Task 2: OverVIEW

Features

- **Create-Mode**:
  - Get an OverVIEW: Create a Database and populate it with your data
  - Test-suite: Run a set of SQL-queries to test the Database
- **Interactive-Mode**: Process SQL-queries on the created Database interactively

Downloads

- Input: Data
- Script: OverVIEW
- Reference: Test-suite tables
- Test-suite query as CSV

How to Use

Setup

To run the OverVIEW script requires the third-party libraries psycopg2 and rich for Python and PostgreSQL as third-party implementation of relational databases. The Python libraries can be installed using pip (lines 1 and 2) while PostgreSQL is provided by apt (line 3).

```
pip install psycopg2-binary
pip install rich
apt install postgresql
```

To use PostgreSQL, it is convenient to create a new SQL-user with a password who is able to create new databases. For that, switch first to the user postgres (line 1). Then start the PostgreSQL interactive terminal psql (line 2). Within psql, create a SQL-user (line 3) and provide them with a password (line 4). Since the script creates a database, give the SQL-user the permission to do that (line 5). To check if the SQL-user was created, list all SQL-users (line 6). Then exit (lines 7 and 8).

```
jan@linux:~$ sudo -iu postgres
postgres@linux:~$ psql
postgres=# CREATE USER ser;
postgres=# \password ser
postgres=# ALTER USER ser CREATEDB;
postgres=# \du
postgres=# \q
postgres@linux:~$ exit
```

Creation of the Database

To organize the data in the file *AlphaTechConsultigEmployees.csv* as SQL database, it has to be located in the same directory as the OverVIEW script. Running the script in create-mode (explicitely with arguments *-c* or *--create*) requests the username and password of the SQL-user created previously. It assembles a database owned by the SQL-user and contains the data of the original file.

```
jan@linux:~$ python3 overview_OOP.py -c

| OverVIEW of AlphaTech Consulting |

Provide login details for the database.
Username: ser
Password: ******

| Create-mode |

Creating the database "alphatech"...
Database created.

Creating and filling the database relations...
Database relations created and filled.

Adding constraints...
Constraints added.

Creating sample queries and output...
Created sample queries and output.
```

Inspection of the Database

To test whether the database was created correctly and allows for queries to extract data, the OverVIEW script runs a Test-suite and writes the results into the file *AlphaTech_tests.txt*. For comparison, the results of the Test-suite can be downloaded.

One example which is answered in the Test-suite is the question for the name, birthday and sex of the employees in the sales department. To get access to all three attributes selected in the first line, the table for the employees has to be joined with the table of the departments by using the department ID as common denominator. By filtering for the sales department, the question is answered. Note that attributes have to be specified with double quotes.

```
SELECT employees."EmployeeName", employees."DateofBirth", employees."Sex"
FROM employees
INNER JOIN departments ON employees."DeptID" = departments."DeptID"
WHERE departments."Department" = 'Sales';
```

The entity-relationship diagram discussed later provides the knowledge to decide when to join and when all requested information is already at hand.

To manually access the database via queries, two options are available. Running the OverVIEW script in interative-mode (no arguments or explicitely with arguments *-i* or *--interactive*) allows to compose and run queries.

```
jan@linux:~$ python3 overview_OOP.py -i

| OverVIEW of AlphaTech Consulting |

Provide login details for the database.
Username: ser
Password: ******

| Interactive-mode |

Enter query:
SELECT * from departments;
+------------------------------+
| DeptID | Department          |
|--------+---------------------|
| 4      | Software Engineering |
| 3      | IT/IS               |
| 1      | Admin Offices       |
| 2      | Sales               |
| 0      | Executive Office    |
+------------------------------+

Press q+Enter to quit or Enter to continue... q
```

The other option is to use psql to access the database. First switch to the user postgres (line 1). Then start the PostgreSQL interactive terminal psql (line 2). Switch over to your created SQL-user (line 3). Check if the database *alphatech* was created by listing all tables (line 4). Connect to the database *alphatech* (line 5). Compose and run a query (line 6).

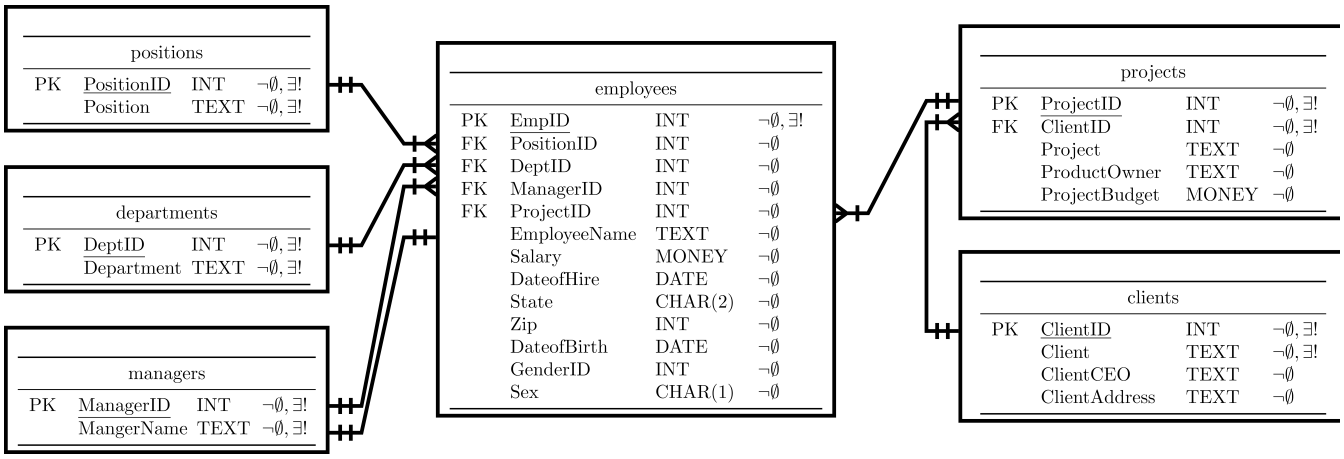
```
jan@linux:~$ sudo -iu postgres
postgres@linux:~$ psql
postgres=# SET ROLE ser;
postgres=> \l
postgres=> \c alphatech
alphatech=# SELECT * from departments;
 DeptID |      Department
--------+----------------------
      4 | Software Engineering
      3 | IT/IS
      1 | Admin Offices
      2 | Sales
      0 | Executive Office
(5 rows)
alphatech=# \q
postgres@linux:~$ exit
```

Note that in queries requesting attributes by name, they have to be put in double quotes (e.g. "EmployeeName").

