Load

NDBI046: Practical class 4

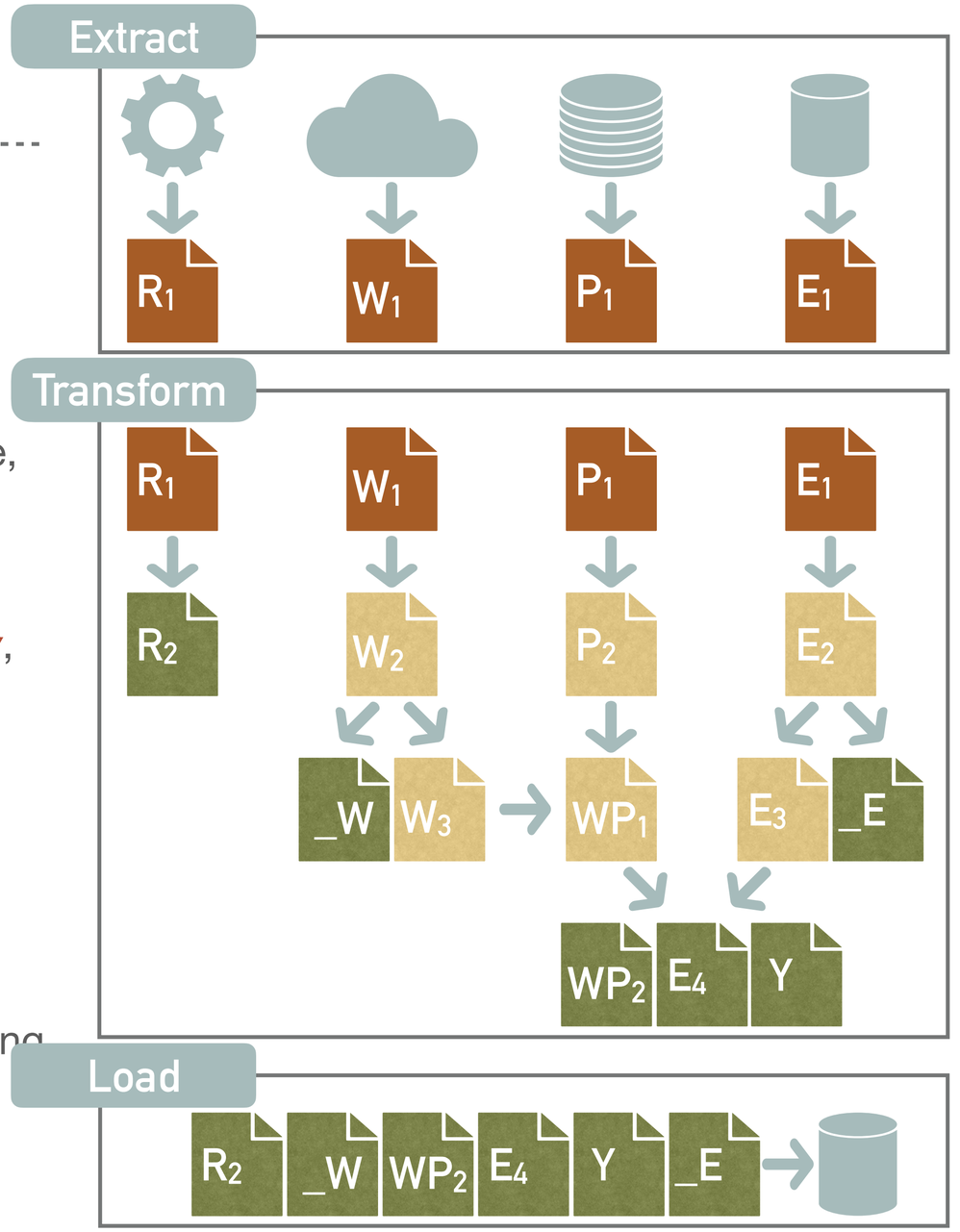
User Story

- The data analysts in our company asked us to prepare waste management datasets and load them to the data warehouse so that they could perform the required analyses
  - Specifically, they need datasets related to waste management in the Czech Republic at the level of individual regions and the extent of funds used to mitigate the environmental burden due to waste management
  - Their aim is to assess municipal and corporate waste production and address impacts, and they require that the amount of waste can be calculated per capita or per unit area
  - As a source of data, we are to use open data on waste from Czech open data portal, data published by the Czech Statistical Office and generally known facts can be extracted from Wikipedia

