DBMS PROJECT

LULU ONLINE RETAIL STORE

By- Akshay Ambaprasad (2021444), Anurag Gupta(2021451)

SCOPE OF PROJECT

The goal of this project is to create a comprehensive system that can handle all of the technological needs and functionalities of the online retail business. The system will be in charge of controlling the inventory, sales, accounts payable, and more. The system must also have a user-friendly interface, be scalable, secure, and accessible around-the-clock. The objective is to increase the efficacy and efficiency of the business' operations and to facilitate data-driven performance evaluation.

These are the project's goals:

1. **To increase the speed and ease with which processes can be completed that cannot be done manually**
2. **To enhance data-driven decision-making and performance monitoring by delivering real-time data and analytics.**
3. **To improve the user experience by offering a web portal or application that makes it simple to add, update, and delete records as well as monitor and control all functionalities.**
4. **Customers can log in, view the merchandise, place a purchase, and track the progress of their order.**
5. **Employees can log in and view sales, inventory, financial information, and distributor information.**

Technical requirements -

The following will be used for the front-end:

MySQL

Python

To ensure the efficient operation of the website, we must assure payment gateway integration, product inventory management, shopping cart and checkout functionality, order tracking and management, delivery management, and raw materials supply.

Functional Standards -

For the Customer-

1.To maintain the security and safety of data and its access, when creating an account and logging in, double-check your username and password.

2.Delete, add, and remove items from the cart

3.Ordering - Will place the order and generate a client bill. A warning will be given to the delivery partners.

4.Checking the status of the order .

5.Viewing the available items

6.Payment management - Allow for the use of various payment methods.

For Employee -

1. **Creating an account and logging in (Both for customers and Employees) - Will verify username and password to ensure the security and protection of data and its access.**

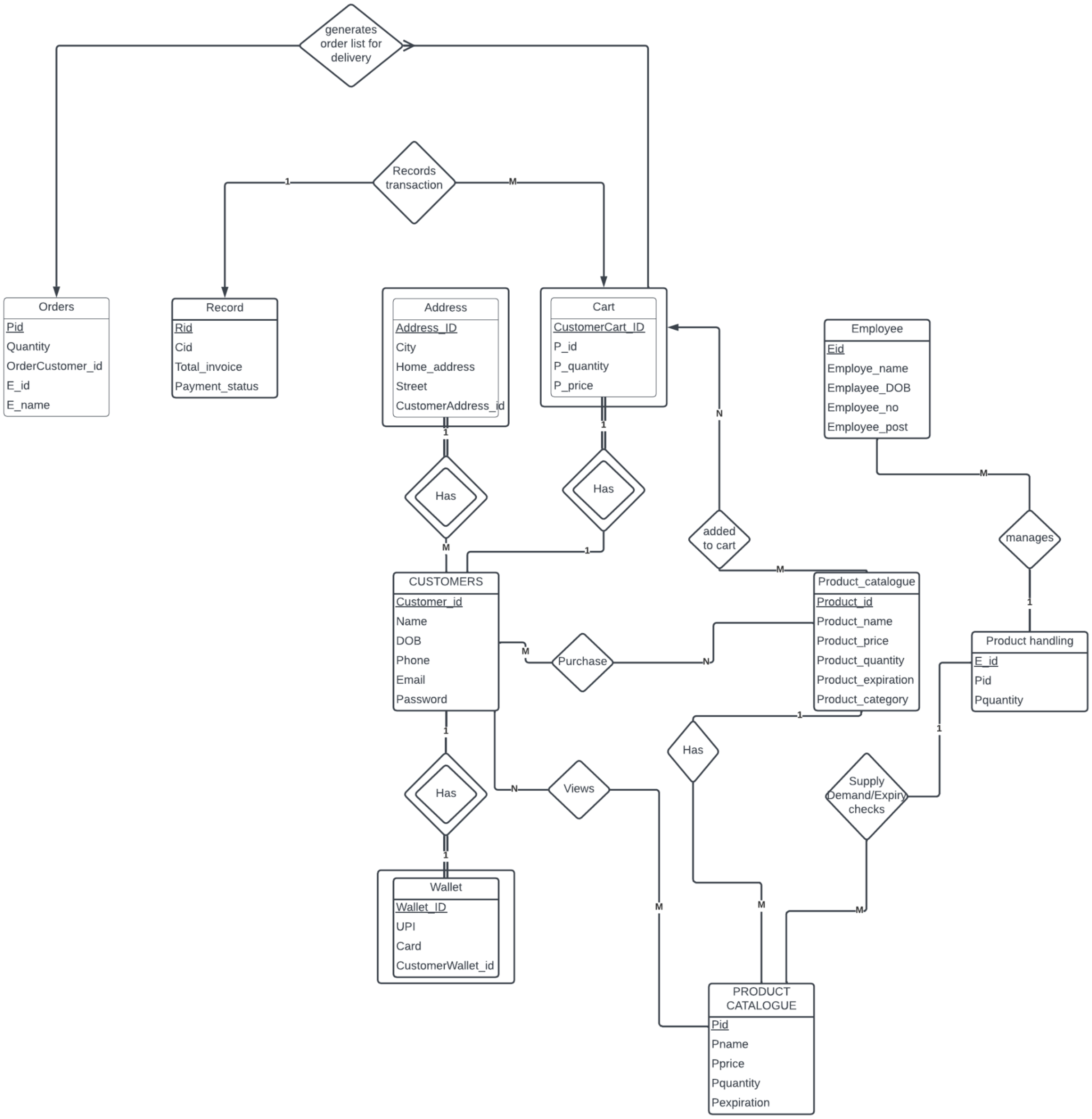
2. **Each Employee will view and access different interfaces and will be able to do different actions based on their Job post and clearance.**

