Database Systems 236363,Spring 2024

Homework 2

Sql programming

- TA in charge: Gal Kesten
- Due date: 28/07/2024
- Questions should be asked in the dedicated forum in the course's Piazza: https://piazza.com/technion.ac.il/spring2024/236363
- Administrative questions should be directed to Shir Rotman
- Submission is in pairs.

1. Introduction

In this assignment, you will develop part of a delivery management application for a new restaurant called "Yummy". Due to the high costs associated with using third-party apps to manage restaurant orders, the owner of Yummy has decided to build a dedicated app for managing the restaurant's orders and needs your help.

Customers of the app will be able to place orders and save their favorite dishes. Managers of the app will be able to add new dishes to the menu, update prices, and track the profits of the restaurant.

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 designs 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, and your code must be contained in the functions given, except for the case of defining basic functions to avoid code duplication. Additionally, you may only use the material learned in class when writing your queries.
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 can use only **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.

- **You can assume that an empty string (not a null string) is valid unless otherwise specified.**
- **You can assume strings can have unlimited length unless otherwise specified.**

Customer

Attributes:

Description	Type	Comments
cust_id	int	The customer's ID.
full_name	string	The customer's name.
phone	string	The customer's phone number.
address	string	The customer's address

Constraints:

1. cust_id is unique across all customers
2. cust_id is positive (>0)
3. address must contain at least 3 characters.
4. All attributes are not optional (not null).

Order

Attributes:

Description	Type	Comments
order_id	int	The order ID.
date	timestamp	The date of the order

Constraints:

1. order_id is unique across all orders
2. order_id is positive (>0)
3. All attributes are not optional (not null).
4. The timestamp should be without time zone and precision should be limited only to seconds (no microseconds).

Dish

Attributes:

Description	Type	Comments
dish_id	int	The dish ID
name	string	The name of the dish.
price	decimal	The price of the dish.
is_active	boolean	Indicates if the dish is currently offered

Constraints:

1. dish_id is unique across all dishes
2. dish_id is positive (>0)
3. price is positive (> 0)
4. name of the dish cannot be empty and must contain at least 3 characters.
5. All attributes are not optional (not null).

3. Dates in SQL and Python

3.1 Dates in SQL

PostgreSQL provides several data types for storing date and time information. One of the most versatile is TIMESTAMP, which stores both the date and time.

Creating a TIMESTAMP that represents the 6th of May 2023 at 14:30:00

This line returns a table with a single row and a single column that has type timestamp and a value of the date 6/5/2023.

```
SELECT TIMESTAMP '2023-05-06 14:30:00'
```

The returned table:

	timestamp timestamp without time zone
1	2023-05-06 14:30:00

You can extract individual components from a timestamp using the EXTRACT function.

This line returns a table with a single row and a single column that has type numeric(also called decimal) and a value of the day from the given timestamp:

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2023-05-06 14:30:00');
```

The returned table:

	extract numeric
1	6

The next two lines will return similar tables with values of month/year/hour/minute/second

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2023-05-06 14:30:00');
SELECT EXTRACT(YEAR FROM TIMESTAMP '2023-05-06 14:30:00');
SELECT EXTRACT(HOUR FROM TIMESTAMP '2023-05-06 14:30:00');
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2023-05-06 14:30:00');
SELECT EXTRACT(SECOND FROM TIMESTAMP '2023-05-06 14:30:00');
```

3.2 Dates in Python

This assignment will be done in python. Because you will be working with timestamps, you will need to use the datetime object from the datetime module in python.

How to use date:

Import:

```
from datetime import datetime
```

Basic constructor:

```
# format: datetime(year: int, month: int, day: int, hour: int, minute: int, second: int)
# example: 24/12/2023 14:30:00
dt = datetime(2023, 12, 24, 14, 30, 0)
```

Basic getters:

```
print(dt.year)   # prints "2023"
print(dt.month)  # prints "12"
print(dt.day)    # prints "24"
print(dt.hour)   # prints "14"
print(dt.minute) # prints "30"
print(dt.second) # prints "0"
```

date.strftime:

A function for getting a string of the datetime object in a specified format. The function uses codes for different representations of year, month, day and time.

