DBMS Project(Deadline -6)

By: Yash Dhiman & Vishnu Mothukuri **Group ->91**

Scope of the project:

The purpose of the aforementioned database is to control how a pharmacy operates. A number of columns in the database are used to store data about customers, staff, memberships, drugs, stocks, orders, carts, bills, and wallets.

The "customer" table contains data about the medical store's patrons, including their name, email address, gender, phone number, address, and location information. The "employee" table contains data about the medical supply store's staff members, such as name, address, phone number, email, salary, and position.

The "membership" database keeps track of details regarding customer memberships, such as membership type, discount %, and start and end dates. Information regarding the medications sold at the store, such as their name and maker, is kept on the "drug" table.

The "Stock" table contains data on the amount of pharmaceuticals in stock, their price, expiration date, and batch information. The "orders_med" table keeps track of details regarding client orders, such as the date, price, buyer, and medicine information.

Information about the items that customers have added to their carts, such as quantity, price, drug information, and customer information, is kept in the "CART" table. The "CART_MEDICINE" table keeps track of the names and cart IDs of the medications put to the cart.

The "Bill" database keeps track of facts about the bills created for consumers, including the date, cost, method of payment, and customer information. The "wallet" table maintains data about customer wallet balances, including wallet amount and customer ID.

An effective way to handle a medical store's business operations, including managing clients, staff, medications, stocks, orders, and payments, is to use the database. The database can assist in keeping correct records, monitoring inventories, and producing reports to enhance the functioning of the store.

User guide for the program:

Main menu

```
Welcome to the MedBag
1. Admin login
2. Customer login
3. run custom queries
4. exit
```

This is the main menu where we can select the Admin login, which we can access using the employee id and employee email or can be accessed. The access for the admin is done by the admin login and password, which is “admin” & “admin”, respectively; the admin commands open up the sets of admin commands, which are displayed here :

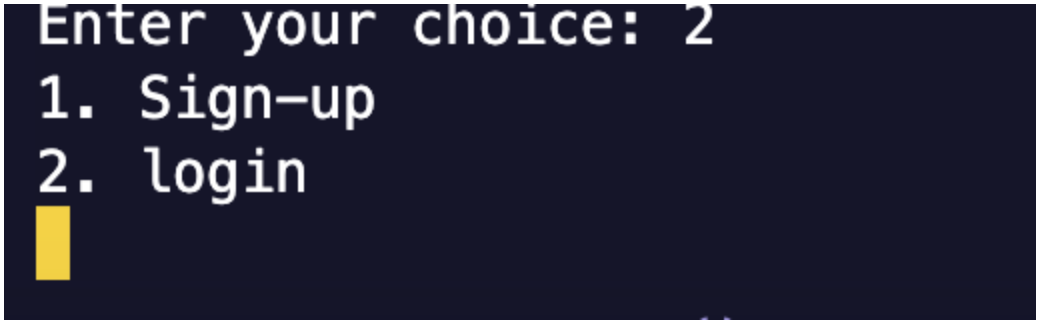
```
Welcome Admin
What all actions would you like to perform?
1.   Generate data (randomly genrated data for testing)
2.   add new employee
3.   add new customer
4.   add new membership
5.   Upgrade the customer membership
6.   Add a new product to the database
7.   Remove the product from the database
8.   view purchases
9.   update stocks
10.  view stocks for certain product
Enter your choice:
```

Commands for Admin:

1. Generate data: in this section, we will add the random data which is being generated by the datagen.py (a file which is which will randomly generate the data and add it to the database accordingly.

