MyMART

SuperMarket Database Management System

Created By:

Dhyey Italiya(2021A7PS1463P)

Rakshit Aggarwal(2021A7PS1458P)

Links to All required submission materials

- Dhyey Italiya (2021A7PS1463P) Explanation Video
- Rakshit Aggarwal (2021A7PS1458P) Explanation Video
- Application Running Explanation Video
- ER Diagram (Also Provided in the zipped folder)
- Relational Tables and 3NF conversions. (Also Provided in the zipped folder)
- Documentation of Project
- ScreenShots of Outputs in SQL terminal

Tech Stack used

- Tkinter-Tcl/Tk Version 8.6
- MySQL- 8.0.32-0
- MySQL connector - Version: 2.2.9
- Python-Python 3.10.6

Documentation for SuperMarket Database Management System(MyMART)

MyMART is a relational database management system designed to manage the operations of a supermarket efficiently. It is a comprehensive system that handles all aspects of the supermarket's operations, including managing employees, customers, suppliers, products, purchases, and sales.

Installation Instructions

To use the Supermart Application (For users using the Application), follow these instructions:

- Download and install MySQL Community Server:
    1. Go to https://dev.mysql.com/downloads/mysql/ to download the MySQL Community Server.
    2. Choose the appropriate version for your operating system and follow the instructions to install it.

- Configure MySQL:
    1. Once installed, you need to configure MySQL to run on your machine.
    2. During installation, you will be prompted to set a root password for MySQL. Make sure to choose a strong password and keep it secure.

3. After installation, you can access the MySQL Server from the command line using the command "mysql -u root -p" and enter the root password when prompted.

- Run the Application:
  1. You need to run the Python file named “FinalLoginPage.py”, and you will be prompted to enter your MySQL username and password. On Authenticating this information, The window will lead you to the application Dashboard, where you can use the application with the requirements specified.

Database Design

MyMART is designed using the relational database model. The database contains five tables: Employee, Customer, Supplier, Product, Purchase, and Sale.

- **Employee Table**: This table contains the personal details of all the supermarket employees. Each employee is assigned a unique ID. The table includes details such as the employee's first name, second name, salary, amount sold, designation, phone number, sex, address line one, address line two, and postal code.

- **Customer Table**: This table contains the personal details of all the supermarket's customers. Each customer is assigned a unique ID, and the table includes details such as the customer's first name, second name, phone number, the amount spent, address line one, address line two, and postal code.

- **Supplier Table**: This table contains details of all the suppliers that provide products to the supermarket. Each supplier is assigned a unique ID, and the table includes details such as the supplier's first name, second name, phone number, the amount earned, address line one, address line two, and postal code.

- **Product Table**: This table contains details of all the products available in the supermarket. Each product is assigned a unique ID, and the table includes details such as the product's name, price, and quantity left.

- **Purchase Table**: This table contains details of all the purchases made by the supermarket from the suppliers. Each purchase is assigned a unique ID, and the table includes details such as the purchase date and the supplier ID.

- **Sale Table**: This table contains details of all the sales made by the supermarket to customers. Each sale is assigned a unique ID, and the table includes details such as the sale date, customer ID, and employee ID.

- **Purchase_has_Product Table**: This table contains details of products bought in each purchase made by the supermarket from the supplier. The table includes multiple records, each with, the purchase ID, the associated product ID and the associated quantity bought of that product.

- **Sale_has_Product Table**: This table contains details of products sold in each sale made by the supermarket to customers. The table includes multiple records, each with, the sale ID, the associated product ID and the associated quantity bought of that product

Data Management

MyMART provides various functionalities to manage the data in the database. These functionalities include:

- **Inserting Data**: The system provides SQL queries to insert data into the database tables. Sample data has been provided in the SQL file for each table.

- **Updating Data**: The system provides SQL queries to update the data in the database tables. The user can edit the employee's amount sold or the customer's amount spent based on their progress.

- **Deleting Data**: The system provides SQL queries to delete data from the database tables. The user can delete a purchase or sale record from the respective tables.

- **Retrieving Data**: The system provides SQL queries to retrieve data from the database tables. The user can retrieve the employee's details based on their ID or the customer's details based on their phone number.

Assumptions :

1. In a single table, all records will have a unique phone number, any phone number cannot be held by multiple people at the same time
2. Any customer coming to buy a product (Adding a Sale) brings the products with him to the counter, meaning the products he brings will be in stock in the SuperMarket beforehand, i.e., after adding a Sale, Quantity left in stock for any product cannot go negative.
3. While deleting a Sale, we assume that the Customer and Employee have made some sales, meaning that, Employee's Amount Sold and the Customer's Amount Spent cannot be negative.