A link to a list of all codes: Python strftime reference cheatsheet

Usage example:

```
print(dt.strftime('%Y-%m-%d %H:%M:%S'))  # prints "2023-12-24 14:30:00"
print(dt.strftime('%A %B %d %Y %I:%M %p'))  # prints "Sunday December 24 2023 02:30 PM"
```

datetime.strptime:

This function is the inverse of strftime. It takes a format and a string and converts it into a datetime object. It uses the same codes as strftime.

Usage example:

```

t = datetime.strptime('2023-12-24 14:30:00', '%Y-%m-%d %H:%M:%S')
print(t)  # prints "2023-12-24 14:30:00"
print(type(t))  # prints "<class 'datetime.datetime'>"
```

Further reading: https://docs.python.org/3.11/library/datetime.html

4. API

In this part you will implement the API for the system.

You should use views whenever possible to avoid code duplication. Points will be deducted for code duplication where views could have been used.

Note: For the functions 'get_order_total_price' and 'get_max_amount_of_money_cust_spent', you must utilize a single view that will serve both functions.

4.1 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 one of them.

4.2 CRUD API

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

While implementing the CRUD API, you should remember that an integral functionality of the system is to track orders and calculate monthly and yearly profits of the restaurant. This requirement may pose challenges when deleting certain objects from the system. Therefore, you should think carefully about your database design.

You will notice that there isn't a function for deleting dishes in the API. This approach was chosen intentionally, as it aligns with the common practice of avoiding the deletion of objects that may affect other objects. This approach can be taken when the business "owns" the information. However, this logic cannot be applied to all objects in the system. For instance, customers should have the ability to remove their personal information.

Notes:

- Python's equivalent to NULL is None.
- You can assume that the arguments to the functions will not be `None` (for example, if an argument is specified as an `int`, we won't send a `None` value). However, the inner attributes of the arguments, such as those within classes, might consist of `None`.

ReturnValue add_customer(customer: Customer)

Add a customer to the database.

Input: A Customer object

Output: ReturnValue with the following conditions:

- OK in case of success
- BAD_PARAMS if any of the parameters are illegal (based on constraints specified above)
- ALREADY_EXISTS if a customer with the same ID already exists
- ERROR in case of a database error

Customer get_customer(customer_id: int)

Get a customer from the database.

Input: The ID of the requested customer

Output: The object of the requested customer if the customer exists, BadCustomer otherwise

ReturnValue delete_customer(customer_id: int)

Delete a customer from the database.

Deleting a customer will remove all their details from the database as if they never existed. However, any orders placed by the customer should not be deleted, allowing the restaurant to track its profits even after the customer is removed.

Input: The ID of the customer to delet

Output: ReturnValue with the following conditions:

- OK in case of success
- NOT_EXISTS if the customer does not exist (also for illegal id)
- ERROR in case of a database error

ReturnValue add_order(order: Order)

Add an order to the database.

Input: An Order object

Output:

ReturnValue with the following conditions:

- OK in case of success
- BAD_PARAMS if any of the parameters are invalid (based on the constraints specified above).
- ALREADY_EXISTS if an order with the same ID already exists
- ERROR in case of database error

Notes:

If you receive an order with a timestamp that includes microseconds and the order is otherwise valid, you should be able to add the order to the database and return ok (assuming your implementation correctly handles the microseconds part of the timestamp).

Order get_order(order_id: int)

Get an order from the database.

Input: The ID of the requested order

Output: The object of the requested order if the order exists, BadOrder otherwise

ReturnValue delete_order(order_id: int)

Delete an order from the database.

Deleting an order will delete it from everywhere as if it never existed.

Input: The id of the order to delete

Output:

ReturnValue with the following conditions:

- OK in case of success
- NOT_EXISTS if the order does not exist (also for illegal id)
- ERROR in case of a database error

ReturnValue add_dish(dish: Dish)

Add a dish to the database.

