Assignment 2

SQL Programming

Due date: 27.12.22

Submission is in pairs.
Please use hw2's piazza forum for any question you may have.

1. Introduction

You are about to take a lead part in the development of the "**MovieStats**" database, a website that holds information about movies, studios, movie ratings, and more.

In **MovieStats**, users with admin privileges (you), can add a Movie, Actor, Studio and so on.

MovieStats is a smart service that gives you statistics about Movies in general.

Your mission is to design the database and implement the data access layer of the system. Typically, the data access layer facilitates the interaction of other components of the system with the database by providing a simplified API that carries out a predefined desired set of operations. A function in the API may receive business objects as Input arguments. These are regular Python classes that hold special semantic meaning in the context of the application (typically, all other system components are familiar with them). The ZIP file that accompanies this document contains the set of business objects to be considered in the assignment, as well as the full (unimplemented) API. Your job is to implement these functions so that they fulfill their purpose as described below.

Please note:

1. The database design is your responsibility. You may create and modify it as you see fit. You will be graded for your database design, so bad and inefficient design will suffer from points reduction.

2. Every calculation involving the data, like filtering and sorting, must be done by querying the database. You are prohibited from performing any calculations on the data using Python. Furthermore, you cannot define your own classes, your code must be contained in the functions given, except for the case of defining basic functions to avoid code duplication. Additionally, when writing your queries, you may only use the material learned in class.

3. It is recommended to go over the relevant Python files and understand their usage.

4. All provided business classes are implemented with a default constructor and getter\setter to each field.

5. You may not use more than **one** SQL query in each function implementation, not including views. Create/Drop/Clear functions are not included!

2. Business Objects

In this section we describe the business objects to be considered in the assignment.

Critic

Attributes:

Description	Type	Comments
Critic ID	Int	The critic's ID
Name	String	The critic's name

Constraints:

1. IDs are unique across all critics.
2. IDs are positive
3. All attributes are not optional (not null)

Notes:

1. In the class Critic you will find the static function badCritic() that returns an invalid Critic.

Movie

Attributes:

Description	Type	Comments
Movie Name	String	The name of the movie.
Year	Int	The release year of the movie.
Genre	String	The genre of the movie.

Constraints:

1. The combinations of Year and Name are unique across all movies.
2. Movie release year is a positive integer bigger or equal to 1895.
3. Genre values are one of "Drama", "Action", "Comedy", or "Horror".
4. All attributes are not optional (not null).

Notes:

1. In the class Movie you will find the static function badMovie() that returns an invalid Movie.

Actor

Attributes:

Description	Type	Comments
Actor ID	Int	The ID of the Actor.
Name	String	The name of the actor.
Age	Int	The Actor's age.
Height	int	The Actor's Height in cm.

Constraints:

1. IDs are unique across all Actors.
2. IDs, Age and Height are positive (>0) integers.
3. All attributes are not optional (not null).

Notes:

1. In the class Actor you will find the static function badActor() that returns an invalid Actor.

Studio

Attributes:

Description	Type	Comments
Studio ID	Int	The studio's ID.
Name	String	The name of the studio.

Constraints:

1. IDs are unique across all Studios.
2. IDs are positive
3. All attributes are not optional (not null).

Notes:

1. In the class Studio you will find the static function badStudio() that returns an invalid Studio.

3. API

3.1 Return Type

For the return value of the API functions, we have defined the following enum type:
ReturnValue (enum):

- OK
- NOT_EXISTS
- ALREADY_EXISTS
- ERROR
- BAD_PARAMS

In case of conflicting return values, return the one that appears first on each section.

3.2 CRUD API

This part handles the CRUD - Create, Read, Update and Delete operations of the business objects in the database. Implementing this part correctly will lead to easier implementations of the more advanced APIs.

Python's equivalent to NULL is None.
You can assume the arguments to the function will not be None, the inner attributes of the argument might consist of None.

ReturnValue addCritic(critic: Critic)

Adds a critic to the database.
Input: critic to be added.
Output: ReturnValue with the following conditions:

* OK in case of success
* BAD_PARAMS in case of illegal parameters.
* ALREADY_EXISTS if a critic with the same ID already exists.
* ERROR in case of a database error

Critic getCriticProfile(critic_id : Int)

Input: ID of the requested critic.
Output: The critic profile (a Critic python object) in case the critic exists. badCritic () otherwise.

