Data Masking with PostgreSQL Anonymizer

A practical guide

DALIBO

Feb. 2023

Contents

Welcome to Paul's Boutique ! 1
- The Story 2
- Objectives 3
- About PostgreSQL Anonymizer 3
- About GDPR 4
- Requirements 4
- The Roles 5
- The Sample database 6
- Authors 6
- License 6
- Credits 6

1 - Static Masking 7
- The story 7
- How it works 7
- Learning Objective 7
- The "customer" table 8
- The "payout" table 8
- Activate the extension 9
- Declare the masking rules 9
- Apply the rules permanently 9
- Exercices 10
    - E101 - Mask the client's first names 10
    - E102 - Hide the last 3 digits of the postcode 10
    - E103 - Count how many clients live in each postcode area? 10
    - E104 - Keep only the year of each birth date 10
    - E105 - Singling out a customer 11
- Solutions 12
    - S101 12
    - S102 12
    - S103 12

S104 13
S105 13

2- How to use Dynamic Masking **15**
The Story 15
How it works 15
Objectives 15
The “company” table 16
The ”supplier” table 16
Activate the extension 17
Dynamic Masking 17
Activate the masking engine 17
Masking a role 17
Masking the supplier names 18
Exercices 19
E201 - Guess who is the CEO of ”Johnny's Shoe Store” 19
E202 - Anonymize the companies 19
E203 - Pseudonymize the company name 19
Solutions 20
S201 20
S202 20
S203 21

3- Anonymous Dumps **23**
The Story 23
How it works 23
Learning Objective 23
Load the data 24
Activate the extension 24
Masking a JSON column 25
Exercices 26
E301 - Dump the anonymized data into a new database 26
E302 - Pseudonymize the meta fields of the comments 27
Solutions 27
S301 27
S302 27

4 - Generalization **29**
The Story 29

How it works 29
Learning Objective 29
The ”employee” table 30
Data suppression 31
K-Anonymity 32
Range and Generalization functions 32
Declaring the indirect identifiers 33
Exercices 34
E401 - Simplify `v_staff_per_month` and decrease granularity 34
E402 - Staff progression over the years 34
E403 - Reaching 2-anonymity for the `v_staff_per_year` view 34
Solutions 34
S401 34
S402 35
S403 35

Conclusion **37**
Clean up ! 37
Many Masking Strategies 38
Many Masking Functions 38
Advantages 38
Drawbacks 39
Also… 39
Help Wanted! 39
This is a 4 hour workshop! 39
Questions? 40

Welcome to Paul's Boutique !

This is a 4 hours workshop that demonstrates various anonymization techniques using the PostgreSQL Anonymizer[1] extension.

[1] https://labs.dalibo.com/postgresql_anonymizer

The Story

Figure 1: 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[2] (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[3] using the PostgreSQL Data Definition Language (DDL) and specify your anonymization strategy inside the table definition itself.

[2] https://en.wikipedia.org/wiki/Personally_identifiable_information
[3] https://postgresql-anonymizer.readthedocs.io/en/stable/declare_masking_rules

---

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

- Anonymous Dumps[4] : Simply export the masked data into an SQL file
- Static Masking[5] : Remove the PII according to the rules
- Dynamic Masking[6] : Hide PII only for the masked users
- **Generalization[7]** 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[8] (Video / French)
- Anonymization, Beyond GDPR[9] (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"

---

[4] https://postgresql-anonymizer.readthedocs.io/en/stable/anonymous_dumps
[5] https://postgresql-anonymizer.readthedocs.io/en/stable/static_masking
[6] https://postgresql-anonymizer.readthedocs.io/en/stable/dynamic_masking
[8] https://www.youtube.com/watch?v=KGSlp4UygdU
[9] https://public.dalibo.com/exports/conferences/_archives/_2019/20191016_anonymisation_beyond_GDPR/anonymisation_beyond_gdpr.pdf

A simple way to deploy a workshop environment is to install Docker Desktop[10] and download the image below:

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

Check out the INSTALL section[11] in the documentation[12] 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