Input: A Dish object

Output: ReturnValue with the following conditions:

- OK in case of success
- BAD_PARAMS if any of the params are illegal (based on constraints specified above)
- ALREADY_EXISTS if a dish with the same ID already exists
- ERROR in case of database error

Dish get_dish(dish_id: int)

Get a dish from the database.

Input: The id of the requested dish

Output: The object of the requested dish if the dish exists, BadDish otherwise

ReturnValue update_dish_price(dish_id: int, price: float)

Update the price of the dish with the given dish_id.

Note: You should only update the current price of the dish, not the price of the dish in previous orders.

Input:

- dish_id: The id of the dish
- price: The new price of the dish

Output:

- OK in case of success
- BAD_PARAMS if the price is illegal (based on what mentioned before)
- NOT_EXISTS if the dish does not exist in the system (also for illegal dish id) or the dish is not active
- ERROR in case of database error

ReturnValue update_dish_active_status(dish_id: int, is_active: bool)

Update the status of a dish with the given dish_id to either active or inactive.

Input:

- dish_id: The id of the dish
- is_active: A boolean value indicating the desired status of the dish. `True` to mark the dish as active, `False` to mark the dish as inactive.

Output:

- OK in case of success
- NOT_EXISTS if the dish does not exist in the system (also for illegal dish_id)
- ERROR in case of database error

ReturnValue customer_placed_order(customer_id: int, order_id: int)

Customer specified by customer_id has placed order with the given order_id

Note: Each order can be associated with at most one customer.

Input:

- customer_id: The id of the customer that placed the order
- order_id: The id of the order

Output:

- OK in case of success
- ALREADY EXISTS If the order is already related to a customer (whether it is the same customer or another one).
- NOT_EXISTS if the customer or the order don't exist (also for illegal ids)
- ERROR in case of database error

customer get_customer_that_placed_order(order_id: int)

Get the customer who placed the order if such a customer exists.

Input:

- order_id: The id of the order

Output:

The object of the requested customer if there is a customer related to the order, BadCustomer otherwise.

ReturnValue order_contains_dish (order_id: int , dish_id: int, amount: int)

Add the dish specified by dish_id to the order specified by order_id, along with the amount ordered and the dish's price at the time of ordering.

Notes:

- You should use the price of the dish stored in the system in that date (think about the structure of insert into you need to use to get this data)
- You should check the active status of the dish on the date the order was placed.
- Each order can contain multiple dishes but each dish should be specified only once with the amount that was ordered per order.

Input:

- order_id: The id of the order
- dish_id: The id of the dish
- amount: The amount that was ordered from that dish

Output:

- OK in case of success
- BAD_PARAMS if amount < 0
- ALREADY_EXISTS if the dish is already specified in the order.
- NOT_EXISTS if the order or the dish don't exist or if the dish is not active (also for illegal ids)
- ERROR in case of database error

Please do not forget to save the price of the dish along with the amount, otherwise points will be deducted.

ReturnValue order_does_not contain_dish (order_id: int , dish_id: int)

Remove the dish with the given dish_id from the order with the given order_id.

Input:

- order_id: The id of the order
- dish_id: The id of the dish

Output:

- OK in case of success
- NOT_EXISTS if the order does not contain the dish (also for illegal ids)
- ERROR in case of database error

List[OrderDish] get_all_order_items(order_id: int)

Get a list of all the dishes that the order with the given order_id contains.

Input:

- order_id : the id of the order

Output:

- A list of distinct OrderDish objects of all the dishes the order contains, ordered by dish_id ascending. If the order does not exist or the order does not have dishes, return an empty list.

ReturnValue customer_likes_dish(cust_id: int, dish_id: int)

Customer with the given cust_id likes the dish with the given dish_id

Input:

- dish_id: the id of the dish
- cust_id: the id of the customer

Output:

- OK in case of success
- ALREADY EXISTS if the customer already liked that dish
- NOT_EXISTS if the customer or the dish don't exist in the system (also for illegal ids)
- ERROR in case of database error

ReturnValue customer_dislike_dish(cust_id: int, dish_id: int)

Customer with the given cust_id dislikes the dish with the given dish_id and does not like it anymore.

