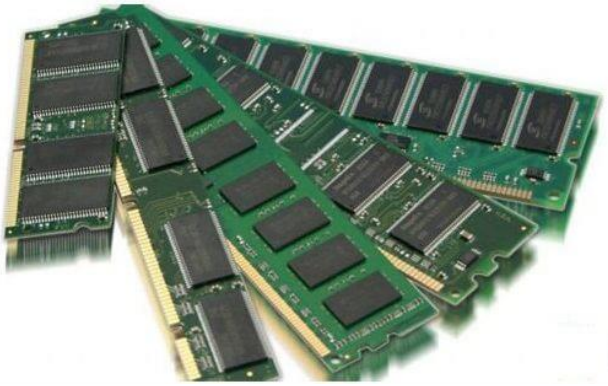
Assignment 2

SQL Programming

Due date: 3.6.21



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 **"Queries"** database, a website that holds information about queries, disks, and RAMs available for use.

In **Queries**, users with admin privileges (you), can add a query, add an available disk, RAM and so on.

Queries is a smart service that gives you statistics about queries, disks and RAMs.

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 with an exception to 'COALESCE' (it is not mandatory). Explanation can be found under https://www.w3schools.com/sql/func_sqlserver_coalesce.asp.

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/transaction in each function implementation, including views. Create/Drop/Clear functions are included!

2. Business Objects

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

Query

Attributes:

Description	Type	Comments
Query's ID	Int	The query id.
Purpose	String	What does the query do?
Disk size needed (in bytes)	Int	The amount of free disk space needed to perform.

Constraints:

1. ID is unique across all queries.
2. IDs are positive, Disk size needed is not negative (>=0).
3. All attributes are not optional (not null).

Notes:

1. In the class Query you will find the static function badQuery() that returns an invalid query.

Disk

Attributes:

Description	Type	Comments
Disk ID	Int	The ID of the disk.
Manufacturing company	String	The name of manufacturing company
Speed	Int	Disk's speed.
Free space	Int	The amount of free space left on disk.
Cost per byte	Int	Cost of using one byte on this disk.

Constraints:

1. IDs are unique across all disks.
2. IDs, speed and cost are positive (>0) integers and free space is not negative.
3. All attributes are not optional (not null).

Notes:

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

RAM

Attributes:

Description	Type	Comments
RAM ID	Int	The ID of the RAM.
Size	Int	The amount of space on disk.
Company	String	Manufacturing company.

Constraints:

1. IDs are unique across all RAMS.
2. IDs and size are positive (>0) integers.
3. All attributes are not optional (not null).

Notes:

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

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 addQuery(Query query)

Adds a **query** to the database.
Input: query 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 query with the same ID already exists.
* ERROR in case of a database error

Query getQueryProfile(Int queryID)

Returns the query profile of
queryID.

Input: Query ID.
Output: The query profile (a query object) in case the query exists. BadQuery() otherwise.

ReturnValue deleteQuery (Query query)

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

* OK in case of success or if query does not exist (ID wise).
* ERROR in case of a database error

Note: do not forget to adjust the free space on disk if the query runs on one. Hint - think about transactions in such cases (there are more in this assignment.

ReturnValue addDisk (Disk disk)

Adds a disk to the database.
Input: disk 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 disk with the same ID already exists.
* ERROR in case of a database error

Disk getDiskProfile (Int diskID)

Returns the disk with diskID as its id.

Input: disk id.
Output: The student with diskID if exists. BadDisk() otherwise.

ReturnValue deleteDisk(Int diskID)

Deletes a disk from the database.
Deleting a disk will delete it from <u>everywhere</u> as if he/she never existed.
Input: disk ID to be deleted.
Output: ReturnValue with the following conditions:

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

ReturnValue addRAM (RAM ram)

Adds a RAM to the database.
Input: RAM 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 RAM with the same ID already exists.
* ERROR in case of a database error

RAM getRAMProfile (Int RAMID)

Returns the RAM with RAMID as its id.

Input: RAM id.
Output: The RAM with RAMID if exists. BadRAM() otherwise.

ReturnValue deleteRAM (Int RAMID)

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

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

ReturnValue addDiskAndQuery (Disk disk, Query query)

Adds a both disk and query to the database.
Input: disk and query to be added.
Output: ReturnValue with the following conditions:

* OK in case of success
* ~~BAD_PARAMS in case of illegal parameters of either of the objects.~~
* ALREADY_EXISTS if a disk/query with the same ID already exists.
* ERROR in case of a database error

Note: in case of failure of one of the queries, **the all** operation must be aborted. You can assume the parameters are 'legal'.

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

3.3 Basic API

ReturnValue addQueryToDisk(Query query, Int diskID)

The **query** with queryID is now executing on disk with **diskID** only if query's size is not larger than free space on disk.
Input: The query that wishes to run on disk with **diskID**.
Output: ReturnValue with the following conditions:

* OK in case of success.
* NOT_EXISTS if query/disk does not exist.
* ALREADY_EXISTS if the query already executing on the disk.
* BAD_PARAMS in case query's size is larger than free space on disk.
* ERROR in case of a database error

Note: do not forget to adjust the free space on disk.

ReturnValue removeQueryFromDisk (Query query, Int diskID)

The **query** with queryID is now removed from the disk with **diskID**.
Input: The **query** with queryID to remove from disk with **diskID**.
Output: ReturnValue with the following conditions:

* OK in case of success (also if query/disk does not exist or query does not run-on disk).
* ERROR in case of a database error

Note: do not forget to adjust the free space on disk.

