Module Y5

Masquage de données & postgresql_anonymizer

24.04

Table des matières

Sur ce document

Formation	Module Y5
Titre	Masquage de données & postgresql_anonymizer
Révision	24.04
PDF	https://dali.bo/y5_pdf
EPUB	https://dali.bo/y5_epub
HTML	https://dali.bo/y5_html
Slides	https://dali.bo/y5_slides

Vous trouverez en ligne les différentes versions complètes de ce document.

Chers lectrices & lecteurs,

Nos formations PostgreSQL sont issues de nombreuses années d'études, d'expérience de terrain et de passion pour les logiciels libres. Pour Dalibo, l'utilisation de PostgreSQL n'est pas une marque d'opportunisme commercial, mais l'expression d'un engagement de longue date. Le choix de l'Open Source est aussi le choix de l'implication dans la communauté du logiciel.

Au-delà du contenu technique en lui-même, notre intention est de transmettre les valeurs qui animent et unissent les développeurs de PostgreSQL depuis toujours : partage, ouverture, transparence, créativité, dynamisme… Le but premier de nos formations est de vous aider à mieux exploiter toute la puissance de PostgreSQL mais nous espérons également qu'elles vous inciteront à devenir un membre actif de la communauté en partageant à votre tour le savoir-faire que vous aurez acquis avec nous.

Nous mettons un point d'honneur à maintenir nos manuels à jour, avec des informations précises et des exemples détaillés. Toutefois malgré nos efforts et nos multiples relectures, il est probable que ce document contienne des oublis, des coquilles, des imprécisions ou des erreurs. Si vous constatez un souci, n'hésitez pas à le signaler via l'adresse formation@dalibo.com[1] !

À propos de DALIBO

DALIBO est le spécialiste français de PostgreSQL. Nous proposons du support, de la formation et du conseil depuis 2005.

Retrouvez toutes nos formations sur https://dalibo.com/formations

[1] mailto:formation@dalibo.com

Remerciements

Ce manuel de formation est une aventure collective qui se transmet au sein de notre société depuis des années. Nous remercions chaleureusement ici toutes les personnes qui ont contribué directement ou indirectement à cet ouvrage, notamment :

Jean-Paul Argudo, Alexandre Anriot, Carole Arnaud, Alexandre Baron, David Bidoc, Sharon Bonan, Franck Boudehen, Arnaud Bruniquel, Pierrick Chovelon, Damien Clochard, Christophe Courtois, Marc Cousin, Gilles Darold, Jehan-Guillaume de Rorthais, Ronan Dunklau, Vik Fearing, Stefan Fercot, Pierre Giraud, Nicolas Gollet, Dimitri Fontaine, Florent Jardin, Virginie Jourdan, Luc Lamarle, Denis Laxalde, Guillaume Lelarge, Alain Lesage, Benoit Lobréau, Jean-Louis Louër, Thibaut Madelaine, Adrien Nayrat, Alexandre Pereira, Flavie Perette, Robin Portigliatti, Thomas Reiss, Maël Rimbault, Julien Rouhaud, Stéphane Schildknecht, Julien Tachoires, Nicolas Thauvin, Be Hai Tran, Christophe Truffier, Cédric Villemain, Thibaud Walkowiak, Frédéric Yhuel.

Forme de ce manuel

Les versions PDF, EPUB ou HTML de ce document sont structurées autour des slides de nos formations. Le texte suivant chaque slide contient le cours et de nombreux détails qui ne peuvent être données à l'oral.

Licence Creative Commons CC-BY-NC-SA

Cela inclut les diapositives, les manuels eux-mêmes et les travaux pratiques. Cette formation peut également contenir quelques images et schémas dont la redistribution est soumise à des licences différentes qui sont alors précisées.

Marques déposées

PostgreSQL® Postgres® et le logo Slonik sont des marques déposées[3] par PostgreSQL Community Association of Canada.

Versions de PostgreSQL couvertes

Ce document ne couvre que les versions supportées de PostgreSQL au moment de sa rédaction, soit les versions 12 à 16.

Sur les versions précédentes susceptibles d'être encore rencontrées en production, seuls quelques points très importants sont évoqués, en plus éventuellement de quelques éléments historiques.

Sauf précision contraire, le système d'exploitation utilisé est Linux.

[3] https://www.postgresql.org/about/policies/trademarks/

1/ Masquage de données & postgresql_anonymizer

1.1 CAS D'USAGE

- Paul : le propriétaire
- Pierre : *data scientist*
- Jack : employé chargé des fournisseurs

La boutique de Paul a beaucoup de clients. Paul demande à son ami Pierre, *data scientist*, des statistiques sur ses clients (âge moyen, etc.).

Pierre demande un accès direct à la base de données pour écrire ses requêtes SQL.

Jack est un employé de Paul, chargé des relations avec les divers fournisseurs de la boutique.

Paul respecte la vie privée de ses fournisseurs. Il doit masquer les informations personnelles à Pierre, mais Jack doit pouvoir lire les vraies données, et y écrire.

Crédits

Cet exemple pratique est un travail collectif de Damien Clochard, Be Hai Tran, Florent Jardin et Frédéric Yhuel.

La version originale en anglais est diffusée sous licence PostgreSQL[1].

Le présent document en est l'adaptation en français.

Paul's Boutique est le second album studio du groupe de hip-hop américain les *Beastie Boys*, sorti le 25 juillet 1989 chez Capitol Records.

La photo ci-dessus est d'Erwin Bernal[2], sous licence CC BY 2.0[3].