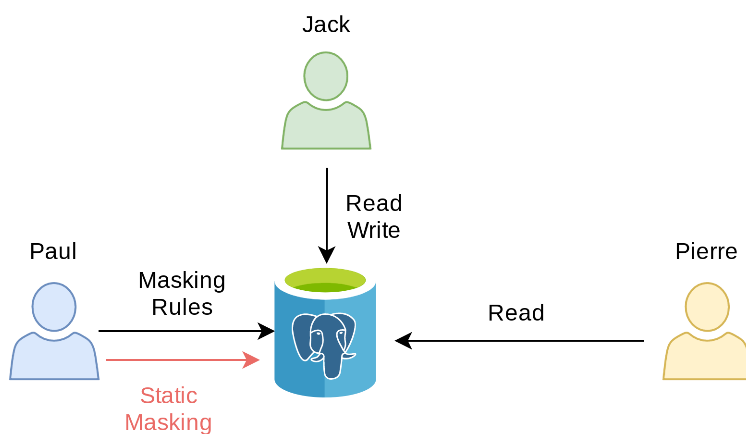
1 - Static Masking

> 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;
```

id	firstname	lastname	phone	birth	postcode
107	Sarah	Conor	060-911-0911	1965-10-10	90016
258	Luke	Skywalker	None	1951-09-25	90120
341	Don	Draper	347-515-3423	1926-06-01	04520

The "payout" table

Sales are tracked in a simple table:

```
CREATE TABLE payout ( id SERIAL PRIMARY KEY, fk_customer_id INT REFERENCES
     customer(id), order_date DATE, payment_date DATE, amount INT );
```

Let's add some orders:

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

Activate the extension

```
CREATE EXTENSION IF NOT EXISTS anon 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');
```

anonymize_table
True

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

id	firstname	lastname	phone
107	Sarah	Morgan	06X-XXX-XX11
258	Luke	Thomas	None
341	Don	Clarke	34X-XXX-XX23

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

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[1] 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)
```

id	firstname	lastname	phone	birth	postcode
341	Don	Clarke	34X-XXX-XX23	1926-06-01	04520

This is called **Singling Out[2] 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[3] section of the documentation[4]

Solutions

S101

```sql
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

```sql
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

```sql
SELECT postcode,
       COUNT(id)
FROM customer
GROUP BY postcode;
```

postcode	count
90xxx	2

postcode	count
04xxx	1

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;
```

run-postgres: 'MASKED WITH FUNCTION make_date(EXTRACT(YEAR FROM birth)::INT,1,1)' is not a valid masking function

S105

Let's mix up the values of the fk_customer_id:

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

shuffle_column
True

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

```
WITH best_client AS
  (SELECT SUM(amount),
          fk_customer_id
   FROM payout
   GROUP BY fk_customer_id
```

```
   ORDER BY 1 DESC
   LIMIT 1)
SELECT c.*
FROM customer c
JOIN best_client b ON (c.id = b.fk_customer_id);
```

id	firstname	lastname	phone	birth	postcode
258	Timothy	Morris	None	1951-09-25	90xxx

WARNING

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

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

In this case, Pierre will not be able to produce a BI report with the shuffle data, because the links between the customers and their payments are fake.

2- How to use Dynamic Masking

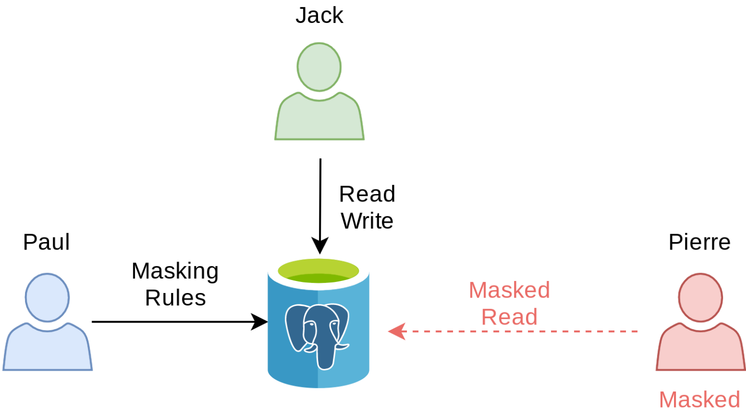
> With Dynamic Masking, the database owner can hide personal data for some users, while other users are still allowed to read and write the authentic data.

The Story

Paul has 2 employees:

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

How it works



Objectives

In this section, we will learn:

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

The “company” table

```
DROP TABLE IF EXISTS supplier CASCADE;
DROP TABLE IF EXISTS company CASCADE;
CREATE TABLE company ( id SERIAL PRIMARY KEY, name TEXT, vat_id TEXT
    UNIQUE );
```

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

