Database Systems 236363, Spring 2025

Homework 2

SQL programming

- TA in charge: Gil Kizner
- Due date: 28/12/2025
- Questions should be asked in the dedicated forum in the course's Piazza.
- Submission is in pairs.

Overview

We recommend the following approach to this assignement:

1. **Set Up Your Environment**: Install PostgreSQL version 17 (version 16 is also supported) by following the instructions in the provided file *"PostgreSQL-install"*.
   Then, set up your work environment (Python, IDE, connect to the local server, run basic tests) by following the instructions in the file "Setting Up the Development Environment and Running Tests".
2. Go over the file titled *"Appendix"* (using the connector, handling dates in Python, and useful tips).
3. **Start Coding:** Read this document carefully, and begin implementing your solution in `Solution.py`.

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 rate 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, implement the data access layer of the system and finally optimize selected queries by applying the appropriate implementation methods.
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.

As you read this assignment, keep in mind that your goal is to write compatible queries for each function. If you would like an example of how a general function in your implementation should look before reading this document in detail, refer to the example.py file.

A significant part of the task involves determining which database tables/views to create to solve this homework. The recommended approach is to first read the entire API documentation to understand the entities in the system and their relationships, then translate these into a database design (an ERD can be helpful here), and finally start implementing the API functions.

Part 1- 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.
age	int	The age of the customer in years
phone	string	The customer's phone number.

Constraints:

1. cust_id is unique across all customers
2. cust_id is positive (>0)
3. age should be between 18 and 120.
4. The phone number should contain exactly 10 characters.
5. 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
delivery_fee	decimal	The delivery fee associated with the order
delivery_address	string	The address for delivery

Constraints:

1. order_id is unique across all orders
2. order_id is positive (>0)
3. delivery_fee is non negative (>=0)
4. delivery_address cannot be empty and must contain at least 5 characters.
5. All attributes are not optional (not null).
6. 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 whether the dish is currently available on the menu

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 4 characters.
5. All attributes are not optional (not null).

Part 2- API

In this section, you will implement the system's API. It includes a description of each function, detailing the inputs, outputs, and functionality.

Implementation Instructions (Read carefully)

1. We have defined the following enum type to be used as the return type for some functions:
   ReturnValue (enum):
   - OK
   - NOT_EXISTS
   - ALREADY_EXISTS
   - ERROR
     BAD_PARAMS

   In edge cases where multiple return values could apply based on the function description, you should return one of them (it doesn't matter what).

2. 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`.

3. Every calculation involving the data, such as filtering and sorting, must be done within the database queries. You are prohibited from performing any data calculations using Python. Points will be deducted if you ignore this rule.

4. You may only use **one** SQL query in each function implementation, not including views or recursive CTE. Create/Drop/Clear functions are not included in this rule.

5. You may only use material covered in lectures and tutorials when writing your queries. If you are unsure about using a particular command, please ask on Piazza. Points will be deducted for queries that include SQL commands not covered in class.

6. Your code must be contained within the provided functions, except for defining basic helper functions to avoid code duplication. You are only allowed to submit a single file, solution.py. All your code should be included there.

7. The database design is your responsibility. You may create and modify it as you see fit. Your database design will be graded, so inefficient or poorly designed databases will result in a reduction of points.

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

9. For the functions ‘get_order_total_price’ and ‘get_customers_spent_max_avg_amount_money’, you must utilize a single view that will serve both functions.

10. It is recommended to review the relevant Python files to understand their usage.

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

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.

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.
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
- ERROR in case of database error

Notes:

- You should use the price of the dish stored in the system (think about the structure of insert into you need to use to get this data)
- You should check the active status of the dish and validate it is active.
- Each order can contain multiple dishes, but each dish should be specified only once with the amount that was ordered per order.
- 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 or if the order does not exist (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.
- OrderDish is a data object created specifically for this function. It doesn't serve as business object and its only usage is to return information

ReturnValue customer_rated_dish(cust_id: int, dish_id: int, rating:int)

Customer with the given cust_id added a rating for the dish with the given dish_id.

Input:

- dish_id: the id of the dish.
- cust_id: the id of the customer
- rating: the rating given by the customer for the dish (must be between 1 and 5)

Output:

- OK in case of success.
- BAD PARAMS The rating is invalid (not between 1 and 5)
- ALREADY EXISTS if the customer has already rated this 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_deleted_rating_on_dish(cust_id: int, dish_id: int)

Customer with the given cust_id deleted their rating for 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.
- NOT_EXISTS if there isn't a record indicates the customer rated the dish (also for illegal ids)
- ERROR in case of database error

List[Tuple[int, int]] get_all_customer_ratings(cust_id: int)

Get a list of all ratings left by a specific customer for dishes in the system.

Input:

- cust_id: the id of the customer

Output:

- A list of distinct tuples containing dish_id and rating for each dish the customer has left.
- Each tuple should be in the form (dish_id, rating).
  Example tuple: (1, 5) where the first entry is the dish id, and the second entry is the rating.
- The returned list should be ordered by dish_id in ascending order. If the customer does not exist, or the customer did not rate 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.
- You can assume you will get legal order_id and that the order_id exists in the system
- fee delivery should also be included in the total price of the order.
- If the order has no dishes related, you should return fee delivery only.
- 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.

List[int] get_customers_spent_max_avg_amount_money()

Get the customer ids of customers who have the highest average total price per order.

Input: None

Output: A list of distinct customer ids where each customer has the maximum average spending amount per order. The list should be ordered by customer id in ascending order.

Note:

- You **must** use a view for this function.
- If no orders exist or none of the orders is related to a customer, return an empty list

Dish get_most_ordered_dish_in_period(start:datetime, end:datetime)

This function returns the dish with the highest amount of orders between start and end (inclusive). If there is a tie, the dish with the lowest id is returned.