3. **The manager can check the sales, inventory, finances,can also control and edit the workforce and add , delete or edit information on the products**

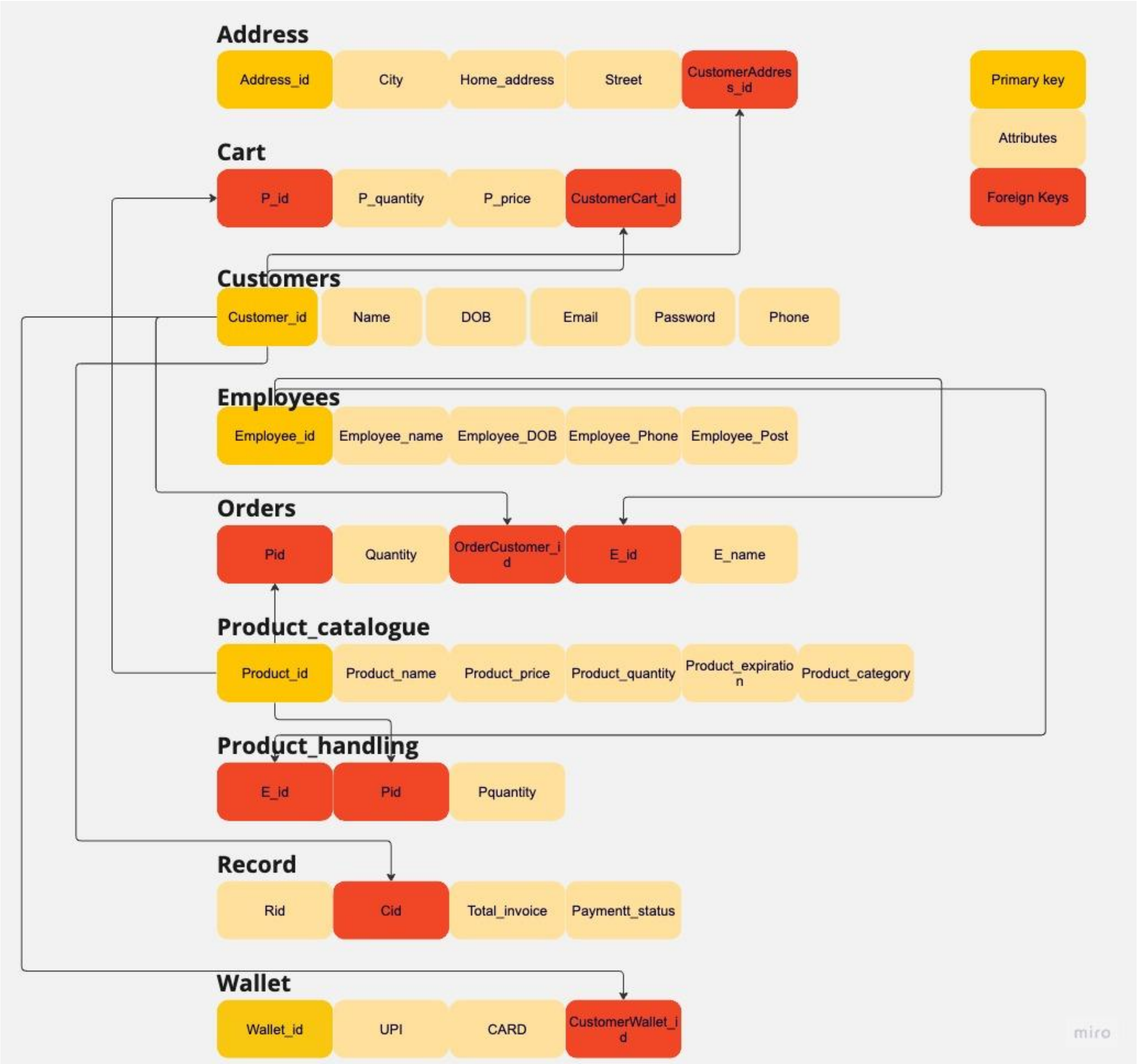
4. **The floor workers are assigned work for which they can view and clock in and clock out**