- **Data Engineer roles:**
  - *Extract datasets* from (various) sources
  - *Transform data* into a uniform form, detect and correct inconsistencies, etc.
  - *Load* the *data to* a *data warehouse* so that analysts can perform analyses

Extract Transform Load (ETL)

- **Data extraction** involves extracting data from homogeneous or heterogeneous sources
- **Data transformation** processes clean and transform data into a suitable format/structure for querying and analysis
- **Data loading** involves the insertion of data into, e.g., operational data store, data warehouse, data lake, or data mart
  - The *slowest part* of the ETL process
    - *Databases* may operate slowly as they *have to maintain concurrency*, *integrity* and *indexes*
  - To improve performance, following may be useful:
    - Performing all *data transformation outside the database*
    - *Disabling integrity constraint* checking (DROP CONSTRAINT) in the target database *during the load*
    - *Removing indexes* on a table or partition (DROP INDEX) before loading *and re-creating* them after the data is loaded (CREATE INDEX)
    - Using *parallel bulk loading* whenever possible



Prerequisite: Setting up Python (Linux, macOS)

- *Check* which *version of Python* is installed (if any)
  - If Python 3 is not installed, download the (latest) version of Python[#1] and follow the installation
- Once installed, *create* any *folder for your NDBI046 project* and navigate to it, e.g., ~/Projects/python-ndbi046
- *Create* your *Python environment*, e.g., ndbi046_env
- *Activate* you Python environment
- *Install* the required *packages*
  - Download the requirements.txt file from the practical class website
  - You may also *install additional packages*
  - You may always export the list of installed packages to a file
- Exit the Python environment after completing the practical class (not before)
  - You can return to the environment at any time by activating it

```
1  % python3 --version
2
3  % mkdir ~/Projects/python-ndbi046
4  % cd ~/Projects/python-ndbi046
5
6  % python3 -m venv ndbi046_env
7
8  % source ndbi046_env/bin/activate
9
10 (ndbi046_env) % pip install -r requirements.txt
11
12 (ndbi046_env) % pip install pandas
13
14 (ndbi046_env) % pip freeze > requirements.txt
15
16 (ndbi046_env) % deactivate
17
18 % cat requirements.txt
```

checking the installed version

creating and activating a virtual environment

installation of the required packages

export of installed packages

list installed packages

deactivating a virtual environment

#1 https://www.python.org/downloads/

Prerequisite: Setting up Python (Windows)

- *Check* which *version of Python* is installed (if any)
  - If Python 3 is not installed, download the (latest) version of Python[#1] and follow the installation

- Once installed, *create* any *folder for your NDBI046 project* and navigate to it, e.g., C:\Projects\python-ndbi046
- *Create* your *Python environment*, e.g., ndbi046_env
- *Activate* you Python environment

- *Install* the required *packages*
  - Download the requirements.txt file from the practical class website
  - You may also *install additional packages*
  - You may always export the list of installed packages to a file

- Exit the Python environment after completing the practical class (not before)
  - You can return to the environment at any time by activating it

checking the installed version

creating and activating a virtual environment

installation of the required packages

export of installed packages

list installed packages

deactivating a virtual environment

```
1  > python3 --version
2
3  > mkdir C:\Projects\python-ndbi046
4  > cd C:\Projects\python-ndbi046
5
6  > python3 -m venv ndbi046_env
7
8  > ndbi046_env\Scripts\activate.bat
9
10 (ndbi046_env) > pip install -r requirements.txt
11
12 (ndbi046_env) > pip install pandas
13
14 (ndbi046_env) > pip freeze > requirements.txt
15
16 (ndbi046_env) > deactivate
17
18 > type requirements.txt
```

#1 https://www.python.org/downloads/

Prerequisite: Setting up PostgreSQL and psycopg2 library

- Before you can use the psycopg2 library in Python, ensure that:
  - *Python 3.5 or newer is installed* on your system as psycopg2 is compatible only with Python 3.5 and above
  - psycopg2 is a PostgreSQL adapter for Python, hence *PostgreSQL must be installed and running* on your system
    - Download: https://www.postgresql.org/
    - Postgres.app: https://postgresapp.com/ (macOS)