1.1.1 Objectifs

Nous allons découvrir :

- comment écrire des règles de masquage
- la différence entre masquage dynamique et masquage statique
- comment implémenter un masquage avancé

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

[2] https://www.flickr.com/photos/edogisgod/16858046971/in/photolist-rFFU3g-4NZreN-xKEkv-h9ZxgT-aMD5mr-8dAvwU-cru1CN-xFfJgh-8QhtH6-6E81fG-zUN3Rg-7dCVPA-5VbEct-ewX2Lc-hA4JqP-psCh1y-dmZpzf-pjkwX-cu5NNQ-fTyVqj-MjtjRc-cLdmvW-3fd5BM-9m6ChY-dwLhRK-9d2A9s-6WsfHq-abVSJd-dWYBD4-gmJyHe-bVyJNn-SHTV2b-BoMvci-abVPJW-5pgugb-r4oJpD-6YeAE3-6kKVpZ-e4LeQN-BLUvZG-do1bDr-o5YACZ-9karFj-dPuSLW-btYwsQ-e4L9h9-abT31z-3eWVAZ-abVN43-btYwvJ

[3] https://creativecommons.org/licenses/by/2.0/deed.fr

2/ PostgreSQL Anonymizer

2.0.1 Principe

Principe :

- Extension
- Déclaratif (DDL)

`postgresql_anonymizer` est une extension pour masquer ou remplacer des données personnelles[1] (ou PII pour *personnally identifiable information*) ou toute donnée sensible dans une base de données PostgreSQL.

Le projet a une **approche déclarative** de l'anonymisation. Vous pouvez déclarer les règles de masquage[2] dans PostgreSQL avec du DDL (*Data Definition Language*, ou langage de définition des données) et spécifier votre stratégie d'anonymisation dans la définition de la table.

2.0.2 Masquages

Principe :

- statique
- dynamique
- sauvegardes anonymisées
- « généralisation »

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

Une fois les règles de masquage définies, vous pouvez accéder aux données anonymisées de quatre manières différentes :

- le masquage statique[3] : supprime les données personnelles en fonction des règles ;
- le masquage dynamique[4] : cache les données personnelles uniquement des utilisateurs masqués ;
- les sauvegardes anonymisées[5] : export des données masquées vers un fichier SQL (sauvegarde logique) ;
- la généralisation[6] : crée des « vues brouillées » des données originales.

Cette présentation n'entrera pas dans le détail du RGPD et des principes généraux d'anonymisation. Pour plus d'informations, référez-vous à la présentation de Damien Clochard ci-dessous :

- Anonymisation, Au-delà du RGPD (vidéo)[7] (PGSession 12, Paris 2019)
- Anonymisation, Au-delà du RGPD (PDF)[8]
- Anonymization, Beyond GDPR (PDF en anglais)[9]

2.0.3 Pré-requis

Cet exemple nécessite :

- une instance PostgreSQL ;
- l'extension **PostgreSQL Anonymizer** (`anon`)
  - installée, initialisée par un super-utilisateur
- une base **boutique**
  - dont le propriétaire est **paul**, super-utilisateur
- les rôles **pierre** et **jack**
  - avec droits de connexion à **boutique**

Voir section « Installation »[10] dans la documentation[11] pour savoir comment installer l'extension dans votre instance PostgreSQL.

[3] https://postgresql-anonymizer.readthedocs.io/en/stable/static_masking/
[4] https://postgresql-anonymizer.readthedocs.io/en/stable/dynamic_masking/
[5] https://postgresql-anonymizer.readthedocs.io/en/stable/anonymous_dumps/
[6] https://postgresql-anonymizer.readthedocs.io/en/stable/generalization/
[7] https://www.youtube.com/watch?v=KGSlp4UygdU
[8] https://public.dalibo.com/exports/conferences/20191210_poss_anonymisation/anonymisation.pdf
[9] https://public.dalibo.com/exports/conferences/20191016_anonymisation_beyond_GDPR/anonymisation_beyond_gdpr.pdf
[10] https://postgresql-anonymizer.readthedocs.io/en/stable/INSTALL
[11] https://postgresql-anonymizer.readthedocs.io/en/stable/

Par exemple :

2.0.3.1 Sous Rocky Linux 8

- Les dépôts du PGDG[12] doivent être installés.
- Lancer :

```
sudo yum install postgresql_anonymizer_14
```

2.0.3.2 Par le PGXN

Sur Debian/Ubuntu, les paquets ne sont pas disponibles au moment où ceci est écrit. Le PGXN permet d'installer l'extension (ici en PostgreSQL 14) :

```
sudo apt install pgxnclient postgresql-server-dev-14
sudo pgxn install postgresql_anonymizer
```

S'il y a plusieurs versions de PostgreSQL installées, indiquer le répertoire des binaires de la bonne version ainsi :

```
sudo PATH=/usr/lib/postgresql/14/bin:$PATH   pgxn install postgresql_anonymizer
```

L'extension sera compilée et installée.

2.0.4 Base d'exemple

pierre, **paul**, **jack** et la base **boutique** :

```
CREATE ROLE paul LOGIN SUPERUSER;
CREATE ROLE pierre LOGIN;
CREATE ROLE jack LOGIN;

-- Define a password for each user with:
-- \password paul  or  ALTER ROLE paul PASSWORD 'change-me';

CREATE DATABASE boutique OWNER paul;

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

Sauf précision contraire, toutes les commandes sont à exécuter en tant que **paul**.

[12] https://yum.postgresql.org/

3/ Masquage statique avec postgresql_anonymizer

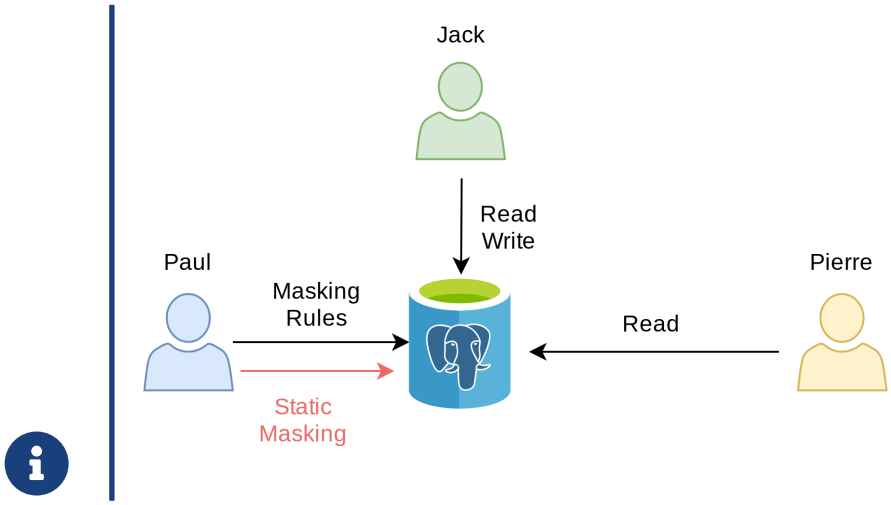
- Le plus simple
- Destructif

Le masquage statique est la manière la plus simple de cacher des données personnelles. L'idée est simplement de détruire les données originales et de les remplacer par des données artificielles.

3.1 L'HISTOIRE

- Au fil des années, Paul a accumulé des données sur ses clients et leurs achats dans une base de données très simple.
- Il a récemment installé un nouveau logiciel de ventes, et l'ancienne base est obsolète.
- Avant de l'archiver, il voudrait en supprimer toutes les données personnelles.

3.2 COMMENT ÇA MARCHE



3.3 OBJECTIFS

Nous allons voir :

- comment écrire des règles de masquage simples
- les intérêts et limitations du masquage statique
- le concept de « singularisation » d'une personne (*singling out*)

3.4 TABLE « CUSTOMER »