5. **The delivery men access information about their deliveries for the day.**

ER Diagram -


generates order list for delivery
Records transaction
1
M
Orders
Pid
Quantity
OrderCustomer_id
E_id
E_name
Record
Rid
Cid
Total_invoice
Payment_status
Address
Address_ID
City
Home_address
Street
CustomerAddress_id
Cart
CustomerCart_ID
P_id
P_quantity
P_price
Employee
Eid
Employe_name
Emplayee_DOB
Employee_no
Employee_post
1
1
Has
Has
N
M
added to cart
manages
M
1
M
CUSTOMERS
Customer_id
Name
DOB
Phone
Email
Password
M
Purchase
N
Product_catalogue
Product_id
Product_name
Product_price
Product_quantity
Product_expiration
Product_category
1
Product handling
E_id
Pid
Pquantity
1
1
Has
Has
Supply Demand/Expiry checks
1
N
Views
Wallet
Wallet_ID
UPI
Card
CustomerWallet_id
M
M
M
PRODUCT CATALOGUE
Pid
Pname
Pprice
Pquantity
Pexpiration

Relational Schema



Address

Address_id | City | Home_address | Street | CustomerAddress_id

Cart

P_id | P_quantity | P_price | CustomerCart_id

Customers

Customer_id | Name | DOB | Email | Password | Phone

Employees

Employee_id | Employee_name | Employee_DOB | Employee_Phone | Employee_Post

Orders

Pid | Quantity | OrderCustomer_id | E_id | E_name

Product_catalogue

Product_id | Product_name | Product_price | Product_quantity | Product_expiration | Product_category