- Once the prerequisites are in place, *install* psycopg2 as follows *in terminal or command prompt*:

make sure that appropriate virtual environment is activated

```
(ndbi046_env) % python3 --version
(ndbi046_env) % python3 -m pip install --upgrade pip
(ndbi046_env) % pip install psycopg2
```

upgrade pip and install psycopg2-binary

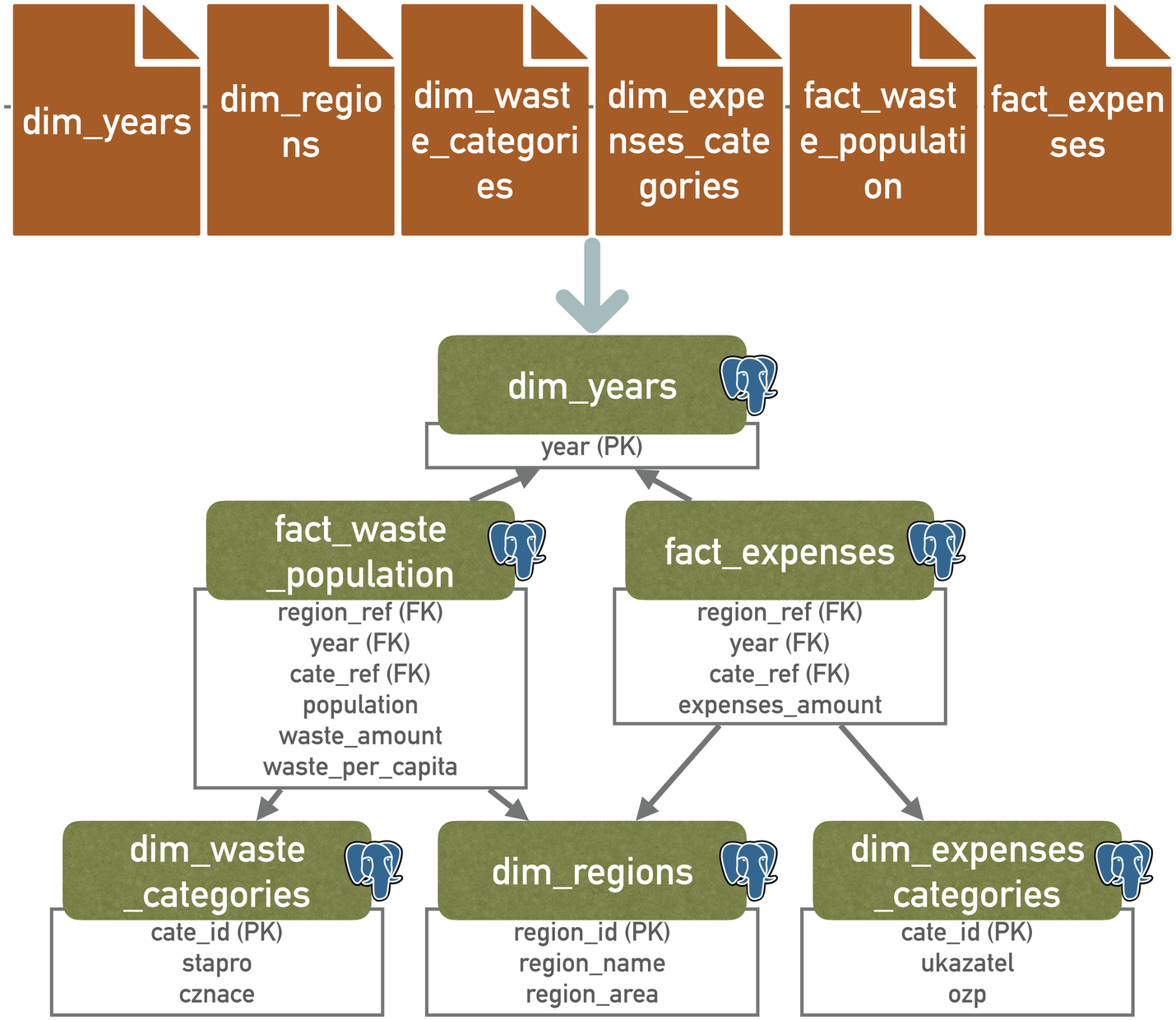
- If the installation fails due to a missing psycopg2-binary library, proceed as follows:

```
(ndbi046_env) % pip install psycopg2-binary
(ndbi046_env) % export export PATH=$PATH:/Applications/Postgres.app/Contents/Versions/15/bin
(ndbi046_env) % pip install psycopg2
```