```
\c boutique paul
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
);
```

3.4.1 Quelques clients

Insertion de quelques personnes :

```
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 |              | 1951-09-25 | 90120
 341 | Don       | Draper    | 347-515-3423 | 1926-06-01 | 04520
(3 lignes)
```

3.5 TABLE « PAYOUT »

Les ventes sont suivies dans cette simple table :

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

3.5.1 Quelques données

Quelques commandes :

```
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')
;
```

3.5.2 Activer l'extension

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

NB : l'extension `pgcrypto` sera installée automatiquement. Ses binaires sont généralement livrés avec PostgreSQL.

3.6 DÉCLARER LES RÈGLES DE MASQUAGE

```
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)';
```

Paul veut masquer le nom de famille et le numéro de téléphone de ses clients.

Pour cela, il utilise les fonctions `fake_last_name()` et `partial()`.

3.7 APPLIQUER LES RÈGLES DE MANIÈRE PERMANENTE

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

Cette fonction ne fait qu'appliquer la règle et doit renvoyer True. Ensuite :

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

```
 id  | firstname | lastname |    phone
-----+-----------+----------+--------------
 107 | Sarah     | Okuneva  | 06X-XXX-XX11
 258 | Luke      | Okuneva  |
 341 | Don       | Boyle    | 34X-XXX-XX23
```

Cette technique est nommée « masquage statique » car la donnée réelle a été détruite de manière définitive. L'anonymisation dynamique et les exports seront vus plus loin.

3.8 EXERCICES

3.8.1 E101 - Masquer les prénoms des clients

> Déclarer une nouvelle règle de masquage et relancer l'anonymisation statique.

3.8.2 E102 - Masquer les 3 derniers chiffres du code postal

Paul réalise que le code postal est un bon indice sur l'endroit où vivent ses clients. Cependant, il voudrait pouvoir faire des statistiques par département.

> Créer une règle de masquage pour remplacer les 3 derniers chiffres du code postal par 'x'.

3.8.3 E103 - Compter le nombre de clients dans chaque département.

> Agréger les clients selon le code postal anonymisé.

3.8.4 E104 - Ne garder que l'année dans les dates de naissance

Paul veut des statistiques selon l'âge. Mais il veut aussi masquer les vraies dates de naissance de ses clients.

> Remplacer toutes les dates de naissance par le 1er janvier, en conservant l'année réelle. Utiliser la fonction make_date[a].
>
> [a] https://www.postgresql.org/docs/current/functions-datetime.html#FUNCTIONS-DATETIME-TABLE

3.8.5 E105 - Identifier un client particulier

Même si un client est correctement anonymisé, il est possible d'isoler un individu grâce à des données d'autres tables. Par exemple, il est possible d'identifier le meilleur client de Paul avec une telle requête :

```
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       | Boyle    | 34X-XXX-XX23 | 1926-06-01 | 04520
```

Ce processus est appelé « singularisation » (*singling out*[1]) d'une personne.

Il faut donc aller plus loin dans l'anonymisation, en supprimant le lien entre une personne et sa société. Dans la table des commandes `order`, ce lien est matérialisé par une clé étrangère sur le champ `fk_company_id`. Mais nous ne pouvons supprimer des valeurs de cette colonne ou y insérer de faux identifiants, car cela briserait la contrainte de clé étrangère.

Comment séparer les clients de leurs paiements tout en respectant l'intégrité des données ?

> Trouver une fonction qui mélange les valeurs de `fk_company_id` dans la table `payout`. Consulter la section *shuffling*[a] de la documentation[b].
>
> [a] https://postgresql-anonymizer.readthedocs.io/en/stable/static_masking/#shuffling
> [b] https://postgresql-anonymizer.readthedocs.io/en/stable/

[1] https://www.pnas.org/content/117/15/8344

3.9 SOLUTIONS

3.9.1 S101

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

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

La table anonymisée devient :

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

 id  | firstname | lastname
-----+-----------+----------
 107 | Hans      | Barton
 258 | Jacqueline | Dare
 341 | Sibyl     | Runte
```

3.9.2 S102

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

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

Le code postal anonymisé devient :

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

 id  | firstname | lastname | postcode
-----+-----------+----------+----------
 107 | Curt      | O'Hara   | 90xxx
 258 | Agusta    | Towne    | 90xxx
 341 | Sid       | Hane     | 04xxx
```

Noter que les noms ont encore changé après application de `anon.anonymize_table()`.

3.9.3 S103

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

 postcode | count
----------+-------
 90xxx    |     2
 04xxx    |     1
```

3.9.4 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');
```

Les dates de naissance anonymisées deviennent :

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