Product_handling

E_id | Pid | Pquantity

Record

Rid | Cid | Total_invoice | Paymentt_status

Wallet

Wallet_id | UPI | CARD | CustomerWallet_id

Primary key

Attributes

Foreign Keys

Database Schema-

```
CREATE TABLE `Customers` (
  `Customer_id` int(11) NOT NULL,
  `Name` varchar(50) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `DOB` date DEFAULT NULL,
  `Email` varchar(50) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `Password` varchar(50) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `Membership` varchar(1) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `Phone` varchar(50) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  PRIMARY KEY (`Customer_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;


CREATE TABLE `Employees` (
  `Employee_id` int(11) NOT NULL,
  `Employee_name` varchar(50) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `Employee_DOB` date DEFAULT NULL,
  `Employee_Phone` varchar(50) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `Employee_Post` varchar(50) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  PRIMARY KEY (`Employee_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;


CREATE TABLE `Product_catalogue` (
  `Product_id` int(11) NOT NULL,
  `Product_name` varchar(50) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `Product_price` int(11) DEFAULT NULL,
  `Product_quantity` int(11) DEFAULT NULL,
  `Product_expiration` date DEFAULT NULL,
  PRIMARY KEY (`Product_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

CREATE TABLE `Address` (
  `Address_id` int(11) NOT NULL,
  `City` varchar(50) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `Home_address` varchar(50) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `Street` varchar(50) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `CustomerAddress_id` int(11) DEFAULT NULL,
  PRIMARY KEY (`Address_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;


CREATE TABLE `Cart` (
  `Cart_id` int(11) NOT NULL,
  `P_id` int(11) DEFAULT NULL,
  `P_quantity` int(11) DEFAULT NULL,
  `P_price` int(11) DEFAULT NULL,
  `CustomerCart_id` int(11) DEFAULT NULL,
  PRIMARY KEY (`Cart_id`),
  KEY `CustomerCart_id` (`CustomerCart_id`),
  CONSTRAINT `Cart_ibfk_1` FOREIGN KEY (`CustomerCart_id`) REFERENCES `Customers` (`Customer_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

```
CREATE TABLE `Product_handling` (
  `E_id` int(11) DEFAULT NULL,
  `Pid` int(11) DEFAULT NULL,
  `Pquantity` int(11) DEFAULT NULL,
  KEY `E_id` (`E_id`),
  KEY `Pid` (`Pid`),
  CONSTRAINT `Product_handling_ibfk_1` FOREIGN KEY (`E_id`) REFERENCES `Employees` (`Employee_id`),
  CONSTRAINT `Product_handling_ibfk_2` FOREIGN KEY (`Pid`) REFERENCES `Product_catalogue` (`Product_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

CREATE TABLE `Record` (
  `Rid` int(11) NOT NULL,
  `Bid` int(11) DEFAULT NULL,
  `Cid` int(11) DEFAULT NULL,
  `Total_invoice` int(11) DEFAULT NULL,
  `Payment_status` varchar(9) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  PRIMARY KEY (`Rid`),
  KEY `Bid` (`Bid`),
  KEY `Cid` (`Cid`),
  CONSTRAINT `Record_ibfk_1` FOREIGN KEY (`Bid`) REFERENCES `Bill` (`Bill_id`),
  CONSTRAINT `Record_ibfk_2` FOREIGN KEY (`Cid`) REFERENCES `Customers` (`Customer_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;

CREATE TABLE `Wallet` (
  `Wallet_id` int(11) NOT NULL,
  `UPI` varchar(50) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `Card` varchar(50) COLLATE utf8mb4_unicode_ci DEFAULT NULL,
  `CustomerWallet_id` int(11) DEFAULT NULL,
  PRIMARY KEY (`Wallet_id`),
  KEY `CustomerWallet_id` (`CustomerWallet_id`),
  CONSTRAINT `Wallet_ibfk_1` FOREIGN KEY (`CustomerWallet_id`) REFERENCES `Customers` (`Customer_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_unicode_ci;
```