ReturnValue deleteCritic (critic_id : int)

Deletes a **critic** from the database.
Deleting a **critic** will delete it from everywhere as if it never existed.
Input: the ID of the critic to be deleted.
Output: ReturnValue with the following conditions:

* OK in case of success.
* NOT_EXISTS if the critic does not exist.
* ERROR in case of a database error

ReturnValue addMovie(movie : Movie)

Adds a **movie** to the database.
Input: movie to be added.
Output: ReturnValue with the following conditions:

* OK in case of success
* BAD_PARAMS in case of illegal parameters.
* ALREADY_EXISTS if a movie with the same name and year already exists.
* ERROR in case of a database error

Movie getMovieProfile(name : str, year : int)

Input: name and year of the requested movie.
Output: The movie profile (a Movie python object) in case the movie exists. badMovie() otherwise.

ReturnValue deleteMovie (name : str, year : int)

Deletes a **movie** from the database.
Deleting a **movie** will delete it from everywhere as if it never existed.
Input: the name and year of the movie to be deleted.
Output: ReturnValue with the following conditions:

* OK in case of success.
* NOT_EXISTS if the movie does not exist.
* ERROR in case of a database error

ReturnValue addActor(actor : Actor)

Adds an **actor** to the database.
Input: Actor to be added.
Output: ReturnValue with the following conditions:

* OK in case of success
* BAD_PARAMS in case of illegal parameters.
* ALREADY_EXISTS if an actor with the same ID already exists.
* ERROR in case of a database error

Actor getActorProfile(actor_id : Int)

Input: ID of the requested actor.
Output: The actor profile (an Actor python object) in case the actor exists. badActor() otherwise.

ReturnValue deleteActor (actor_id : int)

Deletes an **actor** from the database.
Deleting an **actor** will delete it from everywhere as if it never existed.
Input: the ID of the actor to be deleted.
Output: ReturnValue with the following conditions:

* OK in case of success.
* NOT_EXISTS if the actor does not exist.
* ERROR in case of a database error

ReturnValue addStudio(studio : Studio)

Adds a **studio** to the database.
Input: studio to be added.
Output: ReturnValue with the following conditions:

* OK in case of success
* BAD_PARAMS in case of illegal parameters.

* ALREADY_EXISTS if a studio with the same ID already exists.
* ERROR in case of a database error

Movie getStudioProfile(studio_id: int)

Input: ID of the requested Studio.
Output: The studio profile (a Studio python object) in case the studio exists. badStudio () otherwise.

ReturnValue deleteStudio (studio_id : int)

Deletes a **studio** from the database.
Deleting a **studio** will delete it from everywhere as if it never existed.
Input: the ID of the studio to be deleted.
Output: ReturnValue with the following conditions:

* OK in case of success.
* NOT_EXISTS if the studio does not exist.
* ERROR in case of a database error

You may not use getProfile() functions in your implementation, all must be done via SQL.

3.3 Basic API

ReturnValue criticRatedMovie(movie_name: str, movie_year: int, critic_id: int, rating: int)

The **critic** rated the **movie**, and gave it **rating** stars.
Input: The name and year of the **movie**, the ID of the **critic** and the rating.
Output: ReturnValue with the following conditions:

* OK in case of success.
* BAD_PARAMS in case rating is not between 1 and 5.
* NOT_EXISTS if movie/critic does not exist.
* ALREADY_EXISTS if the critic already rated this movie.
* ERROR in case of a database error.

ReturnValue criticDidntRateMovie(movie_name: str, movie_year: int, critic_id: int)

The **critic** did not rate the **movie** and its record should be removed from the database.
Input: The name and year of the **movie**, the ID of the **critic**
Output: ReturnValue with the following conditions:

* OK in case of success.
* NOT_EXISTS if movie/critic does not exist or if the critic didn't already rate this movie.
* ERROR in case of a database error.