```
 id  | firstname | lastname |   birth
-----+-----------+----------+------------
 107 | Pinkie    | Sporer   | 1965-01-01
 258 | Zebulon   | Gerlach  | 1951-01-01
 341 | Erna      | Emmerich | 1926-01-01
```

3.9.5 S105

Pour mélanger les valeurs de `fk_customer_id` :

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

Si l'on essaie à nouveau d'identifier le meilleur client :

```
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 | Zebulon   | Gerlach  |       | 1951-01-01 | 90xxx
```

Noter que le lien entre un client (`customer`) et ses paiements (`payout`) est à présent complètement faux !

Par exemple, si un client A a deux paiements, l'un se retrouvera associé à un client B, et l'autre à un client C. En d'autres termes, cette méthode de mélange respectera la contrainte d'intégrité technique, mais brisera l'intégrité des données. Pour certaines utilisations, ce peut être problématique.

Ici, Pierre ne pourra pas produire de rapport avec les données mélangées.

4/ Masquage dynamique avec postgresql_anonymizer

4.1 PRINCIPE DU MASQUAGE DYNAMIQUE

- Masquer les données personnelles à certains utilisateurs
  - mais pas tous

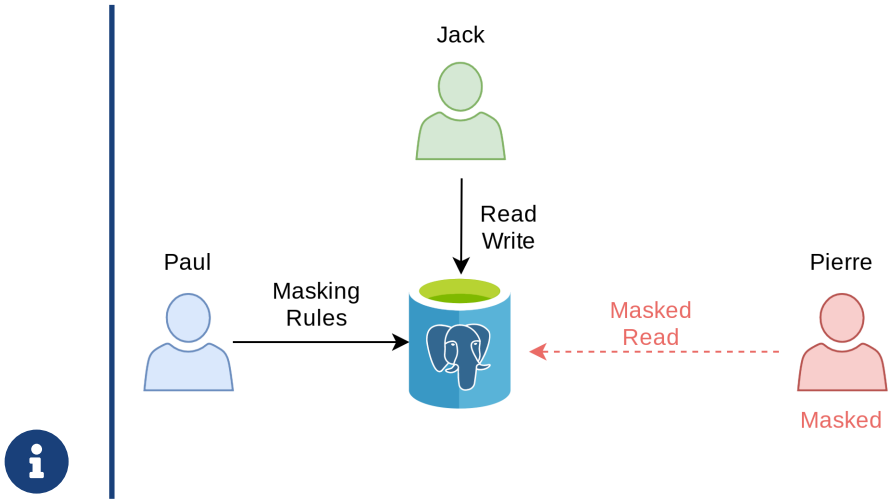
Avec le masquage dynamique, le propriétaire de la base peut masquer les données personnelles à certains utilisateurs, tout en laissant aux autres les droits de lire et modifier les données réelles.

4.2 L'HISTOIRE

Paul a 2 employés :

- **Jack** s'occupe du nouveau logiciel de ventes.
  - il a besoin d'accéder aux vraies données
  - pour le RGPD c'est un « processeur de données »
- **Pierre** est un analyste qui exécute des requêtes statistiques
  - il ne doit pas avoir accès aux données personnelles

4.3 COMMENT ÇA MARCHE



4.4 OBJECTIFS DE LA SECTION

Nous allons voir :

- comment écrire des règles de masquage simple
- les avantages et limitations du masquage dynamique
- le concept de « recoupement » d'une personne (*linkability*)

4.5 TABLE « COMPANY »

```
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
);
```

4.5.1 Quelques données

```
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
```

4.6 TABLE « SUPPLIER »

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

4.6.1 Quelques données

```
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
```

4.7 ACTIVER L'EXTENSION

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

4.8 ACTIVER LE MASQUAGE DYNAMIQUE

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

4.9 RÔLE MASQUÉ

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

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

Le rôle **pierre** devient « masqué », dans le sens où le super-utilisateur va pouvoir lui imposer un masque qui va changer sa vision des données.

En tant que Pierre, on essaie de lire la table des fournisseurs :

```
\c boutique pierre
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
```

Pour le moment, il n'y a pas de règle de masquage : Pierre peut voir les données originales dans chaque table.

4.10 MASQUER LE NOM DES FOURNISSEURS

En tant que Paul, une règle de masquage se définit ainsi :

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

Pierre essaie de lire la table des fournisseurs :

```
\c boutique pierre
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
```

Si Jack essaie de lire les vraies données, ce sont encore les bonnes :

```
\c boutique jack
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
```

4.11 EXERCICES

4.11.1 E201 - Deviner qui est le PDG de « Johnny's Shoe Store »

Masquer le nom du fournisseur n'est pas suffisant pour anonymiser les données.

> Se connecter en tant que Pierre. Écrire une requête simple permettant de recouper certains fournisseurs en se basant sur leur poste et leur société.

Les noms des sociétés et les postes de travail sont disponibles dans de nombreux jeux de données publics. Une simple recherche sur LinkedIn ou Google révèle les noms des principaux dirigeants de la plupart des sociétés…

On nomme « recoupement » la possibilité de rapprocher plusieurs données concernant la même personne.

4.11.2 E202 - Anonymiser les sociétés

Nous devons donc anonymiser aussi la table `company`. Même si elle ne contient pas d'informations personnelles, certains champs peuvent être utilisés pour identifier certains de leurs employés…

> Écrire deux règles de masquage pour la table `company`. La première doit remplacer le champ `nom` avec un faux nom. La seconde remplacer `vat_id` avec une suite aléatoire de dix caractères. NB : dans la documentation[a], consulter les générateurs de données factices[b] et fonctions aléatoires[c] (*faking functions*).
>
> [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

> Vérifier que Pierre ne peut pas voir les vraies données sur la société.

4.11.3 E203 - Pseudonymiser le nom des sociétés

À cause du masquage dynamique, les valeurs artificielles sont différentes **à chaque fois que Pierre lit la table**. Ce n'est pas toujours très pratique.

Pierre préfère appliquer tout le temps les mêmes valeurs artificielles pour une même société. Cela correspond à la « pseudonymisation ».

La **pseudonymisation** consiste à générer systématiquement les mêmes données artificielles pour un individu donné à la place de ses données réelles.

Écrire une nouvelle règle de masquage à partir du champ `name`, grâce à une fonction de pseudonymisation[a].

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

4.12 SOLUTIONS

4.12.1 S201