ReturnValue addRAMToDisk(Int ramID, Int diskID)

The RAM with **ramID** is now a part of the disk with **diskID**.
Input: The RAM with **ramID** which is now a part of the disk with **diskID**.
Output: ReturnValue with the following conditions:

* OK in case of success.
* NOT_EXISTS if RAM/disk does not exist.
* ALREADY_EXISTS if the RAM already a part of the disk.
* ERROR in case of a database error

ReturnValue removeRAMFromDisk (Int ramID, Int diskID)

The RAM with **ramID** is now removed from the disk with **diskID**.
Input: The RAM with **ramID** to remove from disk with **diskID**.
Output: ReturnValue with the following conditions:

* OK in case of success
* NOT_EXISTS if RAM/disk does not exist or RAM is not a part of disk.
* ERROR in case of a database error

Float averageSizeQueriesOnDisk(Int DiskID)

Returns the average size of the queries running on the disk with **diskID**.
Input: disk's ID.
Output:

* The average size in case of success.
* 0 in case of division by 0 ~~(it is the default for AVG)~~ or ID does not exist, -1 in case of other error.

Int diskTotalRAM(Int diskID)

Returns the total amount of RAM available on **diskID**.
Input: diskID of the requested disk.
Output:

* The sum in case of success.
* 0 if the disk does not exist,-1 in case of an error.

Int getCostForPurpose(String purpose)

Returns the total amount of money paid for running queries with **purpose** across all disks (money paid = cost per unit * size).
Input: the cost of the requested **purpose**.
Output:

* The sum in case of success.
* 0 if the **purpose** does not exist, -1 in case of an error.

List<Int> getQueriesCanBeAddedToDisk(Int diskID)
Returns a List (up to size 5) of queries' IDs that can be added to the disk with **diskID** as singles, not all together (even if it already does).
The list should be ordered by IDs in descending order.
Input: The **diskID** in question.
Output:

* List with the queries' IDs.
* Empty List in any other case.

List<Int> getQueriesCanBeAddedToDiskAndRAM(Int diskID)
Returns a List (up to size 5) of queries' IDs that can be added to the disk with **diskID** as singles, not all together (even if it already does) and can also fit in the sum of all the RAMs that belong to the disk with **diskID.**
The list should be ordered by IDs in ascending order.
Input: The **diskID** in question.
Output:

* List with the queries' IDs.
* Empty List in any other case.

Bool isCompanyExclusive(Int diskID)
Returns whether the disk with **diskID** is manufactured by the same company as all its RAMs.

Input: the **diskID**.
Output:

* The result in case of success.
* False in case of an error or the disk does not exist.

3.4 Advanced API

Note: In any of the following functions, if you are required to return a list of size X but there are less than X results, return a shorter list which contains the relevant results.

List<Int> getConflictingDisks()

Returns a list containing conflicting disks' IDs (no duplicates).

Disks are conflicting if and only if they run at least one identical query.

The list should be ordered by diskIDs in ascending order.

Input: None

Output:

*List with the disks' IDs.

*Empty List in any other case.

List<Int> mostAvailableDisks()

Returns a list of up to 5 disks' IDs that can execute the most queries (as singles).

A disk can execute a query if and only if the query's size is not larger than the free space on disk (even if it already does).

The list should be ordered by:

- **Main sort by number of queries in descending order.**
- **Secondary sort by disk's speed in descending order.**
- **Final sort by diskID in ascending order.**

Input: None

Output:

*List with the disks' IDs that satisfy the conditions above (if there are less than 5 disks, return a List with the <5 disks).

*Empty List in any other case.

List<Int> getCloseQueries (Int queryID)

Returns a list of the 10 "close queries" to the query with **queryID**.

Close queries are defined as queries who run on at least (>=) 50% of the disks the query with **queryID** does. Note that query cannot be a close query of itself.

The list should be ordered by IDs in ascending order.

Input: The ID of a query.

Output:

*List with the queries' IDs that meet the conditions described above (if there are less than 10 queries, return a List with the <10 queries).

*Empty List in any other case.

Note: queries can be close in an empty way (query in question does not run on any disk).

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=cs236363
user=username
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") (~~DO NOT forget to use prepared statement in case of user input~~). This will return a tuple of (number of rows affected, results in case of SELECT). Do not forget to commit or abort (rollback) your changes. Also 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.

2. The file <id1>_<id2>.pdf in which you explain in detail your database design and the implantation of the API. 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).

3. The file <id1>_<id2>.txt with nothing inside.

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.

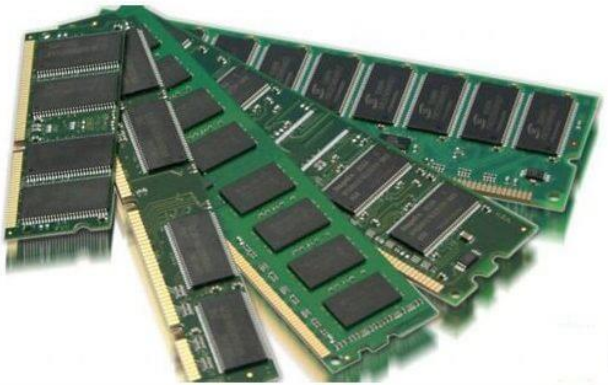
Make sure that is the exact content of the zip with no extra files/directories and no typos by using:

'python check_submission.py <id1>-<id2>.zip'

(script and zip in the same directory).

Any other type of submission will fail the automated tests and result in 0 on the wet part, which is 50% of the total grade (your code will also go through dry exam).

You will not have an option to resubmit in that case!



Good Luck!