Input:

- dish_id: The id of the dish
- cust_id : The id of the customer

Output:

- OK in case of success
- NOT_EXISTS if there isn't a record indicates the customer liked the dish (also for illegal ids)
- ERROR in case of database error

List[Dish] get_all_customer_likes(cust_id: int)

Get a list of all the dishes that customer likes

Input:

- cust_id: the id of the customer

Output:

- A list of distinct Dish objects of all the dishes the customer likes, ordered by dish_id ascending. If the customer does not exist, or the customer did not like any dish return empty list.

4.3 Basic API

float get_order_total_price(order_id: int)

Get the total price of the order specified by order_id.

Input: order_id: The id of the order

Output: The total price of the specified order

Note:

- You **must** use a view for this function and for **get_max_amount_of_money_cust_spent**
- You can assume you will get legal order_id and that the order_id exists in the system
- If the order has no dishes related to it, you should return 0.
- Ensure you return a float type. The result you typically get from querying the datasbase is of decimal type, so make sure to convert it to float.

float get_max_amount_of_money_cust_spent(cust_id: int)

Get the maximum amount of money the customer with the given cust_id has ever spent in one order.

Input:

- cust_id: The id of the customer

Output: The maximum amount of money the customer has spent on an order.

Note:

- You **must** use a view for this function and for **get_order_total_price**
- You can assume you will get a valid cust_id that exist in the system.
- If the customer never placed an order you should return 0.
- Ensure you return a float type. The result you typically get from querying the datasbase is of decimal type, so make sure to convert it to float

Order get_most_expensive_anonymous_order()

Get the order with the maximum total price among all the orders that are not related to any customer.

Input: None

Output: The anonymous order with the maximum total price among all anonymous orders.

Note:

- If there is more than one anonymous order with the maximum total price, return the one with the lower Id
- You can assume there will be at least one anonymous order in the system.
- If an order in the system does not contain dishes, its total price should be considered as 0.

bool is_most_liked_dish_equal_to_most_purchased()

Input: None

Output: Returns True if the most liked dish is also the most purchased, else returns false.

Notes:

- Most liked dish: The dish that has the largest number of likes.
- Most purchased dish: The dish that was purchased the largest number of times, considering the amount bought in each order.
- If there are no orders/ no dishes related to orders/ no likes, return False.
- If there is a tie for the most liked or most purchased dish, select the dish with the lower ID.

4.4 Advanced API

List[int] get_customers_ordered_top_5_dishes ()

Get all customers who have ordered all the dishes in the 5 most liked dishes.

Input: None

Output: A list of distinct customers ids who have ordered all the dishes in the 5 most liked dishes, ordered by customer id ascending.

Notes:

- The "top 5 most liked dishes" are defined as the five dishes that have received the highest number of likes. In case of a tie, select the dishes with the lowest dish_id to make up five dishes in total.
- Dishes with no likes are considered with 0 likes for calculation.
- If no customer has ordered all of the top 5 dishes, return an empty list.
- You can assume there will be at least 5 dishes in the system. There will also be existing customers.
-

List[int] get_non_worth_price_increase()

Get all active dishes ids where the current price results in a lower average profit per order compared to a previous (lower) price.

Input: None

Output: A list of distinct dish ids where each dish is active and its current price results in a lower average profit per order. The list should be ordered by dish id ascending.

Notes:

- Average profit per order (per price)**:** The average amount of the dish in each order (among all orders that contains that dish at that price) multiplied by the price.
- To do a comparison for a specific dish the following conditions must be satisfied:
    - There must exist at least one order that contains the dish with the current price of the dish.
    - There must exist another order that contains the dish with a lower price than the current price.
    -

There is an example in the next page

Example:

An active dish with ID 1 currently costs 50 shekels. There are 4 orders containing the dish with the new price:

- Order 1 contains dish 1 with amount 1 and price 50.
- Order 2 contains dish 1 with amount 2 and price 50.
- Order 3 contains dish 1 with amount 2 and price 40.
- Order 4 contains dish 1 with amount 2 and price 40.