For further information, refer to the file "Queries.sql", which has SQL implementation of the functionalities used in the application.

Queries.sql Explanation

The given MySQL script consists of several stored procedures that allow the user to interact with the database. The documentation for each procedure is given below:

- **insertIntoCustomer:**

```
DELIMITER //
CREATE PROCEDURE insertIntoCustomer(IN input_First_Name VARCHAR(50), IN input_Second_Name
VARCHAR(50), IN input_Phone_Number BIGINT, IN input_Address_Line_One VARCHAR(100), IN
input_Address_Line_Two VARCHAR(100), IN input_Postal_Code INT)
BEGIN
INSERT INTO Customer (Cust_First_Name, Cust_Second_Name, Cust_Phone_Number, Cust_Amount_Spent,
Cust_Address_Line_One, Cust_Address_Line_Two, Cust_Postal_Code) VALUES(input_First_Name,
input_Second_Name, input_Phone_Number, 0, input_Address_Line_One, input_Address_Line_Two,
input_Postal_Code);
END //
DELIMITER ;
```

This stored procedure is used to insert customer details into the 'Customer' table of the database. We initialize; it takes the following parameters:

- input_First_Name: The first name of the customer (VARCHAR).
- input_Second_Name: The last name of the customer (VARCHAR).
- input_Phone_Number: The phone number of the customer (BIGINT).
- input_Address_Line_One: The first line of the customer's address (VARCHAR).
- input_Address_Line_Two: The second line of the customer's address (VARCHAR).
- input_Postal_Code: The postal code of the customer's address (INT).

- ❖ **insertIntoSupplier:**

```
DELIMITER //
CREATE PROCEDURE insertIntoSupplier(IN input_First_Name VARCHAR(50), IN input_Second_Name
VARCHAR(50), IN input_Phone_Number BIGINT, IN input_Address_Line_One VARCHAR(100), IN
input_Address_Line_Two VARCHAR(100), IN input_Postal_Code INT)
BEGIN
INSERT INTO Supplier (Supplier_First_Name, Supplier_Second_Name, Supplier_Phone_Number,
Supplier_Amount_Earnt, Supplier_Address_Line_One, Supplier_Address_Line_Two,
Supplier_Postal_Code) VALUES(input_First_Name, input_Second_Name, input_Phone_Number, 0,
input_Address_Line_One, input_Address_Line_Two, input_Postal_Code);
END //
DELIMITER ;
```

This stored procedure is used to insert supplier details into the 'Supplier' table of the database. It takes the following parameters:

- input_First_Name: The first name of the supplier (VARCHAR).
- input_Second_Name: The last name of the supplier (VARCHAR).
- input_Phone_Number: The phone number of the supplier (BIGINT).
- input_Address_Line_One: The first line of the supplier's address (VARCHAR).
- input_Address_Line_Two: The second line of the supplier's address (VARCHAR).
- input_Postal_Code: The postal code of the supplier's address (INT).

- ❖ **insertIntoProduct:**

```
DELIMITER //
CREATE PROCEDURE insertIntoProduct(IN p_name VARCHAR(50), IN p_price INT)
BEGIN
   INSERT INTO Product (Product_Name, Price, Qty_Left) VALUES (p_name, p_price, 0);
END//
DELIMITER ;
```

This stored procedure is used to insert product details into the 'Product' table of the database. It takes the following parameters:

- p_name: The name of the product (VARCHAR).
- p_price: The price of the product (INT).

- ❖ **insertIntoEmployee:**

```
DELIMITER //
CREATE PROCEDURE insertIntoEmployee(
   IN emp_fname VARCHAR(50),
   IN emp_lname VARCHAR(50),
   IN emp_salary INT,
   IN emp_designation ENUM('Custodian', 'Security', 'Manager', 'Supervisor', 'Cashier'),
   IN emp_phone_number BIGINT,
   IN emp_sex ENUM('M','F'),
   IN emp_address_line_one VARCHAR(100),
   IN emp_address_line_two VARCHAR(100),
   IN emp_postal_code INT
)
BEGIN
   INSERT INTO Employee (Emp_First_Name, Emp_Second_Name, Emp_Salary, Emp_Amount_Sold,
Emp_Designation, Emp_Phone_Number, Emp_Sex, Emp_Address_Line_One, Emp_Address_Line_Two,
Emp_Postal_Code)
   VALUES (emp_fname, emp_lname, emp_salary, 0, emp_designation, emp_phone_number, emp_sex,
emp_address_line_one, emp_address_line_two, emp_postal_code);
END //
DELIMITER ;
```