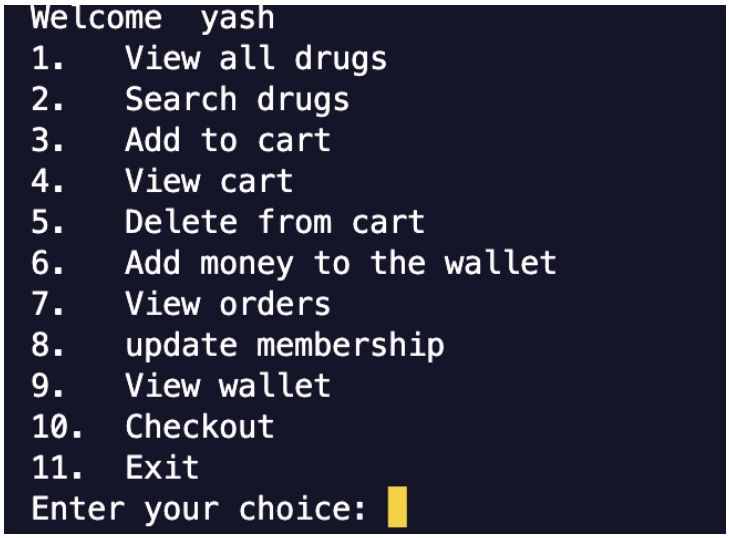
2. Add new employee: the admin adds the details of the employee, such as employee id, name, address etc., where the query will be executed, and the data will be inserted.
3. Add new customer: the admin can add the customer to the customer database.
4. Add new membership: in this command, we upgrade the existing membership or add a new member when once it is confirmed that the customer has performed the payment for the respective membership.
5. Upgrade Membership: In this command, we upgrade the membership type for the customer.
6. Add new Product to the database: In this command, we are adding the stock and a new drug in the database, which will be available for purchase.
7. Remove Product from the database: If the admin wants to remove the product from the database, he/she can simply write the drug id and remove the data from the database.
8. View purchases: The employee can view the purchases done by the customers by typing in the customer id of the person.
9. Update stocks: this command will update the stocks of the of the drugs in case there is a new batch of drugs has arrived.
10. View stocks for a particular product: The admin can view the stock of a particular product by entering the drug id.

Customer login and & Sign up



This command has a basic login section where the customer who has already registered can access commands by simply writing the name and the email of the registered user and can access the customer commands.

Customer Commands



1.) View all drugs: This option allows users to view all the drugs in the pharmacy. This will display a list of all drugs in the database, including their names, prices, and quantities in stock.

2.) Search drugs: This option enables users to search for a specific drug by name or keyword. It will display all drugs that match the search criteria, including their names, prices, and quantities in stock.
3.) Add to cart: This option allows users to add drugs to their cart. Users will be prompted to enter the drug name and the quantity they wish to purchase. The system will then add the selected drugs to the user's cart.
4.) View cart: This option allows users to view the drugs they have added to their cart. The system will display a list of all drugs in the user's cart, including their names, prices, and quantities.
5.) Delete from the cart: This option enables users to remove drugs from their cart. Users will be prompted to select the drug they want to remove from their cart, and the system will then remove it from the user's cart.
6.) Add money to the wallet: This option allows users to add money to their wallet balance. Users will be prompted to enter the amount they want to add to their wallet, and the system will then update the user's wallet balance accordingly.
7.) View orders: This option enables users to view their previous orders. The system will display a list of all previous orders, including the date, time, and total cost of each order.
8.) Update membership: This option allows users to update their membership status. Users will be prompted to select their desired membership status (e.g., regular or premium), and the system will then update their membership status in the database.
9.) View wallet: This option allows users to view their current wallet balance. The system will display the user's current wallet balance.
10.) Checkout: This option allows users to complete their purchase. Users will be prompted to confirm their order and pay using their wallet balance. The system will then update the database with the new order details and adjust the drug quantities in stock accordingly.
11.) Exit: This option allows users to exit the system and end their session.

What is a translation query?

A set of one or more SQL queries that are performed as a single unit of work, meaning that they must all succeed or all fail, is referred to as a "transaction query." Because it

enables data consistency and integrity, the concept of transactions is crucial in databases.

A database management system (DBMS) in a transaction ensures that all operations are completed or that the system will undo all changes made by the transaction. This guarantees that even if a transactional mistake occurs, the database will still be in a consistent state.Depending on whether the transaction was successful or not, transaction queries are normally initiated with the "START TRANSACTION" statement and ended with either a "COMMIT" or "ROLLBACK" statement.

Custom Query:

```
Which type of query do you want to perform?
1. run 4 transation queries
2. run 4 non conflicting queries
3. Run olap queries
4. Run embedded queries
5. Run triggers
6. Transations with  two schedules which are conflict serializable and non-conflict
 serializable
7. Exit
```

The given code allows the user to select various types of queries to perform on a database. The following are the types of queries that the user can perform:

1.) Run 4 transaction queries
2.) Run 4 non-conflicting queries
3.) Run OLAP queries
4.) Run embedded queries

5.)Run triggers
6.) Transactions with two schedules which are conflict serializable and non-conflict serializable.
7.) Exit

The user is prompted to specify the type of query they want to run. The associated block of code is run depending on the user's input.