```
SELECT *
FROM company;
```

id	name	vat_id
952	Shadrach	FR62684255667
194	Johnny's Shoe Store	CHE670945644
346	Capitol Records	GB663829617823

The ”supplier” table

```
CREATE TABLE supplier ( id SERIAL PRIMARY KEY, fk_company_id INT
    REFERENCES company(id), contact TEXT, phone TEXT, job_title TEXT );
```

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

```
SELECT *
FROM supplier;
```

id	fk_company_id	contact	phone	job_title
299	194	Johnny Ryall	597-500-569	CEO
157	346	George Clinton	131-002-530	Sales manager

Activate the extension

```
CREATE EXTENSION IF NOT EXISTS anon CASCADE;

SELECT anon.init();

SELECT setseed(0);
```

Dynamic Masking

Activate the masking engine

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

start_dynamic_masking
True

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;
```

id	fk_company_id	contact	phone	job_title
299	194	Johnny Ryall	597-500-569	CEO
157	346	George Clinton	131-002-530	Sales manager

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

Masking the supplier names

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

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

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

```
SELECT *
FROM supplier;
```

id	fk_company_id	contact	phone	job_title
299	194	CONFIDENTIAL	597-500-569	CEO
157	346	CONFIDENTIAL	131-002-530	Sales manager

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

```
SELECT *
FROM supplier;
```

id	fk_company_id	contact	phone	job_title
299	194	Johnny Ryall	597-500-569	CEO
157	346	George Clinton	131-002-530	Sales manager

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[a] and look at the faking functions[b] and random functions[c]!
>
> [a] https://postgresql-anonymizer.readthedocs.io/en/stable/
> [b] https://postgresql-anonymizer.readthedocs.io/en/stable/masking_functions#faking
> [c] https://postgresql-anonymizer.readthedocs.io/en/stable/masking_functions#randomization

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[1].

Solutions

S201

```
SELECT s.id,
       s.contact,
       s.job_title,
       c.name
FROM supplier s
JOIN company c ON s.fk_company_id = c.id;
```

id	contact	job_title	name
299	CONFIDENTIAL	CEO	Johnny's Shoe Store
157	CONFIDENTIAL	Sales manager	Capitol Records

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;
```

[1] https://postgresql-anonymizer.readthedocs.io/en/stable/masking_functions#pseudonymization

id	name	vat_id
952	Ware, Brown and Ellis	NZ8CRZ71PH
194	Mueller-Thompson	KEKYV45BNG
346	Lee, Kane and Fowler	JLL07SLXZ5

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

```
SELECT *
FROM company;
```

id	name	vat_id
952	Adams Inc	L5J06N7RGX
194	Patel-Gutierrez	0SXSIBAPT2
346	Bowers Group	U1IAOGUX6B

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;
```

id	name	vat_id
952	Wilkinson LLC	R2GP7MBBVZ
194	Johnson PLC	COBQ8XWMLA
346	Young-Carpenter	8YIHG64ZGC

```
SELECT *
FROM company;
```

id	name	vat_id
952	Wilkinson LLC	8HTICU54SH
194	Johnson PLC	OGC7ZECP5G
346	Young-Carpenter	IVQH873EC3

Now the fake company name is always the same.

3- Anonymous Dumps

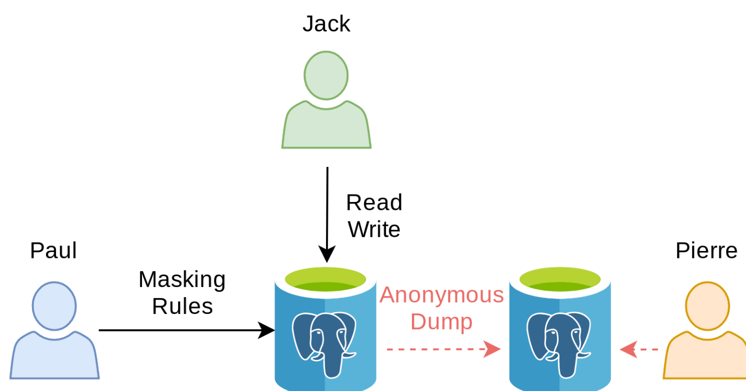
> In many situation, what we want is simply to export the anonymized data into another database (for testing or to produce statistics). This is what **pg_dump_anon** does!