```
CREATE TABLE `Orders` (`Pid` int NOT NULL,`Quantity` int DEFAULT NULL,`OrderCustomer_id` int DEFAULT NULL,`E_Id` int DEFAULT NULL,`E_name` varchar(50) )
```

SQL Queries:

1.INSERT INTO Customers VALUES(101,"Yaksh Patel","1998-01-01","yaksh21444@iiitd.ac.in","oliverallen","","123456789");

This is an insert operation, so it does not have a relational algebraic expression.

2.SELECT MAX(Total_invoice) AS Largest_order FROM Record;

Π<sub>Total_invoice</sub>(σ<sub>Largest_order=Total_invoice</sub>(Record))

3.SELECT AVG(Total_invoice) AS Largest_order FROM Record;

Π<sub>Total_invoice</sub>(γ<sub>Avg(Total_invoice)</sub>(Record))

4.UPDATE Customers SET Phone = "987654321" WHERE Customer_id = 101;

This is an update operation, so it does not have a relational algebraic expression.

5.INSERT INTO Product_handling (Pid) SELECT Product_id FROM Product_catalogue WHERE Product_quantity=2;

π<sub>Product_id</sub>(σ<sub>Product_quantity=2</sub>(Product_catalogue)) ⟶ <sub>Pid</sub> Product_handling

6.SELECT * FROM Address JOIN Bill ON Address.CustomerAddress_id = Bill.C_id WHERE Address.City = 'Lake Marques' AND Bill.Total_price > 1000;

σ<sub>City='Lake Marques' and Total_price>1000</sub>(Address ⋈<sub>CustomerAddress_id=C_id</sub> Bill)

7.SELECT ph.E_id, ph.Pid, pc.Product_name, pc.Product_price, ph.Pquantity FROM Product_handling ph INNER JOIN Product_catalogue pc ON ph.Pid = pc.Product_id;

Π<sub>E_id,Pid,Product_name,Product_price,Pquantity</sub>(Product_handling ⋈<sub>Pid=Product_id</sub> Product_catalogue)

8.SELECT * FROM Cart JOIN Product_catalogue ON Cart.P_id = Product_catalogue.Product_id WHERE Cart.P_price IS NOT NULL;

σ<sub>P_price is not null</sub>(Cart ⋈<sub>P_id=Product_id</sub> Product_catalogue)

9.SELECT Cid, SUM(Total_invoice) as total_invoice_amount FROM Record GROUP BY Cid;

γ<sub>Cid, total_invoice_amount: SUM(Total_invoice)</sub>(Record)

10.SELECT R.Rid, R.Total_invoice, B.Total_price FROM Record R LEFT
OUTER JOIN Bill B ON R.Bid = B.Bill_id;

R ◁<sub>Bid=Bill_id</sub> B

(Note that this is already a left outer join, so the result is the same as the original query.)

Four OLAP Queries:

1. SELECT Product_name, SUM(P_quantity * P_price) AS Total_sales FROM Cart JOIN Product_catalogue ON Cart.P_id = Product_catalogue.Product_id GROUP BY Product_name;

This SQL query retrieves the total sales amount for each product in the shopping cart by joining the "Cart" and "Product_catalogue" tables based on the "P_id" and "Product_id" columns, respectively.

The query first selects the "Product_name" column from the "Product_catalogue" table and then calculates the total sales for each product by multiplying the quantity of each item purchased ("P_quantity") by its price ("P_price") in the "Cart" table. The results are grouped by product name using the "GROUP BY" clause.

Therefore, the output of this query will show a list of all the product names in the shopping cart and their respective total sales amounts.

```
2. SELECT Name, SUM(P_quantity * P_price) AS Total_sales FROM Cart JOIN Customers ON Cart.CustomerCart_id =
Customers.Customer_id JOIN Product_catalogue ON Cart.P_id = Product_catalogue.Product_id GROUP BY Name;
```

This query retrieves the total sales made by each customer, grouped by their name. It joins three tables: "Cart," "Customers," and "Product_catalogue."

The "Cart" table contains information about customer purchases, including the customer ID, product ID, quantity purchased, and the price per unit.

The "Customers" table contains customer information, including their name and customer ID.

The "Product_catalogue" table contains product information, including the product ID and price.

The query uses the "JOIN" statement to combine information from the three tables based on their respective ID fields. Then, it calculates the total sales for each customer by multiplying the quantity of each product purchased by its price and summing up the results. Finally, it groups the results by customer name using the "GROUP BY" statement.