```
\c boutique pierre
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
```

4.12.2 S202

```
\c boutique paul
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)';
```

En tant Pierre, relire la table :

```
\c boutique pierre
SELECT * FROM company;
```

```
 id  |            name              |   vat_id
-----+------------------------------+------------
 952 | Graham, Davis and Bauer      | LYFVSI3WT5
 194 | Martinez-Smith               | 9N62K8M6JD
 346 | James, Rodriguez and Nelson  | OHB20OZ4Q3
```

À chaque lecture de la table, Pierre voit des données différentes :

```
SELECT * FROM company;
```

```
 id  |            name              |   vat_id
-----+------------------------------+------------
 952 | Holt, Moreno and Richardson  | KPAJP2Q4PK
 194 | Castillo Group               | NVGHZ1K50Z
 346 | Mccarthy-Davis               | GS3AHXBQTK
```

4.12.3 S203

```
\c boutique paul
ALTER FUNCTION anon.pseudo_company SECURITY DEFINER;

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

Pour Pierre, les valeurs pseudonymisées resteront identiques entre deux appels (mais pas le code TVA) :

```
\c boutique pierre
SELECT * FROM company;

 id  |      name       |   vat_id
-----+-----------------+------------
 952 | Wilkinson LLC   | IKL88GJVT4
 194 | Johnson PLC     | VOOJ6UKR6H
 346 | Young-Carpenter | DUR78F15VD

SELECT * FROM company;

 id  |      name       |   vat_id
-----+-----------------+------------
 952 | Wilkinson LLC   | DIUAMTI653
 194 | Johnson PLC     | UND2DQGL4S
 346 | Young-Carpenter | X6EOT023AK
```

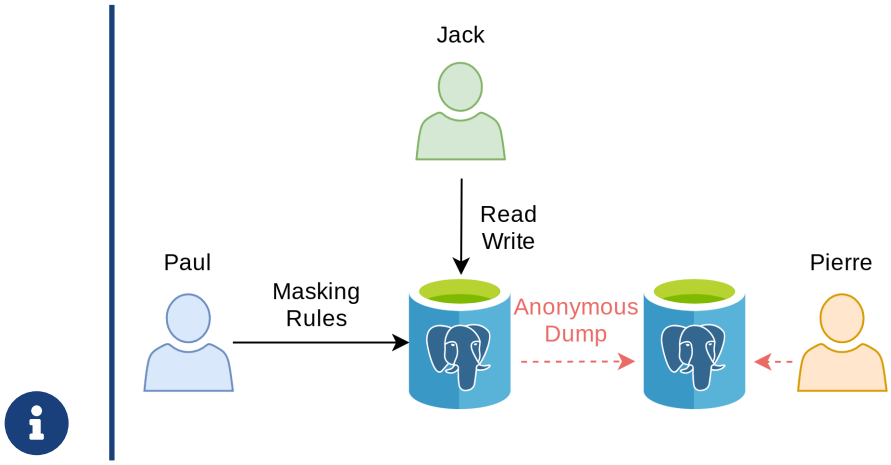
5/ Sauvegardes anonymes avec postgresql_anonymizer

Dans beaucoup de situations, le besoin est simplement d'exporter les données anonymisées pour les importer dans une autre base de données, pour mener des tests ou produire des statistiques. C'est ce que permet de faire l'outil **pg_dump_anon**.

5.1 L'HISTOIRE

- Paul a un site web qui dispose d'une section commentaires où les utilisateurs peuvent partager leurs points de vue.
- Paul a engagé un prestataire pour développer le nouveau design de son site web.
- Le prestataire lui demande un export de la base de données.
- Paul veut « nettoyer » le dump et y retirer toute information personnelle qui pourrait figurer dans la section commentaire.

5.2 COMMENT ÇA MARCHE ?



5.3 OBJECTIFS

- Extraire les données anonymisées de la base de données
- Écrire une fonction de masquage personnalisée pour gérer une colonne de type JSON

5.4 TABLE « WEBSITE_COMMENT »

```
\c boutique paul
DROP TABLE IF EXISTS website_comment CASCADE ;

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

5.4.1 Quelques données

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

    1   {"meta": {"name": "Lee Perry", "ip_addr": "40.87.29.113"},
     ↪  "content": "Hello Nasty!"}
    2   {"meta": {"name": "", "email": "biz@bizmarkie.com"}, "content":
     ↪  "Great Shop"}
    3   {"meta": {"name": "Jimmy"}, "content": "Hi! This is me, Jimmy
     ↪  James "}
```

```
\c boutique paul
\copy website_comment from '/tmp/website_comment.tsv'

SELECT jsonb_pretty(message)
FROM website_comment
ORDER BY id ASC
LIMIT 1 ;

           jsonb_pretty
-----------------------------------
 {                                 +
     "meta": {                     +
         "name": "Lee Perry",      +
         "ip_addr": "40.87.29.113"+
     },                            +
     "content": "Hello Nasty!"     +
 }
```

5.5 ACTIVER L'EXTENSION

```
\c boutique paul

CREATE EXTENSION IF NOT EXISTS anon CASCADE;

SELECT anon.init();

SELECT setseed(0);
```

5.6 MASQUER UNE COLONNE DE TYPE JSON

Généralement, les données non structurées sont difficiles à masquer…

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

La colonne `comment` contient beaucoup d'informations personnelles. Le fait de ne pas utiliser un schéma standard pour les commentaires rend ici la tâche plus complexe.

Comme on peut le voir, les visiteurs du site peuvent écrire toutes sortes d'informations dans la section « commentaire ». La meilleure option serait donc de supprimer entièrement la clé JSON car il est impossible d'y exclure les données sensibles.