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
```

name	content
Lee Perry	Hello Nasty!
	Great Shop
Jimmy	Hi ! This is me, Jimmy James

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;
```

?column?
{'meta': {'name': 'Lee Perry', 'ip_addr': '40.87.29.113'}}

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

```
CREATE SCHEMA IF NOT EXISTS my_masks;

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

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

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

Let's try it!

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

remove_content
{'meta': {'name': 'Lee Perry', 'ip_addr': '40.87.29.113'}}
{'meta': {'name': '', 'email': 'biz@bizmarkie.com'}}
{'meta': {'name': 'Jimmy'}}

And now we can use it in a masking rule:

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

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;
```

clean_comment
{'meta': {'name': 'Morgan', 'email': None, 'ip_address': '1d8cbcdef988d55982af1536922ddcd1'}}
{'meta': {'name': 'Thomas', 'email': None, 'ip_address': None}}
{'meta': {'name': 'Clarke', 'email': None, 'ip_address': None}}

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

4 - Generalization

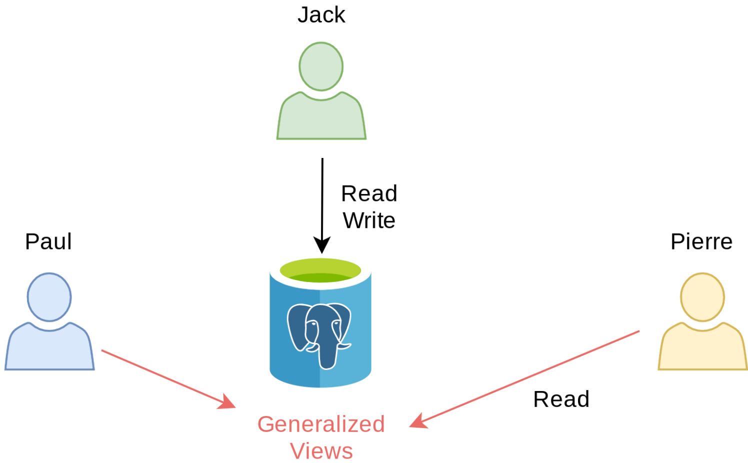
> The main idea of generalization is to ”blur” the original data. For example, instead of saying ”Mister X was born on July 25, 1989”, we can say ”Mister X was born is the 80's”. The information is still true, but it is less precise and it can't be used to reidentify the subject.

The Story

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

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

How it works



Learning Objective

In this section, we will learn:

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

The "employee" table

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

This is awkward and illegal.

Loading the data:

```
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;
```

count
16

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

full_name	first_day	hair	size	asthma
Luna Dickens	2018-07-22	blond	L	True
Paul Wolf	2020-01-15	bald	M	False
Rowan Hoeger	2018-12-01	dark	XXL	True

Data suppression

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

He provides the following view to Pierre.

```
DROP MATERIALIZED VIEW IF EXISTS v_asthma_eyes;


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

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

eyes	asthma
blue	True
brown	False
blue	True

Pierre can now write queries over this view.

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

eyes	asthma_rate
green	100
brown	37
blue	33

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';
```

run-postgres: 'INDIRECT IDENTIFIER' is not a valid label for a column

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

k_anonymity
None

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;
```

first_day	last_day
[2018-07-01, 2018-08-01)	[2018-12-01, 2019-01-01)
[2020-01-01, 2020-02-01)	(None, None)
[2018-12-01, 2019-01-01)	[2018-12-01, 2019-01-01)

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

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

count
4

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');
```

run-postgres: 'INDIRECT IDENTIFIER' is not a valid label for a column

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;
```

period
[2018, 2019)
[2020, None)
[2018, 2019)

S402

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

year	count
2018	4
2019	6
2020	9
2021	10

S403

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


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

run-postgres: 'INDIRECT IDENTIFIER' is not a valid label for a column

Conclusion

Clean up !

```sql
DROP EXTENSION anon CASCADE;
```

```sql
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;
```

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

run-postgres: database “boutique” is being accessed by other users DETAIL: There is 1 other session using the database.