Composition of queries

To compose queries to answer questions, one needs a layout of the database: Which tables are there and how are they related. Such a layout is provided by an entity-relationship diagram (ERD) which is shown below for the *alphatech* database. Central to everything are the employees with their employee ID as unique identifier. They are connected to the positions, departments, managers and projects by means of their unique IDs. Since the clients are connected to the employees only via the projects the employees work on, they are only directly connected to the projects by means of their unique client ID. A special kind of connection is between the EmployeeName and the ManagerName: Since a manager is an employee, the name of an employee can be that of a manager.

The keys in the first, the datatype in the third and further constraints in the fourth column of the tables are not relevant to compose queries that only access the data without altering it. Also the types of relationship denoted by different connectors are not relevant for this type of queries. More on these topics will be discussed in the section on Tasks & Design choices.



To get access to the name, birthday and sex of the employees in the sales department, one first has to join the table of employees and departments. After that, one can filter the employees based on their affiliation to a department by the name of the department.

Dependencies

Python Standard Library:

- sys
- getpass
- csv
- itertools
- textwrap
- functools
- datetime
- readline

Third-party libraries:

- psycopg2
- rich

Third-party software:

- PostgreSQL

Technical Summary

The OverVIEW script organizes the data provided by AlphaTech Consulting in a database using Python as interface and PostgreSQL to access and manipulate the data. It provides two modes, one to create the database and one to interact with it. For both modes the user specifies an SQL-user and their password to own the database.

In Create mode, first the database *alphatech* is created. Then the relations are created and filled with the data from *AlphaTechConsultigEmployees.csv* . After adding foreign key constraints, the Testing-suite consisting of several SQL-queries is executed.

In Interactive mode, the user can enter their own SQL-queries to access and manipulate the database.

Features & Extensibility

- **Create-Mode**: In Create-Mode, the OverVIEW script extracts the data from the source file *AlphaTechConsultigEmployees.csv* present in the directory of the script and creates the database *alphatech*. Both, source file and database name, are hard-coded for convenience.

  To **create another database using different data**, the variables *db_name*, *db_fname* and *db_tests*, as well as the Database methods *initialize_relations* and *test_suite* have to be adapted.

  Note first, that the OverVIEW script expects the source file to be formatted as CSV.

  In *initialize_relations*, the relation name, attribute names, attribute types, relation keys and attribute constraints have to be specified for the new database. Attribute names have to match the header names in the source file. Foreign keys are specified by the relation name and attribute name to which the foreign key links. Since primary keys automatically satisfy the *UNIQUE* and *NOT NULL* contraints, the constraint entry of the primary key has to be empty.

  In *test_suite*, the *tasks* describing the test cases and the *queries* containing the test cases have to be adapted.

  To make the script more flexible, one can read the source file name, database name and file name of the Test-suite output from the user and add three methods *get_name(self,name),* get_fname(self,fname) and *get_tests(self,tests)* to the *Database* class to alter the instance attributes *self.name*, *self.fname* and *self.tests*.
- **Interactive-Mode**: In Interactive-Mode, the database *alphatech* can be accessed. The name of the database is hard-coded for convenience.

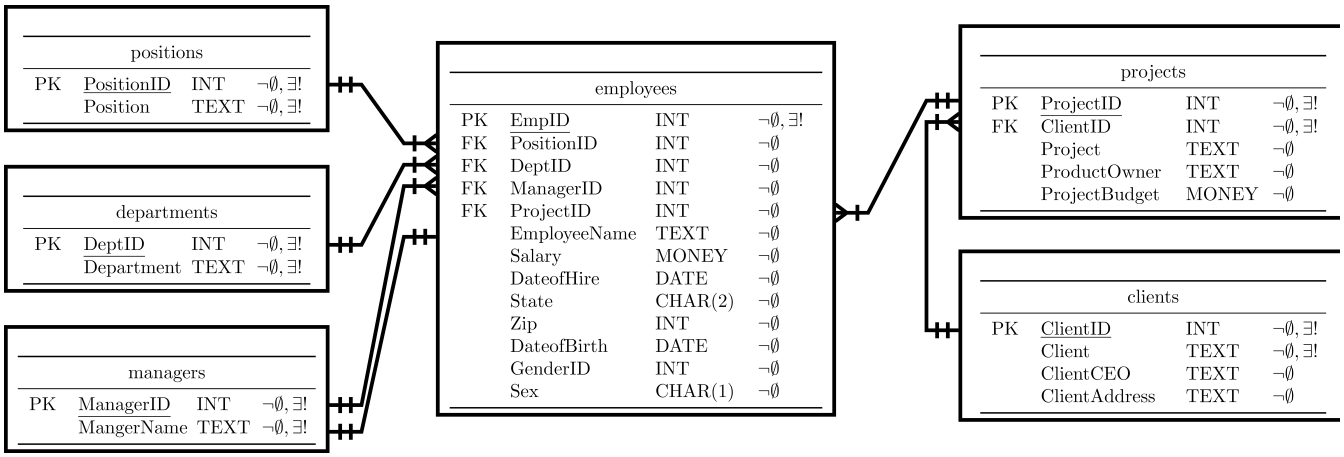
  To **access other databases**, the variable *db_name* has to be adapted. To make the scipt more flexible, one can read the database name from the user and add the method *get_name(self,name)* to the *Database* class to alter the instance attribute *self.name.*
- **Test-suite**: The Test-suite is executed when running the OverVIEW script in Create-Mode and its output is stored in the file with hard-coded name *AlphaTech_tests.txt*.

  To **extend the Test-suite**, the Database method *test_suite* has to be adapted. In it, the *tasks* describing the test cases and the *queries* containing the test cases have to be extended.

  To make the script more flexible, one can read the file name of the Test-suite output from the user and add a method *get_tests(self,tests)* to the *Database* class to alter the instance attribute *self.tests*.

Tasks & Design choices

Entity-relationship diagram



Relations

Consider a relation **employees** encompassing all data. Following the mnemonic "The key, the whole key and nothing but the key", the first three normal-forms during normalization require:

- **0NF**: Each tuple in a relation must be unique. Double occurences must be removed.
- **1NF**: Each relation must only have one (compound) primary key. Alternate keys must be moved into a new relation with the (compound) primary key as foreign key.
- **2NF**: Each attribute must depend only on the whole key. Attributes depending on parts of the primary key must be moved into a new relation with the part of the compund primary key as full primary key.
- **3NF**: Each attribute must depend directly on the primary key and not only transitively. Attributes which only depend transitively on the primary key must be moved into a new relation with the "linking" attribute as primary key.