Il est possible de nettoyer la colonne `comment` en supprimant la clé `content` :

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

5.6.1 Fonctions de masquage personnalisées

```
\c boutique paul

CREATE SCHEMA IF NOT EXISTS my_masks;

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

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

- Super-utilisateurs seulement !

Créer en premier lieu un schéma dédié, ici **my_masks**, et le déclarer en `trusted` (« de confiance »). Cela signifie que l'extension `anon` va considérer les fonctions de ce schéma comme des fonctions de masquage valides.

Seul un super-utilisateur devrait être capable d'ajouter des fonctions dans ce schéma !

Cette fonction de masquage se contente de supprimer du JSON le champ avec le message :

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

Exécuter la fonction :

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

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

La fonction va pouvoir ensuite être utilisée dans une règle de masquage.

5.6.2 Utilisation de la fonction de masquage personnalisée

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

5.6.3 Sauvegarde anonymisée

- Export

```
pg_dump_anon.sh -U paul -d boutique --table=website_comment >
↪  /tmp/dump.sql
```

- Limitations : risque d'inconsistence, format plain

Enfin, une **sauvegarde logique anonymisée** de la table peut être exportée avec l'utilitaire `pg_dump_anon`. Celui-ci est un script, livré avec l'extension, disponible dans le répertoire des binaires :

L'outil utilise `pg_dump`, il vaut mieux qu'il n'y ait pas d'ambiguïté sur le chemin :

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

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

En ne demandant que les données (option `-a`), le résultat contient notamment :

```
COPY public.website_comment (id, message) FROM stdin;
1       {"meta": {"name": "Lee Perry", "ip_addr": "40.87.29.113"}, "content": "Hello
↪  Nasty!"}
2       {"meta": {"name": "", "email": "biz@bizmarkie.com"}, "content": "Great Shop"}
3       {"meta": {"name": "Jimmy"}, "content": "Hi ! This is me, Jimmy James "}
\.
```

pg_dump_anon ne vise pas à réimplémenter toutes les fonctionnalités de pg_dump. Il n'en supporte qu'une partie, notamment l'extraction d'objets précis, mais pas la compression par exemple. De plus, en raison de son fonctionnement interne, il y a un risque que la restauration des données soit incohérente, notamment en cas de DML ou DDL pendant la sauvegarde. Une sauvegarde totalement cohérente impose de passer un masquage statique et un export classique par pg_dump.

Le produit est en développement actif et la situation peut avoir changé depuis que ces lignes ont été écrites. Il est notamment prévu que le script bash soit remplacé par un script en go.

5.7 EXERCICES

5.7.1 E301 - Exporter les données anonymisées dans une nouvelle base de données

> Créer une base de données nommée **boutique_anon**. y insérer les données anonymisées provenant de la base de données **boutique**.

5.7.2 E302 - Pseudonymiser les métadonnées du commentaire

Pierre compte extraire des informations générales depuis les métadonnées. Par exemple, il souhaite calculer le nombre de visiteurs uniques sur la base des adresses IP des visiteurs, mais une adresse IP est un **identifiant indirect**.

Paul doit donc anonymiser cette colonne tout en conservant la possibilité de faire le lien entre les enregistrements provenant de la même adresse.

> Remplacer la fonction `remove_content` par la fonction `clean_comment` (ci-dessous), qui : - supprime la clé JSON `content` ; - remplace la valeur dans la colonne `name` par un faux nom ; - remplace l'adresse IP dans la colonne `ip_address` par sa somme de contrôle `md5` ; - met à `NULL` la clé `email`.

```
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'];
$func$;
```

5.8 SOLUTIONS

5.8.1 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.sh boutique | psql --quiet boutique_anon
```

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

```
 count
-------
     3
```

5.8.2 S302

Suite à utilisation de la fonction personnalisée `clean_comment`, les données n'ont plus rien à voir :

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

```
                                          clean_comment
-------------------------------------------------------------------------------
 ↪  -----------
 {"meta": {"name": "Heller", "email": null, "ip_address":
  ↪  "1d8cbcdef988d55982af1536922ddcd1"}}
 {"meta": {"name": "Christiansen", "email": null, "ip_address": null}}
 {"meta": {"name": "Frami", "email": null, "ip_address": null}}
(3 lignes)
```

On applique le masquage comme à l'habitude :

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

6/ Généralisation avec postgresql_anonymizer

6.1 PRINCIPE

- Flouter les données
- Par ex : 25 juillet 1989 ⇒ années 1980

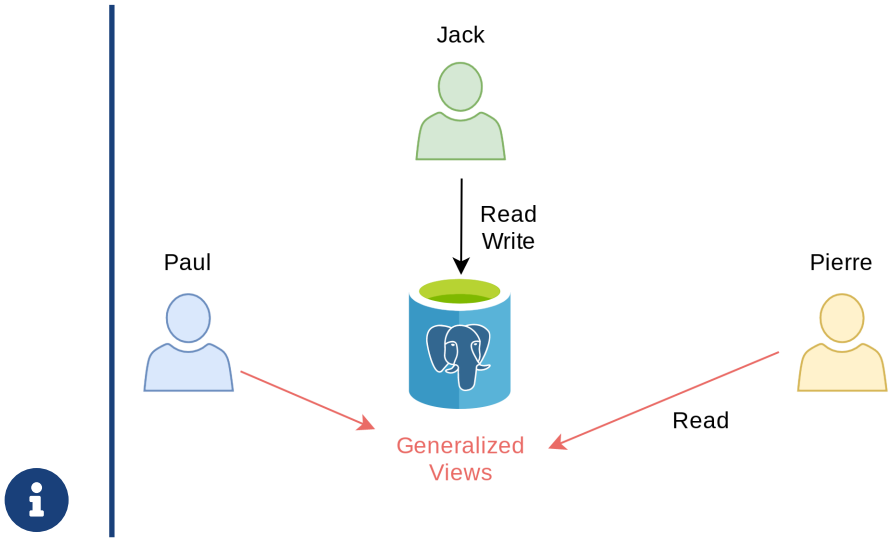
L'idée derrière la **généralisation** est de pouvoir flouter une donnée originale.