ReturnValue actorPlayedInMovie(movie_name : str, movie_year : int, actor_id: int, salary: int, roles: List[str])

The **actor** played the listed roles in the **movie**, and was paid **salary** dollars.
Input: The name and year of the **movie**, the ID of the **actor**, salary and list of roles.
Output: ReturnValue with the following conditions:

* OK in case of success.
* BAD_PARAMS in case salary is not positive (>0), if any of the roles is None or if roles is an empty list.
* NOT_EXISTS if movie/actor does not exist.
* ALREADY_EXISTS if the actor already plays in this movie.
* ERROR in case of a database error.

IMPORTANT NOTE: as an exception, in **actorPlayedInMovie** you may use **two** queries for supporting the multiple roles. We suggest executing them at once (in one command, separated by ;) for easier error maintenance.

ReturnValue actorDidntPlayInMovie (movie_name: str, movie_year: int, actor_id: int)

The **actor** did not play in **movie**, if the actor did, delete its record along with all the roles.
Input: The name and year of the **movie** and the ID of the **actor**.
Output: ReturnValue with the following conditions:

* OK in case of success.
* NOT_EXISTS if movie/actor does not exist or actor did not already play in the movie.
* ERROR in case of a database error.

List<str> getActorsRoleInMovie(actor_id : int, movie_name : str, movieYear :int):

Input: id of the actor, name and year of the movie
Output: a list of roles the actor played in the movie, sorted lexicographically in descending order.
If the actor did not play in the movie, either actor or movie don't exist or in case of any other error, return an empty list

NOTE:

This function was not included in the original assignment and was added later as a bonus. There will be 105 points instead of 100 in the automated tests, with this question worth an additional five points. Submissions scoring over 100 points will receive a final grade of 100.

ReturnValue studioProducedMovie(studio_id : int, movie_name: str, movieYear: int, budget : int, revenue : int)

The **studio** provided **budget** dollars for the production of the **movie**. In box office sales, **revenue** dollars were made.
Input: The name and year of the **movie**, ID of the **studio**, **budget** and **revenue**.
Output: ReturnValue with the following conditions:

* OK in case of success.
* BAD_PARAMS in case budget or revenue are negative (<0).
* ALREADY_EXISTS if the movie is already produced by any studio.
* NOT_EXISTS if movie/studio does not exist.
* ERROR in case of a database error.

ReturnValue studioDidntProduceMovie(studio_id : int, movie_name: str, movie_year: int, budget : int, revenue : int)

The **studio** did not produce the **movie**, and the record of that should be removed from the database.
Input: The name and year of the **movie** and the ID of the **studio**.
Output: ReturnValue with the following conditions:

* OK in case of success.
* NOT_EXISTS if movie/studio does not exist or if the **movie** is not already produced by the **studio**.
* ERROR in case of a database error.

Float averageRating(movie_name: str, movie_year: int)

Returns the average rating of the **movie**.
Input: The name and year of the **movie** which we want the average rating of.
Output:

* The average rating in case of success.
* 0 in case of division by 0, if the movie does not exist or in case of other error.

Float averageActorRating(actor_id: int)

Input: ID of the actor
Output: the average of average ratings of movies in which the actor played,
or 0 if the actor did not play in any movie.
if any movie has no ratings, it is counted as having average rating of 0.

In case the actor does not exist, or have not played in any movies with ratings, return 0.

Example:
The movie "Titanic", released in 1997 was rated by 10 critics, each of them gave it 5 stars.
The movie "Inception", released in 2010 was rated by 1 critic, and was given a 1-star review.
The movie "Shutter Island", released in 2010 was not rated by any critic.

Leonardo DiCaprio played in all 3 movies, so his average rating is 2.

Movie bestPerformance(actor_id : int)

Input: ID of the actor
Output: Movie object of the best rated (by average rating) movie the actor has played in.
If multiple movies share the highest average rating, tie breaker is done by choosing the earlier release, and for movies released in the same year choosing the movie with greater name (lexicographically)
In case the actor doesn't exist or did not play in any movies, return badMovie().

Int stageCrewBudget(movie_name: str, movie_year: int)

Input: name and year of the movie
Output: the difference between the budget of the movie and the sum of salaries of actors who play in the movie. Movies that was not produced by any studio, are considered to have a budget of 0.
In case the movie does not exist, return -1.