Input:

- start: The start datetime (inclusive) of the period to consider.
- end: The end datetime (inclusive) of the period to consider.

Output: The dish that was ordered the largest number of times across all orders made during the specified period.

Note:

- If there is more than one dish with the highest purchase count, return the one with the lower dish_id.
- If there are no orders to return then you need to return a *BadDish.*

bool did_customer_order_top_rated_dishes(int cust_id)

Returns true if the customer with the given cust_id has ever ordered a dish from the top-rated dishes list.

Top-rated dishes: Dishes with the highest average rating. The maximum length of the top-rated dishes list is 5 (if there are more than 5 dishes in the system).

Input:

- cust_id: the id of the customer

Output: Returns true if the customer has ordered any dish from the top-rated dishes list. Otherwise, returns false.

Notes:

- if a dish does not have any rating, consider it with a rating of 3.
- If there is a tie for one of the top-rated dishes, select dishes with the lower dish_id.
- Return false if the customer does not exist, has no orders, or if there are no dishes in the system.

4.4 Advanced API

List[int] get_customers_rated_but_not_ordered()

Get all customers who have left a bad rating on one of the 5 lowest-rated dishes but have never ordered that dish.

Input: None

Output: A list of distinct customer ids who have left a bad review (rating < 3) on one of the 5 lowest-rated dishes and have never ordered that dish. The list should be ordered by customer id in ascending order. If no customer fulfills the condition, return an empty list.

Notes:

- The "5 lowest-rated dishes" are defined as the five dishes with the lowest average rating among all dishes. In case of a tie, choose dishes with the lowest dish_id to complete the list of five.
- Dishes with no ratings are considered to have an average rating of 3.
- A "bad rating" is defined as a rating less than 3.
- 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.

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_cumulative_profit_per_month(year: int)

Calculate the cumulative profit for each month of a specific year across all orders.

The cumulative profit for each month is the total profit from the beginning of the year up to and including that month.

I**nput:** year

Output: A list of distinct tuples, each containing month and the cumulative profit for that month. If no profit was made up to a specific month, the cumulative profit should be 0 for that month.

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

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.

Note:

- You can assume year will be a valid year.
- Ensure that the cumulative profit is converted to float in each returned tuple.
- You cannot use any SQL commands or functions that have not been covered during class.

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 rated 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 rated at least 4 by similar customers but not yet ordered by the given customer.

The list should be ordered by dish_id ascending.

Notes:

- Similar Customers: This is a transitive property. Two customers are similar if there exist a dish, they both rated at least 4. If Customer A is similar to B and customer B is similar to C, then A and C are also similar. So, you should compute the full transitive closure of all such similar customers starting from the given customer.
- Potential Dish Recommendations: Dishes rated at least 4 by similar customers that the given customer has not yet ordered.
- If customer doesn't exist or there are no dishes in the system return empty list.
- You can assume that the customer with the given id did not rate dishes they didn't order.

Example:

Assume customer 1 has rated with rating >= 4 dishes with ids 1,2,3 and order all of them.

Customer 2 has rated dishes 1,4 with rating >= 4.

Customer 3 has rated dishes 4, 5,7 with rating >= 4

Customer 4 has rated dish 6 with rating >= 4

- Customer 1 and customer 2 are similar because they both rated dish 1 with high rating.
- Customer 1 and customer 3 are similar because customer 2 and customer 3 are similar (transitivity).
- Therefore, the group of similar users to customer 1 is : {customer 2,customer 3}
- Customer 2 also highly rated dish 4 and customer 3 also highly rated dishes 5 and 7. Customer 1 did not order dishes 4,5,7. Therefore, we should return them as possible recommendations.

4.3 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).

Part 3- Implementation

a. Because of misuse by some app users, several dishes in the system were also mistakenly inserted into the Customer table, with the same phone number ranging the digits from 0 to 9 (012-345-6789).
   To identify these incorrect tuples, the following query was executed:

```
SELECT C.cust_id
FROM Customer R, Dish D
WHERE R.full_name = D.name
```

   Assume the following:
   - Each memory block contains 1024 bytes.
   - Each tuple occupies 50 bytes.
   - There are 60 dishes and 200 customers in the system.
   - The number of tuples with phone numbers 0–9 is at most 20.
   - The number of available memory blocks is B = 2.

   Design the optimal query execution plan based on block nested loop join, and explicitly calculate and analyze the I/O cost of your plan (The solution is not required to follow exactly the version presented in class. You are encouraged to design and implement the most efficient approach you can, with respect to I/O performance).

b. You and the other app developers noticed that a certain query is executed very frequently. The query receives a date as input and returns two values:
   1. The average delivery fee of all orders made on that date.
   2. The number of delivery addresses located in Haifa on that date.

To optimize the query, three different indexes were proposed:
1. CREATE INDEX date_idx ON Order(date)
2. CREATE INDEX date_idx ON Order(delivery_address, date, delivery_fee)
3. CREATE INDEX date_idx ON Order(date, delivery_fee, delivery_address)

Each index is implemented using a B+ tree with d=10, and each index is 10,000 blocks. The Order table contains 100 tuples per date.
Your task is to rank the three indexes by their I/O efficiency, with a detailed calculation of the exact I/O cost for each index and listing them from most efficient to least efficient.

Part 4- Submission

Please submit the following:

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

1. The file Solution.py (this exact name) that should have all your code (your code will also go through manual testing) and remember to change only the “TODO” parts, and not anything else (do not change function names, for example).
2. The file <id1>.pdf or <id1>_<id2>.pdf in which you should include your part 3 (implementation) answer.

Resubmissions will not be allowed if your submission does not follow these instructions!