add PostgreSQL bin directory to PATH

Objectives of the practical class

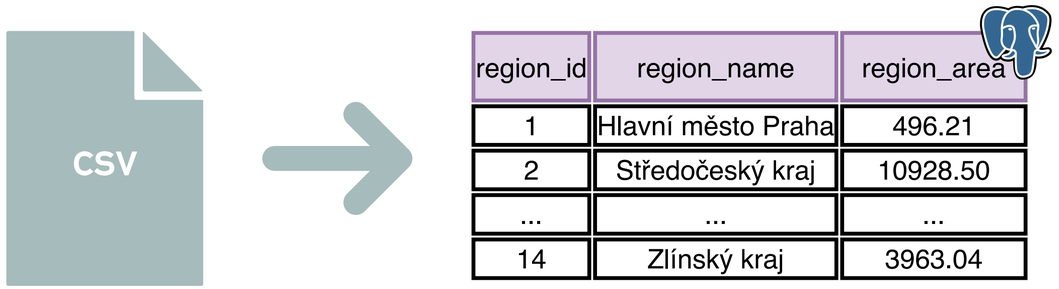
- Having prepared the *datasets* in csv format, we *load* them *into the data warehouse*
- For data warehouse we choose *PostgreSQL* database system
  - Easy to start with, but it's an OLTP system (certain limitation)
- Data warehouse *galaxy schema*:
  - Fact tables fact_waste_population, fact_expenses
  - Dimension tables dim_years, dim_regions, dim_waste_categories, dim_expenses_categories
- Three different methods of loading datasets will be utilized:
  - Row by row loading (simple, but slowest)
  - Bulk loading
  - *Bulk loading with integrity constraints deactivated* (fastest, but most complex)



Example 4.1: Load dataset 'Regions' into dim_regions (row by row)

- Write a *Python script to load the dataset* dim_regions.csv (see Example 3.2) *into the table* dim_regions in the data warehouse
  - The dimension table will contain the columns region_id (PRIMARY KEY), region_name, and region_area
    - Use *appropriate data types* to represent individual *columns* (e.g., INTEGER, FLOAT, TEXT, VARCHAR)
  - Insert data into the table row by row, i.e., *for each data row generate and execute the insert statement separately*

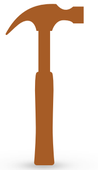
- Read the *user credentials* and information for connecting to PostgreSQL *from the configuration file* (e.g., credentials.json)



- **Tip:** A suitable choice for interaction with the PostgreSQL database system in Python is, e.g., the Psycopg2[#2] library (version 2.9.9).

[#2] https://pypi.org/project/psycopg2/

Example 4.1: Load dataset 'Regions' into dim_regions (row by row) (Solution)



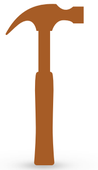
```python
import json
import logging
import sys
from typing import Any, Dict

import pandas as pd
from psycopg2 import Error, connect

logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")

def read_credentials_file(credentials_file: str) -> Dict[str, Any]:
  pass

def read_data_from_file(file_path: str) -> pd.DataFrame:
  pass

def execute_ddl(conn_params: Dict[str, Any], ddl_statement: str) -> None:
  pass

def insert_data(conn_params: Dict[str, Any], insert_query: str, data_df: pd.DataFrame
) -> None:
  pass
```

import library for interaction with PostgreSQL

logging into the console

program decomposition:
(1) loading credentials
(2) loading the dataset
(3) executing the DDL statements (i.e., create table, drop table)
(4) inserting data

Example 4.1: Load dataset 'Regions' into dim_regions (row by row) (Solution)

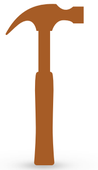


```python
def read_credentials_file(credentials_file: str) -> Dict[str, Any]:
    try:
        with open(credentials_file, "r") as file:
            credentials = json.load(file)
    except FileNotFoundError:
        logging.error(f"Credentials file '{credentials_file}' not found.")
        raise
    except Exception as e:
        logging.error(f"An error occurred while reading credentials file: {e}")
        raise
    return credentials


def read_data_from_file(file_path: str) -> pd.DataFrame:
    try:
        data_df = pd.read_csv(file_path, dtype=str)
    except FileNotFoundError:
        logging.error(f"File '{file_path}' not found.")
        raise
    except Exception as e:
        logging.error(f"An error occurred while reading data from file: {e}")
        raise
    return data_df
```

reading credentials from json file

load values as text strings to avoid errors such as "An unexpected error occurred: can't adapt type 'numpy.int64'" when inserting data into PostgreSQL