The average profit per order for dish 1 at price 50:

avg_amount * 50 = (1+2)/2*50=75

The average profit per order for dish 1 at price 40:

avg_amount * 40 = (2+2)/2*40=80

The current price of dish 1 is 50. The average profit per order for dish 1 for price 50 is 75 shekels. We also see the average profit per order for dish 1 for price 40 was 80. Therefore, we will return dish 1.

List[Tuple[int, floatl]] get_total_profit_per_month(year: int)

Calculate the total profit per month for a specific year across all orders.

I**nput:** year

Output: A list of distinct tuples containing month, and total profit for each month, including months where no profit was made**.**

Each tuple should be of the form (month, profit).

Example tuple: (1, 10000.897) where the first entry is the month, and the second entry is the profit.

The returned list should be ordered by month in **descending** order.

Note:

- If no profit was made in a specific month, the profit should be 0 in that month.
- You can assume year will be a valid year.
- Make sure you covert total profit to float in each returned tuple.

List[int] get_potential_dish_recommendations(cust_id: int)

Get all dish IDs that might be of interest to a given customer based on the dishes liked by similar customers.

Input:

cust_id: The id of the given customer for whom we want to find new dish recommendations.

Output:

A list of distinct dish IDs that the given customer might like. These are dishes liked by similar customers but not yet liked by the given customer.

The list should be ordered by dish id ascending .

Notes:

- Similar Customers: Customers who have liked at least 3 dishes that the given customer has liked.
- Potential Dish Recommendations: Dishes liked by similar customers that the given customer has not yet liked.
- If customer doesn't exist or no dishes/likes, return empty list

Example:

Assume customer 1 has liked dishes with ids 2, 3, 4 and 6.

Customer 2 has liked dishes 1,2,3,4 and 5.

Customer 3 has liked dishes 3,4,5,6 and 7.

Customer 4 has liked dish 2,6 and 7.

- The group of similar users to customer 1 is : {customer 2,customer 3} .
- Customer 2 also liked dishes 1 and 5, and customer 3 also liked dishes 5 and 7. Therefore, customer 1 might like one of the dishes they also liked.
- The result of the query : [1,5,7] (5 appears once)

5. Database

5.1 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 (pay attention to the order you create and delete tables/views).

5.2 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 function will return the number of affected rows and a ResultSet object.
You can find the full implementation of ResultSet in DBConnector.py.
Here are a few examples of how to use it:

```
conn = Connector.DBConnector()
query = "SELECT * FROM customers"
# execute the query, result will be a ResultSet object
rows_affected, result = conn.execute(query)

# get the first tuple in the result as a dictionary, the keys are the column
# names, the value are the values of the tuple
tuple = result[0]

# get the id column from the first tuple in the result
customer_id = result[0]['id']

# get the id column as a list
id_column = result['id']

# get the id of the first tuple
tuple_id = column[0]

# iterate over the rows of the result set. tuple is a dictionary, same as the
# example above
for tuple in result:
   # do something
   pass
```

This will return a tuple of (number of rows affected, results in case of SELECT).
Make sure to close your session using:

```
conn.close()
```

5.3 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
  print(e)
```

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

5.4 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!

6. 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 that should have all your code (your code will also go through manual testing).
2. The file <id1>_<id2>.pdf in which should include detailed explanations of your database design and the implementation of the API (explain each function and view). You should explain your database design choices in detail. You are NOT required to draw a formal ERD but it is recommended. You are limited to 6 pages total for the dry part.

You must use the check_submission.py script to check that your submission is in the correct format. Usage example:

```
python check_submission.py 123456789-987654321.zip
```

If your zip file is in the correct format you will get:

```
Success, IDs are: 123456789, 987654321
```

Any other type of submission will fail the automated tests and result in 0 on the wet part.

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

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.