This stored procedure is used to insert employee details into the 'Employee' table of the database. It takes the following parameters:

- emp_fname: The first name of the employee (VARCHAR).
- emp_lname: The last name of the employee (VARCHAR).
- emp_salary: The salary of the employee (INT).
- emp_designation: The job designation of the employee (ENUM).
- emp_phone_number: The phone number of the employee (BIGINT).
- emp_sex: The sex of the employee (ENUM).
- emp_address_line_one: The first line of the employee's address (VARCHAR).
- emp_address_line_two: The second line of the employee's address (VARCHAR).
- emp_postal_code: The postal code of the employee's address (INT).

❖ **updateSupplierContact:**

```
DELIMITER //
CREATE PROCEDURE updateSupplierContact(IN s_id INT, IN s_pno BIGINT)
BEGIN
   UPDATE Supplier SET Supplier_Phone_Number = s_pno WHERE Supplier_ID = s_id;
END //
DELIMITER ;
```

This stored procedure is used to update the phone number of a supplier in the 'Supplier' table of the database. It takes the following parameters:

- s_id: The ID of the supplier (INT).
- s_pno: The new phone number of the supplier (BIGINT).

❖ **updateProductQuantity:**

```
DELIMITER //
CREATE PROCEDURE updateProductQuantity(IN p_id INT, IN qty INT)
BEGIN
   UPDATE Product SET Product.Qty_Left = qty WHERE Product_ID = p_id;
```

```
END //
DELIMITER ;
```

This stored procedure is used to update the quantity of a product in the 'Product' table of the database. It takes the following parameters:

- p_id: The ID of the product (INT).
- qty: The new quantity of the product (INT).

❖ **updateJobPosition:**

```
DELIMITER //
CREATE PROCEDURE updateJobPosition(IN ep_id INT, IN pos ENUM('Custodian', 'Security', 'Manager',
'Supervisor', 'Cashier'))
BEGIN
   UPDATE Employee SET Emp_Designation = pos WHERE Emp_ID = ep_id;
END //
DELIMITER ;
```

This stored procedure is used to update the job position of an employee in the 'Employee' table of the database. It takes the following parameters:

- ep_id: The ID of the employee (INT).
- pos: The new job position of the employee (ENUM).

❖ **Add_purchase:**

```
DELIMITER //
CREATE PROCEDURE addPurchase (
   IN supplier_id INT,
   IN products VARCHAR(255),
   IN quantities VARCHAR(255)
)
-- Taking array of product IDs and their quantities in an array as inputs
BEGIN
   -- Inserting into the table Purchase
   INSERT INTO Purchase (Purchase_Date, Supplier_ID)
   VALUES (CURRENT_DATE(), supplier_id);
   -- Storing unique purchase id into a variable
   SET @purchase_id = LAST_INSERT_ID();
   SET @products = products;
   SET @quantities = quantities;
   SET @products_array = JSON_EXTRACT(@products, '$');
   SET @quantities_array = JSON_EXTRACT(@quantities, '$');
   -- Setting variable for keeping index of while loop
   SET @i = 0;
   WHILE (@i < JSON_LENGTH(@products_array)) DO
       -- Storing product ID and their quantity bought in variables
       -- Finding ith product and quantity
```

```sql
        SET @product_id = JSON_EXTRACT(@products_array, CONCAT('$[', @i, ']'));
        SET @qty_bought = JSON_EXTRACT(@quantities_array, CONCAT('$[', @i, ']'));
        -- Inserting record into Purchase_has_Product
        INSERT INTO Purchase_has_Product (Purchase_ID, Product_ID, Qty_Bought)
        VALUES (@purchase_id, @product_id, @qty_bought);
        -- Updating value of Quantity left of Product in the supermarket
        UPDATE Product SET Qty_Left = Qty_Left + @qty_bought WHERE Product_ID = @product_id;
        -- Incrementing value of index
        SET @i = @i + 1;
    END WHILE;
    -- Storing sum of Price of total purchase
    SET @total_purchase = (
        SELECT SUM(Price * Qty_Bought)
        FROM Purchase_has_Product
        JOIN Product ON Product.Product_ID = Purchase_has_Product.Product_ID
        WHERE Purchase_ID = @purchase_id
    );
    -- Update Supplier's amount earnt by above sum
    UPDATE Supplier
    SET Supplier_Amount_Earnt = Supplier_Amount_Earnt + @total_purchase
    WHERE Supplier_ID = supplier_id;
END //
DELIMITER ;
```