Example 4.1: Load dataset 'Regions' into dim_regions (row by row) (Solution)



required parameters for connection to the database server

DDL statement to be executed

```
def execute_ddl(conn_params: Dict[str, Any], ddl_statement: str) -> None:
  try:
    conn = connect(**conn_params)

    cur = conn.cursor()

    cur.execute(ddl_statement)

    conn.commit()
  except Error as e:
    logging.error(f"Error altering table: {e}")
    raise
  finally:
    cur.close()
    conn.close()
```

connect to the database server and create a cursor that allows you to execute commands in the database

execution of DDL statement followed by committing a transaction

closing the cursor and the database connection to prevent leakage of system resources

Example 4.1: Load dataset 'Regions' into dim_regions (row by row) (Solution)

```python
def insert_data(conn_params: Dict[str, Any], insert_query: str, data_df: pd.DataFrame
) -> None:
  try:
    conn = connect(**conn_params)
    cur = conn.cursor()

    for index, row in data_df.iterrows():
      cur.execute(insert_query, tuple(row))
      conn.commit()
  except Error as e:
    logging.error(f"Error inserting data: {e}")
    raise
  finally:
    cur.close()
    conn.close()
```

each
row of input data is inserted
individually within the scope of one
transaction

Example 4.1: Load dataset 'Regions' into dim_regions (row by row) (Solution)

```
class DimRegionsQueries:
  drop_table_query = """
  DROP TABLE IF EXISTS dim_regions;
  """

  create_table_query = """
  CREATE TABLE dim_regions (
    region_id INTEGER PRIMARY KEY,
    region_name VARCHAR(255),
    region_area FLOAT
  );
  """

  insert_query = """
  INSERT INTO dim_regions (region_id, region_name, region_area)
  VALUES (%s, %s, %s);
  """
```

DDL statement to delete a table if the table exists

DDL statement to create a table with primary key region_id

DML statement for inserting one row of data into the dim_regions

Example 4.1: Load dataset 'Regions' into dim_regions (row by row) (Solution)

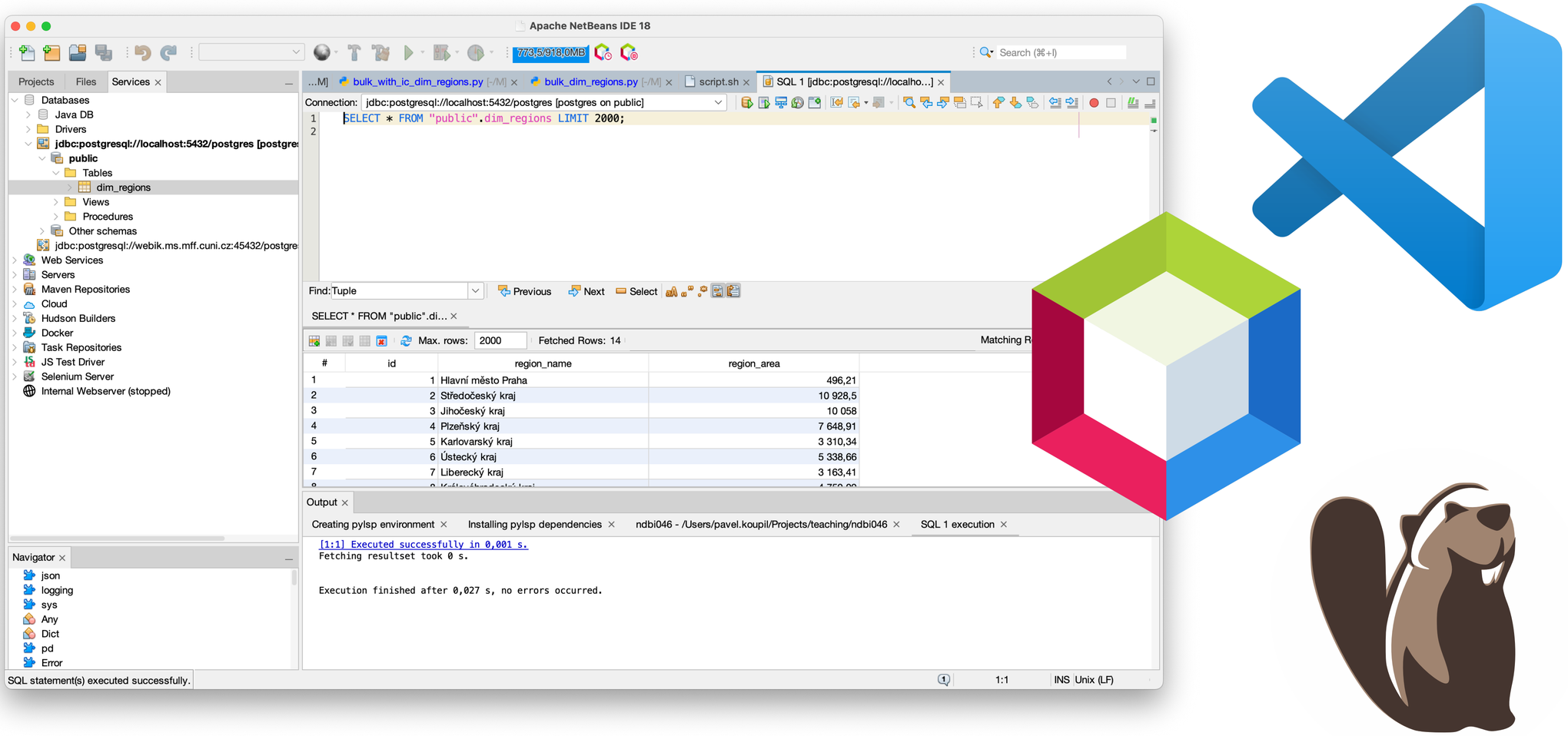
```python
if __name__ == "__main__":
    if len(sys.argv) != 3:
        logging.error("Usage: python script.py <credentials_file> <dataset_file>")
        sys.exit(1)

    credentials_file = sys.argv[1]
    dataset_file = sys.argv[2]

    try:
        conn_params = read_credentials_file(credentials_file)

        data_df = read_data_from_file(dataset_file)

        execute_ddl(conn_params, DimRegionsQueries.drop_table_query)

        execute_ddl(conn_params, DimRegionsQueries.create_table_query)

        insert_data(conn_params, DimRegionsQueries.insert_query, data_df)

        logging.info("Data insertion completed successfully.")
    except Exception as e:
        logging.error(f"An unexpected error occurred: {e}")
```

