Practice Session N° 2: Databases and Data Preparation

This is the initial version of the document, shared in beginning of the lab session

In this tutorial 1st you will learn:

- How to setup the database server
- How to manage the database using admin-client
- How to manage the database using command line interface
- How to store/access the data using command line interface
- How the create a simple database schema with tables and relations
- How to populate the database with the data
- How to setup python database client
- How to store/access the data using python database client

In 2nd half ot the lab we will deal with preprocessing of the mobility data

1st Part

PostgreSQL

The PostgreSQL is an advanced open source relational database management system (RDBMS) originatng from Berkeley, it supports classics DB transactions following ACID properties as well as many features seen in commercial databases (triggers, views, indexes etc.). It supports most popular operations systems and can be used standalone as well as in cluster deployment. Multiple well known open source projects rely on PostgreSQL for managing the data, for us most notably: OpenStreetMap, PostGIS, CartoDB, GRASS GIS, QGIS, Mapnik. The current upstream of Postgre is 13.2 which we are going to demonstrate in this tutorial.

Installing in general

The PostgreSQL documentation is well organized and provides detailed instructions along with command line (CLI) examples. The
corresponding downloads section contains pre-compiled packages as well as the source code archives.

https://www.postgresql.org/download/ (https://www.postgresql.org/download/)

Installing on Linux

Most of the modern Linux distributions include software repositories simplifying the installation of specific packages. PostgreSQL is included along with its dependencies in APT based (Debian, Ubuntu, Mintu); YUM based (Fedora, RedHat, CentOS); YaST based SUSE Linux. One notable disadvantage of repository based distributions - the repo may not contain the latest version of the software. The up-to-date version of the software in this case is installed through custom repository (if exists); or manually compiled. In the following sections we list the install routines for most popular Linux distributions.

Installing on Debian/Ubuntu/Mintu Linux

```
In [46]:

%bash
# Figure out the required distro, generate apt-sources repository line:
echo "deb http://apt.postgresql.org/pub/repos/apt $(lsb_release -cs)-pgdg main"
echo "deb http://apt.postgresql.org/pub/pgadmin/pgadmin4/apt/$(lsb_release -cs) pgadmin4 main"

deb http://apt.postgresql.org/pub/repos/apt (http://apt.postgresql.org/pub/repos/apt) buster-pgdg main

Add the generated lines into repositories list */etc/apt/sources.list*

Alternatively, allow script to modify the repositories list:

```
In []:

%bash
# Generate the apt-sources line, add it to the repositories lists:
echo "deb http://apt.postgresql.org/pub/repos/apt $(lsb_release -cs)-pgdg main" > /tmp/tmp_apt.list
echo "deb http://apt.postgresql.org/pub/pgadmin/pgadmin4/apt/$(lsb_release -cs) pgadmin4 main" >> /tmp/tmp_apt.list
sudo bash -c 'cat /tmp/tmp_apt.list > /etc/apt/sources.list.d/pgdg.list'

Next, fetch the PostgreSQL repository PGP public keys, and add it into *apt* keys

```
Finally install the packages *postgresql* and *pgadmin4*.
In [ ]:

```bash
# Install PostgreSQL server and pgAdmin
sudo apt install postgresql pgadmin4
```

**Task 001:** install the PostgreSQL related packages specific to your platform

---

**PgAdmin graphical administration utility**

The PostgreSQL comes with nice administration GUI (desktop or web-based application). The PgAdmin includes an educational sandbox setup, which allows to get the basic knowledge without damaging the local PostgreSQL server setup.

[https://www.pgadmin.org/try/](https://www.pgadmin.org/try/)

**Optional Task:** Follow the online demo, make yourself familiar to PgAdmin

**Question:** How many tables are there in the *pagila* schema?

---

**Managing PostgreSQL server using command line**

*Checking status of the PostgreSQL database server*

In [51]:

```bash
# In Linux or MacOS command line
pg_lsclusters
```

```
13 main 5432 online postgres /var/lib/postgresql/13/main /var/log/postgresql/postgresql-13-main.log
```

In [52]:

```bash
# In Windows
# @TODO: check if 'pg_lsclusters' is available in command line (cmd)
```
Starting/Stopping/Reloading the server daemon process

In [ ]: # In command line:
# Assuming we have cluster 'main' of version 13 (check with pg_lsclusters)

# To check the status
pg_ctlcluster 13 main status

# To stop
pg_ctlcluster 13 main stop
# To start
pg_ctlcluster 13 main start
# To restart
pg_ctlcluster 13 main restart

Changing the forgotten password for PostgreSQL root user

It is often a case when a password of the root PostgreSQL user is forgotten and needs to be reset. Here are some hints how to reset it.

In [ ]: # In Linux the PostgreSQL database server process is running in separate user space (postgres)
# The corresponding postgres user has access to the server with no password
#
# Switch to super user first
# by providin the system root password
su -
# or through 'sudo' providing your local user password
sudo su -

# Once switched to root user, you can switch next to 'postgres' user (there will be no need for it's su - postgres

# Finally you can access PostgreSQL shell and modify anything you want in the 'postgres' schema
psql
In [ ]: # You will notice the PostgreSQL shell appeared in place of standard bash
postgres=#
# the '#' here stands for root space (root has all permissions)
# Next we can change the password of the root user (it will ask for a new password twice)
\password postgres
# Once password is changed you can exit the PostgreSQL shell
\q

In [ ]: # Once the password is changed you can access the administrative account (root) of the PostgreSQL from
# knowing the password
#
# * localhost user using command line
psql -h localhost -W postgres postgres

# * remote user using command line (assuming <ip> is PostgreSQL server's IP address)
psql -h <ip> -W postgres postgres

Finally you can access the administrative account through pgAdmin:
Open the pgAdmin GUI, then add new database server connection File->Object->Create->Server. Next fill in the following settings:

- **Name**: PostgreSQL Admin
- **Host**: localhost
- **Port**: 5432
- **Service**: 
  - **Maintainance DB**: postgres
  - **Username**: postgres

Optionally you can store password

Once added the new connection, double click it and establish the connection, it then provide the overview:
Creating the user role and associating to a database

Once we acquired the admin account on the database server we can create schemas and roles for our needs. Pay attention with local PostgreSQL setup it is always possible to have the admin access to database server. In case of remote database server setup, the admin access account access is not always the case.

Next we will show how to create new roles and databases using admin account. We will create a sandbox database first, and next we will create a user tester with password 12345. Finally we will associate the user tester to database sandbox.

```
In [ ]:
# In command line first login into administrative account of PostgreSQL:
psql -h localhost -W postgres postgres
# Make sure you can see the root shell of PostgreSQL:
postgres=#
```

Refer to database creation documentation:

https://www.postgresql.org/docs/current/sql-createdatabase.html

```
In [ ]:
# Create the 'sandbox' database as follows
CREATE DATABASE sandbox;
```
Next we create the role tester, the corresponding documentation:

https://www.postgresql.org/docs/13/sql-createrole.html

In [ ]:

```
# Create the user 'tester' with password '12345'
CREATE USER tester WITH PASSWORD '12345';
```

Alternate the database sandbox, grant all permissions on database sandbox to user tester:

https://www.postgresql.org/docs/current/sql-grant.html

In [ ]:

```
GRANT ALL PRIVILEGES ON DATABASE sandbox to tester;
```

Accessing the created database using created role and password

Once user tester is granted with the privileges, we can log out out administrative user and access the sandbox database from user space

In [ ]:

```
# Log out of admin account of PostgreSQL
# type \q in command line
\q

# Login to PostgreSQL database 'sandbox' with user 'tester', this will ask for password which we set
psql -h localhost -W sandbox tester

# After successful login we will see the PostgreSQL shell again, inside the 'sandbox' database
# this time with '>' indicating user space:
# sandbox=>