This stored procedure is used to add a new purchase to the 'Purchase' table of the database. It takes the following parameters:

- in_product_id (IN INT): The ID of the product being purchased.
- in_supplier_id (IN INT): The supplier ID from the purchased product.
- in_purchase_price (IN DECIMAL(10,2)): The purchase price of the product.
- in_purchase_date (IN DATE): The date on which the purchase was made.
- in_quantity_purchased (IN INT): The quantity of the product that was purchased.

Functionality:

1. Inserts a new row into the Purchase table with the current date, the Supplier ID.
2. Gets the ID of the newly inserted purchase row using the LAST_INSERT_ID() function and stores it in a variable.
3. Extracts the product IDs and quantities from the JSON-encoded products and quantities input string arrays and stores them in corresponding variables.
4. Iterates each product in the products array and inserts a new row into the Purchase_has_Product table with the Purchase ID, the product ID, and the quantity bought.
5. Updates the Qty_Left column in the Product table for each product based on the quantity bought in through each iteration.

6. Calculates the total purchase amount by summing the prices of each product bought using the purchase ID and joins between the Purchase_has_Product and Product tables. Stores the total purchase amount in a variable.
7. Updates the Supplier_Amount_Spent column in the Supplier table for the supplier who purchased by adding the total purchase amount to their respective columns.

- **Add_sale:**

```
DELIMITER //
CREATE PROCEDURE addSale (
   IN customer_id INT,
   IN products VARCHAR(255),
   IN quantities VARCHAR(255),
   IN employee_id INT
)
-- Taking array of product IDs and their quantities in an array as inputs
BEGIN
   -- Inserting into the table Sale
   INSERT INTO Sale (Sale_Date, Cust_ID, Emp_ID)
   VALUES (CURRENT_DATE(), customer_id, employee_id);
   -- Storing unique Sale id into a variable
   SET @sale_id = LAST_INSERT_ID();
   SET @products = products;
   SET @quantities = quantities;
   SET @products_array = JSON_EXTRACT(@products, '$');
   SET @quantities_array = JSON_EXTRACT(@quantities, '$');
   -- Setting variable for keeping index of while loop
   SET @i = 0;
   WHILE (@i < JSON_LENGTH(@products_array)) DO
       -- Storing product ID and their quantity sold in variables
       -- Finding ith product and quantity
       SET @product_id = JSON_EXTRACT(@products_array, CONCAT('$[', @i, ']'));
       SET @qty_sale = JSON_EXTRACT(@quantities_array, CONCAT('$[', @i, ']'));
       -- Inserting record into Sale_has_Product
       INSERT INTO Sale_has_Product (Sale_ID, Product_ID, Qty_Bought)
       VALUES (@sale_id, @product_id, @qty_sale);
       -- Updating value of Quantity left of Product in the supermarket
       UPDATE Product SET Qty_Left = Qty_Left - @qty_sale WHERE Product_ID = @product_id;
       -- Incrementing value of index
       SET @i = @i + 1;
   END WHILE;
   -- Storing sum of Price of total Sale
   SET @total_sale = (
       SELECT SUM(Price * Qty_Bought)
       FROM Sale_has_Product
       JOIN Product ON Product.Product_ID = Sale_has_Product.Product_ID
```

```
        WHERE Sale_ID = @sale_id
   );
   -- Update Customer's amount earnt by above sum
   UPDATE Customer
   SET Cust_Amount_Spent = Cust_Amount_Spent + @total_sale
   WHERE Cust_ID = customer_id;
   -- Update Employee's amount earnt by above sum
   UPDATE Employee
   SET Emp_Amount_Sold = Emp_Amount_Sold + @total_sale
   WHERE Emp_ID = employee_id;

END //
DELIMITER ;
```

Input parameters:

- customer_id: an integer representing the customer's ID who made the purchase.
- products: a JSON-encoded string containing an array of product IDs.
- quantities: a JSON-encoded string containing an array of quantities for each product.
- employee_id: an integer representing the ID of the employee who made the sale.

