Beyond GDPR

# PostgreSQL Anonymizer

# PostgreSQL Anonymizer

Beyond GDPR

TITRE : PostgreSQL Anonymizer
SOUS-TITRE : Beyond GDPR

## WHO I AM

- Damien Clochard
- PostgreSQL DBA & Co-founder at Dalibo
- President of PostgreSQLFr Association

---

## WHO I AM NOT

- I Am Not A Lawyer
- I Am Not A Privacy Expert
- Don't take my word for it / Check the links !

---

## MY JOURNEY

---

## MENU

- Why Anonymization is hard
- Anonymization Pipelines
- PostgreSQL Anonymizer

---

## WHY ANONYMIZATION IS HARD

- Singling out
- Linkability
- Inference

(source: WP29 Opinion on Anonymisation Techniques)

---

## SINGLING OUT

The possibility to isolate a record and identify a subject in the dataset.

```
SELECT * FROM employees;
```

```
  id  |  name          | job  | salary
------+----------------+------+--------
 1578 | xkjefus3sfzd   | NULL |   1498
 2552 | cksnd2se5dfa   | NULL |   2257
 5301 | fnefckndc2xn   | NULL |  45489
 7114 | npodn5ltyp3d   | NULL |   1821
```

---

## LINKABILITY

Identify a subject in the dataset using other datasets

- Netflix Ratings + IMDB Ratings
- Hospital visits + State voting records

(sources: Netflix prize + Hospital Reidentification )

## INFERENCE

Identify a subject using a set of indirect identifiers.

87% of the U.S. population are uniquely identified by date of birth, gender and zip code

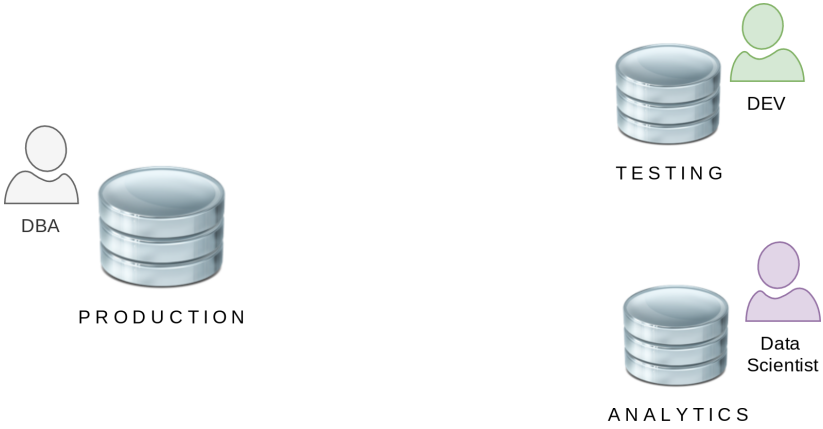
(source : Latanya Sweeney)

## ANONYMIZATION PIPELINES

Minimizing the risk of data leaks by reducing the attack surface

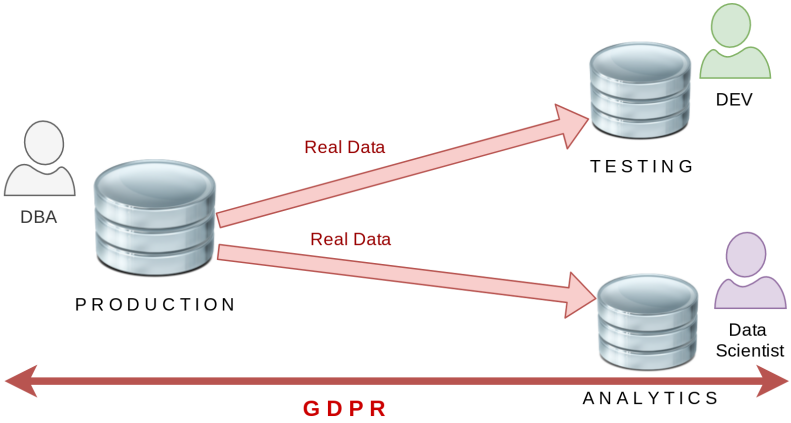
---

### BASIC EXAMPLE



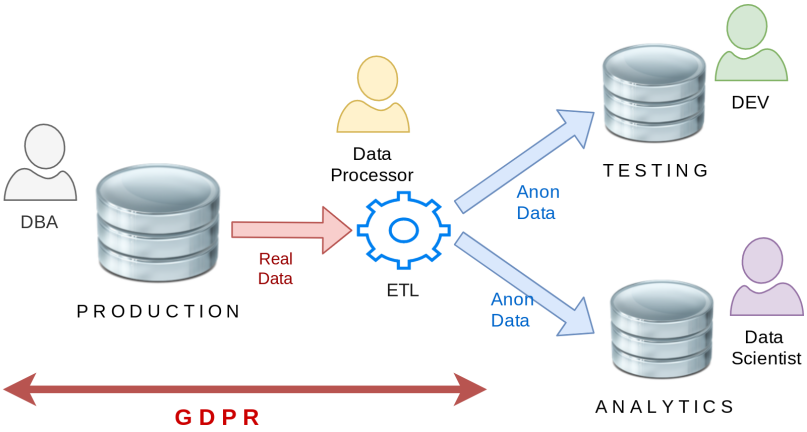

---

### WORST SCENARIO

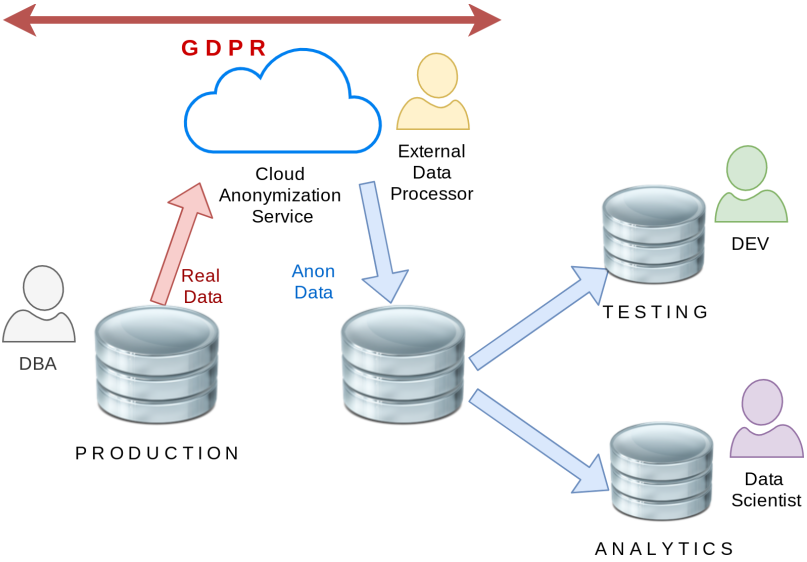



---

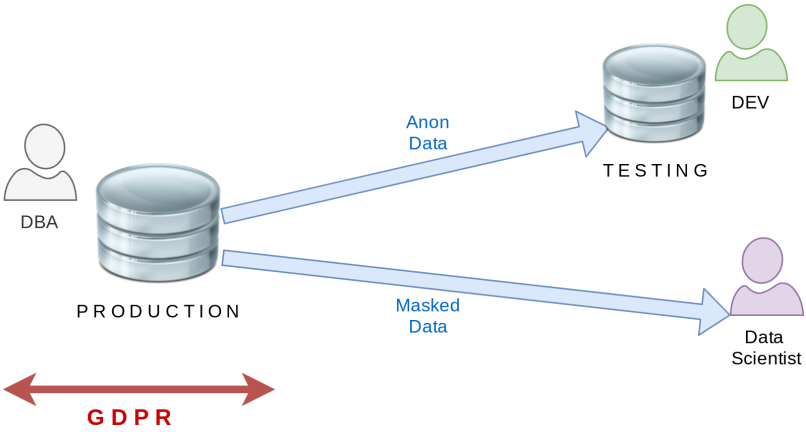
## ETL




## CLOUD ANONYMIZATION



## POSTGRESQL ANONYMIZER



# PostgreSQL Anonymizer

## WHAT IS THIS ?

- Started as a research project in 2018
- Now part of the “Dalibo Labs” initiative
- Currently in beta
- Version 1.0 is coming by the end of 2020

## GOALS

- Declare masking rules within the database model
- Anonymization is done internally
- Dynamic Masking / Anonymous Export / In-Place Masking
- Batteries included : Builtin masking functions
- Inspired by MS SQL Server Dynamic Data Masking

