DBMS PROJECT - GROUP 26

Aryaman Raina
Faizan Haider
Mohammad Aflah Khan
Shivaansh Mital

Online Retail Management System

Scope & Objectives

With the technological advancements the world has encountered & the growth of internet users worldwide, the scope for e-commerce activities is expanding day by day. Especially with the onset of the COVID-19 Pandemic, several people have resorted to online shopping to fulfill their short-term as well as long-term requirements.

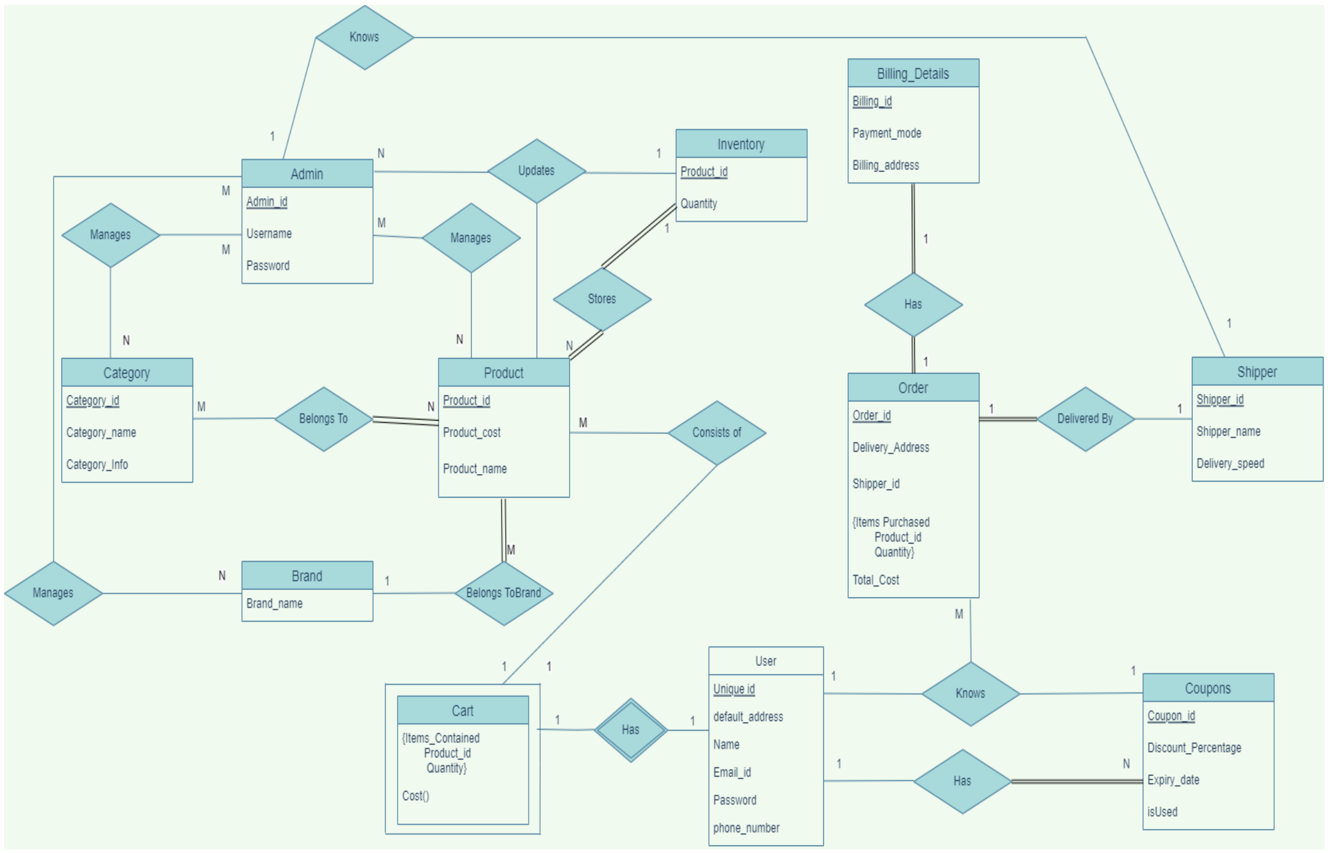
With this project, we aim to design our own online retail management system. The main objective of this project is to store & manage the details of various products, orders, shippers & users that would engage in online retail activities. This provides an easy gateway for users to navigate & search for products as compared to offline retail stores.