Bool overlyInvestedInMovie(movie_name: str, movie_year: int, actor_id: int)

Input: The name and year of the **movie** and the ID of the **actor.**

Output:

* Returns True if the **actor** with actor_id plays at least half of all the roles in the **movie**. False otherwise

* Returns False if either the movie or actor do not exist, or if the actor does not play any role in the movie.

For example:

In the movie "Austin Powers" released in 2002, mike Myers plays the roles of Austin powers, doctor Evil, and Goldmember - 3 roles in total.

Beyonce plays 1 role, Foxy Cleopatra.

Tom Cruise plays 2 roles, as Himself (the role of 'Tom Cruise') and Famous Austin.

Given that Mike Myers ID is 1234

overlyInvestedInMovie("Austin Powers", 2002, 1234) will return true, because Mike Myers plays 3 of 6 total roles.

3.4 **Advanced API**

list<(string, int)> franchiseRevenue():

Input: None

Output: list of (movie_name, total_revenue). Where total_revenue is the sum of all revenues movies with movie_name made for studios. If a movie was not produced by any studio, its revenue should be counted as 0.
the movies should be ordered by name in descending order.

Example:

Sony produced the movie "Spiderman" in 2002, and made 825,000,000 dollars in revenue.
Marvel produced the movie "Spiderman" in 2017, and made 880,000,000 dollars in revenue.
Marvel also produced the movie "Avengers" in 2019 and made 2,720,000,000 dollars in revenue.
Marvel also produced the movie "Captain Marvel" in 20019 and made 1,120,000,000 dollars in revenue.

So franchiseRevenue() will return this list:

[("Spiderman", 1705000,000),
("Captain Marvel", 1120000000),
("Avengers", 2720000000)]

list<(string, int)> studioRevenueByYear():

Input: None

Output:

list of (studio_id, year, total_revenue). Where total_revenue is the sum of all revenues movies with made for the studio with studio_id during that year.

The tuples should be ordered by studio_id in descending order, and tuples with the same studios should be ordered by year in descending order.

Example:

In our database, sony studio has studio_id of 1, and Marvel studio has studio_id of 2

Sony produced the movie “Spiderman” in 2002, and made 825,000,000 dollars in revenue.
Marvel produced the movie “Spiderman” in 2017, and made 880,000,000 dollars in revenue.
Marvel also produced the movie “Avengers” in 2019 and made 2,720,000,000 dollars in revenue.
Marvel also produced the movie “Captain Marvel” in 20019 and made 1,120,000,000 dollars in revenue.

So studioRevenueByYear () will return this list:

[(2, 2019, 3840000000),
(2, 2007, 880000000),
(1, 2002, 825000000)]

We will define a critic to be a fan of a studio, if he rated every movie produced by the studio.

list<(int, int)> getFanCritics():

Input: None

Output: list of (critic_id, studio_id) where the critic with critic_id is a fan of studio with studio_id

The list should be ordered by CriticID in descending order, and tuples with the same critic should be ordered by StudioID in descending order.

list<(string, float)> averageAgeByGenre():

Input: None

Output: list of (genre, average_age) where average_age is the average age of actors who play in at least one movie of the genre.

NOTE: an actor might play in multiple movies from the same genre, they still should only be counted once.

list<(int, int)> getExclusiveActors():

Input: None
Output: a list of (actor_id, studio_id) where the actor with actor_id played only in movies Produced by studio with Studio_id.
The list should be ordered by actor_id in descending order.

Example:

In our database, Paramount Studio_id is 0, and Universal Studio_id is 1.

Paramount studio produced the movies:
"Grease" from 1978, "Top Gun" from 1986, "Top Gun" from 2022, and "Titanic" from 1997

Universal studio produced the movies:
"The aviator" from 2004 and "HulK" from 2003.