## EXAMPLE: REAL DATA

```
=# SELECT * FROM customer;
 id  |   full_name      |   birth    | zipcode | fk_shop
-----+------------------+------------+---------+---------
 911 | Chuck Norris     | 1940-03-10 | 75001   | 12
 112 | David Hasselhoff | 1952-07-17 | 90001   | 423
```

---

## EXAMPLE: ANONYMIZED DATA

```
=# SELECT * FROM customer;
 id  |     full_name     |   birth    | zipcode | fk_shop
-----+-------------------+------------+---------+---------
 911 | Michel Duffus     | 1970-03-24 | 63824   | 12
 112 | Andromache Tulip  | 1921-03-24 | 38199   | 423
```

---

## INSTALL

Using the Community RPM Repo:

```
$ yum install https://.../pgdg-redhat-repo-latest.noarch.rpm
$ yum install postgresql_anonymizer12
```

---

## LOAD & INIT

```
ALTER DATABASE foo SET session_preload_libraries = 'anon';
CREATE EXTENSION anon CASCADE;
SELECT anon.init();
```

---

## DECLARE A MASKING RULE

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

---

## EXAMPLE

```
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';
```

---

## NOW WE HAVE 3 OPTIONS

- In-Place Anonymization
- Anonymous Dumps
- Dynamic Masking

---

## IN-PLACE ANONYMIZATION

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

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

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

---

## IN-PLACE ANONYMIZATION

This will update all lines of all tables containing at least one masking rule.

This is gonna be slow and trigger heavy write workloads.

---

## ANONYMOUS DUMPS

```
$ pg_dump_anon -h localhost -U bob foo > anonymous_dump.sql
```

---

## DYNAMIC MASKING

Let's take a basic example :

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

---

## DYNAMIC MASKING

Step 1 : Activate the dynamic masking engine

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

---

## DYNAMIC MASKING

Step 2 : Declare a masked user

```
CREATE ROLE skynet LOGIN;

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

The masked user has a read-only access to the anonymized data of the masked tables.

---

## DYNAMIC MASKING

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)';
```

---

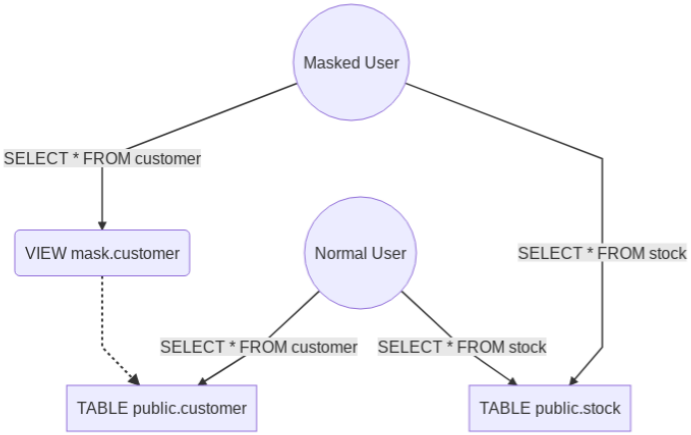
## DYNAMIC MASKING

Step 4 : Connect with the masked user

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

---

## HOW IT WORKS




---

## BATTERIES INCLUDED: 10 MASKING TECHNIQUES

- Destruction
- Noise Addition
- Shuffling / Permutation
- Randomization
- Faking / Synthetizing
- Advanced Faking
- Pseudonymization
- Hashing
- Partial Scrambling
- Generalization

---

## DESTRUCTION

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

---

## DESTRUCTION

- Simple, Fast, Efficient
- Required for `NOT NULL` columns

---

## NOISE ADDITION

```
=# SECURITY LABEL FOR anon
-# ON COLUMN employee.salary
-# IS 'MASKED WITH FUNCTION
-#     anon.add_noise_on_numeric_column(user, salary, 0.33)
-# ';
```

All values of the column will be randomly shifted with a ratio of +/- 33%

---

## NOISE ADDITION

- The dataset remains meaningful
- `AVG()` and `SUM()` are similar to the original
- works only for dates and numeric values
- “extreme values” may cause re-identification (“singling out”)
- risk repetition attack, especially with dynamic masking

---

## SHUFFLING

```
SECURITY LABEL FOR anon
   ON COLUMN employee.fk_company
   IS 'MASKED WITH FUNCTION
     anon.shuffle_column(employee, fk_company, id)
';
```