If the user selects option 1, the database is subjected to four transactional queries. A new drug is added to the database in the first query, a drug's stock is updated in the second, a drug's quantity and price are updated in the third, and a new drug and its matching stock are added to the database in the fourth. Each query's modifications are shown on the terminal.

If the user selects option 2, the database is queried using four non-conflicting queries. The first query increases the stock of a medicine with ID 1 by 100 units, whereas the second query decreases the stock of the same drug by 50 units. Each query's modifications are shown on the terminal.

If the user selects option 3, the OLAP module is imported and launched in order to perform an OLAP (Online Analytical Processing) query on the database.

If the user selects option 4, the query module is imported and executed on the database in order to run an embedded SQL query.

If the user selects option 5, the triggers module is imported and run in order to execute a trigger on the database.

If the user selects option 6, two transactions—one conflict serializable and the other non-conflict serialisable—are carried out on the database. Each transaction's modifications are shown on the console.

If the user selects option 7, the application ends.

Code:

```
 if choice4 ==6:
        try:
            cursor = db.cursor()
            cursor.execute("START TRANSACTION")
            # Insert new customer into the database
            customer_insert_query = "INSERT INTO customer (customer_name,
customer_email, customer_gender, customer_phone, customer_address,
customer_zipcode, customer_street, customer_city, customer_state) VALUES
(%s, %s, %s, %s, %s, %s, %s, %s, %s)"
            customer_insert_val = ("John Doe", "johndoe@gmail.com", "M",
1234567890, "123 Main St", 12345, "Main St", "Anytown", "CA")
            cursor.execute(customer_insert_query, customer_insert_val)
            customer_id = cursor.lastrowid
            # Update customer's wallet with 100
            wallet_insert_query = "INSERT INTO wallet (customer_id, wallet_amount)
VALUES (%s, %s)"
            wallet_insert_val = (customer_id, 100)
            cursor.execute(wallet_insert_query, wallet_insert_val)
            # Commit transaction 1
            db.commit()
            print("Transaction 1 completed successfully")
        except:
            # Rollback transaction 1 in case of any errors
            db.rollback()
            print("Transaction 1 failed")
        # Start transaction 2: Place an order for a drug and update the stock
quantity
        try:
            cursor.execute("START TRANSACTION")
            # Insert new order into the database
            order_insert_query = "INSERT INTO orders_med (order_date, order_price,
customer_id, drug_id) VALUES (%s, %s, %s, %s)"
            order_insert_val = ("2023-04-23", 50, customer_id, 1)
```

```
            cursor.execute(order_insert_query, order_insert_val)
            # Update stock quantity
            stock_update_query = "UPDATE Stock SET stock_quantity =
stock_quantity - %s WHERE drug_id = %s"
            stock_update_val = (10, 1)
            cursor.execute(stock_update_query, stock_update_val)
            # Commit transaction 2
            db.commit()
            print("Transaction 2 completed successfully")
        except:
            # Rollback transaction 2 in case of any errors
            db.rollback()
            print("Transaction 2 failed")
    # Schedule 1: T1 followed by T2
        try:
            cursor.execute("START TRANSACTION")
            # Execute T1
            customer_insert_query = "INSERT INTO customer (customer_name,
customer_email, customer_gender, customer_phone, customer_address,
customer_zipcode, customer_street, customer_city, customer_state) VALUES
(%s, %s, %s, %s, %s, %s, %s, %s, %s)"
            customer_insert_val = ("Jane Smith", "janesmith@gmail.com", "F",
9876543210, "456 Main St", 54321, "Main St", "Anytown", "CA")
            cursor.execute(customer_insert_query, customer_insert_val)
            customer_id = cursor.lastrowid
            wallet_insert_query = "INSERT INTO wallet (customer_id, wallet_amount)
VALUES (%s, %s)"
            wallet_insert_val = (customer_id, 200)
            cursor.execute(wallet_insert_query, wallet_insert_val)
            # Execute T2
            order_insert_query = "INSERT INTO orders_med (order_date, order_price,
customer_id, drug_id) VALUES (%s, %s, %s, %s)"
            order_insert_val = ("2023-04-23", 50, customer_id, 1)
            cursor.execute(order_insert_query, order_insert_val)
            stock_update_query = "UPDATE Stock SET stock_quantity =
stock_quantity - %s WHERE drug_id = %s"
```

```
            stock_update_val = (10, 1)
            cursor.execute(stock_update_query, stock_update_val)
            # Commit both transactions
            db.commit()
            print("Both transactions completed successfully")
        except:
            # Rollback both transactions in case of any errors
            db.rollback()
            print("Both transactions failed")
        # Schedule 2: T2 followed by T1
        try:
            cursor.execute("START TRANSACTION")
            # Execute T2
            order_insert_query = "INSERT INTO orders_med (order_date, order_price,
customer_id, drug_id) VALUES (%s, %s, %s, %s)"
            order_insert_val = ("2023-04-23", 50, customer_id, 1)
            cursor.execute(order_insert_query, order_insert_val)
            stock_update_query = "UPDATE Stock SET stock_quantity =
stock_quantity - %s WHERE drug_id = %s"
            stock_update_val = (10, 1)
            cursor.execute(stock_update_query, stock_update_val)
            # Execute T1
            order_insert_query = "INSERT INTO orders_med (order_date, order_price,
customer_id, drug_id) VALUES (%s, %s, %s, %s)"
            order_insert_val = ("2023-04-23", 50, customer_id, 1)
            cursor.execute(order_insert_query, order_insert_val)
            wallet_insert_query = "INSERT INTO wallet (customer_id, wallet_amount)
VALUES (%s, %s)"
            wallet_insert_val = (customer_id, 200)
            cursor.execute(wallet_insert_query, wallet_insert_val)
            # Commit both transactions
            db.commit()
            print("Both transactions completed successfully")
        except:
            # Rollback both transactions in case of any errors
            db.rollback()
```

```
        print("Both transactions failed")
```