To satisfy 1NF, the **primary key** for the relation denoting an employee is identified as the employee ID **EmpID**. Since this is not a compound primary key, 2NF is also satisfied. Concerning 3NF, there are several transitive dependencies:

- EmpID - PositionID - Position
- EmpID - DeptID - Department
- EmpID - ManagerID - ManagerName
- EmpID - ProjectID - ClientID, Project, ProductOwner, ProjectBudget
- ProjectID - ClientID - Client, ClientCEO, ClientAddress

Therefore, they are transferred into their own **relations** denoted as **positions**, **departments**, **managers**, **projects** and **clients** with the **primary keys** **PositionID**, **DeptID**, **ManagerID**, **ProjectID** and **ClientID**. Note that the attributes

- PositionID : Position
- DeptID : Department
- ManagerID : Manager
- ProjectID : Project
- ClientID : Client

can be mapped 1:1, so either of them could be used as a primary key. Since a numeric value is unique while a string can differ in capitalization, the IDs are used as primary keys. Also note, that GenderID and Sex are not listed here, since one attribute is essentially a copy of the other. To reduce redundancy one of them could be removed from the table. However to faithfully preserve completeness of the provided data, both are kept.

Be aware, that one cannot combine the managers relation with the departments relation, i.e. make the managers attributes part of the departments relation and vice versa:

- Example 1: The CTO manages the Software Engineering department, but is also a member of the department. Asking who manages the CTO would lead via the department to the CTO himself.
- Example 2: The CTO is managed by the CEO. While the CEO is member of the Executive office, the CTO is member of the Software Engineering department.

The relationship between employees, departments and managers is also not transitive:

- Example: The CTO belongs to the Software Engineering department, but is managed by the CEO. The association to a department does not determine the manager and vice versa.

One also cannot move the positions relation into the departments relation without having PositionID as foreign key in the employees relation. The problem here is, that requesting the position of an employee would only provide all possible positions in a department, rather than the actual position the employee holds. In other words, the relation between employees, departments and positions is not transitive and the position directly depends on the employee and not merely on the department of that employee.

Relationships

To allow *JOIN* operations, the primary keys of positions, departments, managers and projects have to be added as foreign keys to the employees. Likewise, the primary key of clients has to be added as foreign key to projects. These primary key-foreign key pairs also form the majority of relationships between relations:

- One or multiple employees with EmpID can **hold** one and only one Position with PositionID.
- One or multiple employees with EmpID can **belong to** one and only one Department with DeptID.
- One or multiple employees with EmpID are **managed by** one and only one Manager with ManagerID.
- One or multiple employees with EmpID can **work on** one and only one Project with ProjectID.
- One or multiple projects with ProjectID can **be commissioned by** one and only one Client with ClientID.

There is one additional relationship: One and only one Manager **is** at the same time one and only one employee, i.e. the ManagerName and EmployeeName agree.

Note that if any of those many-to-one relationships change into a many-to-many relationship, a new relation is added with a new ID as primary key which maps to multiple primary keys, e.g. four positions allow for 12 combinations of a primary position and a secondary position.

Constraints

Each primary key is by default *NOT NULL* and *UNIQUE*, see Constraints in the PostgreSQL documentation. Since the attributes

- PositionID : Position
- DeptID : Department
- ManagerID : Manager
- ProjectID : Project
- ClientID : Client

can be mapped 1:1, they inherit the same constraints from the primary keys. All other attributes are constrained to be *NOT NULL* to ensure reliable data.

Datatypes

For IDs and the Zip code, the INT datatype is used. For string type data, the datatype TEXT is used since it is flexible and in PostgreSQL nothing is gained by using VARCHAR(n), see Character Types in the documentation. For monetary values the datatype MONEY is used since it provides the correct precision and adds a currency symbol to the display. For dates, the datatype DATE is used. Note the issues occuring when only two digits of the year are specified on stackoverflow and the section Data Type Formatting Functions in the PostgreSQL documentation. For the State and Sex, two and one letter long *CHAR*s are used.

Main function

Using the hard-coded database name, source file name and file name of the Test-suite output, a *Database* object is created. After creating a *User* object, it asks the user to specify the SQL-user to own the database and their password. These credentials are checked and requested anew if they do not allow access to the *postgres* database.

Depending on the command line argument, the Create-Mode or Interactive-Mode is selected. The default mode without any argument is Interactive-Mode.

In Create-Mode, first the database is (re-)created. Then the relations are initialized and created within the Database. Thereby also primary keys, *NOT NULL* and *UNIQUE* constraints are specified. Next, additional constraints (here: foreign key constraints) are applied to the database. In case the database already exists and is currently in use (e.g. in psql), the user is notified. Finally, the Test-suite is executed.

In Interactive-Mode, an interactive SQL-session is started. If no connection to the database can be established, the user is informed. This is accomplished via a decorator added during definition of the function.

```
print("\n| OverVIEW of AlphaTech Consulting |\n")

db_name = "alphatech"
db_fname = "AlphaTechConsultigEmployees.csv"
db_tests = "AlphaTech_tests.txt"
db = Database(db_name,db_fname,db_tests)

# get login data
print("Provide login details for the database.")
user = User()
connected = False
while (not connected):
    user.get_login()
    connected = db.check_credentials(user)

if (len(sys.argv)==2):
    mode = sys.argv[1]
else:
    mode = "-i"

if (mode=="-c" or mode=="--create"):
    print("\n| Create-mode |\n")

    try:
        # create database
        print(f"Creating the database \"{db_name}\"...")
        db.create_database(user)
        print("Database created.\n")

        # create and fill relations
        print("Creating and filling the database relations...")
        db.initialize_relations()
        db.setup_relations(user)
        print("Database relations created and filled.\n")

        # set foreign key constraints
        print("Adding constraints...")
        db.add_constraints(user)
        print("Constraints added.\n")

    except psycopg2.errors.ObjectInUse:
        msg = "Error: Database cannot be rebuild " \
                +"since it is currently in use.\n" \
                +"Using existing database."
        print(msg)

    # run test suite
    print("Creating sample queries and output...")
    db.test_suite(user)
    print("Created sample queries and output.")

elif (mode=="-i" or mode=="--interactive"):
    print("\n| Interactive-mode |\n")

    # run queries
    interactive_queries(user,db.name)
```

- **Q**: Why are the database name, source file name and file name of the Test-suite output hard-coded?
  A: Since there is only one database, one source file and one Test-suite output, the variables are hard-coded for convenience. An extension is discussed in the section on Features & Extensibility.
- **Q**: Why is Interactive-Mode the default mode?
  A: Since Create-Mode *DROP*s the database and re-creates it, any changes in the database since creation would be lost by accidentally forgetting the command-line argument. The worst that can happen in Interactive-Mode is that no connection to the database can be established and that is handled.
- **Q**: What is meant by (re-)created?
  A: In case the database already exists, it is *DROP*ped and created anew. If it does not exist, it is created.
- **Q**: What is meant by initialization of relations and setup of relations?
  A: During initialization of relations, the *Relation* objects in Python are created. During setup of relations, the relations are created in the database.