Par exemple, au lieu de dire « Monsieur X est né le 25 juillet 1989 », on peut dire « Monsieur X est né dans les années 1980 ». L'information reste vraie, bien que moins précise, et elle rend plus difficile l'identification de la personne.

6.2 L'HISTOIRE

- Paul a embauché des dizaines de salariés au fil du temps.
- Il conserve une trace sur la couleur de leurs cheveux, leurs tailles, et leurs conditions médicales.
- Paul souhaite extraire des statistiques depuis ces détails.
- Il fournit des vues **généralisées** à Pierre.

6.3 COMMENT ÇA MARCHE ?



6.4 OBJECTIFS

Nous allons voir :

- la différence entre le **masquage** et la **généralisation**
- le concept de « *k*-anonymat »

6.5 TABLE « EMPLOYEE »

```
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']))
);
```

- Légal ?

Bien sûr, stocker les caractéristiques physiques d'employés est généralement illégal. Quoi qu'il en soit, il sera impératif de les masquer.

6.6 QUELQUES DONNÉES

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

\c boutique paul
\COPY employee FROM '/tmp/employee.tsv'
```

Ce fichier charge 16 lignes, dont :

```
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    | t
 Paul Wolf    | 2020-01-15 | bald  | M    | f
 Rowan Hoeger | 2018-12-01 | dark  | XXL  | t
```

6.7 SUPPRESSION DE DONNÉES

Pierre peut trouver un lien entre asthme et yeux verts :

```
\c boutique paul
DROP MATERIALIZED VIEW IF EXISTS v_asthma_eyes ;

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

Paul souhaite savoir s'il y a une corrélation entre l'asthme et la couleur des yeux.

Il fournit à Pierre la vue ci-dessus, qui peut désormais écrire des requêtes sur cette vue :

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

```
 eyes  | asthma
-------+--------
 blue  | t
 brown | f
 blue  | t
```

```
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
```

Paul vient de prouver que l'asthme est favorisé par les yeux verts, et surtout de trouver une corrélation entre deux champs.

6.8 CALCULER LE K-ANONYMAT

- Les colonnes `asthma` et `eyes` sont considérés comme des identifiants indirects.

```
\c boutique paul
SECURITY LABEL FOR anon ON COLUMN v_asthma_eyes.eyes
IS 'INDIRECT IDENTIFIER';

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

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

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

```
 k_anonymity
-------------
           2
```

La vue `v_asthma_eyes` a le niveau « 2-anonymity ». Cela signifie que chaque combinaison de quasi-identifiants (`eyes` - `asthma`) apparaît au moins 2 fois dans le jeu de données.

En d'autres termes, cela veut dire qu'un individu ne peut pas être distingué d'au moins un autre individu (k-1) dans cette vue.

Pour les détails sur le K-anonymat, voir cet article sur Wikipédia[1].

[1] https://en.wikipedia.org/wiki/K-anonymity

6.9 FONCTIONS D'INTERVALLE ET DE GÉNÉRALISATION

```
\c boutique paul
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 ;

GRANT SELECT ON v_staff_per_month TO pierre ;
```

```
\c boutique pierre
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) | (,)
 [2018-12-01,2019-01-01) | [2018-12-01,2019-01-01)
```

Pierre peut écrire une requête pour trouver le nombre d'employés embauchés en novembre 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
```

6.9.1 Déclarer les identifiants indirects

Calculer le facteur de *k*-anonymat de cette vue :

```
\c boutique paul
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');
```

Dans ce cas, le résultat est 1, ce qui veut dire qu'au moins une personne peut être directement identifiée par les dates de ses premier et dernier jour en poste.

Dans ce cas, la généralisation est insuffisante.

6.10 EXERCICES

6.10.1 E401 - Simplifier la vue v_staff_per_month pour en réduire la granularité.

Généraliser les dates en mois n'est pas suffisant. > Écrire une autre vue > v_staff_per_year qui va généraliser les dates en années. > Simplifier également la vue en utilisant un intervalle de `int` pour stocker > l'année, plutôt qu'un intervalle de `date`.

6.10.2 E402 - Progression du personnel au fil des années

> Combien de personnes ont travaillé pour Paul chaque année entre 2018 et 2021 ?

6.10.3 E403 - Atteindre le facteur *2-anonymity* sur la vue v_staff_per_year

> Quel est le facteur *k*-anonymat de la vue v_staff_per_year ?

6.11 SOLUTIONS

6.11.1 S401

Cette vue généralise les dates en années :

```
\c boutique paul
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,
    '[]' -- include upper bound
  ) AS period
FROM employee;

SELECT *
FROM v_staff_per_year
LIMIT 3;

   period
-------------
 [2018,2019)
 [2020,)
 [2018,2019)
```

6.11.2 S402

Les personnes ayant travaillé pour Paul entre 2018 et 2021 sont :

```
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
```

6.11.3 S403

Le k-anonymat de cette vue est meilleur :

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

SELECT anon.k_anonymity('v_staff_per_year');