Explaining transactions and then making two schedules that are conflict serializable and non-conflict serializable, respectively, for these transactions.

Then, the code generates Schedule 1 and Schedule 2, two schedules.

Transaction 1 is carried out by Schedule 1 first, followed by Transaction 2.

Transaction 2 is carried out by Schedule 2 first, followed by Transaction 1.

Due to the fact that there is no conflict between the two transactions, Schedule 1 is conflict serializable. While Transaction 2 simply adds a new order and updates the stock level, Transaction 1 only adds a new customer to the database and updates the customer's wallet. The data being altered by the two transactions do not overlap, so they can be conducted in any sequence without running afoul of one another.

Contrarily, Schedule 2 causes a conflict between the two transactions and is not conflict serializable. The stock quantity will be changed prior to the creation of the new customer's wallet since Transaction 2 updates the stock quantity before Transaction 1 inserts a new customer. This could lead to a situation where the stock amount is decreased prior to the creation of the customer's wallet, which could complicate the accounting for the order. Schedule 2 cannot be serialised to avoid conflicts as a result.

Summary of the code:

If the first transaction completes properly, the code produces a message stating as much, and the transaction is committed, making the changes to the database permanent. The function rolls back the transaction if there are any mistakes in the initial transaction, which undoes all changes performed as a result of the transaction. It also prints a message indicating that the transaction failed.

The code then performs two further transactions that are identical to the original transaction but with different operation orderings. In both cases, a new customer is added, their wallet is updated with $200, and an order for medicine is placed while the stock level is updated. The order in which these processes are performed differs.

In contrast to the third transaction, which accomplishes the opposite, the second transaction places the order first and updates the customer's wallet second. If the transactions in either situation are successful, the code outputs a message stating that the transactions were committed and that both transactions were successful. The code rolls back both transactions if there are any issues and prints a message stating that both transactions failed.

In conclusion, this code shows how transactions may be used in a database system to protect the consistency and integrity of data and to deal with problems that may arise when carrying out several activities.

CONFLICT SERIALIZABLE SCHEDULE:

T1	T2
-	START
R1(A)	-

W1(B)	-
START	-
-	R2(B)
W2(A)	-
-	COMMIT
COMMIT	-

NON CONFLICT SERIALIZABLE SCHEDULE:

STEP	T1	T2
1		START TRANSACTION
2	R1(A)	
3	W1(B)	
4	START TRANSACTION	
5		R2(B)
6		W2(A)
7	W1(C)	
8		R2(C)
9	R1(D)	
10	W1(E)	
11		W2(D)
12	W1(F)	
13	R1(G)	

14	W1(H)	
15		W2(I)
16	R1(J)	
17	W1(K)	
18		COMMIT
19	COMMIT	