- **Q**: Why are not all constraints handled while setting up the relations?
  A: Some constraints of one relation require the existence of another relation, e.g. foreign key constraints require the existence of the primary keys of the linked relation. With only one-sided primary key-foreign key constraints, there exists in principle an order in which the tables would have to be created such that foreign key constraints can be added during creation. In case of two-sided primary key-foreign key constraints, such an order no longer exists. To be general and not complicate the code unnecessarily, such constraints are added after all relations have been created and set to deferred mode to not interfere when adding new entries.

Relation class

The *Relation* class has five public and five "private" instance attributes:

- *self.name* (*str*): Name of the relation
- *self.attrs* (*tuple* of *str*): Attribute names of the relation
- *self.types* (*tuple* of *str*): Data types of the attributes
- *self.keys* (*tuple* of *str*): Strings specifying private and foreign keys
- *self.cstrs* (*tuple* of *str*): Strings specifying *NOT NULL* and *UNIQUE* constraints
- *self._sql_name*: SQL-conform version of *self.name*
- *self._sql_attrs*: SQL-conform version of *self. attrs*
- *self._sql_types*: SQL-conform version of *self.types*
- *self._sql_cstrs*: SQL-conform version of *self.cstrs*
- *self._attr_dict* (dict of str:int): Dictionary to translate lines of the input into attributes of the relation, see method *attr_dict*.

The datatypes of the SQL-conform versions can be inferred from the code block and the psycopg2 documentation.

```
def __init__(self,name,attrs,types,keys,cstrs):
    """Constructs necessary attributes of the Relation object."""

    self.name = name
    self.attrs = attrs
    self.types = types
    self.keys = keys
    self.cstrs = cstrs

    self._sql_name = sql.Identifier(self.name)
    self._sql_attrs = tuple(map(sql.Identifier,self.attrs))
    self._sql_types = tuple(map(sql.SQL,self.types))
    self._sql_cstrs = tuple(map(sql.SQL,self.cstrs))

    self._attr_dict = {}
```

- **Q**: Why are SQL-conform representations of strings needed?
  A: In case data provided by the user contains SQL-statements and quotes to counter escaping quotes, the user can execute SQL queries although they should not have permission to do so. This issue is called SQL-injection. The psycopg2 documentation elaborates on this page more on the topic and has the sql module to counter it. The webcomic xkcd has a nice illustration of the possible consequences.
- **Q**: Why is there no SQL-conform version of the key introduced?
  A: The keys encompass primary and foreign keys. Since they are applied at different times and require different handling, they are kept as tuples of strings until processed.
- **Q**: Why is *_attr_dict* only an instance attribute and not a class attribute?
  A: If *_attr_dict* were a class attribute, all relations created by the class would belong to the same source file. To allow for multiple source files with different headers, *_attr_dict* has to be an instance attribute. For the same reason the source file name is not a class attribute.

- **Task**: To add data to a relational database consisting of tables, the tables first have to be created. This is accomplished with CREATE queries.
  - **Implementation**: The primary key is extracted from the tuple keys and converted into an SQL-conform string. To fit the different entries properly into a query, they are first alternatingly combined into tuples using the *zip_longest* method of the *itertools* module (e. g. zip[(a,b),(c,d)] = (a,c),(b,d)). In contrast to the *zip* method, here the longest iterable determines the length of the resulting object. All "empty" parts of shorter iterables are filled with the *fillvalue* (e.g. zip_longest[(a,b,c),(d,e),fillvalue=h] = (a,d),(b,e),(c,h)). Next, the arguments are flattened into comma separated groups of space separated keywords (e.g. (a,d),(b,e),(c,h) a" "d, b" "e, c" "h). Finally, the names and flattened arguments are inserted into the query SQL-string, where each argument of the format method is entered at the location of a set of empty braces while preserving the order. This gives the format expected for a query used to create a relation (e.g. CREATE TABLE IF NOT EXISTS <table name> (<attr1> <type1> <cstrs1>, <attr2> <type2> <cstrs2>, ...)) which is used in the method *setup_relations*. Note that the clause "IF NOT EXISTS" prevents a table from being created if it already exists.

    ```
    def query_create(self):
        """Constructs query to CREATE the relation."""

        # get primary key
        pkey = tuple(key for key in self.keys if key=="PRIMARY KEY")
        sql_pkey = tuple(map(sql.SQL,pkey))
        # combine parts of arguments
        args_zip = tuple(itertools.zip_longest(self._sql_attrs, \
    ```

```
                                        self._sql_types, \
                                        sql_pkey, \
                                        self._sql_cstrs, \
                                        fillvalue=sql.SQL(""))))
    args_flat = sql.Composed(sql.SQL(', ').join( \
                            [sql.SQL(' ').join(tpl) for tpl in args_zip]))

    query = sql.SQL("CREATE TABLE IF NOT EXISTS {} ({});") \
           .format(self._sql_name,args_flat)

    return query
```

- **Task**: After creating a table, it has to be filled with data. This is accomplished with INSERT queries.
  - **Implementation**: Following the documentation of the psycopg2 sql module, queries can be assembled in a two-step process. First, relation names and attributes are entered at placeholders marked by empty braces. Furthermore, additional placeholders for actual values denoted by *sql.Placeholder()* given by *%s* are inserted at the last set of empty braces. The resulting query with placeholders (e.g. INSERT INTO <table name> (<attr1>, <attr2>, ...) VALUES (%s, %s, ...)) can then be used together with actual data to fill the relation. It is used in the method *setup_relations*. Note that the clause "ON CONFLICT DO NOTHING" prevents a tuple (row) from being added twice.

```
def query_insert(self):
    """Constructs query to INSERT tuples into the relation."""

    query = sql.SQL("""INSERT INTO {} ({}) VALUES ({})
                       ON CONFLICT DO NOTHING;""").format( \
            sql.Identifier(self.name), \
            sql.SQL(', ').join(self._sql_attrs), \
            sql.SQL(', ').join(sql.Placeholder() * len(self.attrs)))

    return query
```