```
3. SELECT IFNULL(c.Name, 'Total') AS Customer_Name, SUM(b.Total_price) AS Total_Sales FROM Customers c LEFT JOIN
Bill b ON c.Customer_id = b.C_id GROUP BY c.Name WITH ROLLUP;
```

This SQL query retrieves the total sales for each customer in the Customers table along with the total sales across all customers. It uses a LEFT JOIN to join the Customers table with the Bill table on the Customer_id and C_id columns respectively.

The IFNULL function is used to replace any null customer names with the string "Total" in the result set.

The GROUP BY clause groups the results by customer name and the WITH ROLLUP modifier adds an extra row at the end of the result set that displays the total sales across all customers.

The SUM function is used to calculate the total sales for each customer by summing the Total_price column from the Bill table.

In summary, this query retrieves a report of total sales by customer and includes a summary row at the end showing the total sales across all customers.

```
4. SELECT Employee_name, Product_name, SUM(Pquantity) AS TotalQuantityHandled
 FROM Product_handling JOIN Employees ON Product_handling.E_id = Employees.Employee_id JOIN Product_catalogue
ON Product_handling.Pid = Product_catalogue.Product_id
 GROUP BY Employee_name, Product_name
 WITH ROLLUP;
```

This query retrieves the total quantity of each product handled by each employee and also provides a total for each employee and for all employees combined.

It joins the "Product_handling" table with "Employees" and "Product_catalogue" tables using their respective IDs. It calculates the sum of Pquantity for each combination of employee and product using the GROUP BY clause.

Finally, it adds a WITH ROLLUP statement to include additional rows that provide subtotals for each employee and a grand total for all employees. The resulting table will have three columns: Employee_name, Product_name, and TotalQuantityHandled.

TWO TRIGGERS:

```
1.DELIMITER //
 CREATE TRIGGER check_customer_balance BEFORE INSERT ON Cart
 FOR EACH ROW
 BEGIN
 DECLARE customer_balance INT;
 SELECT Membership , Phone , Name , Password , DOB , Email , Customer_id INTO @Membership, @Phone, @Name, @Password,@DOB, @Email, @Cid FROM Customers WHERE Customer_id = NEW. CustomerCart_id ;
 SELECT SUM( P_quantity * P_price ) INTO @total_price FROM Cart WHERE CustomerCart_id = NEW. CustomerCart_id ; SELECT Balance INTO customer_balance FROM Customers WHERE Customer_id = NEW. CustomerCart_id ;
 IF (@Membership = 'Y' AND @total_price > 1000) THEN
 SET @discounted_price = @total_price * 0.9;
 ELSE
 SET @discounted_price = @total_price;
 END IF;
 IF (@discounted_price > customer_balance) THEN
 SIGNAL SQLSTATE '45000'
 SET MESSAGE_TEXT = 'Insufficient balance';
 END IF;
 END
 DELIMITER ;
```

```
2.DELIMITER //

CREATE TRIGGER delete_expired_product BEFORE INSERT ON Product_catalogue FOR EACH ROW BEGIN
 IF NEW.Product_expiration < NOW() THEN
 DELETE FROM Product_catalogue WHERE Product_id = NEW. Product_id ;

END IF; END //
```

```
DELIMITER ;
```

Here are 4 non-conflicting transactions:

Transaction 1:

sql

Copy code

```
BEGIN;

INSERT INTO Customers (Customer_id, Name, DOB, Email, Password, Membership, Phone) VALUES (1, 'John Doe', '1990-05-20', 'johndoe@email.com', 'password123', 'Y', '123-456-7890');

INSERT INTO Employees (Employee_id, Employee_name, Employee_DOB, Employee_Phone, Employee_Post) VALUES (1, 'Jane Smith', '1985-02-10', '234-567-8901', 'Manager');

COMMIT;
```

Transaction 2:

sql

Copy code

```
BEGIN;

INSERT INTO Product_catalogue (Product_id, Product_name, Product_price, Product_quantity, Product_expiration) VALUES (1, 'Product 1', 100, 50, '2024-01-01');

INSERT INTO Address (Address_id, City, Home_address, Street, CustomerAddress_id) VALUES (1, 'New York', '123 Main St', 'Main St', 1);