the program accepts two arguments: (1) the path to the credentials file and (2) the path to the data file

assembling a row by row insertion from individual steps

Example 4.1: Load dataset 'Regions' into dim_regions (row by row) (Solution)

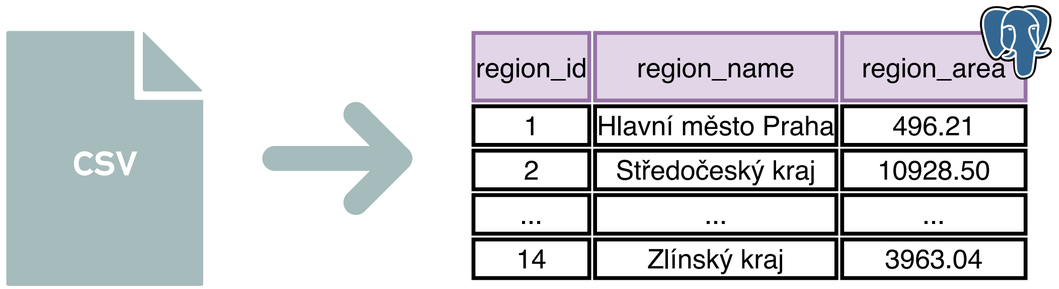
- Use your favorite IDE or database explorer to verify that the data was successfully loaded



Example 4.2: Load dataset 'Regions' into dim_regions (bulk loading)

- Write a *Python script to load the dataset* dim_regions.csv (see Example 3.2) *into the table* dim_regions in the data warehouse
  - The dimension table will contain the columns region_id (PRIMARY KEY), region_name, and region_area
    - Use *appropriate data types* to represent individual *columns* (e.g., INTEGER, FLOAT, TEXT, VARCHAR)
  - Insert data into the table utilizing *bulk loading*, i.e., *insert multiple rows of data (or entire dataset) at once*

- Read the *user credentials* and information for connecting to PostgreSQL *from the configuration file* (e.g., credentials.json)



- **Tip:** A suitable choice for interaction with the PostgreSQL database system in Python is, e.g., the Psycopg2[#2] library (version 2.9.9).