# We can check what are the schemas and users available in database:
\dg  # <will list users>

\dn  # <will list schemas>

\?   # <will print the available commands>
```

Alternatively you can setup another connection in pgAdmin, using the following settings:
- Name: PostgreSQL Admin
- Host: localhost
- Port: 5432
- Service:
- Maintenance DB: postgres
- Username: postgres

The corresponding database appears as follows:
Managing the tables in the public schema

We have just created a separate database sandbox for user tester, next let's create some simple table structures. By default all the tables will be created in the public schema if not specified otherwise. We will follow a basic SQL tutorial of PostgreSQL:

https://www.postgresql.org/docs/13/tutorial-sql.html

Section 2.3 suggest the creation of the first table weather; here we can refer to psql command line shell or we can use pgAdmin GUI application. Whereas simple commands creating users, altering roles, granting permissions are OK to perform in command line; more complex and structured expressions are better to perform in a Query Tool of pgAdmin.

To open the Query Tool, first make sure you have connected to the sandbox database; next click on the sandbox database in the tree view in the left, next locate the Query Tool in the Menu (Tools->Query Tool)
Pay attention to used datatypes! Here is the list of all possible datatypes in PostgreSQL:

https://www.postgresql.org/docs/13/datatype.html

Alternatively you can store the SQL query into a file and execute it in command line:
In [16]: %bash
# In Linux, storing the content into a file:
cat << EOF > /tmp/weather_table.sql
CREATE TABLE weather (  
city varchar(80),  
temp_lo int, -- low temperature  
temp_hi int, -- high temperature  
prcp real, -- precipitation  
date date
);
EOF

In [ ]: %bash
# In Linux, executing SQL script from command line into 'sandbox' database as user 'tester'
psql -h localhost -W sandbox tester < /tmp/weather_table.sql

Now the table weather is created, you can see it in pgAdmin in database tree: sandbox -> schemas -> public -> tables
Populating the table with the data

We can populate the data using insert commands:

https://www.postgresql.org/docs/13/tutorial-populate.html

```python
In [ ]:

# Inserting values using default columns in default order
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');

# Inserting values specifying columns and order
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date) VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');

# Inserting values specifying columns and order, omitting unknown values
INSERT INTO weather (date, city, temp_hi, temp_lo) VALUES ('1994-11-29', 'Hayward', 54, 37);
```

Alternatively you can populate your tables from text files directly using COPY command:

https://www.postgresql.org/docs/13/sql-copy.html

Pay attention PostgreSQL has two versions of COPY command with same syntax:

- COPY - for server side usage
- \COPY - for client side usage

It is important to understand the concept of copying the data from a file to database table:

- COPY - assumes a file resides on server side (server file system)
- \COPY - assumes a file resides on client side (client file system)

In our case both client (pgAdmin, psql) and server are residing on the host (PC or Laptop) yet the communication runs through the loopback network (127.0.0.0/8). Therefore in our SQL scripts we must use \COPY command. An attempt of using the COPY will trigger a permission error.

Let's now try to use both of the commands, and let's take a look what will happen. We will use the weather dataset:

weather_data_raw.csv (weather_data_raw.csv)

NB! the data is not clean some (sensor values are unrealistic)!
Quering the data

Quering the data is done through SELECT command:

https://www.postgresql.org/docs/13/sql-select.html

In [72]:

```
# Indeed being on client side makes no sence to refer to use COPY, the hint suggests \COPY as well
# Let's load the data correct way!
psql -h localhost -U tester -f /tmp/load_weather_data_v1.csv sandbox
COPY 101368
```

In [88]:

```
# COPY 101368 - suggests we have successfully uploaded 101368 lines from CSV
```

And again we have multiple options to execute the query:

- writing an SQL script and executing it with psql
- provisioning the SQL expression directly to psql
- executing SQL query in pgAdmin
In [73]:

```bash
# Writing SQL script to /tmp/count-weather.sql
cat << EOF > /tmp/count-weather.sql
SELECT count(*) FROM weather;
EOF

# Executing the /tmp/count-weather.sql
psql -h localhost -U tester -f /tmp/count-weather.sql sandbox

count
--------
101368
(1 row)
```

In [74]:

```bash
# Provisioning psql client with explicit SQL statement:
psql -h localhost -U tester -c 'SELECT count(*) FROM weather;' sandbox

count
--------
101368
(1 row)
```

Executing SELECT query in pgAdmin:
More complex queries possible using aggregate functions:

https://www.postgresql.org/docs/13/functions-aggregate.html

In [ ]:

```sql
# An example of getting average values per-city
SELECT
city,
  avg(temp_lo) as avg_temp_lo,
  avg(temp_hi) as avg_temp_hi,
  avg(prcp)  as avg_prcp
FROM weather GROUP BY (city);
```
In [78]:

```bash
%%bash
# ... and executing it as psql fed script:
cat << EOF | psql -h localhost -U tester sandbox
SELECT
city,
avg(temp_lo) as avg_temp_lo,
avg(temp_hi) as avg_temp_hi,
avg(prcp) as avg_prcp
FROM weather GROUP BY (city);
EOF
```

<table>
<thead>
<tr>
<th>city</th>
<th>avg_temp_lo</th>
<th>avg_temp_hi</th>
<th>avg_prcp</th>
</tr>
</thead>
<tbody>
<tr>
<td>NARVA</td>
<td>1.3505014499758337</td>
<td>31.9614850169163847</td>
<td>17.29857419043016</td>
</tr>
<tr>
<td>TALLINN</td>
<td>0.86601210251542477456</td>
<td>31.6927503559563360</td>
<td>17.055054579971525</td>
</tr>
<tr>
<td>TARTU</td>
<td>-0.87230840472331558231</td>
<td>33.1177066450567261</td>
<td>15.573165084510304</td>
</tr>
</tbody>
</table>

(3 rows)

Even more complex examples are possible if we have relations. Let's create another table `cities` which would contain the coordinates of each city in our `weather` table

```sql
CREATE TABLE cities (
    name            varchar(80),
    location        point,
    altitude        real
);
```

We will populate the `cities` as follows:

```sql
INSERT INTO cities (name, location, altitude) VALUES ('TARTU', POINT(58.0000, 26.7331), 58.0);
INSERT INTO cities (name, location, altitude) VALUES ('TALLINN', POINT(59.3981, 24.6028), 33.0);
INSERT INTO cities (name, location, altitude) VALUES ('NARVA', POINT(59.3892, 28.1128), 28.0);
```

The data in `cities` table now looks as follows:
Next let's combine two tables joining on city name, let's modify our previous query and include altitude of the city and sort all the records by altitude value using ascending order:

In [80]:

```bash
# Executing simple select as an explicit psql command:
psql -h localhost -U tester -c 'select * from cities;' sandbox

<table>
<thead>
<tr>
<th>name</th>
<th>location</th>
<th>altitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>TARTU</td>
<td>(58.3,26.7331)</td>
<td>58</td>
</tr>
<tr>
<td>TALLINN</td>
<td>(59.3981,24.6028)</td>
<td>33</td>
</tr>
<tr>
<td>NARVA</td>
<td>(59.3892,28.1128)</td>
<td>28</td>
</tr>
</tbody>
</table>
(3 rows)
```

Next let's combine two tables joining on city name, let's modify our previous query and include altitude of the city and sort all the records by altitude value using ascending order:

In [ ]:

```sql
SELECT cities.name,                  -- taken from cities table
cities.altitude,                    --
avg(temp_lo) as avg_temp_lo,        -- taken from weather table
avg(temp_hi) as avg_temp_hi,        --
avg(prcp) as avg_prcp               --
FROM weather, cities                -- 2 tables involved
WHERE cities.name = weather.city    -- joining 2 tables
GROUP BY (cities.name, cities.altitude)  
ORDER BY cities.altitude ASC;       -- ordering ascending
```
Deleting data, dropping tables

Finally if some of the records are not needed anymore, DELETE FROM command can be used:

https://www.postgresql.org/docs/13/sql-delete.html

Let's find and remove some abnormal high temperature readings (remember our data is not clean, some sensors were faulty)

In [ ]:  

# Are there any records beyod 50.0 C in temp_hi  
select count(temp_hi)  
from weather  
where temp_hi > 50.0;
Finally if we want to clear the whole table we can use DELETE with no conditions:

```bash
# DELETE 39910 -- confirms the deletion was successful
```

And in case we want to delete also the table definition, we can refer to DROP TABLE command:

```
https://www.postgresql.org/docs/13/sql-droptable.html
```
In [ ]:

```sql
# Dropping definitions of both weather and cities tables
# In case tables are populated with the data - it will gone as well
DROP TABLE weather;
DROP TABLE cities;
```

Pay attention: we did not actually drop the table neither cleaned them completely, as we will need the data for the next sections

**Accessing the database using Python**

The `psql` command line client is a powerful tool, assuming it is operated in `bash` shell. Could we avoid using `shell` at all? and run SQL queries directly from Python?

The `psycopg2` module of Python adds missing cooperation between Python and PostgreSQL

[https://www.psycopg.org/](https://www.psycopg.org/)

According to *install* documentation:

[https://www.psycopg.org/install/](https://www.psycopg.org/install/)

All we need to do is:

In [90]:

```bash
pip install psycopg2
```

Collecting psycopg2
  Downloading psycopg2-2.8.6.tar.gz (383 kB)
Building wheels for collected packages: psycopg2
  Building wheel for psycopg2 (setup.py): started
  Building wheel for psycopg2 (setup.py): finished with status 'done'
  Created wheel for psycopg2: filename=psycopg2-2.8.6-cp37-cp37m-linux_x86_64.whl size=476560 sha256=f63da4edf3d0b95387ee04f1c912694a240752900986295cf2063f77000a1b7
  Stored in directory: /home/hp/.cache/pip/wheels/25/78/75/9c0323f7e1fb42143cbd2439302beb7850a1034a
  Successfully built psycopg2
Installing collected packages: psycopg2
Successfully installed psycopg2-2.8.6
Next let's try to follow the *psycopg2* tutorial with respect to our sandbox database

[https://www.psycopg.org/docs/usage.html](https://www.psycopg.org/docs/usage.html)

```python
# Importing the module
import psycopg2
psycopg2.__version__
```

```
'2.8.6 (dt dec pq3 ext lo64)'
```
In [95]:
# Check how to use the 'connect' method of 'psycopg2'
help(psycopg2.connect)

Help on function connect in module psycopg2:

    connect(dsn=None, connection_factory=None, cursor_factory=None, **kwargs)
    Create a new database connection.

    The connection parameters can be specified as a string:

        conn = psycopg2.connect("dbname=test user=postgres password=secret")

    or using a set of keyword arguments:

        conn = psycopg2.connect(database="test", user="postgres", password="secret")

    Or as a mix of both. The basic connection parameters are:

    - *dbname*: the database name
    - *database*: the database name (only as keyword argument)
    - *user*: user name used to authenticate
    - *password*: password used to authenticate
    - *host*: database host address (defaults to UNIX socket if not provided)
    - *port*: connection port number (defaults to 5432 if not provided)

    Using the *connection_factory* parameter a different class or connections
    factory can be specified. It should be a callable object taking a dsn
    argument.

    Using the *cursor_factory* parameter, a new default cursor factory will be
    used by cursor().

    Using *async*=True an asynchronous connection will be created. *async_* is
    a valid alias (for Python versions where `async` is a keyword).

    Any other keyword parameter will be passed to the underlying client
    library: the list of supported parameters depends on the library version.
In [98]: # Similar to 'pgAdmin' and 'psql', we need to provide some information in order to connect

    # Let's use keyword arguments for simplicity
    connect_kwargs = {
        'database': 'sandbox',
        'user': 'tester',
        'host': 'localhost',
        'port': 5433,
        'password': '12345'
    }

    # Simple connection wrapper using 'with' expression:
    # Here 'with' will ensure the connection will be closed in the end
    with psycopg2.connect(**connect_kwargs) as _conn:
        # Cursor will act as pipe, it has to be fed with query
        # ... and the results can be fetched
        # each pipe has to be closed in the end, hence 'with'
        with _conn.cursor() as _cur:
            _cur.execute('SELECT count(*) FROM cities;')
            # Fetch and print one line of results
            print(_cur.fetchone())

    (3,)

In [100]: # Let's try another one with more data

    # ... reusing the 'connect_kwargs'
    with psycopg2.connect(**connect_kwargs) as _conn:
        # Cursor will act as pipe, it has to be fed with query
        # ... and the results can be fetched
        # each pipe has to be closed in the end, hence 'with'
        with _conn.cursor() as _cur:
            _cur.execute('SELECT * FROM cities;')
            # Fetch and print all of the results
            for rec in _cur.fetchall():
                print(rec)

    ('TARTU', '(58.3,26.7331)', 58.0)
    ('TALLINN', '(59.3981,24.6028)', 33.0)
    ('NARVA', '(59.3892,28.1128)', 28.0)

Pay attention our type POINT() of PostgreSQL is lost, and simple Python string is shown instead!
In [104]: # Let's do the big one we had with grouping, averaging and sorting ...
# ... it was well complex, let's store it separately not to pollute the Python code:
sql_q0 = '''
SELECT
  cities.name,                         -- taken from cities table
  cities.altitude,                    --
  avg(temp_lo) as avg_temp_lo,        -- taken from weather table
  avg(temp_hi) as avg_temp_hi,        --
  avg(prcp) as avg_prcp               --
FROM weather, cities                  -- 2 tables involved
WHERE cities.name = weather.city      -- joining 2 tables
GROUP BY (cities.name, cities.altitude)
ORDER BY cities.altitude ASC;         -- ordering ascending
'''

with psycopg2.connect(**connect_kwargs) as _conn:
  # Cursor will act as pipe, it has to be fed with query
  # ... and the results can be fetched
  # each pipe has to be closed in the end, hence 'with'
  with _conn.cursor() as _cur:
    _cur.execute(sql_q0)
    # Show the column header as well
    print(tuple([col.name for col in _cur.description]))
    # Fetch and print all of the results
    for rec in _cur.fetchall():
      print(rec)

('name', 'altitude', 'avg_temp_lo', 'avg_temp_hi', 'avg_prcp')
('NARVA', 28.0, Decimal('-21.7374808063797117'), Decimal('-4.5782356728911784'), 13.901282876814106)
('TALLINN', 33.0, Decimal('-21.6438521814839896'), Decimal('-2.5311502370126729'), 14.26569604333946)
('TARTU', 58.0, Decimal('-23.7994658897790726'), Decimal('-5.9544549647972809'), 11.588395241563486)

Looks almost a Pandas dataframe, however these are just 4 python lists printed. Can we retrieve the database into Pandas dataframe?

**Reading the SQL query into Pandas dataframe**
Luckily Pandas is already aware of psycopg2 and all we need to do is to replace the iterating over a cursor and instead feed the opened connection and needed SQL script to Pandas.

