PostgreSQL Anonymizer Documentation

Release 0.6.1

Henning Kage

Jul 13, 2021

Contents

Contents:

CHAPTER 1

1 PostgreSQL Anonymizer

A commandline tool to anonymize PostgreSQL databases for DSGVO/GDPR purposes.

It uses a YAML file to define which tables and fields should be anonymized and provides various methods of anonymization. The tool requires a direct PostgreSQL connection to perform the anonymization.

no-web no-pdf

Contents

1.1 1.1 Features

- Intentionally compatible with Python 2.7 (for old, productive platforms)
- Anonymize PostgreSQL tables on data level entry with various methods (s. table below)
- Exclude data for anonymization depending on regular expressions
- Truncate entire tables for unwanted data

Field	Value	Provider	Output
`first_name`	John	`choice`	(Bob|Larry|Lisa)
`title`	Dr.	`clear`	
`street`	Irving St	`faker.street_name`	Miller Station
`password`	dsf82hFxcM	`mask`	XXXXXXXXXX
`email`	jane.doe@example.com	`md5`	0cba00ca3da1b283a57287bcceb17e35
`email`	jane.doe@example.com	`faker.unique.email`	alex7@sample.com
`phone_num`	65923473	`md5` as_number: True	3948293448
`ip`	157.50.1.20	`set`	127.0.0.1
`uuid_col`	00010203-0405-.....	`uuid4`	f7c1bd87-4d....

- Note: `faker.unique.[provider]` only supported on Python 3.5+ (Faker library min. supported python version)
- Note: `uuid4` - only for (native uuid4) columns

See the documentation for a more detailed description of the provided anonymization methods.

1.2 1.2 Installation

The default installation method is to use `pip`:

```
$ pip install pganonymize
```

1.3 1.3 Usage

```
usage: pganonymize [-h] [-v] [-l] [--schema SCHEMA] [--dbname DBNAME]
                   [--user USER] [--password PASSWORD] [--host HOST]
                   [--port PORT] [--dry-run] [--dump-file DUMP_FILE]

Anonymize data of a PostgreSQL database

optional arguments:
-h, --help            show this help message and exit
-v, --verbose         Increase verbosity
-l, --list-providers  Show a list of all available providers
--schema SCHEMA       A YAML schema file that contains the anonymization
                        rules
--dbname DBNAME       Name of the database
--user USER           Name of the database user
--password PASSWORD   Password for the database user
--host HOST           Database hostname
--port PORT           Port of the database
--dry-run             Don't commit changes made on the database
--dump-file DUMP_FILE
                        Create a database dump file with the given name
--init-sql INIT_SQL   SQL to run before starting anonymization
```

Despite the database connection values, you will have to define a YAML schema file, that includes all anonymization rules for that database. Take a look at the schema documentation or the YAML sample schema.

Example call:

```
$ pganonymize --schema=myschema.yml \
    --dbname=test_database \
    --user=username \
    --password=mysecret \
    --host=db.host.example.com \
    -v

$ pganonymize --schema=myschema.yml \
    --dbname=test_database \
    --user=username \
    --password=mysecret \
    --host=db.host.example.com \
    --init-sql "set search_path to non_public_search_path; set work_mem to '1GB';" \
    -v
```

1.3.1 1.3.1 Database dump

With the `--dump-file` argument it is possible to create a dump file after anonymizing the database. Please note, that the `pg_dump` command from the `postgresql-client-common` library is necessary to create the dump file for the database, e.g. under Linux:

```
sudo apt-get install postgresql-client-common
```

Example call:

```
$ pganonymize --schema=myschema.yml \
    --dbname=test_database \
    --user=username \
    --password=mysecret \
    --host=db.host.example.com \
    --dump-file=/tmp/dump.gz \
    -v
```

1.4 1.4 Quickstart

Clone repo:

```
$ git clone git@github.com:rheinwerk-verlag/postgresql-anonymizer.git
$ cd postgresql-anonymizer
```

For making changes and developing pganonymizer, you need to install `poetry`:

```
$ sudo pip install poetry
```

Now you can install all requirements and activate the virtualenv:

```
$ poetry install
$ poetry shell
```

1.5 1.5 Docker

If you want to run the anonymizer within a Docker container you first have to build the image:

```
$ docker build -t pganonymizer .
```

After that you can pass a schema file to the container, using Docker volumes, and call the anonymizer:

```
$ docker run \
    -v <path to your schema>:/schema.yml \
    -it pganonymizer \
    /usr/local/bin/pganonymize \
    --schema=/schema.yml \
    --dbname=<database> \
    --user=<user> \
    --password=<password> \
    --host=<host> \
    -v
```

CHAPTER 2

Schema

`pganonymize` uses a YAML based schema definition for the anonymization rules.

2.1 Top level

2.1.1 `tables`

On the top level a list of tables can be defined with the `tables` keyword. This will define which tables should be anonymized.

Example:

```
tables:
 - table_a:
    fields:
     - field_a: ...
     - field_b: ...
 - table_b:
    fields:
     - field_a: ...
     - field_b: ...
```

2.1.2 `truncate`

You can also specify a list of tables that should be cleared instead of anonymized with the *truncated* key. This is useful if you don't need the table data for development purposes or the reduce the size of the database dump.

Example:

```
truncate:
 - django_session
 - my_other_table
```

If two tables have a foreign key relation and you don't need to keep one of the table's data, just add the second table and they will be truncated at once, without causing a constraint error.

2.2 Table level

2.2.1 `primary_key`

Defines the name of the primary key field for the current table. The default is `id`.

Example:

```
tables:
 - my_table:
    primary_key: my_primary_key
    fields: ...
```

2.2.2 `fields`

Starting with the keyword `fields` you can specify all fields of a table, that should be available for the anonymization process. Each field entry has its own `provider` that defines how the field should be treated.

Example:

```
tables:
 - auth_user:
    fields:
     - first_name:
        provider:
          name: clear
 - customer_email:
    fields:
     - email:
        provider:
          name: md5
        append: @localhost
```

2.2.3 `excludes`

For each table you can also specify a list of `excludes`. Each entry has to be a field name which contains a list of exclude patterns. If one of these patterns matches, the whole table row won't be anonymized.

Example:

```
tables:
 - auth_user:
    primary_key: id
    fields:
     - first_name:
        provider:
          name: clear
    excludes:
     - email:
       - "\\S[^@]*@example\\.com"
```

This will exclude all data from the table `auth_user` that have an `email` field which matches the regular expression pattern (the backslash is to escape the string for YAML).

2.2.4 search

You can also specify a (SQL WHERE) *search_condition*, to filter the table for rows to be anonymized. This is useful if you need to anonymize one or more specific records, eg for "Right to be forgotten" (GDPR etc) purpose.

Example:

```
tables:
 - auth_user:
    search: id BETWEEN 18 AND 140 AND user_type = 'customer'
    fields:
     - first_name:
        provider:
          name: clear
```

2.2.5 chunk_size

Defines how many data rows should be fetched for each iteration of anonymizing the current table. The default is 2000.

Example:

```
tables:
 - auth_user:
    chunk_size: 5000
    fields: ...
```

2.3 Field level

2.3.1 provider

Providers are the tools, which means functions, used to alter the data within the database. You can specify on field level which provider should be used to alter the specific field. The reference a provider you will have can use the `name` attribute.

Example:

```
tables:
 - auth_user:
    fields:
     - first_name:
        provider:
          name: set
          value: "Foo"
```

For a complete list of providers see the next section.

2.3.2 append

This argument will append a value at the end of the altered value:

Example usage:

```
tables:
 - auth_user:
    fields:
     - email:
        provider:
          name: md5
        append: "@example.com"
```

2.4 Provider

2.4.1 choice

This provider will define a list of possible values for a database field and will randomly make a choice from this list.

Arguments:

- `values`: All list of values

Example usage:

```
tables:
 - auth_user:
    fields:
     - first_name:
        provider:
          name: choice
          values:
            - "John"
            - "Lisa"
            - "Tom"
```

2.4.2 clear

Arguments: none

The `clear` provider will set a database field to `null`.

Note: But remember, that you can set fields to `null` only if the database field allows null values.

Example usage:

```
tables:
 - auth_user:
    fields:
     - first_name:
        provider:
          name: clear
```

2.4.3 fake

Arguments: none

pganonymize supports all providers from the Python library Faker. All you have to do is prefix the provider with fake and then use the function name from the Faker library, e.g:

- fake.first_name
- fake.street_name

Note: Please note: using the Faker library will generate randomly generated data for each data row within a table. This will dramatically slow down the anonymization process.

Example usage:

```
tables:
 - auth_user:
    fields:
     - email:
        provider:
          name: fake.email
```

See the Faker documentation for a full set of providers.

2.4.4 mask

Arguments:

- sign: The sign to be used to replace the original characters (default X).

This provider will replace each character with a static sign.

Example usage:

```
tables:
 - auth_user:
    fields:
     - last_name:
        provider:
          name: mask
          sign: '?'
```

2.4.5 md5

Arguments:

- as_number (default False): Return the MD5 hash as an integer.
- as_number_length (default 8): The length of the integer representation.

This provider will hash the given field value with the MD5 algorithm.

Example usage:

```
tables:
 - auth_user:
    fields:
     - password:
        provider:
          name: md5
          as_number: True
```

2.4.6 set

Arguments:

- `value`: The value to set

Example usage:

```
tables:
 - auth_user:
    fields:
     - first_name:
        provider:
          name: set
          value: "Foo"
```

The value can also be a dictionary for JSONB columns:

```
tables:
 - auth_user:
    fields:
     - first_name:
        provider:
          name: set
          value: '{"foo": "bar", "baz": 1}'
```

2.4.7 uuid4

Arguments: none

This provider will replace values with a unique UUID4.

Note: The provider will only generate native UUIDs. If you want to use UUIDs for character based columns, use `fake.uuid4` instead.

Example usage:

```
tables:
 - auth_user:
    fields:
     - first_name:
        provider:
          name: uuid4
```

CHAPTER 3

API

3.1 pganonymizer package

3.1.1 Submodules

3.1.2 pganonymizer.cli module

3.1.3 pganonymizer.constants module

3.1.4 pganonymizer.exceptions module

exception pganonymizer.exceptions.**BadDataFormat**
Bases: *pganonymizer.exceptions.PgAnonymizeException*

Raised if the anonymized data cannot be copied.

exception pganonymizer.exceptions.**InvalidFieldProvider**
Bases: *pganonymizer.exceptions.PgAnonymizeException*

Raised if an unknown field provider was used in the schema.

exception pganonymizer.exceptions.**InvalidProvider**
Bases: *pganonymizer.exceptions.PgAnonymizeException*

Raised if an unknown provider class was requested.

exception pganonymizer.exceptions.**InvalidProviderArgument**
Bases: *pganonymizer.exceptions.PgAnonymizeException*

Raised if an argument is unknown or invalid for a provider.

exception pganonymizer.exceptions.**PgAnonymizeException**
Bases: Exception

Base exception for all pganonymize errors.

3.1.5 pganonymizer.providers module

3.1.6 pganonymizer.utils module

3.1.7 pganonymizer.version module

3.1.8 Module contents

CHAPTER 4

Tests

For testing you have to install tox, either system-wide via your distribution's package manager, e.g. on debian/Ubuntu with:

```
$ sudo apt-get install python-tox
```

or via pip:

```
$ sudo pip install tox
```

Run the tests via tox for all Python versions configured in `tox.ini`:

```
$ tox
```

If you want to test only against your default Python version, just run:

```
$ make test
```

or activate the virtualenv and run:

```
$ poetry shell
$ pytest -v
```

To see all available make target just run `make` without arguments.

CHAPTER 5

Documentation

Package documentation is generated by Sphinx and uploaded to readthedocs.io. To build the documentation manually just call:

```
$ make docs
```

After a successful build the documentation index is opened in your web browser. You can override the command to open the browser (default `xdg-open`) with the `BROWSER` make variable, e.g.:

```
$ make BROWSER=chromium-browser docs
```

CHAPTER 6

Deploy

To create a new release you will have to install `twine` first:

```
$ pip install twine
```

Then you have to create a new distribution file:

```
$ make dist
```

Finally you can upload the file to PyPi:

```
$ twine upload dist/*
```

CHAPTER 7

License

CHAPTER 8

Changelog

8.1 Development

8.2 0.6.1 (2021-07-13)

- Added missing dependencies for the `setup.py`

8.3 0.6.0 (2021-07-13)

- #28: Add json support (nurikk)
- #27: Better anonymisation (nurikk)
- #25: Remove column specification for `cursor.copy_from` call (nurikk)

8.4 0.5.0 (2021-06-30)

- #22: Fix table and column name quotes in `cursor.copy_from` call (nurikk)
- #23: Allow uniq faker (nurikk)

8.5 0.4.1 (2021-05-27)

- #19: Make chunk size in the table definition dynamic (halilkaya)

8.6 0.4.0 (2021-05-05)

- #18: Specify (SQL WHERE) search_condition, to filter the table for rows to be anonymized (bobslee)
- #17: Fix anonymizing error if there is a JSONB column in a table (koptelovav)

8.7 0.3.3 (2021-04-16)

- #16: Preserve column and table cases during the copy process

8.8 0.3.2 (2021-01-25)

- #15: Fix for exclude bug (abhinavvaidya90)

8.9 0.3.1 (2020-12-04)

- #13: Fixed a syntax error if no truncated tables are defined (ray-man)

8.10 0.3.0 (2020-02-11)

- Use python-poetry for requirements management
- Added commandline argument to list all available providers (#4)
- Added commandline argument to create a dump file (#5)
- Execute table truncation in one statement to avoid foreign key constraint errors (thanks to W1ldPo1nter)

8.11 0.2.4 (2020-01-03)

- Fixed several issues with the usage of `dict.keys` and Python 3

8.12 0.2.3 (2020-01-02)

- Fixed the wrong cStringIO import for Python 3
- Removed Travis-CI file in favor of the Github actions

8.13 0.2.2 (2020-01-02)

- Hide the progressbar completely if verbose is set to `False`
- Restructured the requirement files and added flake8 to Travis CI

8.14 0.2.1 (2019-12-20)

- Added field based, regular expression excludes (to skip data under certain conditions). Currently only regular expressions are supported and the exclusion affects the whole row, not just one single column.

8.15 0.2.0 (2019-12-20)

- Added provider classes
- Added new providers:
  - choice - returns a random list element
  - mask - replaces the original value with a static sign

8.16 0.1.1 (2019-12-18)

Changed setup.py

8.17 0.1.0 (2019-12-16)

Initial release of the prototype

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

B

I

P