#2 *https://pypi.org/project/psycopg2/*

Example 4.2: Load dataset 'Regions' into dim_regions (bulk loading) (Solution)

necessary to import the function execute_values for bulk loading

```python
from psycopg2.extras import execute_values
```

```python
def insert_data(conn_params: Dict[str, Any], insert_query: str, data_df: pd.DataFrame
) -> None:
  try:
    conn = connect(**conn_params)
    cur = conn.cursor()

    execute_values(cur, insert_query, data_df.to_numpy())
    conn.commit()
  except Error as e:
    logging.error(f"Error inserting data: {e}")
    raise
  finally:
    cur.close()
    conn.close()
```

modify the insert_data function to insert all rows of data in a single transaction

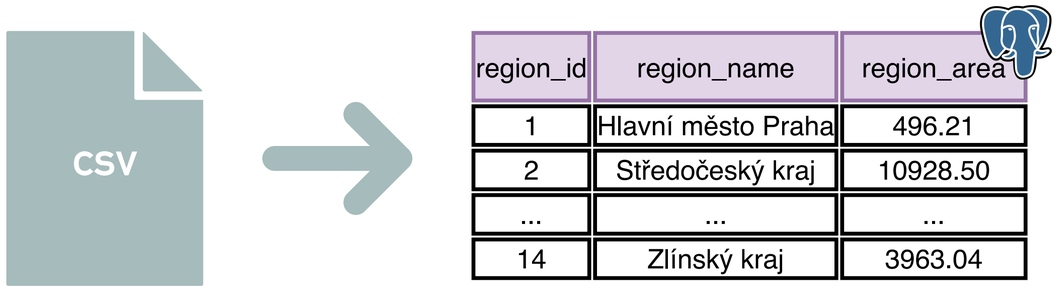
```python
  insert_query = """
  INSERT INTO dim_regions (region_id, region_name, region_area)
  VALUES %s;
  """
```

```python
  insert_data(conn_params, DimRegionsQueries.insert_query, data)
```

Example 4.3: Load dataset 'Regions' into dim_regions (bulk loading & alter table)

- Write a *Python script to load the dataset* dim_regions.csv (see Example 3.2) *into the table* dim_regions in the data warehouse
  - The dimension table will contain the columns region_id (PRIMARY KEY), region_name, and region_area
    - Use *appropriate data types* to represent individual *columns* (e.g., INTEGER, FLOAT, TEXT, VARCHAR)
  - Insert data into the table utilizing *bulk loading*, i.e., *insert multiple rows of data (or entire dataset) at once*
    - Load data efficiently, i.e., *set integrity constraints* only after *all data has been loaded* into the table dim_regions

- Read the *user credentials* and information for connecting to PostgreSQL *from the configuration file* (e.g., credentials.json)



- **Tip:** A suitable choice for interaction with the PostgreSQL database system in Python is, e.g., the Psycopg2[#2] library (version 2.9.9).

#2 *https://pypi.org/project/psycopg2/*

Example 4.3: Load dataset 'Regions' into dim_regions (bulk loading & alter table) (Solution)

```
1  class DimRegionsQueries:
2  ...
3    create_table_query = """
4    CREATE TABLE dim_regions (
5      region_id INTEGER,
6      region_name VARCHAR(255),
7      region_area FLOAT
8    );
9    """
10
11   alter_table_query = """
12   ALTER TABLE dim_regions
13   ADD CONSTRAINT dim_regions_pk PRIMARY KEY (region_id);
14   """
```

create a table without primary key (IC)

DDL statement for adding integrity constraint dim_regions_pk

```
14 ...
14     execute_ddl(conn_params, DimRegionsQueries.drop_table_query)
14
14     execute_ddl(conn_params, DimRegionsQueries.create_table_query)
14
14     insert_data(conn_params, DimRegionsQueries.insert_query, data)
14
14     execute_ddl(conn_params, DimRegionsQueries.alter_table_query)
14 ...
```