```python
In [107]:
# Let's execute the last query once again through pandas
import pandas as pd

# We will reuse the connection properties once more:
# In addition we will reuse the previous SQL
with psycopg2.connect(**connect_kwargs) as _conn:
    # Dataframe once populated will be available in 'df'
    df = pd.read_sql_query(sql_q0, _conn)

def

Out[107]:

<table>
<thead>
<tr>
<th>name</th>
<th>altitude</th>
<th>avg_temp_lo</th>
<th>avg_temp_hi</th>
<th>avg_prcp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NARVA</td>
<td>28.0</td>
<td>-21.737481</td>
<td>-4.578236</td>
</tr>
<tr>
<td>1</td>
<td>TALLINN</td>
<td>33.0</td>
<td>-21.643852</td>
<td>-2.531150</td>
</tr>
<tr>
<td>2</td>
<td>TARTU</td>
<td>58.0</td>
<td>-23.799466</td>
<td>-5.954455</td>
</tr>
</tbody>
</table>

In [108]:
# What about simple ones?
with psycopg2.connect(**connect_kwargs) as _conn:
    # Dataframe once populated will be available in 'df'
    df = pd.read_sql_query('SELECT * FROM cities;', _conn)

def

Out[108]:

<table>
<thead>
<tr>
<th>name</th>
<th>location</th>
<th>altitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TARTU</td>
<td>58.0</td>
</tr>
<tr>
<td>1</td>
<td>TALLINN</td>
<td>33.0</td>
</tr>
<tr>
<td>2</td>
<td>NARVA</td>
<td>28.0</td>
</tr>
</tbody>
</table>
In [113]: # Pay attention Pandas still did not manage to properly convert our POINT() - strings are used again
df.location.values

Out[113]: array(['(58.3,26.7331)', '(59.3981,24.6028)', '(59.3892,28.1128)'],
dtype=object)

Exercises:

Refer to official documentation:

https://www.postgresql.org/docs/13/tutorial-sql.html

Task 002:

Using either CLI or PgAdmin create the required tables and relations in your local postgreSQL SQL

See the entity-relationship diagram below

![Entity-relationship diagram]

Task 003:

Fill in the created tables with the data
Accessing the database using Python

Task 004:
Install the psycopg2 module
Fetch the data from your previously created schema

Reading the SQL query into Pandas dataframe

Task 005:
Fetch the data from your previously created schema into a Pandas dataframe

Second Part

Preprocessing of the mobility data

We will refer to the and exiting tutorial of scikit-mobility You can copy paste commands directly to your Jupyter notebooks


Since we start with the *preprocessing* section, make sure the corresponding data is loaded first

In [ ]: