Case Study

During the development stage of a project, you may want to have a full reset of the database. Believe it or not, even when the underlying table is deleted, stored procedures/functions and/or triggers may persists. In some cases, this is good, because you can update your table and the database can run again after you finish your update. In other cases, you don't know the effect that a lingering trigger may have on your table.

Since the project will be evaluated from a blank state, we will also need to be able to do a full reset of the database. Before we proceed with the description, we have a large warning.

WARNING

Following this, the `public` schema on your database will be wiped clean. So far, we have been using `public` schema for everything. Basically, all your working will be lost. Make sure you have a backup of those.

Continue with this only if you understand the risk.

Ok, so make sure you understand the risk as once you have done the following two simple lines, there is no going back.

WARNING

Are you sure you understand and want to continue?

While we try to ensure that the test cases are all correct, there may be mistake which may only be found during actual grading.

So the code is as follows:

```
1 DROP SCHEMA public CASCADE;
2 CREATE SCHEMA public;
```

The first line will drop schema. Unlike our definition of schema, in PostgreSQL, schema is like a namespace. When you create a table `T`, if it belongs to a schema `S`, then the qualified name of the table is actually `S.T`. But because we have only been using public, there is no other namespace and there is no other table called `T`. Even if there is, you may not have access to those.

Test Cases

With that preliminary in mind, we can now start to explain how the test cases for our project can work. We will provide a different `DB.py` file, one that has an added method called `reset` to fully reset the `public` schema as laid out above.

Files

You are provided with the following files from Canvas "`Files > Cases > Reset`" as our "main files". We will not give an explanation of the other files as they are utilities file created long ago.

File Name	Description
`DB.py`	A Python class encapsulating database connection.
`Config.py`	A configuration file that contains your password. **DO NOT SHARE THIS FILE**.
`Utilities.py`	A file containing a function to pretty print a table.
`ProjectChecker.py`	The "public" test cases for the project.
`DDL.sql`	The ddl for project 3.
`P03.sql`	A blank file provided so that you can put your solution for testing.

As a side-note, we will have "private" test cases. We try to make the "public" test cases as extensive as possible, but we may have missed some cases. The main thing about "private" test cases is to avoid students from "hardcoding" the solution. For instance,

```
1 SELECT 'ans1' AS row1, 'ans2' AS row2;
```

Python

To use the file, you will need Python installed, preferably Python 3.11 and above. You can download Python at the following website: https://www.python.org/downloads/. Follow the instruction depending on your operating system:

- Windows: https://docs.python.org/3/using/windows.html#installation-steps
- MacOS: https://docs.python.org/3/using/mac.html#getting-and-installing-python

We would recommend

- **Uncheck** the box for "Install launcher for all users (recommended)".
- **Check** the box for "Add Python 3.XX to PATH".

PsycoPG

Once you have installed Python, follow the steps below to install PsycoPG.

1. Run one of the following commands on terminal:

   ```
   pip install psycopg
   ```

   ```
   python -m pip install psycopg
   ```

   If those do not work, you may try either `pip3` or `python3` before trying the fix below.

2. Update `PG_HOME` or `PATH` if step 1 failed.

   NOTE: The following fix was needed on MacOS but was not needed on Windows.

   Run both commands on terminal:

   ```
   export PG_HOME=/Library/PostgreSQL/16
   export PATH=$PATH:$PG_HOME/bin
   ```

   You may need to change `16` to your version of PostgreSQL. Once you are done, try step 1 again.

3. Install missing binaries if there is still an error in running `DB.py`. Run one of the following additional commands on terminal:

   ```
   pip install "psycopg[binary,pool]"
   ```

   ```
   python -m pip install "psycopg[binary,pool]"
   ```

Running the Test Cases

The test cases in `ProjectChecker.py` are arranged as follows:

1. Connect to the database and reset: `db = DB(**config).reset()`.
   After this point,
   - the variable `db` will contain an active connection with PostgreSQL, and
   - `public` schema has been dropped and recreated.
2. Setting up the database. This will be a series of `INSERT` run using `db.exec('INSERT ...')`.
   - We setup the database before any stored procedures/functions and/or triggers are initialized.
   - This allows us more flexibility in our `INSERT`.
   - We guarantee that the initial setup will produce **valid** tables with respect to the constraints.

3. Checking the database consistency: `db_check = unordered(db.fetch('...'), expect)`.
    - We check one of the tables against an expected value `expect`.
    - This ensures that we start from a correct data.
4. Initialize **ALL** the functions/procedures/triggers from your answer.
5. Testing. This will be a series of statements of the following form:

```
db.exec('... command to run ...')
db_check = unordered(db.fetch('SELECT ...').res[-1], expect)
count += check_db_state(db_check, 'Test Name', tbl)
```

    - We first run the command (*e.g.*, `CALL return_car(...)`).
    - Then we check if the affected tables have the correct value by executing `db.fetch('SELECT ...')` on the affected table.
    - We get the **last** result `db.fetch('SELECT ...').res[-1]`.
    - Then we compare this against the expected value `expect`.
    - There are 2 kinds of comparison
        (a) Unordered check `unordered(actual, expect)`.
        (b) Ordered check `ordered(actual, expect)`.
    - Lastly, we record the status on a table `tbl`.
6. Close the connection: `db.close()`.
    - Do **NOT** forget to always close the connection.
    - Also do **NOT** forget to close any cursor.
7. Print summary: `print(table(['Check', 'Comment'], tbl))`.

Note that we have a few “printing" to help you know the current status of the test cases. This may help in the case that the code raises an exception or stuck in some infinite loop. The printing are as follows:

- **Before Step 1:** print the test name.
- **After Step 1:** print whitespace.
- **After Step 2:** print a light box character.
- **After Step 4:** print a medium box character.
- **After *EACH* Test Case in Step 5**: print heavy box character.
- **After Step 6:** Print black box character.

Checking

Equality Checking

The essence of the checking if a resulting table is the same is as follows:

- **Unordered:**
  1. Check if there is a result: `type(act) != tuple`.
     - If the actual result `act` is not a `tuple`, it means there is no result (*e.g.*, an exception is raised instead).
     - Return `'Error'` if there is no result, otherwise, proceed to next step.
  2. Check if the number of rows are the same: `len(act) != len(exp)`.
     - If the actual result `act` has different number of rows from expected result `exp`, it means the test has failed.
     - Return `'Mismatch length'` if they have different number of rows, otherwise, proceed to next step.
  3. Convert to `set` and check: `set(act) != set(exp)`.
     - Return `'Mismatch body'` if they are different, otherwise, proceed to next step.
  4. At this point, they are correct in the case of “unordered".
     - Return `'Correct'`.
- **Ordered:**
  1. Check if there is a result: `type(act) != tuple`.
     - If the actual result `act` is not a `tuple`, it means there is no result (*e.g.*, an exception is raised instead).
     - Return `'Error'` if there is no result, otherwise, proceed to next step.
  2. Check if the number of rows are the same: `len(act) != len(exp)`.
     - If the actual result `act` has different number of rows from expected result `exp`, it means the test has failed.
     - Return `'Mismatch length'` if they have different number of rows, otherwise, proceed to next step.
  3. Do not convert and simply check: `act != exp`.
     - Return `'Mismatch body'` if they are different, otherwise, proceed to next step.
  4. At this point, they are correct in the case of “ordered".
     - Return `'Correct'`.

While technically Step 3 of “Ordered" can be omitted, we kept it for simplicity.

Insertion Checking

In the case of insertion operation, we may want to check if an insertion was successful or not.

- **Successful Insertion**
  - Check for `INSERT 0 1` since all insertion will only insert 1 row.
- **Failed Insertion**
  - Check for `INSERT 0 0` in the case an insertion is prevented.
  - Check exception is raised `type(res) == psycopg.errors.RaiseException` in case an exception is used to prevent insertion.

Both of these appear in the function `insert(res, exp, cmt, tbl)`.

Testing Your Solution

To test your solution, you need to do the following.

1. Unzip `Reset.zip` to get the files listed in this guide.
2. Open `Config.py` and replace `'********'` with your PostgreSQL password.
   - You may also change the other setting if you are not using the default setting.
3. Open `P03.sql` and copy your solution to this file.
4. Open `ProjectChecker.py`, go to the bottom of the file and *uncomment* the test you want to perform.
   - For instance, if you want to test "Trigger 3", then you uncomment the function call to `trigger3(triggers)`.
   - If you are using IDLE, you can perform the following shortcut ALT + 3 (*for Windows*) or CTRL + 3 (*for MacOS*).
5. Execute `ProjectChecker.py` (*e.g.*, press F5).

Ideally, you should pass all the test cases. As we have not finalized the grading, we have not decided on if any partial grading is to be given.