---

## SHUFFLING

- The dataset remains meaningful
- Perfect for Foreign Keys
- Works bad with low distribution (ex: boolean)
- The table must have a primary key

---

## RANDOMIZATION

```
SECURITY LABEL FOR anon
   ON COLUMN employee.birth
   IS 'MASKED WITH FUNCTION
     anon.random_date_between(''01/01/1920'',now())
';
```

---

## RANDOMIZATION

- Simple and Fast
- Usefull for columns with `NOT NULL` constraints
- Useless for analytics

---

## FAKING

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

---

## FAKING

- Just a more realistic version of Randomization
- Great for developpers and CI tests
- You can load your own dictionnaries !
- Very basic implementation

---

## ADVANCED FAKING

```
CREATE EXTENSION faker SCHEMA faker CASCADE;

SELECT faker.faker('it_IT');

SELECT faker.name();
     name
--------------------------
   Sig. Alighieri Monti
```

---

## ADVANCED FAKING

- Based on the well-known Python `Faker` library
- Complete, Powerful, Extensible
- Slow

---

## PARTIAL SCRAMBLING

```
=# SECURITY LABEL FOR anon
-# ON COLUMN employee.phone
-# IS 'MASKED WITH FUNCTION anon.partial(phone,4,'******',2)';
```

+33142928107 becomes +331******07

---

## PARTIAL DESTRUCTION

- Similar to the “Destruction” approach
- Perfect for phone number, credit cards, etc.
- The user can still recognize his/her own data
- Transformation is IMMUTABLE
- Works only for TEXT / VARCHAR types

---

## PSEUDONYMIZATION

```
SECURITY LABEL FOR anon
  ON COLUMN users.city
  IS 'MASKED WITH FUNCTION anon.pseudo_city(users.email) ';
```

---

## PSEUDONYMIZATION

This is an IMMUTABLE transformation:

```
 SELECT anon.pseudo_city('bob@gmail.com');
 pseudo_city
-------------
 Moriki

SELECT anon.pseudo_city('bob@gmail.com');
 pseudo_city
-------------
 Moriki
```

---

## PSEUDONYMIZATION

- Useful for Foreing Keys and UNIQUE columns
- You can build an index on pseudonymized columns
- Pseudonymized Data are still covered by GDPR !

---

## HASHING

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

---

## HASHING

```
SELECT anon.hash('bob@gmail');
                  hash
-------------------------------------------------------------------
95b6accef02c5a725a8c9abf19ab5575f99ca3d9997984181e4b3f81d96cbca4d0
```

---

## GENERALIZATION

```
SELECT * FROM patient;
     ssn      | firstname |  zip  |   birth    |  disease
--------------+-----------+-------+------------+--------------
 253-51-6170 | Alice     | 47012 | 1989-12-29 | Flu
 091-20-0543 | Bob       | 42678 | 1979-03-22 | Allergy
 565-94-1926 | Caroline  | 42678 | 1971-07-22 | Flu
 510-56-7882 | Eleanor   | 47909 | 1989-12-15 | Acne
```

---

## GENERALIZATION

```
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;
```

---

## GENERALIZATION

```
SELECT * FROM generalized_patient;
 firstname |      zip      |          birth          |  disease
-----------+---------------+-------------------------+--------------
 REDACTED  | [47000,48000) | [1980-01-01,1990-01-01) | Flu
 REDACTED  | [42000,43000) | [1970-01-01,1980-01-01) | Allergy
 REDACTED  | [42000,43000) | [1970-01-01,1980-01-01) | Flu
 REDACTED  | [47000,48000) | [1980-01-01,1990-01-01) | Acne
```

---

## GENERALIZATION

- The data remains **true** but less precise
- Ideal for data science, reporting and analytics (RANGE types)
- The degree of Anonymization can be measured with the k-anonymity function
- Dynamic masking won't work (because the data model has changed)
- Can't be used in CI

---

## WRITE YOUR OWN MASKS !

- Use your own set of fake data
- write simple SQL functions, easy to test and maintain
- Useful for JSON columns

---

## IN A NUTSHELL

- Write your masking rules inside the database
- Different strategies for different use cases
- Combine with other tools (pg_sample, pg_audit, etc.)

---

## THANKS !


- Contact : damien.clochard@dalibo.com
- Follow : @daamien
- Other Projects : Dalibo Labs