- **Task**: To preserve referential integrity, i.e. agreement of values, of the primary key of one relation and a copy of it in another relation used for *JOIN*s, foreign key constraints are added to the relation.
  - **Implementation**: The strings specifying foreign keys are extracted from the tuple keys and split into the referenced relation and attribute. Next, the referenced attribute is used to identify the attribute acting as foreign key. Finally, all components are used to assemble the query (e.g. ALTER TABLE <table name> ADD CONSTRAINT <constraint name> FOREIGN KEY (<foreign key attribute>) REFERENCES <referenced table name> (<referenced primary key>) INITIALLY DEFERRED). To prevent conflicts when adding new entries, the constraints are set to deferred mode.

```
def fk_constraints(self):
    """Sets foreign key constraints for the relation."""

    # get foreign keys
    fkeys = tuple(key for key in self.keys if key!="PRIMARY KEY")
    for fkey in fkeys:
        # name, primary key of other relation
        pkey_rel = fkey.split()[0]
        pkey_attr = fkey.split()[1]
        # foreign key of self
        fkey_attr = self.attrs[self.keys.index(fkey)]

        query = sql.SQL("""ALTER TABLE {} ADD CONSTRAINT {}
                           FOREIGN KEY ({}) REFERENCES {} ({})
                           INITIALLY DEFERRED;""") \
               .format(self._sql_name, \
                       sql.Identifier("fk_"+fkey_attr), \
                       sql.Identifier(fkey_attr), \
                       sql.Identifier(pkey_rel), \
                       sql.Identifier(pkey_attr))

        yield query
```

  - **Q**: Why is the function a generator?
    A: There are multiple relations, each with possibly multiple foreign key constraints. To set all of these constraints for all relations, two loops are required. However a *Relation* object only knows about its own foreign keys, while the calling function knows about all relations. To restrict the keys to the scope of the *Relation* object, the loop over the foreign keys is kept inside and made accessbible by using a generator.
- **Task**: PostgreSQL is pecuiliar in how it expands a year given only by the last two digits "xy" in a date. It expands the date to 19xy or 20xy, depending which year is closer to 2020, see the situation described on stackoverflow and this page of the PostgrerSQL documentation

(search for "2020"). The problem with this is, that the resulting date can lie in the future. For example, xy=65 becomes 2065 instead of 1965. Since usually dates are referenced that have already passed, this is a problem.

- **Implementation**: The date is broken up at the separator "/". Then the length of the year string is checked. If it consists only of two digits, the number 20 is prepended and the resulting *year_20* is compared to the current year. If it is larger than the current year, i.e. lies in the future, the year is changed to lie in the 1900s, otherwise it is set to lie in the 2000s. Finally the list is joined into a full string again.

```
@staticmethod
def _check_date(old_date):
    """
    Expand year in old_date to four digits while assuming
    the youngest timespan possible (e.g. 18 years rather
    than 100 years).
    """

    new_date = old_date
    date_list = old_date.split("/")

    # check if digits in year are missing
    if (len(date_list[2])<4):
        year_20 = int("20"+date_list[2])
        current_year = date.today().year

        # check if year has already passed
        if ((year_20-current_year) > 0):
            date_list[2] = "19"+date_list[2]
        else:
            date_list[2] = year_20

        new_date = "/".join(date_list)

    return new_date
```

- **Q**: Why is this a static method?
  A: A static method is owned by the class which does not use any of its attributes or methods. It is used for utility functions for which it makes sense to associate them to a class. The *check_date* function checks the date for every relation, but does not need any attributes from the class. It thus checks these boxes.

- **Task**: A given source file contains columns which are to be separated into different SQL-relations. These columns thus have to be associated with the correct attribute of the correct relation. Also each line received from the source file may need to be modified, for example with the method *check_date*.
  - **Implementation**: In *create_attr_dict*, the translation between columns from the source file and attributes of a relation is implemented as a dictionary. Given the column/attribute name, it returns the index of the entry in a row. It is created using the header of the source file. This identification by name is also the reason, why capitalized attributes are entered into the database which require double quotes.

    In *convert_line*, the line is sorted using the *_attr_dict* and the line is modified using *check_date*.

```
def create_attr_dict(self,src_attr):
    """
    Creates dictionary to sort lines of input with attributes
    src_attr to match attributes of the relation.
    """

    self._attr_dict = {attr:src_attr.index(attr) for attr in self.attrs}


def convert_line(self,line):
    """
    Converts line of input to properly fit into the database.
    Tasks:
        *Sorts line to match attributes of the relation.
        *Modifies the date to handle a postgreSQL implementation
            choice (years with less than four digits are adjusted to
            be closest to 2020, see
            https://www.postgresql.org/docs/current/functions-formatting.html)
    """

    line_sorted = [line[self._attr_dict[attr]] \
                   for attr in self._attr_dict.keys()]
    line_checked = [Relation._check_date(line_sorted[ii]) \
                    if (self.types[ii]=="DATE") else line_sorted[ii] \
```

```
                    for ii in range(len(line_sorted))]

    return line_checked
```

- **Task**: To present the results of queries of the Test-suite in a clear manner, the table representations of relations are created and written to file or console. To allow further processing of the results of a query, a method to write them to a CSV file is added.
    - **Implementation**: The rich module is used to display the relations as tables. For compatibility reasons, the ouput is set to ASCII. The *create_table* method is used prior to both, the *print_table* and *write_table* methods. The *write_table* method is used in the *test_suite* method and *print_table* is used in the *interactive_queries* function. To write data to a CSV file, the csv module is used.

```
@staticmethod
def create_table(header,content):
    """
    Creates table representation of a query with header
    and content.
    """

    # setup
    table = Table(box=rich.box.ASCII)
    # columns
    for attr in header:
        table.add_column(attr)
    # add rows
    for entry in content:
        table.add_row(*entry)

    return table


@staticmethod
def print_table(table):
    """Displays table representation of a query."""

    console = Console()
    console.print(table)


@staticmethod
def write_table(table,fname):
    """Writes table representation of a query to file fname."""

    rich.print(table,file=fname)


@staticmethod
def export_csv(content,fname):
    """Export query as csv file."""

    try:
        with open(fname,"w",newline="") as csvfile:
            writer = csv.writer(csvfile,delimiter=",",quotechar="\"", \
                                quoting=csv.QUOTE_MINIMAL)
            for line in content:
                writer.writerow(line)
    except PermissionError:
        msg = f"Error: You lack permission to create {fname}."
        print(msg)
```

    - **Q**: Why is this a static method?
      A: A static method is owned by the class which does not use any of its attributes or methods. It is used for utility functions for which it makes sense to associate them to a class. The functions *create_table*, *print_table* and *write_table* are used to represent a query result as a table for output to file or console. Since in PostgreSQL a query result can be saved as a *VIEW* and used as a relation, it makes sense to add these methods to the *Relation* class.

Decorator

- **Task**: Many methods establish a connection with the PostgreSQL-server and thus encounter the same errors. To handle these errors without repeating code, a decorator is used.

- **Implementation**: A decorator is a wrapper around a function which modifies the wrapped functions behavior. The outer function name will be used for the decorator as *@check_db_connection_exists* and added above the definition of a function. To preserve the *__name__* and *__doc__* attributes of the wrapped function, the *@wraps* decorator of the functools module is used. Within the decorated function, catching the generic OperationalError informs the user of a failed connection to the database. This can have multiple reasons: the wrong SQL-user, the wrong password or a non-existing database. There do not seem to be more specific exceptions for the different reasons as stackoverflow1 and stackoverflow2 suggest.

```
def check_db_connection(function):
    """Decorator checking if the database exists."""

    @wraps(function)
    def decorated(*args):
        try:
            function(*args)
        except psycopg2.errors.OperationalError:
            msg = "Error: Connection to database failed."
            print(msg)
    return decorated
```

Database class

The *Database* class has four public instance attributes:

- *self.name* (*str*): Name of the database
- *self.fname* (s*tr*): File name of the source file
- *self.tests*(*str*): File name of the Test-suite output
- *self.relations* (*tuple* of *Relation* objects): Tuple of relations belonging to the database

```
def __init__(self,name,fname,tests):
    """Constructs necessary attributes of the Database object."""

    self.name = name
    self.fname = fname
    self.tests = tests
    self.relations = None
```

- **Task**: Check if the credentials for the SQL-user are correcly provided.
    - **Implementation**: Try to establish a connection to the *postgres* database using the provided credentials. Return False if the credentials are insufficient.

```
@staticmethod
def check_credentials(user):
    """Checks Username and Password."""

    connected = False
    try:
        conn = psycopg2.connect(dbname="postgres",
                                host="localhost",
                                port="5432",
                                user=user.name,
                                password=user.passwd)
        conn.close()
    except psycopg2.errors.OperationalError:
        msg = "Error: Wrong Username or Password."
        print(msg)
    else:
        connected = True

    return connected
```

- **Q**: Why is this a static method?
  A: A static method is owned by the class which does not use any of its attributes or methods. It is used for utility functions for which it makes sense to associate them to a class. The *check_credentials* function checks the login credentials of the SQL-user by connecting to the *postgres* database, but does not need any attributes from the class. It thus checks these boxes.

- **Task**: Create a database.
  - **Implementation**: After establishing a connection to the PostgreSQL server, the query to create the database is assembled and executed. In case the database already exists, it is dropped. Note that to create a database, the query has to be treated as a transaction block by setting *conn.autocommit* to *True*.

```
def create_database(self,user):
    """Creates database."""

    conn = psycopg2.connect(dbname="postgres",
                            host="localhost",
                            port="5432",
                            user=user.name,
                            password=user.passwd)
    conn.autocommit = True
    cursor = conn.cursor()

    # remove database if it exists already
    query = sql.SQL("DROP DATABASE IF EXISTS {};") \
           .format(sql.Identifier(self.name))
    cursor.execute(query)
    # create database
    query = sql.SQL("CREATE DATABASE {};") \
           .format(sql.Identifier(self.name))
    cursor.execute(query)

    cursor.close()
    conn.close()
```

  - **Q**: Why is the header and footer containing the setup and closure of the connection to the PostgreSQL-server not moved into a decorator?
    A: Moving the connection setup and closure into a decorator would be possible with a decorator with arguments discussed on this an d this website, if the *cursor* variable defined therein would not be needed in the wrapped function. To make the *cursor* variable accessible to the wrapped function, one would either have to work with unnamed arguments *args and **kwargs and extend kwargs in the decorator or add the cursor variable to the global namespace of the decorated function. The prior option makes the code hard to understand and the latter makes the code not threadsafe, which contradicts the ACID properties database transactions should possess. See more on stackoverflow. Since such a decorator would make the code in fact less readable, this is not pursued.

- **Task**: To create relations, all their names, attributes, attribute types, primary keys, foreign keys and other constraints have to be set.
  - **Implementation**: The names, attributes, attribute types, primary keys, foreign keys and other constraints of all relations to be created are separated into *tuples* and used to initialize the corresponding *Relation* objects. Note that this is an instance method since it sets the *relations* attribute of a *Database* object.

    To shorten the notation, constructs like *["INT"]*5 are used to create five entries of "INT" in a row. Note, that the tuples do not have to have the same length and only the attributes which should have constraints have to be specified. Attributes without constraints standing before attributes with constraints have to be entered as empty strings (e.g. a PRIMARY KEY already fulfills the *NOT NULL* and *UNIQUE* constraints). Note that tuples containing only one argument need a **trailing comma**. Otherwise they are not considered an iterable. Also note that the attribute names must match the names in the source file header to associate the columns with the attributes. This leads in the end to the requirement for double quotes when requesting an attribute by name in a query.

```
def initialize_relations(self):
    """Initializes relations for the database."""

    # initialize Relation objects
    # employees
    name = "employees"
    attrs = ("EmpID","PositionID","DeptID","ManagerID","ProjectID", \
             "EmployeeName","Salary","DateofHire","State","Zip", \
             "DateofBirth","GenderID","Sex")
    types = (*["INT"]*5,"TEXT","MONEY","DATE","CHAR(2)","INT", \
                "DATE","INT","CHAR(1)")
    keys = ("PRIMARY KEY","positions PositionID","departments DeptID", \
            "managers ManagerID", "projects ProjectID")
    cstrs = ("",*["NOT NULL"]*(len(attrs)-1))
    employees = Relation(name,attrs,types,keys,cstrs)
    # departments
    name = "departments"
    attrs = ("DeptID","Department")
    types = ("INT","TEXT")
    keys = ("PRIMARY KEY",)
```

```
    cstrs = ("","NOT NULL UNIQUE")
    departments = Relation(name,attrs,types,keys,cstrs)
    # positions
    name = "positions"
    attrs = ("PositionID","Position")
    types = ("INT","TEXT")
    keys = ("PRIMARY KEY",)
    cstrs = ("","NOT NULL UNIQUE")
    positions = Relation(name,attrs,types,keys,cstrs)
    # managers
    name = "managers"
    attrs = ("ManagerID","ManagerName")
    types = ("INT","TEXT")
    keys = ("PRIMARY KEY",)
    cstrs = ("","NOT NULL UNIQUE")
    managers = Relation(name,attrs,types,keys,cstrs)
    # projects
    name = "projects"
    attrs = ("ProjectID","ClientID","Project", \
             "ProductOwner","ProjectBudget")
    types = (*["INT"]*2,*["TEXT"]*2,"MONEY")
    keys = ("PRIMARY KEY","clients ClientID")
    cstrs = ("","NOT NULL","NOT NULL UNIQUE",*["NOT NULL"]*2)
    projects = Relation(name,attrs,types,keys,cstrs)
    # clients
    name = "clients"
    attrs = ("ClientID","Client","ClientCEO","ClientAddress")
    types = ("INT",*["TEXT"]*3)
    keys = ("PRIMARY KEY",)
    cstrs = ("","NOT NULL UNIQUE",*["NOT NULL"]*2)
    clients = Relation(name,attrs,types,keys,cstrs)
    # relations
    self.relations = [employees,departments,positions,managers, \
                      projects,clients]
```

  - **Q**: Why are the attributes "Department", "Position", "ManagerName", "Project" and "Client" set to *UNIQUE*?
    A: These attributes are alone in their relation with the primary key, which is *NOT NULL* and *UNIQUE*. Since there is a 1:1 relationship between them, these properties are explicitely transferred to them.
- **Task**: Create the relations of the database.
  - **Implementation**: After establishing a connection to the PostgreSQL server, the source file is opened as a CSV formatted file. Then, for each relation separately, the header is used to create the dictionary to translate between the source file and the attributes of each relation. Next, the relation is created using the query obtained from the method query_create. Afterwards, the rest of the source file is read. For each relation, the currently read line is converted to fit the relation attributes and fix dates using the method check_date. Then the converted line is inserted in the relation.

    In case the source file does not exist or the user does not have permission to read it, they are informed. Note that if no connection to the database can be established, the decorator *check_db_connection* informs the user.

```
@check_db_connection
def setup_relations(self,user):
    """Creates and fills relations."""

    conn = psycopg2.connect(dbname=self.name,
                            host="localhost",
                            port="5432",
                            user=user.name,
                            password=user.passwd)
    cursor = conn.cursor()

    try:
        with open(self.fname,newline="") as csvfile:
            reader = csv.reader(csvfile, delimiter = ",")

            # create relations
            header = next(reader)
            for relation in self.relations:
                relation.create_attr_dict(header)
                query = relation.query_create()
                cursor.execute(query)

            # fill relations
```

```
            for line in reader:
                for relation in self.relations:
                    line_converted = relation.convert_line(line)

                    query = relation.query_insert()
                    cursor.execute(query,line_converted)

    except FileNotFoundError:
        msg = f"Error: The file {csvfile} does not exist."
        print(msg)
    except PermissionError:
        msg = f"Error: You lack permission to read {csvfile}."
        print(msg)

    conn.commit()
    cursor.close()
    conn.close()
```

- **Q**: Why is the option *conn.autocommit=True* not used?
  A: The autocommit option would "save" the creation of a table and each line after it is inserted into the table. Besides not loosing everything upon a crash, it would help with debugging to see, after which line a crash occurs. Here, it is left out since it is not strictly necessary in contrast to the method create_database. Furthermore, the OverVIEW script does not provide a recovery from a crash and the database would have to be re-created anyway.
- **Q**: Why is the dictionary to translate between header and attributes not only created once for the entire *Relation* class?
  A: Defining the dictionary as class attribute would link all *Relation* objects to the same source file. In case there are multiple source files, another class would be required. To be able to use one class for multiple source files, the dictionary either has to be an instance attribute or a list of dictionaries for each source file. For the current purpose, the instance attribute is simplest and does not pose a bottleneck.

- **Task**: Add constraints to the attributes of the relations.
  - **Implementation**: After establishing a connection to the PostgreSQL server, all relations are passed through and all foreign-key constraints are set. It uses the fk_constraints method.

    Note that if no connection to the database can be established, the decorator *check_db_connection* informs the user.

```
@check_db_connection
def add_constraints(self,user):
    """Adds constraints to database."""

    conn = psycopg2.connect(dbname=self.name,
                            host="localhost",
                            port="5432",
                            user=user.name,
                            password=user.passwd)
    cursor = conn.cursor()

    for relation in self.relations:
        for query in relation.fk_constraints():
            cursor.execute(query)

    conn.commit()
    cursor.close()
    conn.close()
```

  - **Q**: Why is the option *conn.autocommit=True* not used?
    A: The autocommit option would "save" the creation of a table and each line after it is inserted into the table. Besides not loosing everything upon a crash, it would help with debugging to see, after which line a crash occurs. Here, it is left out since it is not strictly necessary in contrast to the method create_database. Furthermore, the OverVIEW script does not provide a recovery from a crash and the database would have to be re-created anyway.
- **Task**: To provide the user with a possibility to check if the OverVIEW script fulfilled its task, a Test-suite of several queries is assembled and executed.
  - **Implementation**: The Test-suite consists of a list of short task descriptions and a list of queries to address them. Each query is executed one after another and the output consists of the task description, the executed query as text and the output as ASCII table. Note that the attributes in the queries are put in double quotes. The indentation of the task description and queries is removed using the *dedent* method of the *textwrap* module. To illustrate the export into a CSV file, the results of the first query are written to file.