Several products can be stored in the database that can belong to certain categories and brands. Users can choose from a wide range of products that are available and add them to their respective carts. Once the cart is filled, users can checkout to place an order and pay using their billing details. The order is delivered to the user by the shipper, based on the shipper's delivery speed. With the provision of Coupons, users can also avail discounts on their orders.

Stakeholders Identified

1. **Users:** The users can select products that they want to purchase & expect that their products will be delivered based on the shipper's speed.
2. **Shippers:** The shippers can get the details of the various orders placed by users from the database & perform their delivery actions accordingly.
3. **Admin:** The Admin can be in charge of managing or updating any database details of the retail store system, such as the products, their categories, inventories, etc. whenever necessary.

Entity Relationship Diagram



Entities, Attributes & Schemas

1. admin_table

   Schema: admin_table(admin_id, username, password)

admin_id	INT PRIMARY KEY NOT NULL AUTO_INCREMENT
username	VARCHAR(50) NOT NULL
password	VARCHAR(50) NOT NULL

2. product

   Schema: product(product_id, product_name, product_cost, brand_name(FK))

product_id	INT PRIMARY KEY NOT NULL AUTO_INCREMENT
product_name	VARCHAR(50) NOT NULL
product_cost	VARCHAR(50) NOT NULL CHECK product_cost>0
brand_name	VARCHAR(50) NOT NULL FOREIGN KEY REFERENCES Brand

3. order_table

Schema: order_table(Order_id, Delivery_Address, Shipper_id, DateTime, Unique_id, Billing_id, couponID)

Order_id	INT PRIMARY KEY NOT NULL AUTO_INCREMENT
Delivery_Address	VARCHAR(50) NOT NULL
Shipper_id	INT NOT NULL FOREIGN KEY REFERENCES SHIPPER
DateTime	DATE NOT NULL
Unique_id	INT NOT NULL FOREIGN KEY REFERENCES USER
Billing_id	INT NOT NULL FOREIGN KEY REFERENCES BILLING_DETAILS
couponID	Varchar(40) Default NULL Foreign Key references Coupon_Data

4. shipper

Schema: shipper(Shipper_id, Shipper_name, Delivery_speed)

Shipper_id	INT PRIMARY KEY NOT NULL AUTO_INCREMENT
Shipper_name	VARCHAR(50) NOT NULL
Delivery_speed	INT, NOT NULL

5. user

Schema: user(id, Address, Name, EmailID, Password, PhoneNumber)

id	INT PRIMARY KEY NOT NULL AUTO_INCREMENT
Address	VARCHAR(50) NOT NULL
Name	VARCHAR(50) NOT NULL
EmailID	VARCHAR(50) NOT NULL UNIQUE
Password	VARCHAR(50) NOT NULL
PhoneNumber	VARCHAR(50) NOT NULL

6. coupon_data

Schema: coupon_data(Coupon_id, Discount, Expiry_Date, Unique_id,isUsed)

Coupon_id	VARCHAR(40) PRIMARY KEY NOT NULL AUTO_INCREMENT
Discount	INT NOT NULL CHECK DISCOUNT>0
Expiry_Date	DATE NOT NULL
Unique_id	INT NOT NULL FOREIGN KEY REFERENCES USER
isUsed	INT Default value 0

7. billing_details

Schema: billing_details(billing_id, payment_mode, billing_address)

billing_id	INT PRIMARY KEY NOT NULL AUTO_INCREMENT
payment_mode	VARCHAR(30) NOT NULL
billing_address	VARCHAR(50) NOT NULL

8. brand

Schema: brand(<u>brand_name</u>)

brand_name	VARCHAR(50) PRIMARY KEY NOT NULL

9. inventory

Schema: inventory(<u>product_id</u>, quantity)

product_id	INT PRIMARY KEY NOT NULL FOREIGN KEY REFERENCES PRODUCT
quantity	INT NOT NULL CHECK Quantity > 0

10. cart_data

Schema: cart_data(<u>Unique_ID</u>)

Unique_id	INT PRIMARY KEY NOT NULL FOREIGN KEY REFERENCES User CHECK Quantity > 0

11. category

Schema: category(<u>category_id</u>, category_name, category_info)

category_id	INT PRIMARY KEY NOT NULL AUTO_INCREMENT

category_name	VARCHAR(50) NOT NULL
category_info	VARCHAR(50)

Schemas for Relations & Multivalued Attributes

1. items_contained (in Cart)

   Schema: items_contained(<u>Unique_ID, Product_ID, Quantity</u>)

Unique_id	INT NOT NULL FOREIGN KEY REFERENCES Cart_Data
Product_ID	INT NOT NULL
Quantity	INT NOT NULL CHECK Quantity>0

   PRIMARY KEY (Unique_id, Product_ID, Quantity)

2. items_purchased (in Order)

   Schema: items_purchased(<u>Order_id, Product_ID,</u> Quantity, Cost)

Order_id	INT NOT NULL FOREIGN KEY REFERENCES Order
Product_ID	INT NOT NULL
Quantity	INT NOT NULL Check Quantity > 0

Cost	INT DEFAULT=0 Check Cost > 0

PRIMARY KEY (Order_id, Product_ID)

3. belongsTo (relation b/w Product & Category)

   Schema: belongsTo(product_id,category_id)

product_id	INT NOT NULL FOREIGN KEY REFERENCES Product
category_id	INT NOT NULL FOREIGN KEY REFERENCES Category

PRIMARY KEY (product_id,category_id)

Weak entity:

Taking inspiration from Big Bazaar, we tried to imitate the functioning of our database model similar to it. Since on Big Bazaar, there is no functionality of a guest cart therefore similar to that we have made our cart which needs to be associated to a user, therefore, Cart is a weak entity in our database model. It is a weak entity because a cart is related to a user and only when we have a user does it make sense since the potential products that the user adds to the cart will be his.

Ternary Relation:

1. **Between Inventory, Admin, and Product:** The admin is responsible for both managing the inventory as well as the products and the products also share a relationship with the inventory. Hence the three will have a ternary relationship where the Admin can modify Inventory and Product, Changes in Product quantity reflect in Inventory as Inventory uses Product IDs and Order placing or removal of Products also must reflect in the Inventory
2. **Between User, Orders, and Coupons:** Whenever an order is placed the user might use a coupon to place it, hence there exists a ternary relationship between the three entities. The user can book orders and view orders as well as his/her coupons. The coupon also is related to order because if it is being used there and has an isUsed attribute set to 1 indicating used. Orders also know which user they belong to and which coupon was used (if any).

Views & Grants Used

In our project the two of the stakeholders are user and admin and we have two users named `customer` and `administer` corresponding to the user and admin respectively. Now in our database, we have an entity user which contains their emails and passwords similarly the admin table contains a username and passkey. Now, these users are identified by their given password and the user is their username. Now, corresponding to the user customer we grant it access to select products, brands, shippers, category, belongsTo since the user needs to be able to see the available products, brands, shippers, category, and the products the which belongs to a category. However, he should only have select access and not any other since he is just a customer and should only be allowed to view these things and not change anything about them.

The user has update access on order_table, inventory, items_purchased, billing_details, items_contained, and coupon_data the reason being that when the user applies a coupon it should set isUsed attribute of that coupon, the inventory needs to be updated when he buys a product, now when the user buys products he can change the quantity, therefore, needing to update items_purchased, now since the user can update the items in his cart he can add same products that are already present in his cart causing him to modify the quantity of that product in his cart, therefore, he should be given update access for that. Further, the customer has been given access to insert in items_contained, billing_details,order_table, and items_purchased since when the user adds items to his cart he needs to have insert access for items_contained, when he buys products he needs to insert in billing_details,order_table, and items_purchased. He also has access to delete elements from items_purchased, order_table, and billing_details since he can delete orders. He also has to delete access from items_contained since he has the option to remove items from his cart.

The user administrator has all privileges to almost all the tables except for admin_table where he can only see other admins but he cannot add/delete/update any entries in the table that privilege has only been given to the root. He does not have access to cart_details and items_contained since the administrator does not need to know/should not have access to a user's cart and the items in it.

For customer:

```
CREATE USER customer@localhost IDENTIFIED BY '1234'

grant select on retaildb.usableCouponView to customer@localhost

grant select on retaildb.userProductView to customer@localhost

grant select on retaildb.categoryUserView to customer@localhost

grant select,update on retaildb.inventory to customer@localhost

grant select, update,insert,delete on retaildb.items_purchased to customer@localhost

grant select, update,insert,delete on retaildb.order_table to customer@localhost

grant select on retaildb.shipper to customer@localhost")

grant select, insert,update,delete on retaildb.billing_details to customer@localhost

grant select on retaildb.belongsto to customer@localhost

grant select on retaildb.product to customer@localhost
grant select on retaildb.brand to customer@localhost

grant select on retaildb.category to customer@localhost

grant   select,update,insert,delete   on   retaildb.items_contained   to
customer@localhost

grant select on retaildb.cart_data to customer@localhost

grant select,update on retaildb.coupon_data to customer@localhost

grant select on retaildb.protectedUserView to customer@localhost
```

For Administer:

```
CREATE USER administer@localhost IDENTIFIED BY '{passw}'

grant all on retaildb.user to administer@localhost

grant all on retaildb.coupon_data to administer@localhost

grant all on retaildb.category to administer@localhost

grant all on retaildb.brand to administer@localhost

grant all on retaildb.product to administer@localhost

grant all on retaildb.belongsto to administer@localhost

grant all on retaildb.billing_details to administer@localhost

grant select on retaildb.admin_table to administer@localhost

grant all on retaildb.shipper to administer@localhost

grant  select,update,insert,delete,create,drop  on  retaildb.order_table  to
administer@localhost

grant  select,update,insert,delete,create,drop  on  retaildb.items_purchased
to administer@localhost
```

```
grant all on retaildb.inventory to administer@localhost

grant all on retaildb.userProductView to administer@localhost

grant all on retaildb.categoryUserView to administer@localhost

grant all on retaildb.protectedUserView to administer@localhost

grant all on retaildb.usableCouponView to administer@localhost
```

We have **four views** named userProductView, used to show the customer just the product name its brand, and its cost since the user does not need to know the product id of the product. Similarly, categoryUserView is used to hide the category used internally from the customer's view, protectedUserView is used to show the admin the users on our online retail store but used to protect their password by hiding it from the administer's view. Furthermore, a usableCouponView is used to display only the coupons which can be used i.e have not expired and have not been used till now.

```
-- View products using customer priviliges

Create VIEW userProductView AS

SELECT product_name, product_cost, brand_name

From product;

-- View categories from customer privileges

create view categoryUserView as

select category_name, category_info from category;

-- View users using admin privileges

Create VIEW protectedUserView AS

SELECT id, address, name, EmailID, PhoneNumber

From user;
```

```
-- View Usable Coupons

CREATE VIEW usableCouponView AS

select Coupon_id, Discount, ExpiryDate, Unique_id

from coupon_data

where coupon_data.ExpiryDate > CURRENT_DATE AND coupon_data.isUsed = 0;
```

Indexing Attributes

```
create unique index categoryname on category(category_name);
```

The above index is useful for queries involving searching directly by category name (category_name's are unique)

```
create index shipperspeed on shipper(Delivery_speed);
```

The above index is really helpful in listing out shippers by there speed

```
create index brandn on brand(brand_name);
```

It is used fir searching via brandname

```
create unique index prod_id on product(product_id);
```

This index comes really handy while using queries involving product_id.product_id is used at a number of places like searching deletion updation and other queries

```
create unique index product_name_brand_name on
product(product_name,brand_name);
```

This index helps to fasten the process of searching using product names and brand since.The pair of product name and brand name is unique

```
create index billing on billing_details(billing_id);
```

Since there are queries that involve deletion based on billing_id this index helps in those queries

```
create unique index username_password on user(EmailID,Password);
```

This index helps in searching for a user when he enters his username and password(the username and password combination is unique)

```
create unique index items_contained_index on items_contained(Unique_id ,
Product_ID, Quantity);
```

This index helps in queries where we add products that are already present in the cart where updation of that product's quantity is used.

Triggers Implemented

We've implemented two Triggers one which handles product costs in the order table and another which handles coupons being used for a given order.

Trigger 1:

```
DELIMITER $$

CREATE TRIGGER `getCurrentCost` BEFORE INSERT ON `items_purchased`

FOR EACH ROW BEGIN

      SET NEW.Cost = (Select product_cost*NEW.Quantity From product Where
product_id = NEW.Product_ID);

    UPDATE order_table

    Set totalCost =

    Case

      WHEN (couponID IS NOT NULL) THEN totalCost + (NEW.Cost)*(1-((Select
Discount From coupon_data Where coupon_data.Coupon_id = couponID)/100))

    ELSE totalCost + (NEW.Cost)

    END

    WHERE NEW.Order_id = order_table.Order_id;

END $$

DELIMITER ;

DELIMITER $$
```

The need for these arise as a product's cost can be traced in the product table using product ID however the admin may change the cost and then there is no way to retrieve the cost of the product when an order was placed in the past, hence at the time of order, whenever an item is added to items_purchased we don't give a cost value for that item and rather use the trigger to get the current cost from the product table multiply it by the quantity and set it as the item cost in the items_purchased row entry. At the same time, we also set the total cost of the order in the order table after each insertion in the items_purchased table by incrementing it by the cost while taking care of coupons used by applying a discount if any.

Trigger 2:

```
DELIMITER $$

CREATE TRIGGER `setCoupon` BEFORE INSERT ON `items_purchased`

FOR EACH ROW BEGIN

    UPDATE coupon_data, order_table

    Set isUsed =

    Case

    WHEN (couponID IS NOT NULL) THEN 1

    ELSE 0

    END

     WHERE couponID IS NOT NULL AND couponID = coupon_data.Coupon_id AND
NEW.Order_id = order_table.Order_id;

END $$

DELIMITER;
```

The other trigger is raised to check if a coupon has been applied, if it has been then we set the isUsed value for that row entry in the coupon_data table to 1.

Advanced Aggregate Functions

1) One of the queries uses windowing to show the average purchased cost of each product category next to purchased products by PARTITIONING BY category_id.

```
SELECT T.Product_ID,T.category_id,T.Cost,AVG(T.Cost) OVER( PARTITION
BY T.category_id) AS Avg_Category_Cost

FROM (SELECT P.Product_ID AS Product_ID,B.category_id AS
category_id, SUM(P.Cost) AS Cost FROM items_purchased P NATURAL JOIN
belongsTo B GROUP BY P.Product_ID) AS T;
```

2) One of the queries uses ranking to sort items on the homepage on the basis of how many items of a particular product were sold.

```
Select Product_ID, HighestSeller from (

            SELECT  Product_ID, SUM(Quantity) Quantity, rank () over
(order by Quantity desc) as HighestSeller

            FROM    items_purchased

            GROUP BY Product_ID) as H
```

3) Another query uses ranking to sort items on the homepage on the basis of how much money was spent on each product.

```
 Select Product_ID, HighestSeller from (

            SELECT  Product_ID, Quantity, SUM(Cost) Cost, rank ()
over (order by Cost desc) as HighestSeller

            FROM    items_purchased

            GROUP BY Product_ID) as H
```

List of SQL Queries (Embedded + Non-Embedded)

Show the average purchased cost of each product:

This query uses windowing to show the average purchased cost of each product category next to purchased products by PARTITIONING BY category_id.

```
SELECT T.Product_ID,T.category_id,T.Cost,AVG(T.Cost) OVER( PARTITION BY
T.category_id) AS Avg_Category_Cost

FROM (SELECT P.Product_ID AS Product_ID,B.category_id AS category_id,
SUM(P.Cost) AS Cost FROM items_purchased P NATURAL JOIN belongsTo B GROUP
BY P.Product_ID) AS T;
```

Rank items based on their selling quantity :

This query uses ranking to sort items on the homepage on the basis of how many items of a particular product were sold.

```
Select Product_ID, HighestSeller from (

                 SELECT  Product_ID, SUM(Quantity) Quantity, rank () over
      (order by Quantity desc) as HighestSeller

                 FROM    items_purchased

                 GROUP BY Product_ID) as H
```

Rank items based on their selling prices:

Another query uses ranking to sort items on the homepage on the basis of how much money was spent on each product.

```
       Select Product_ID, HighestSeller from (

                 SELECT  Product_ID, Quantity, SUM(Cost) Cost, rank ()
      over (order by Cost desc) as HighestSeller

                 FROM    items_purchased

                 GROUP BY Product_ID) as H
```

Calculate total cart cost:

This query can be used to calculate the total value of the cart of all the users when products are in the inventory.

```
select Temp.Username, SUM(Temp.Total) as "Total cost"

              from  (select  I.Unique_id,I.Product_ID,U.NAME  as  Username,
SUM(I.Quantity*P.product_cost) as Total

        from User U, items_contained I,product P

               where  P.product_id=I.Product_ID  and  I.Unique_id=U.id  and
P.product_id IN (select product_id from inventory where quantity>0) Group
BY I.Unique_id,I.Product_ID) as Temp

        group by Temp.Username;
```

Cart total post coupon:

```
cursor.execute(f"""

            Select *

            From coupon_data

            where Coupon_id = {coupon_id}

        """)

        coupon_details = cursor.fetchall()

        if (len(coupon_details) == 0):

            return "Coupon Not Found"

        if (coupon_details[0][-1] == 1):

            return "Coupon is Used"
        dateExpiry = coupon_details[0][2]

        print(dateExpiry)

        dateToday = datetime.date.today()

        if (dateExpiry < dateToday):

            return "Expired Coupon"

        cursor.execute(f"""

                    select * from (select Temp.Unique_id, Temp.Username,
SUM(Temp.Total) as "Total cost"

                  from (select I.Unique_id,I.Product_ID,U.NAME as Username,
SUM(I.Quantity*P.product_cost) as Total

            from User U, items_contained I,product P

                  where P.product_id=I.Product_ID and I.Unique_id=U.id and
P.product_id IN (select product_id from inventory where quantity>0) Group
BY I.Unique_id,I.Product_ID) as Temp

             group by Temp.Unique_id) as BigTemp where BigTemp.Unique_id =
{user_id}

            """)

        result = cursor.fetchall()

        totalCost = float(result[0][-1])

        discount = float(int(coupon_details[0][1])/100)

        discountedCost = totalCost - totalCost * discount

        return flask.jsonify(discountedCost)
```

Buy Now:

This query allows a user to purchase his cart and place an order all while emptying his cart and updating the inventory. It also sets the coupon as used if it is given optionally.

```
select product_id, product_name,brand_name from product where product_id
IN (select items_contained.Product_ID as id from items_contained,inventory
where items_contained.Unique_id=1 and
items_contained.Product_ID=inventory.product_id and
items_contained.Quantity>inventory.quantity);
```

```
insert into billing_details (payment_mode, billing_address) values ('Net
Banking', '2923 Street');
```

```
select max(billing_id) from billing_details;
```

```
insert into order_table (Delivery_Address,Shipper_id, Date_Time,
Unique_id, billing_id ) values ('2923 Street',1,CURDATE(), 1, 31);
```

```
insert into order_table (Delivery_Address,Shipper_id, Date_Time,
Unique_id, billing_id,couponID ) values ('2923 Street',1,CURDATE(), 1,
31,'67697c5b-3ee5-4f67-921e-ae8dbb7d31d0');
```

```
Select max(Order_id) from order_table;
```

```
INSERT INTO items_purchased (Order_id, Product_ID, Quantity) SELECT 31,
Product_ID, Quantity FROM items_contained where Unique_id=1;
```

```
UPDATE inventory, items_contained SET inventory.quantity =
inventory.quantity - items_contained.Quantity WHERE
items_contained.Unique_id=1 and
items_contained.Product_ID=inventory.product_id;
```

List orders with products in some category:

This query lists all orders where there exists at least one product belonging to a particular category (say Electronics).

```
SELECT * from order_table where order_id in (
        SELECT  DISTINCT  I.order_id  from  items_purchased  I  where
I.product_id in ( SELECT product_id from belongsTo where category_id = (

   SELECT category_id from category where category_name = 'Electronics')

    )

);
```

List out categories and information about them:

This query simply lists out all the available categories and the information about them.

Updating the inventory of a product by searching for its name and brand name:

This query allows one to update the inventory of a product by searching for its name and brand name.

List out orders based on whether some product belongs to a particular category:

This query lists out for us those orders which contain at least one product belonging to some category.

Canceling Order

This query cancels a given order for a user and updates the inventory as well with the items which were present in the canceled order.

```
-- cancel order

-- check whether it is undelivered or not

Select order_table.billing_id From order_table, shipper

Where     order_table.Order_id=4     AND     order_table.Shipper_id     =
shipper.shipper_id                and                DATEDIFF(CURRENT_DATE,
DATE_ADD(order_table.Date_Time, INTERVAL shipper.Delivery_speed DAY)) < 0;

-- update inventory
```

```
UPDATE inventory, items_purchased SET inventory.quantity = inventory.quantity + items_purchased.Quantity WHERE items_purchased.Order_id=4 and items_purchased.Product_ID=inventory.product_id;
```

```
-- delete order
```

```
delete from billing_details where billing_details.billing_id IN (Select billing_id From order_table where order_table.Order_id = 4);
```

Update product cost:

This query allows the admin to update the cost of a product.

Insert a new product:

This query allows the admin to insert a new product which involves adding the details of a product that could possibly involve adding a new brand and further updating the inventory of the added product.

Provide cart summary:

This query allows a user to get the summary of the products added to his cart along with their quantity and the cost of said quantity of those products.

```
-- list out cart summary provied everything is in stock

select P.product_name as "Name",P.brand_name As brand ,P.product_cost as "Product Cost",I.Quantity,P.product_cost*I.Quantity As Cost

from product P,items_contained I

where P.product_id=I.Product_ID and I.Unique_id = 1 and P.product_id IN (select product_id from inventory where quantity>0);
```

Remove a product from the cart:

This query helps the user by allowing him to remove a particular product from his cart.

Remove a user:

This query allows for a user to be deleted only if the user has no pending orders. It also deletes all the order and billing details associated with the user account and removes the cart as well. It also removes the coupons the user possessed.

```
delete from billing_details where billing_details.billing_id

IN (Select billing_id From order_table where order_table.Unique_id = 2)

AND NOT EXISTS (Select * From order_table,shipper

Where order_table.Unique_id=2 and DATEDIFF(CURRENT_DATE,
DATE_ADD(order_table.Date_Time, INTERVAL shipper.Delivery_speed DAY)) < 0

AND order_table.Shipper_id = shipper.shipper_id);

delete from user where user.id = 2 AND NOT EXISTS (Select * From
order_table where order_table.Unique_id=2);
```

Delete Expired Coupons:

This query deletes all coupons the user possessed which are expired and can no longer be used

View Users But Which Restricted Information (Admin Use):

Used views to hide password data from admins hence the query returns all user data apart from passwords.

List all Items Purchased by the User

This query lists all the items purchased by the User across different orders similar to how Amazon displays its order items sequentially.

SQL Data Dump & Code

The Data Dump is present in the code files submitted and is also present as a gist here.

Instructions Required to Run

1) Install Flutter SDK 2.13 or above, Dart, Flask, MySQL Connector, Python 3
2) Install Virtual Devices, iOS/Android, go to frontend/online_retail_store/lib/main.dart
3) Compile & Run the app

Contributions of Individual Members

Mohammad Aflah Khan (2020082)	Aryaman Raina (2020034)	Faizan Haider (2020083)	Shivaansh Mital (2020122)
❖ Contributed to building ER Diagram and identifying identities/ relations ❖ Identified and implemented some User and Coupon based queries ❖ Helped in Data Population of several entities ❖ Helped in creating Project Report. ❖ Creation of Triggers in the database ❖ Made Embedded SQL queries with the creation of Flask API Endpoints in Python.	❖ Helped in providing ideas of some Entities that can be used. ❖ Contributed to ER Diagram formation ❖ Data population of some Entities such as Brand, Inventory, And Billing Details ❖ Forming & coding entity related SQL Queries ❖ Helped in the Creation of Flask API Endpoints for the backend for embedded SQL. ❖ Formation of some queries with advanced aggregate functions.	❖ Helped in identifying key entities and relationships during initial ideation for ER Diagram ❖ Implemented Order and Cart Related queries which involve multiple entities ❖ Helped in creating relationship schemas for the Project Report ❖ Creation of some API Endpoints ❖ Responsible for the front-end of the project using Flutter.	❖ Contributed to the making of the ER diagram which involved identifying entities along with their attributes and the relationship of entities with each other along with others. ❖ Worked on data population of products, categories and admin ❖ Worked on queries related to product categories, admin some related to cart and users. ❖ Worked on queries on the document. ❖ Collected products, categories and brands image links. ❖ Made API endpoints & embedded SQL

❖ Collected product details and image links. ❖ Made queries involving advanced aggregate functions.	❖ Collection of products and image links for frontend ❖ Formation of Project Report Details		queries. ❖ Responsible for making views & grants. ❖ Created the Indices for attributes.

Just a note most of the work done for this submission was a group effort and was done by all of us while on a meet trying to assist each other in any way needed