In our database we have another movie that is not produces by any studio.
This movie is "Pulp Fiction" from 1994. (this type of movie is called an "Indie movie")

In our database we have 5 actors with the following Ids:
0: Jhon Travolta
1: Tom Cruise
2: Leonardo DiCaprio
4: Stan Lee
5: Uma Thurman

Jhon Travolta is playing in Grease as "Danny" and in Pulp Fiction as "Vincent Vega"
He is not exclusive Paramount because he is playing in a Movie that was not produced by Paramount

Tom Cruise is playing in both Top Gun movies as "L.T Pitt Maverick".
He is exclusive to Paramount studio because he plays in no other films.

Leonardo DiCaprio is playing in Titanic as "Jack" and in The Aviator as "Howard Hughes"
He is not exclusive to any studio because he plays in films produced by both.

Stan lee is playing in Hulk as "security guard #2"
He is exclusive to Universal studios because he is not playing in other movies in our database.

Uma Thurman is playing in Pulp Fiction as "Mia Wallace"
She is not exclusive to any studio because she didn't play in any movie that was produced by one.

4. Database

6.1 Basic Database functions

In addition to the above, you should also implement the following functions:

void createTables()
Creates the tables and views for your solution.

void clearTables()
Clears the tables for your solution (leaves tables in place but without any data).

void dropTables()
Drops the tables and views from the DB.

Make sure to implement them correctly.

6.2 Connecting to the Database using Python

Each of you should download, install and run a local PosgtreSQL server from https://www.postgresql.org. You may find the guide provided helpful.

To connect to that server, we have implemented for you the DBConnector class that creates a *Connection* instance that you should work with to interact with the database.

For establishing successful connection with the database, you should provide a proper configuration file to be located under the folder Utility of the project. A default configuration file has already been provided to you under the name database.ini. Its content is the following:

[postgresql]
host=localhost
database=postgres
user=postgres
password=password
port=5432

Make sure that port (default: 5432), database name (default: cs236363), username (default: username), and password (default: password) are those you specified when setting up the database.

To get the Connection instance, you should create an object using conn = Connector.DBConnector() (after importing "import Utility.DBConnector as Connector" as in Example.py). To submit a query to your database, simply perform conn.execute("query here"). This will return a tuple of (number of rows affected, results in case of SELECT).
Make sure to close your session using .close().

6.3 SQL Exceptions

When preparing or executing a query, an SQL Exception might be thrown. It is thus needed to use the try/catch (try/except in python) mechanism to handle the exception. For your convenience, the DatabaseException enum type has been provided to you. It captures the error codes that can be returned by the database due to error or inappropriate use. The codes are listed here:

NOT_NULL_VIOLATION (23502), *FOREIGN_KEY_VIOLATION*(23503), *UNIQUE_VIOLATION*(23505), *CHECK_VIOLIATION* (23514);

To check the returned error code, the following code should be used inside the except block: (here we check whether the error code *CHECK_VIOLIATION* has been returned)

```
except DatabaseException.CHECK_VIOLATION as e:
    # Do stuff
```

Notice you can print more details about your errors using print(e).

Tips

1. Create auxiliary functions that convert a record of ResultSet to an instance of the corresponding business object.
2. Use the enum type DatabaseException. It is highly recommended to use the exceptions mechanism to validate Input, rather than use Python's “if else”.
3. Devise a convenient database design for you to work with.
4. Before you start programming, think which Views you should define to avoid code duplication and make your queries readable and maintainable.

   (Think which sub-queries appear in multiple queries).
5. Use the constraints mechanisms taught in class to maintain a consistent database. Use the enum type DatabaseException in case of violation of the given constraints.
6. Remember - you are also graded on your database design (tables, views).
7. Please review and run Example.py for additional information and implementation methods.
8. AGAIN, USE VIEWS!

Submission

Please submit the following:

A zip file named <id1>-<id2>.zip (for example 123456789-987654321.zip) that contains the following files:

1. The file Solution.py where all your code should be written in (your code will also go through dry exam).

2. The file <id1>_<id2>.pdf in which you explain in detail your database design and the implantation of the API (each function and view). Is it **NOT** required to draw a formal ERD but it is indeed important to explain every design decision and it is highly recommended to include a draw of the design (again, it is **NOT** required to draw a formal ERD).

Note that you can use the unit tests framework (unittest) as explained in detail in the PDF about installing IDE, but no unit test should be submitted.

Good Luck!