    In case the user does not have permission to write to file, they are informed. Note that if no connection to the database can be established, the decorator *check_db_connection* informs the user.

```
@check_db_connection
def test_suite(self,user):
```

```
    """
    Provides sample queries and their output to verify
    the created database.
    """

    tasks = ["Name, birthday and sex of the Sales department", \
             "Female share", \
             "Gender pay gap", \
             "Hiring over three years", \
             "Age of oldest employee", \
             "Average age", \
             "Total salary for Apple projects", \
             """
             Name, position and department of employees
             working on the projects \"Google Cloud Platform\"
             and \"AWS\"
             """, \
             "Positions in the Software Engineering department", \
             "Number of Managers", \
             "Management team"]

    queries = ["""
                SELECT employees.\"EmployeeName\", employees.\"DateofBirth\", employees.\"Sex\"
                FROM employees
                INNER JOIN departments ON employees.\"DeptID\" = departments.\"DeptID\"
                WHERE departments.\"Department\" = 'Sales';
                """,
                """
                SELECT TO_CHAR(CAST(COUNT(*) FILTER (WHERE \"Sex\"='F') AS decimal)/(COUNT
(*)),'0.99')
                AS \"Ratio Woman/Total\" FROM employees;
                """,
                """
                SELECT CAST(AVG(\"Salary\"::numeric) FILTER (WHERE \"Sex\"='F') AS MONEY)
                AS \"Average Salary Women\",
                CAST(AVG(\"Salary\"::numeric) FILTER (WHERE \"Sex\"='M') AS MONEY)
                AS \"Average Salary Man\" FROM employees;
                """,
                """
                SELECT COUNT(*) FROM employees WHERE \"DateofHire\" >= (CURRENT_DATE - INTERVAL
'3 year');
                """,
                """
                SELECT TO_CHAR(AGE(CURRENT_DATE, MIN(\"DateofBirth\")),
                'YY "Years" mm "Months" DD "Days"')
                AS \"Age of oldest employee\" FROM employees;
                """,
                """
                SELECT TO_CHAR(AVG(AGE(CURRENT_DATE, \"DateofBirth\")),
                'YY "Years" mm "Months" DD "Days"')
                AS \"Average Age\" FROM employees;
                """,
                """
                SELECT SUM(employees.\"Salary\") AS \"Total salary for Apple projects\" FROM
employees
                INNER JOIN projects ON employees.\"ProjectID\"=projects.\"ProjectID\"
                INNER JOIN clients ON projects.\"ClientID\"=clients.\"ClientID\"
                WHERE clients.\"Client\"='Apple';
                """,
                """
                SELECT employees.\"EmployeeName\", positions.\"Position\", departments.\"
Department\"
                FROM employees
                INNER JOIN positions ON employees.\"PositionID\"=positions.\"PositionID\"
                INNER JOIN departments ON employees.\"DeptID\"=departments.\"DeptID\"
                INNER JOIN projects ON employees.\"ProjectID\"=projects.\"ProjectID\"
                WHERE projects.\"Project\"='Google Cloud Platform' OR projects.\"Project\"
='AWS';
                """,
                """
                SELECT DISTINCT positions.\"Position\" AS \"Positions\" FROM employees
```

```
            INNER JOIN positions ON employees.\"PositionID\"=positions.\"PositionID\"
            INNER JOIN departments ON employees.\"DeptID\"=departments.\"DeptID\"
            WHERE departments.\"Department\"='Software Engineering';
            """,
            """
            SELECT COUNT(*) AS \"Number of Managers\" FROM employees
            INNER JOIN managers ON employees.\"EmployeeName\"=managers.\"ManagerName\";
            """,
            """
            SELECT managers.\"ManagerName\", positions.\"Position\", employees.\"Salary\"
            FROM employees
            INNER JOIN managers ON employees.\"EmployeeName\"=managers.\"ManagerName\"
            INNER JOIN positions ON employees.\"PositionID\"=positions.\"PositionID\";
            """]

    conn = psycopg2.connect(dbname=self.name,
                            host="localhost",
                            port="5432",
                            user=user.name,
                            password=user.passwd)
    cursor = conn.cursor()


    # table representation
    try:
        with open(self.tests,"w",newline="") as test_file:
            for ii in range(len(tasks)):
                test_file.write("\n" \
                                +textwrap.dedent(tasks[ii]).strip() \
                                +":\n")
                cursor.execute(queries[ii])
                header = tuple(name[0] for name in cursor.description)
                response = [tuple(map(str,entry)) \
                            for entry in cursor.fetchall()]
                table = Relation.create_table(header,response)
                test_file.write(textwrap.dedent(queries[ii])+"\n")
                Relation.write_table(table,test_file)

    except PermissionError:
        msg = f"Error: You lack permission to create {fname}."
        print(msg)

    # csv file
    cursor.execute(queries[0])
    header = list(name[0] for name in cursor.description)
    response = [list(map(str,entry)) \
                for entry in cursor.fetchall()]
    Relation.export_csv([header,*response], \
                        self.tests.split(".")[0]+".csv")

    conn.commit()
    cursor.close()
    conn.close()
```

User class

The User class has two public instance attributes:

- *self.name (str)*: Name of the user
- *self.passwd (str)*: Password of the user

- **Task**: To connect to a PostgreSQL-server, a SQL-user and their password are required.
    - **Implementation**: The User class serves to obtain and store the SQL-user and their password.

```
class User:
...
    def __init__(self):
        """Constructs all necessary attributes for the User object."""
```

```
        self.name = ""
        self.passwd = ""


    def get_login(self):
        """Gets login data for SQL server."""

        self.name = input("Username: ")
        self.passwd = getpass("Password: ")
```

Interactive-Mode

- **Task**: Allow the user direct access to the created database.
    - **Implementation**: After establishing a connection to the PostgreSQL server, the user can enter multi-line SQL-queries using lists and the *join* method of strings. Note that with the readline module also corrections using the arrow keys are possible for the user. Typos in attributes and relations are handled and after the user is done, they can quit.

      Note that if no connection to the database can be established, the decorator *check_db_connection* informs the user.

```
@check_db_connection
def interactive_queries(user,db_name):
    """Run queries interactively."""

    conn = psycopg2.connect(dbname=db_name,
                            host="localhost",
                            port="5432",
                            user=user.name,
                            password=user.passwd)
    cursor = conn.cursor()

    input_quit = ""
    while (input_quit!="q"):

        # read query
        print("Enter query:")
        input_list = []
        while True:
            line = input()
            input_list.append(line)
            if (";" in line):
                line = line[:line.index(";")+1]
                input_list[-1] = line
                break
        query = " ".join(input_list)

        cursor.execute('SAVEPOINT sp;')
        try:
            cursor.execute(query)
        except psycopg2.errors.UndefinedColumn:
            msg = "Error: Cannot find attribute."
            print(msg)
            cursor.execute('ROLLBACK TO SAVEPOINT sp;')
        except psycopg2.errors.UndefinedTable:
            msg = "Error: Cannot find table."
            print(msg)
            cursor.execute('ROLLBACK TO SAVEPOINT sp;')
        else:
            # display table
            header = tuple(name[0] for name in cursor.description)
            response = [tuple(map(str,entry)) for entry in cursor.fetchall()]
            table = Relation.create_table(header,response)
            Relation.print_table(table)

        input_quit = input("\nPress q+Enter to quit or Enter to continue... ")

    conn.commit()
    cursor.close()
    conn.close()
```

- **Q**: Why is the option *conn.autocommit=True* not used?
  A: The autocommit option would "save" the creation of a table and each line after it is inserted into the table. Besides not loosing everything upon a crash, it would help with debugging to see, after which line a crash occurs. Here, it is left out since it is not strictly necessary in contrast to the method create_database. Furthermore, the OverVIEW script does not provide a recovery from a crash and the database would have to be re-created anyway.

Planned features

- **Append-Mode**: Allow appending data from a file to the database.
- **Generalization**: Extend the OverVIEW script to create multiple databases for multiple source files.
- **User-friendly Input**: Externalize the specification of the relations and Test-suite as input files with user-friendly formatting. Create a GUI to enter relations and tests.
- **Complexity**: Allow for compound keys.
- **Source file format**: Allow the user to specify the delimeter of the source file.