controlled by the superusers.

See the Configure section for more details.

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.