COMMIT;
```

Transaction 3:

sql

Copy code

```
BEGIN;

UPDATE Customers SET Membership = 'N' WHERE Customer_id = 1;

UPDATE Product_catalogue SET Product_quantity = Product_quantity - 10 WHERE Product_id = 1;

COMMIT;
```

Transaction 4:

sql

Copy code

```
BEGIN;

DELETE FROM Product_catalogue WHERE Product_quantity = 0;

DELETE FROM Customers WHERE Membership = 'N' AND Phone IS NULL;

COMMIT;
```

And here are 2 conflicting transactions:

Conflicting Transaction 1:

sql

Copy code

```
-- Transaction 1

BEGIN;
```

```
UPDATE Customers SET Membership = 'N' WHERE Customer_id = 1;

COMMIT;


-- Transaction 2 (conflicting with Transaction 1)

BEGIN;

UPDATE Customers SET Phone = '555-123-4567' WHERE Customer_id = 1 AND Membership = 'Y';

COMMIT;
```

Conflicting Transaction 2:

sql

Copy code

```
-- Transaction 1

BEGIN;

UPDATE Product_catalogue SET Product_quantity = Product_quantity - 20 WHERE Product_id = 1;

COMMIT;


-- Transaction 2 (conflicting with Transaction 1)

BEGIN;

UPDATE Product_catalogue SET Product_price = Product_price * 0.9 WHERE Product_id = 1 AND Product_quantity > 20;

COMMIT;
```

Transaction	Operations	Table(s)	Read/Write
T1	Insert into Customers Insert into Employees	Customers Employees	Write
T2	Insert into Product_catalogue Insert into Address	Product_catalogue Address	Write
T3	Update Customers Update Product_catalogue	Customers Product_catalogue	Write
T4	Delete from Product_catalogue Delete from Customers	Product_catalogue Customers	Write
Conflict 1	Update Customers (Membership = 'N') Update Customers (Phone = '555-123-4567')	Customers	Write
Conflict 2	Update Product_catalogue (Product_quantity - 20) Update Product_catalogue (Product_price * 0.9)	Product_catalogue	Write

User Guide-

When the code is run, a menu is displayed with login options for our stakeholders i.e Customer , Employee

Upon logging in using username and password, several features are available.

Or Name and ID for Employees

For Employees:

There are four types of employees in the Lulu Hypermarket Management System:

Employee Manager: This employee has access to the following features:

> View Employees: The manager can view all the employees in the Employee table.
>
> View Sales: The manager can view all the sales records in the Record table.
>
> View Products Handling: The manager can view all the product handling records in the Product_handling table.

Edit Workforce: The manager can add, remove, or update employee records in the Employee table.

Employee Floor: This employee has access to the following features:

View Assignment Details: The employee can view the product handling assignments they have been given.

Clock In: The employee can clock in for work.

Clock Out: The employee can clock out from work.

Employee Product Management: This employee has access to the following features:

View Products: The employee can view all the products in the Product_catalogue table.

Add Products: The employee can add new products to the Product_catalogue table.

Remove Products: The employee can remove products from the Product_catalogue table.

Update Products: The employee can update the details of products in the Product_catalogue table.

View Product Handling Details: The employee can view the product handling assignments in the Product_handling table, and can also assign employees to these assignments.

Order Delivery: This employee has access to the following features:

View Orders to Deliver: The employee can view the orders that they are assigned to deliver. This involves selecting the Pid, Quantity and display the address of the customer from the Address table.

Overall, the different features available to each employee depend on their role within the Lulu Hypermarket Management System. The system is designed to provide different levels of access and functionality to employees based on their responsibilities and tasks within the hypermarket.

For Customers:

The code consists of three main parts: customer login, customer shopping, and manage account.

When the customer logs in, they can either manage their account details, go shopping or logout. When the customer chooses to go shopping, they can either view products, view their cart, checkout or go back. They can also add or remove products to/from their cart. When the customer is ready to checkout, the details of the order are stored in the database.

When the customer chooses to manage their account, they can either view or edit their account details such as name, address, phone number, email, and password.