no integrity checking or index creation occurs during data insertion

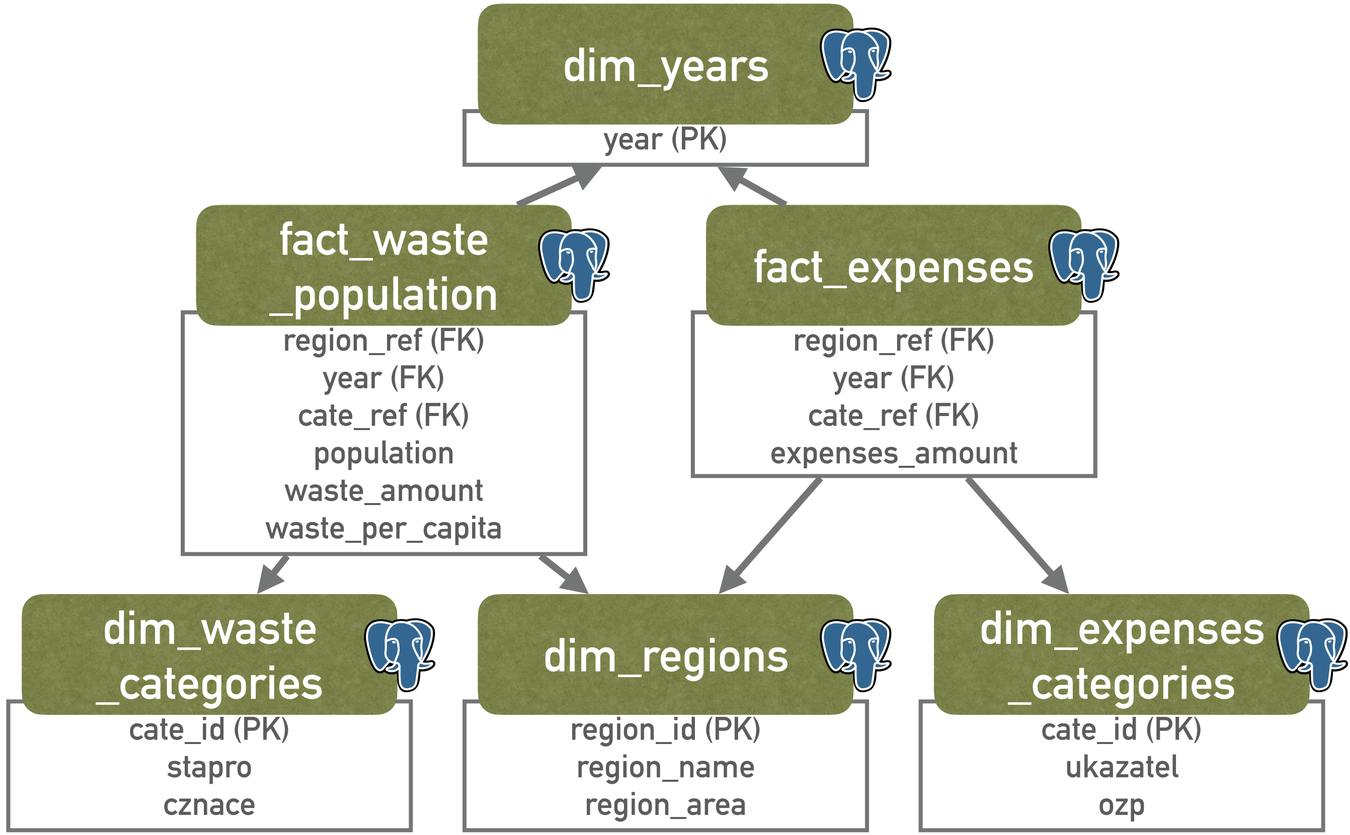
add integrity constraints after all data has been inserted

Exercise 4.4: Efficient and repeatable bulk loading of all datasets

- Extend the script from Example 4.4 to allow *bulk loading* of the following *datasets into the corresponding tables* in the data warehouse:
  - dim_regions.csv (already solved)
  - dim_regions.csv
  - dim_waste_dategories.csv
  - dim_expenses.categories.csv
  - fact_waste_population.csv
  - fact_expenses.csv

- Use *appropriate data types* to represent individual *columns*
- Load into the tables utilizing *bulk loading*, i.e., insert *multiple rows* of data (or entire dataset) *at once*
- Load data efficiently, i.e., *set integrity constraints* only *after all* data *has been loaded*
- Ensure *error-free repetition of script execution*
  - Determine the *appropriate order* of DDL and DML statements

- Read the *user credentials* and information for connecting to data warehouse *from the configuration file* (e.g., credentials.json)



References

Python

- Python 3.x (LATEST) documentation: https://docs.python.org/3/
- venv documentation: https://docs.python.org/3/library/venv.html
- Python W3Schools Tutorial: https://www.w3schools.com/python/
- psycopg2: https://pypi.org/project/psycopg2/

PostgreSQL

- Documentation: https://www.postgresql.org/docs/
- SQL W3Schools Tutorial: https://www.w3schools.com/sql/

IDEs and Database Tools

- Apache NetBeans: https://netbeans.apache.org/front/main/index.html
- Visual Studio Code: https://code.visualstudio.com/
- DBeaver: https://dbeaver.io/