pganonymize Documentation

Release 0.11.0

Henning Kage

Mar 18, 2024

CONTENTS

Contents:

CHAPTER

ONE

PGANONYMIZE

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 Features

- Intentionally compatible with Python 2.7 (for old, productive platforms)
- Anonymize PostgreSQL tables on data level entry with various providers (some examples in the table below)
- Exclude data for anonymization depending on regular expressions or SQL `WHERE` clauses
- 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
credit_card	1234-567-890	partial_mask	1??????????0
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 3.6+ (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 Installation

The default installation method is to use pip:

```
$ pip install pganonymize
```

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 calls:

```
$ 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 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
```

1.3.2 Docker

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

```
$ docker build -t pganonymize .
```

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 pganonymize \
    /usr/local/bin/pganonymize \
    --schema=/schema.yml \
    --dbname=<database> \
    --user=<user> \
    --password=<password> \
```

(continues on next page)

(continued from previous page)

```
    --host=<host> \
    -v
```

CHAPTER

TWO

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 `truncate` keyword. This is useful if you don't need the actual table data (e.g. for a local development) or to 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:
 - user:
    primary_key: user_id
    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

You can also specify a list of `excludes` for each table. Each entry has to be a field name which contains a list of exclude patterns. If one of these patterns matches, the record 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 records 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, e.g. 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
```

YAML schema file supports placeholders with environment variables, e.g.:

```
!ENV ${HOST}
!ENV '/var/${LOG_PATH}'
```

So you can construct dynamic filter conditions like:

```
$ export COMPANY_ID=123
$ export ACTION_TO_BE_TAKEN=clear
$ pganonymize
```

Example:

```
- login:
    search: id = '!ENV ${COMPANY_ID}'
    search2: id = ${COMPANY_ID}
    search3: username = '${USER_TO_BE_SEARCHED}'
    fields:
     - first_name:
        provider:
          name: ${ACTION_TO_BE_TAKEN}
```

2.2.5 chunk_size

Defines how many records 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. To reference a provider you will have to 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
```

(continues on next page)

(continued from previous page)

```
          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 to prefix the provider with `fake` and then use the function name from the Faker library, e.g:

- `fake.first_name`
- `fake.street_name`

Some fake functions allow additional parameters to be passed, these can be specified in the schema as `kwargs`.

For localization options see *Localization*.

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
     - birth_date:
        provider:
          name: fake.date_of_birth
```

(continues on next page)

(continued from previous page)

```
          kwargs:
            minimum_age: 18
```

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 partial_mask

Arguments:

- `sign`: The sign to be used to replace the original characters (default `X`).
- `unmasked_left`: The number of characters on the left side to leave unmasked (default 1).
- `unmasked_right`: The number of characters on the right side to leave unmasked (default 1).

This provider will replace some characters with a static sign. It will leave some characters on the left and right unmasked, you can determine how many by providing `unmasked_left` and `unmasked_right` arguments.

Example usage:

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

2.4.6 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
```

2.4.7 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}'
```

2.4.8 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
```

2.4.9 update_json

Arguments:

- `update_values_type`

This provider will replace json and jsonb data values with a specified provider configuration per data type.

Example usage:

```
tables:
  - payment_transaction:
      fields:
        - data:
            provider:
              name: update_json
              update_values_type:
                str:
                  provider:
                    name: uuid4
                int:
                  provider:
                    name: fake.pyint
                float:
                  provider:
                    name: fake.pyfloat
```

CHAPTER

THREE

LOCALIZATION

It's possible to use the localization feature of `Faker` to localize the generated data.

To localize the data, add the locales to use as a global option to the YAML schema:

```
tables:
  auth_user:
    fields:
      - name:
        provider:
          name: fake.name
      - street:
        provider:
          name: fake.street_address
      - city:
        provider:
          name: fake.city

options:
  faker:
    locales:
      - de_DE
      - en_US
```

Now any field using the `Faker` provider will generate localized data. When multiple locales are configured, `Faker` will use its Multiple Locale Mode. In the example above, `Faker` selects the locale randomly for each field and row.