```sql
DROP ROLE IF EXISTS jack;

DROP ROLE IF EXISTS paul;

DROP ROLE IF EXISTS pierre;
```

run-postgres: role "jack" cannot be dropped because some objects depend on it DETAIL: 5 objects in database boutique

---

Many Masking Strategies

- Static Masking[1] : perfect for "once-and-for-all" anonymization
- Dynamic Masking[2] : useful when one user is untrusted
- Anonymous Dumps[3] : can be used in CI/CD workflows
- **Generalization[4]** 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[5]

Advantages

- Masking rules written in SQL
- Masking rules stored in the database schema

[1] https://postgresql-anonymizer.readthedocs.io/en/stable/static_masking/
[2] https://postgresql-anonymizer.readthedocs.io/en/stable/dynamic_masking/
[3] https://postgresql-anonymizer.readthedocs.io/en/stable/anonymous_dumps/
[5] https://postgresql-anonymizer.readthedocs.io/en/latest/masking_functions/

- 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[6] : extract a small dataset from a larger PostgreSQL database
- PostgreSQL Faker[7] : An advanced faking extension based on the python Faker lib

Help Wanted!

This is a free and open project!

labs.dalibo.com/postgresql_anonymizer[8]

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

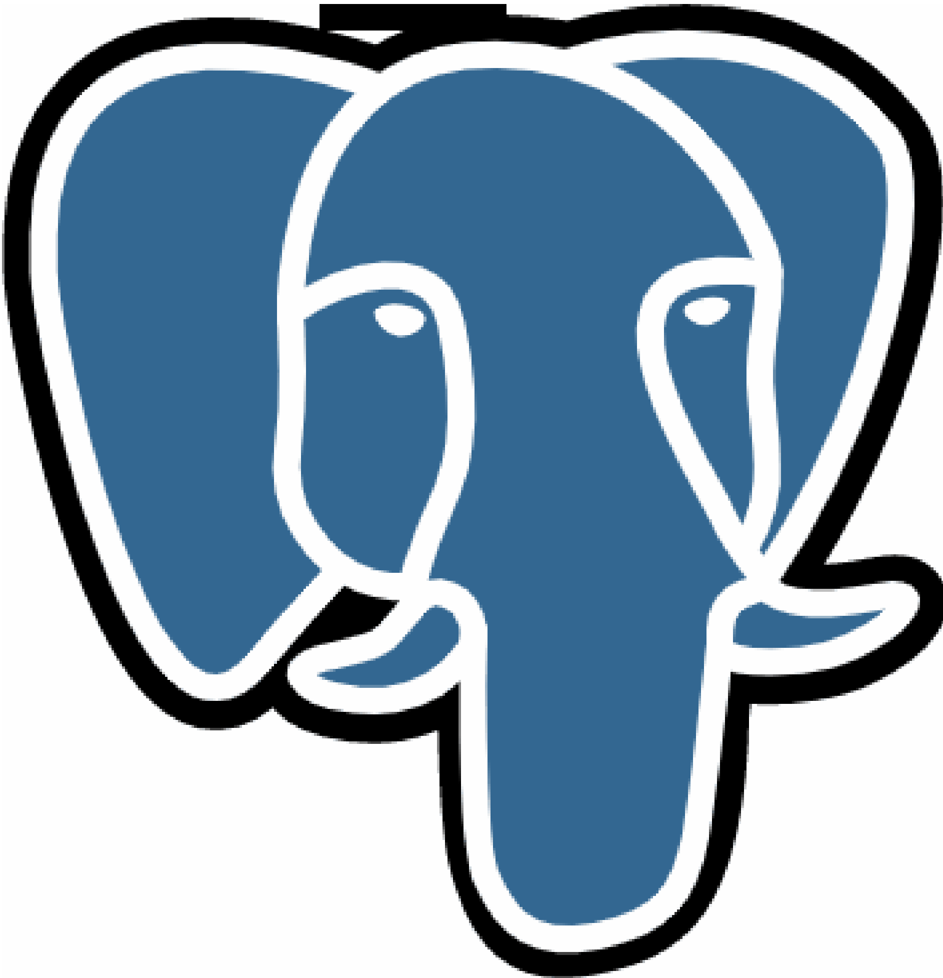
This is a 4 hour workshop!

Sources are here: gitlab.com/dalibo/postgresql_anonymizer[9]

Download the PDF Handout[10]

[6] https://github.com/mla/pg_sample
[7] https://gitlab.com/dalibo/postgresql_faker
[8] https://labs.dalibo.com/postgresql_anonymizer
[9] https://gitlab.com/dalibo/postgresql_anonymizer/-/tree/master/docs/how-to
[10] https://dalibo.gitlab.io/postgresql_anonymizer/how-to.handout.pdf

Questions?



:::