Functionality:

8. Inserts a new row into the Sale table with the current date, the customer ID, and the employee ID.
9. Gets the ID of the newly inserted sale row using the LAST_INSERT_ID() function and stores it in a variable.
10. Extracts the product IDs and quantities from the JSON-encoded products and quantities inputs and stores them in corresponding variables.
11. Iterates each product in the products array and inserts a new row into the Sale_has_Product table with the sale ID, the product ID, and the quantity bought.
12. Updates the Qty_Left column in the Product table for each product based on the quantity bought.
13. Calculates the total sale amount by summing the prices of each product bought using the sale ID and joins between the Sale_has_Product and Product tables. Stores the total sale amount in a variable.
14. Updates the Cust_Amount_Spent column in the Customer table for the customer who bought the sale, and the Emp_Amount_Sold column in the Employee table for the employee who made the sale, by adding the total sale amount to their respective column.

❖ **deleteSale:**

```
DELIMITER //
CREATE PROCEDURE deleteSale(IN sale_id INT)
```

```
BEGIN
   -- Declaring variables for index for while loop, and number of different records in
Sale_has_Product with sale ID same as given
   DECLARE i INT DEFAULT 0;
   DECLARE n INT DEFAULT 0;
   -- Declaring a variable for storing total money involved in a Sale
   SELECT SUM(P.Price*S.Qty_Bought) FROM Product as P, Sale_has_Product as S WHERE P.Product_ID =
S.Product_ID AND S.Sale_ID = sale_id INTO @sum;
   -- Updating Money Spent by Customer by the above @sum declared
   UPDATE Customer SET Customer.Cust_Amount_Spent = (Customer.Cust_Amount_Spent - @sum) WHERE
Customer.Cust_ID = (SELECT Sale.Cust_ID FROM Sale Where Sale.Sale_ID = sale_id LIMIT 1);
   -- Updating Money Sold by Employee by the above @sum declared
   UPDATE Employee SET Employee.Emp_Amount_Sold = (Employee.Emp_Amount_Sold - @sum) WHERE
Employee.Emp_ID = (SELECT Sale.Emp_ID FROM Sale Where Sale.Sale_ID = sale_id LIMIT 1);
   -- Temporary table, only containing those records of Sale_has_Product, which have sale id same
as given
   CREATE TEMPORARY TABLE Specific_Sale SELECT * FROM Sale_has_Product WHERE
Sale_has_Product.Sale_ID = sale_id;
   -- Giving n value of size of Specific_Sale table
   SET n = (SELECT COUNT(*) FROM Specific_Sale);
   WHILE ( i < n ) DO
       -- Storing Product ID and Qty Bought of each record of table in variables in each
iteration of the loop
       SELECT Specific_Sale.Product_ID from Specific_Sale limit i,1 INTO @prod_id;
       SELECT Specific_Sale.Qty_Bought from Specific_Sale limit i,1 INTO @qty_bt;
       -- Updating Quantity Left in Stock in Product table using these variable values
       UPDATE Product SET Product.Qty_Left = (Product.Qty_Left - @qty_bt)
       WHERE Product.Product_ID = @prod_id;
       -- Incrementing index value for while loop
       SET i = i+1;
   END WHILE;
   -- Deleting records from Sale_has_Product and Sale, associated with the given Sale_id
   DELETE FROM Sale_has_Product WHERE Sale_has_Product.Sale_ID = sale_id;
   DELETE FROM Sale WHERE Sale.Sale_ID = sale_id;
END //
DELIMITER ;
```

Input parameters:

- sale_id: an integer representing the ID of the sale to be deleted.

Functionality:

1. Calculates the total sale amount by summing the prices of each product bought using the sale ID and joins between the Sale_has_Product and Product tables. Stores the total sale amount in a variable.
2. Updates the Cust_Amount_Spent column in the Customer table for the customer who made the purchase, and the Emp_Amount_Sold column in the Employee table for the

employee who made the sale, by subtracting the total sale amount from their respective columns.

3. Creates a temporary table Specific_Sale to hold all the rows from the Sale_has_Product table for the sale ID to be deleted.
4. Iterates each product in Specific_Sale and updates the Qty_Left column in the Product table based on the quantity bought.
5. Deletes all rows from the Sale_has_Product table for the sale ID to be deleted.
6. Deletes the row from the Sale table for the sale ID to be deleted.

Conclusion

MyMART is a comprehensive system that provides functionalities to manage the operations of a supermarket efficiently. It uses the relational database model and provides functionalities to insert, update, delete, and retrieve data from the database tables. The system is easy to install and use and can be customised based on the requirements of the supermarket.