 k_anonymity
-------------
           2
```

7/ Conclusion sur postgresql_anonymizer

7.1 BEAUCOUP DE STRATÉGIES DE MASQUAGE

- Masquage statique[1] : parfait pour une anonymisation « une fois pour toute »
- Masquage dynamique[2] : utile pour masquer des informations à certains utilisateurs
- Sauvegardes anonymisées[3] : peuvent être utilisées dans des traitement CI/CD
- Généralisation[4] : adaptée aux statistiques et à l'analyse de données

7.2 BEAUCOUP DE FONCTIONS DE MASQUAGE

- Destruction partielle ou totale
- Ajout de bruit
- Randomisation
- Falsification et falsification avancée
- Pseudonymisation
- Hachage générique
- Masquage personnalisé

RTFM -> Fonctions de masquage[5]

7.3 AVANTAGES

- Règles de masquage en SQL
- Règles de masquage stockées dans le schéma de la base
- Pas besoin d'un ETL
- Fonctionne avec toutes les versions actuelles de PostgreSQL
- Multiples stratégies, multiples fonctions.

7.4 INCONVÉNIENTS

- Ne fonctionne pas avec d'autres systèmes de gestion de bases de données (comme le nom l'indique)
- Peu de retour d'expérience sur de gros volumes (> 10 To)

7.5 POUR ALLER PLUS LOIN

D'autres projets qui pourraient vous plaire :

- pg_sample[6] : Extraire un petit jeu de données d'une base de données volumineuse
- PostgreSQL Faker[7] : Une extension de falsification avancée basée sur la bibliothèque python `Faker`.

7.6 CONTRIBUEZ !

C'est un projet libre !
labs.dalibo.com/postgresql_anonymizer[8]
Merci de vos retours sur la manière dont vous l'utilisez, comment il répond ou non à vos attentes, etc.

7.6.1 Questions

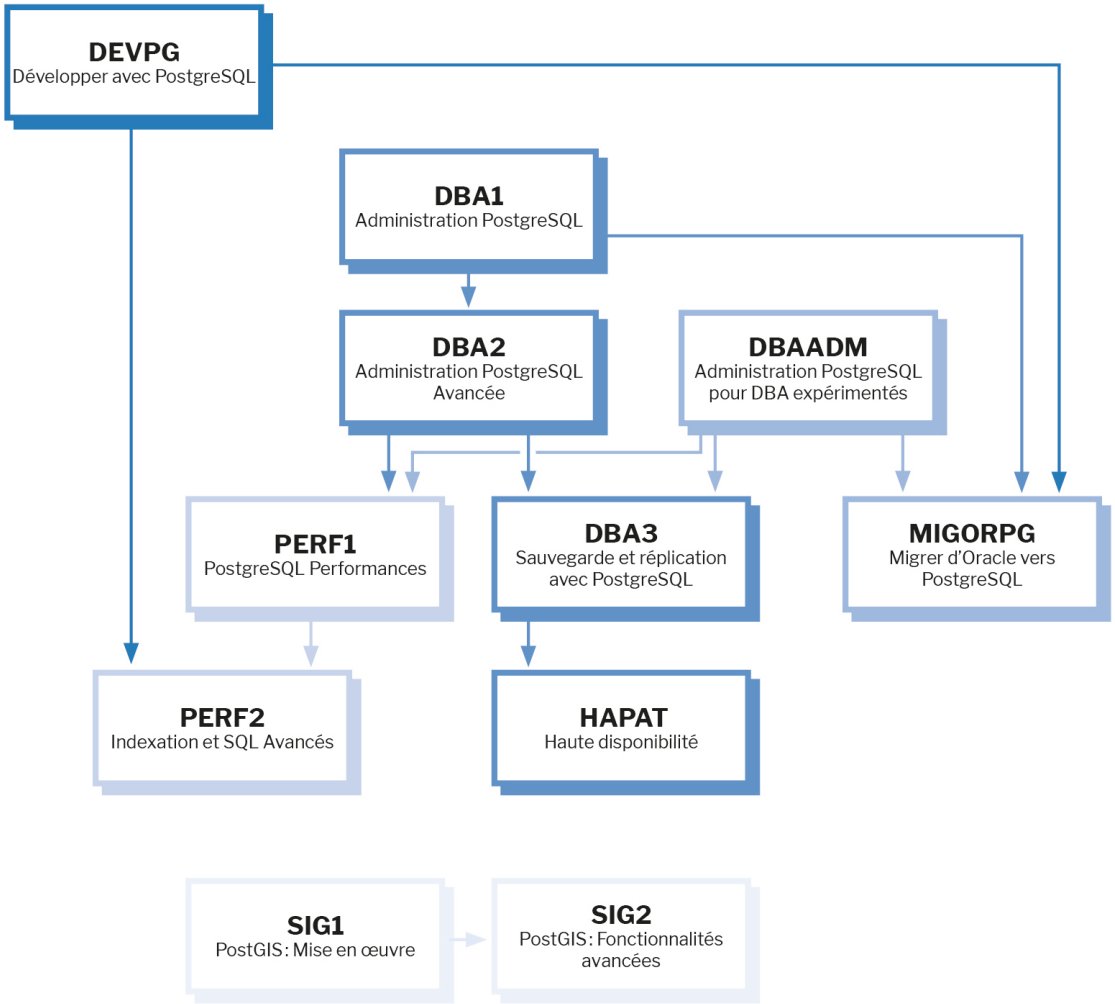
N'hésitez pas, c'est le moment !

Les formations Dalibo

Retrouvez nos formations et le calendrier sur https://dali.bo/formation

Pour toute information ou question, n'hésitez pas à nous écrire sur contact@dalibo.com.

Cursus des formations



Retrouvez nos formations dans leur dernière version :

- DBA1 : Administration PostgreSQL
  https://dali.bo/dba1
- DBA2 : Administration PostgreSQL avancé
  https://dali.bo/dba2
- DBA3 : Sauvegarde et réplication avec PostgreSQL
  https://dali.bo/dba3
- DEVPG : Développer avec PostgreSQL
  https://dali.bo/devpg
- PERF1 : PostgreSQL Performances
  https://dali.bo/perf1
- PERF2 : Indexation et SQL avancés
  https://dali.bo/perf2
- MIGORPG : Migrer d'Oracle à PostgreSQL
  https://dali.bo/migorpg
- HAPAT : Haute disponibilité avec PostgreSQL
  https://dali.bo/hapat

Les livres blancs

- Migrer d'Oracle à PostgreSQL
  https://dali.bo/dlb01
- Industrialiser PostgreSQL
  https://dali.bo/dlb02
- Bonnes pratiques de modélisation avec PostgreSQL
  https://dali.bo/dlb04
- Bonnes pratiques de développement avec PostgreSQL
  https://dali.bo/dlb05

Téléchargement gratuit

Les versions électroniques de nos publications sont disponibles gratuitement sous licence open source ou sous licence Creative Commons.