It's also possible to define the locale to use on field level and to define a default locale:

```
tables:
  - user:
      primary_key: id
      fields:
        - name:
            provider:
              # No locale entry at all, use configured default_locale "de_DE"
              name: fake.name
        - city:
            provider:
              # Use "en_US"
              name: fake.city
              locale: en_US
        - street:
```

(continues on next page)

(continued from previous page)

```
            provider:
              # Use "cs_CZ"
              name: fake.street_address
              locale: cs_CZ
        - zipcode:
            provider:
              # Use empty locale to ignore default_locale and to randomly select locale
              name: fake.postcode
              locale:

options:
  faker:
    locales:
      - de_DE
      - en_US
      - cs_CZ
    default_locale: de_DE
```

Attention: Make sure that the `Faker` provider (e.g. `street_name`) is supported by the Localized Provider.

CHAPTER

FOUR

API

4.1 pganonymize package

4.1.1 Submodules

4.1.2 pganonymize.cli module

4.1.3 pganonymize.constants module

4.1.4 pganonymize.exceptions module

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

Raised if the anonymized data cannot be copied.

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

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

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

Raised if an unknown provider class was requested.

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

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

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

Base exception for all pganonymize errors.

exception pganonymize.exceptions.**ProviderAlreadyRegistered**
Bases: *pganonymize.exceptions.PgAnonymizeException*

Raised if another provider with the same id has already been registered.

4.1.5 pganonymize.providers module

4.1.6 pganonymize.utils module

4.1.7 pganonymize.version module

4.1.8 Module contents

CHAPTER

FIVE

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
```

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

CHAPTER

SIX

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

SEVEN

DEPLOY

A new release (PyPi package) will be created automatically if a tag was created using Github actions. If the release has to be uploaded manually, 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

EIGHT

LICENSE

CHAPTER

NINE

CHANGELOG

9.1 Development

- #56: Add Trusted Publisher Management workflow

9.2 0.11.0 (2024-02-29)

- #52: Add update_json provider (bobslee)

9.3 0.10.0 (2022-11-29)

- #49: Configure psycopg2 to support UUID objects
- #48: Add support for localized “Faker” data

9.4 0.9.0 (2022-11-23)

- #46: Broken Python 2.7 compatibility
- #45: Add partial masked provider (Tilley)
- #44: Pass kwargs through to faker functions from schema (Tilley)

9.5 0.8.0 (2022-03-15)

- #39: Renamed project to “pganonymize”
- #38: Allow environment variables in schema definition (nurikk)

9.6 0.7.0 (2021-11-30)

- #34: Subprocess "run" being used on Python2.7
- #35: parmap no longer supports Python 2.7 * Dropped Python 3.5 support * Pinned libraries Python 2.7
- #32: Fixed pg_dump arguments (korsar182)
- Simplified provider registration (no metaclass usage anymore)

9.7 0.6.1 (2021-07-13)

- Added missing dependencies for the *setup.py*

9.8 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)

9.9 0.5.0 (2021-06-30)

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

9.10 0.4.1 (2021-05-27)

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

9.11 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)

9.12 0.3.3 (2021-04-16)

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

9.13 0.3.2 (2021-01-25)

- #15: Fix for exclude bug (abhinavvaidya90)

9.14 0.3.1 (2020-12-04)

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

9.15 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)

9.16 0.2.4 (2020-01-03)

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

9.17 0.2.3 (2020-01-02)

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

9.18 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

9.19 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.

9.20 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

9.21 0.1.1 (2019-12-18)

Changed setup.py

9.22 0.1.0 (2019-12-16)

Initial release of the prototype

CHAPTER

TEN

LINKS

The following links refer to projects that have a similar purpose of anonymizing a PostgreSQL database. Thanks to the authors of these projects. Some of them inspired the author of this project, e.g. pgantomizer for using a human readable declaration file in YAML.

- PostgreSQL Anonymizer: Anonymization & Data Masking for PostgreSQL
- pg-anonymizer: Dump anonymized PostgreSQL database with a NodeJS CLI
- pgantomizer: Anonymize data in your PostgreSQL dabatase with ease

CHAPTER

ELEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX