Online Retail Store

Siddharth 2021424
Sunny 2021429

Project Report

April 24, 2023

Scope & Objectives

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

We want several products stored in the database belonging to specific categories and brands. As for users, we want Products to be decluttered so that Users can choose from a wide range of available products 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 supply of Coupons for every user category, users can also avail of discounts on their orders. 1

- Stakeholders
    1. Users
    2. Admin

- **Tech Stack:**

  HTML, CSS, JavaScript, Django, React
  MySQL, Python.

- **Functional Requirements:**

For user:

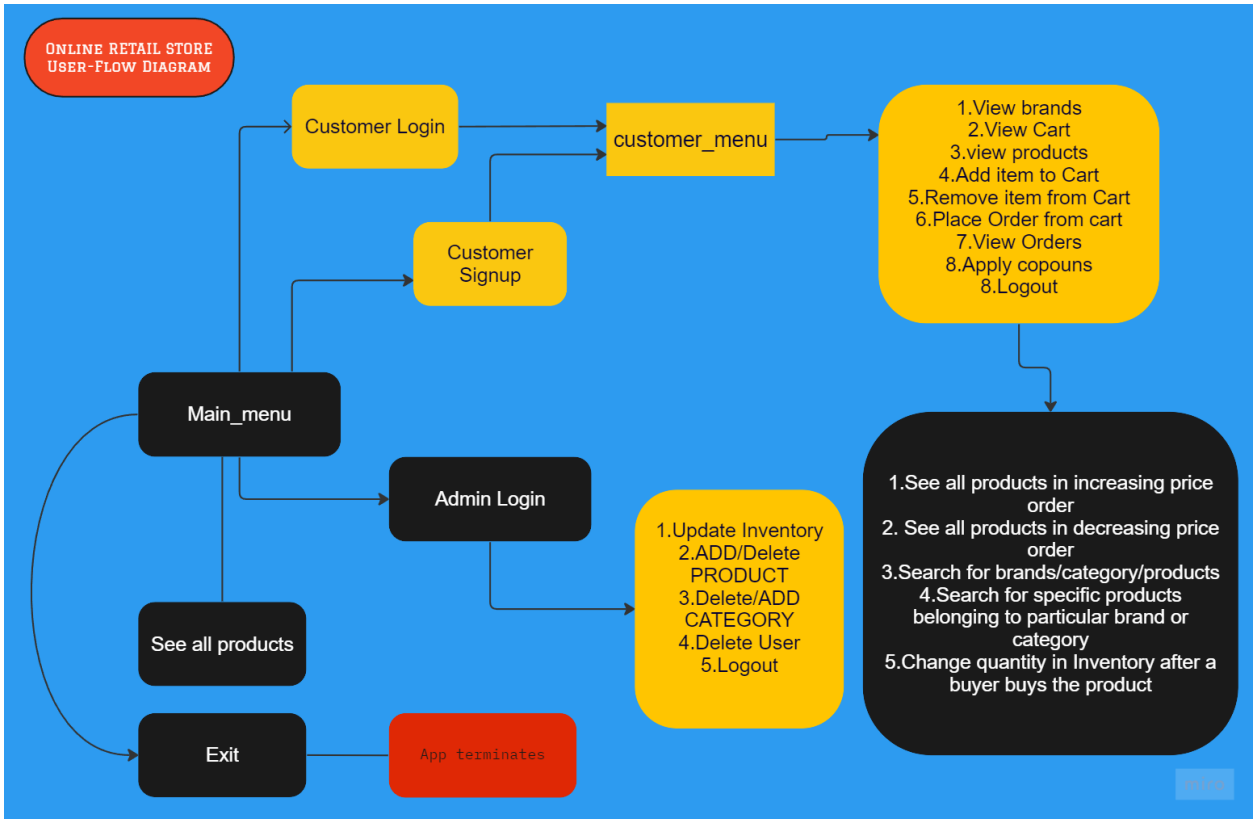
Login/SignUp
Logout/Sign out
Explore Product Catalog
Searching several products from different categories
Adding products to the cart
Returning products
View Discounted prices
Join a Loyalty membership to avail higher discounts
Make Payment
Sorting products based on various filters

For Admin:

Add/remove several products
Add/remove categories
View customer data
Add giveaway deals
Set Discount on Product
Edit product prices

- Store
    1. Admin: admin_id, username,password
    2. Product: product_id ,product_cost
    3. Order: Order_id, delivery_address,cart_data
    4. User: user_id, name , email phone number
    5. Coupon_data: Coupon_id,discount

User Guide



1. Product Catalogue:

Our users will be able to browse all the products available for purchase on your platform within different Categories. Each product will have a detailed description, including its name and price. Users can also consider product reviews or ratings from other customers.

2. Add to Cart:

Once a user finds a product they want to buy, they can add it to their cart by specifying the desired quantity. The cart will keep track of all the items the user has selected to purchase.

3. Remove from Cart:

If a user changes their mind or wants to update their order, they can easily remove items from their cart before proceeding to checkout. This feature will ensure that users have complete control over their purchases and can make changes as needed.

4. Place Order:

Once the user has added all the items they wish to purchase to their cart, they can proceed to the checkout to place their order. During the checkout process, they will be asked to provide their delivery address, contact information, and any other relevant details. If the user has an available coupon and the order meets the minimum order value, they can apply the coupon during checkout. You may also consider providing users with multiple payment options, such as credit card, debit card, or online banking.

5. We can also browse products without logging in as customer.

6. We can search for Specific Brands and their product line.

7. Also users can browse products in Increasing and decreasing order of their prices

8. If a customer wants to see which products are present in the cart, they can easily see that and they add/ remove products from their cart also. User can also choose the quantity of the products they want to buy.

9. After placing the order, the cart gets empty itself.

10. Customers can't add products to their cart if the product quantity in inventory is equal to zero

11. Also, after placing the order, the respective quantity of the products gets decreased from the inventory.

12. The customer also has the option of applying coupon to their order to get a discounted total price of the cart.

13. Admin Menu

The admin has overall access over the stakeholders and has right to remove any such. The admin can also update the product details such as its price and quantity.

The product catalogue feature ensures consistency by providing users with accurate and detailed information about each product. This includes the product name, price, dosage, ingredients, and any relevant warnings or instructions which ensures that users can easily compare products and make informed purchasing decisions.

The add to cart and remove from cart features ensure data integrity by accurately tracking the items that the user has selected to purchase. Users can easily remove items from their cart if they change their minds or want to update their order, which ensures that the system remains up-to-date with the user's intent.

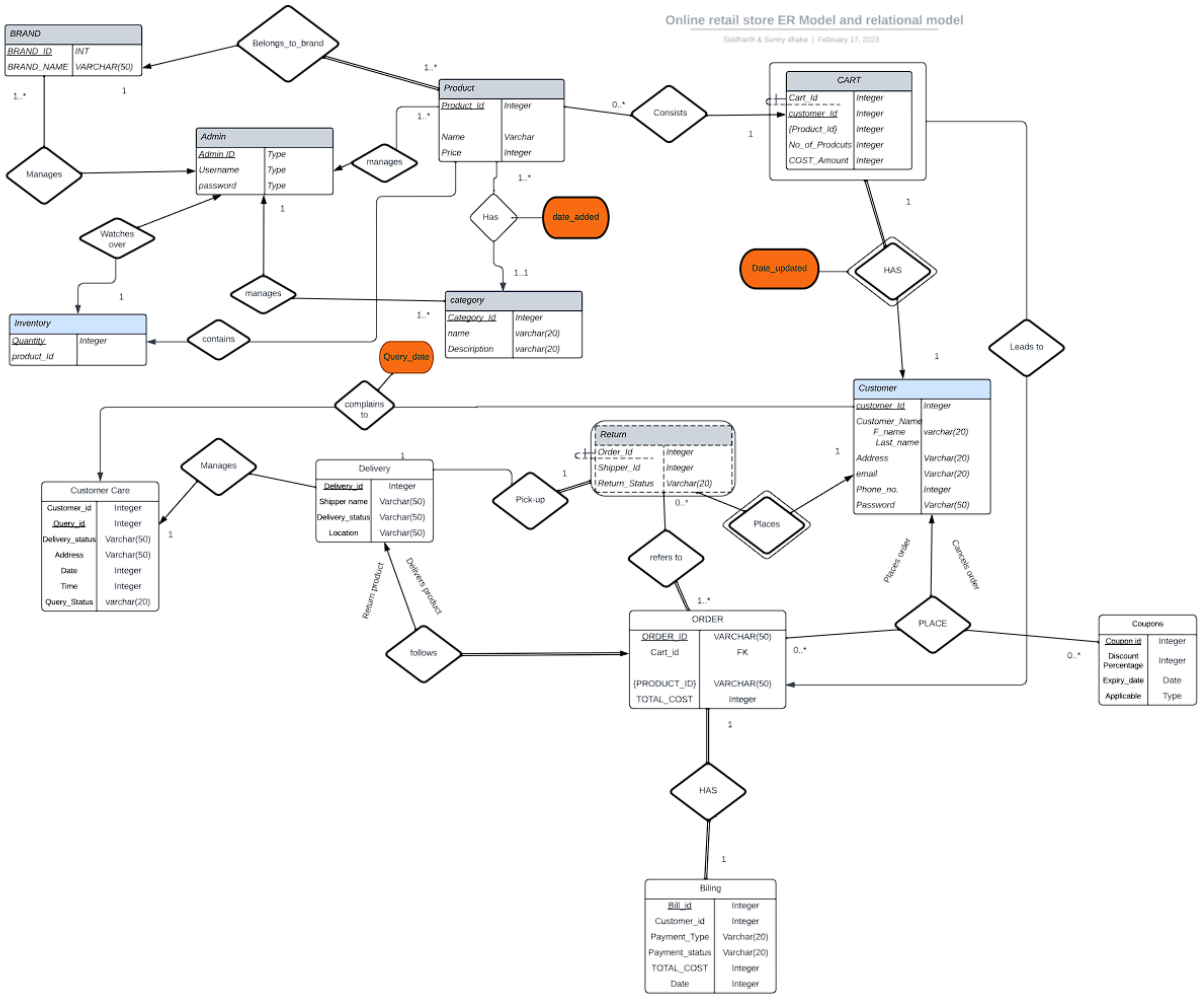
The place order feature ensures data integrity by requiring users to provide accurate information during the checkout process. This includes their delivery address, contact information, and any other relevant details. By verifying this information, the system can ensure that orders are delivered to the correct location and that users can be contacted if there are any issues with their order.

The order history feature ensures consistency by providing users with a comprehensive record of all the orders they have placed on the platform. Users can easily access this

information and use it to reorder items they have previously bought. The feature also ensures data integrity by accurately tracking the status of each order and providing users with up-to-date information about their purchase.

Overall, these features demonstrate a commitment to consistency, integrity, and other important parameters by providing users with accurate information, tracking user intent, and verifying user information. By implementing these features, the platform can ensure a seamless and convenient purchasing experience for users, while also providing valuable feedback for the platform and other potential customers.

Entity Relationship Diagram:



Relational schema

Foreign keys: { } , Primary Keys: Underlined

admin_table (admin_id, username, password)

category (category_id , category_name, category_Description)

brand (brand_id, brand_name)

product (product_id , product_name , product_cost , {brand_id})

has ({product_id , category_id })

inventory ({product_id} , quantity)

customer (customer_Id, Address, Name, EmailID, Password, PhoneNumber)

cart ({Unique_id}, Product_ID, Quantity)

coupon_data (Coupon_id, Discount, ExpiryDate, {Unique_id}, isUsed)

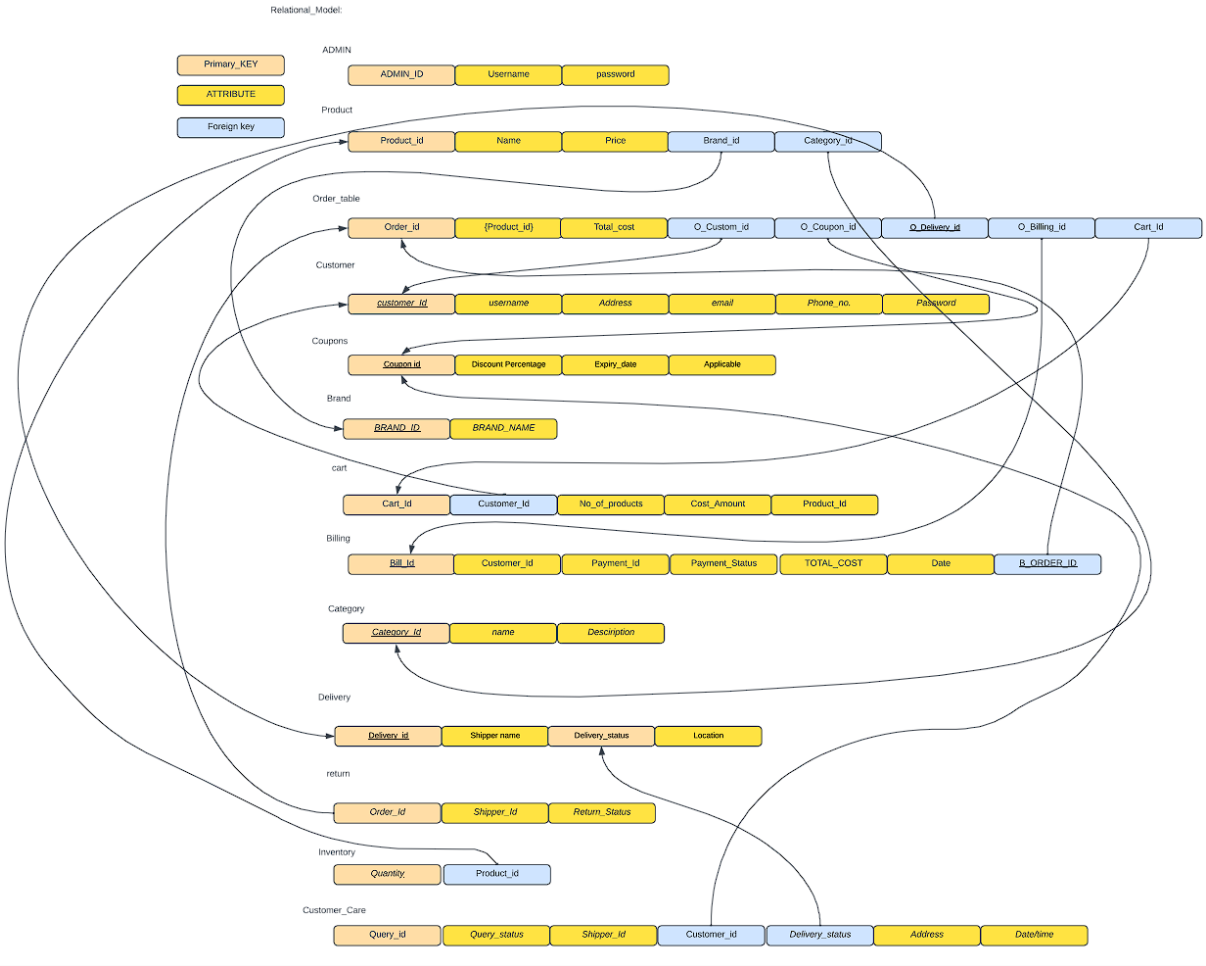
billing_details (billing_id , payment_mode, billing_address)

shipper (shipper_id , shipper_name, Delivery_speed)

order_table (Order_id, Delivery_Address, {Shipper_id} , Date_Time, {Unique_id} , Product_ID, Quantity, {billing_id} , totalcost , {couponID})

returner ({ Order_id, Shipper_id} , Return_Status)

Relational Diagram:



OLAP Queries, Triggers

- Embedded Queries

Query 1:

```
SELECT S.shipper_name, S.delivery_speed FROM shipper S WHERE
S.Delivery_speed >= 2;
```

Query 2:

```
Select * From billing_details, order_table Where
billing_details.billing_id = order_table.billing_id AND
billing_details.billing_id = 1;
```

- OLAP queries

Query 1:

Query to calculate the total sales and profit of each brand and category combination, including subtotals for each brand and category and a grand total:

```
SELECT
    b.brand_name,
    c.category_name,
    SUM(p.product_cost * i.quantity) AS total_sales
FROM
    brand b
    JOIN product p ON b.brand_id = p.brand_id
    JOIN has h ON p.product_id = h.product_id
```

```
    JOIN category c ON h.category_id = c.category_id
    JOIN inventory i ON p.product_id = i.product_id
GROUP BY b.brand_name, c.category_name WITH ROLLUP
ORDER BY b.brand_name, c.category_name;
```

Query 2: used to retrieve sales data from the database by brand and month to generate subtotals and grand totals.

```
SELECT b.brand_name, MONTH(o.Date_Time) AS month,
SUM(p.product_cost * o.Quantity) AS total_sales
FROM brand b
JOIN product p ON b.brand_id = p.brand_id
JOIN order_table o ON p.product_id = o.Product_ID
GROUP BY b.brand_name, MONTH(o.Date_Time) WITH ROLLUP
ORDER BY b.brand_name, month;
```

Query 3:
This query retrieves the average product cost for each brand over the years with the help of the GROUP BY clause and ROLLUP.

```
SELECT
    b.brand_name,
    YEAR(o.date_time) AS year,
    AVG(p.product_cost) AS avg_product_cost
FROM
    brand b
    JOIN product p ON b.brand_id = p.brand_id
    JOIN order_table o ON p.product_id = o.product_id
GROUP BY b.brand_name, YEAR(o.date_time) with ROLLUP
ORDER BY
    b.brand_name, YEAR(o.date_time);
```

Query 4:

This query is selecting sales and profit data for the year 2022 broken down by month. It is joining the order_table, cart, product, and inventory tables to calculate the total sales and total profit for each product in each order.
It is filtering the results to only include orders placed between January 1, 2022, and December 31, 2022. It is grouping the results by year and month, and including a ROLLUP clause to add subtotals for each year and a grand total at the end.

```
SELECT
  YEAR(o.date_time) AS year,
  MONTHNAME(o.date_time) AS month,
  SUM(p.product_cost * oi.quantity) AS total_sales,
  SUM((p.product_cost - p.product_cost * 0.2) * oi.quantity)
AS total_profit
FROM
  order_table o
  JOIN cart ct ON o.unique_id = ct.unique_id
  JOIN product p ON ct.product_id = p.product_id
  JOIN inventory oi ON ct.product_id = oi.product_id
WHERE
  o.date_time BETWEEN '2022-01-01' AND '2022-12-31'
GROUP BY
  YEAR(o.date_time),
  MONTHNAME(o.date_time) WITH ROLLUP
ORDER BY
  YEAR(o.date_time),
  MONTHNAME(o.date_time)
LIMIT 0, 1000;
```

- Triggers

Trigger 1: To check before deleting a customer record whether they have any order or not

```
delimiter //
CREATE TRIGGER prevent_customer_deletion
BEFORE DELETE ON customer
FOR EACH ROW
BEGIN
  IF (SELECT COUNT(*) FROM order_table WHERE Unique_id = OLD.customer_id)
> 0 THEN
  -- unique_id in order_table references Customer_id in Customer table
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Cannot delete customer because orders exist for this
customer.';
  END IF;
END;//
-- Query to check the working of this trigger

DELETE FROM customer WHERE customer_id = 98;
```

Trigger 2: To check if each entry of Customer has a Unique Phone number only

```
delimiter //
CREATE TRIGGER check_phone_number_unique
BEFORE INSERT ON customer
FOR EACH ROW
BEGIN
    IF (SELECT COUNT(*) FROM customer WHERE PhoneNumber =
NEW.PhoneNumber) > 0 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Phone number must be
unique.';
    END IF;
END;//
```

- A Query example to check whether the trigger is working or not

```
-- Insert a new customer with a unique phone number

INSERT INTO customer (customer_Id, Address, Name, EmailID, Password,
PhoneNumber)
VALUES (104, '123 Main St', 'John Doe', 'johndoe@example.com', 'password123',
'555-1234');

-- This should not cause an error

-- Insert a new customer with the same phone number as the first one

INSERT INTO customer (customer_Id, Address, Name, EmailID, Password,
PhoneNumber)
VALUES (105, '456 Maple St', 'Jane Smith', 'janesmith@example.com',
'password456', '555-1234');

-- This should raise an error due to the duplicate phone number
```

Transactions

T1:
START TRANSACTION
SELECT quantity FROM inventory WHERE product_id = 1 FOR UPDATE (Read A)
UPDATE inventory SET quantity = quantity + 1 WHERE product_id = 1 (Write A)
UPDATE customer SET name = ‘Rajesh’ WHERE customer_Id= 1; (Write B)
COMMIT;

T2:
START TRANSACTION
SELECT quantity FROM inventory WHERE product_id = 1 FOR UPDATE (Read A)
UPDATE inventory SET quantity = quantity - 1 WHERE product_id = 1 (Write A)
UPDATE customer SET name = ‘harkesh’ WHERE customer_Id= 1; (Write B)
COMMIT;

SERIAL Schedule

T1	T2
SELECT quantity FROM inventory WHERE product_id = 1 FOR UPDATE	
UPDATE inventory SET quantity = quantity + 1 WHERE product_id = 1	
UPDATE customer SET name = ‘Rajesh’ WHERE customer_Id= 1;	
	SELECT quantity FROM inventory WHERE product_id = 1 FOR UPDATE
	UPDATE inventory SET quantity = quantity - 1 WHERE product_id = 1
	UPDATE customer SET name = ‘Harkesh’ WHERE customer_Id= 1;

Conflict-Serializable Schedule:

T1	T2
SELECT quantity FROM inventory WHERE product_id = 1 FOR UPDATE	
UPDATE inventory SET quantity = quantity + 1 WHERE product_id = 1	
	SELECT quantity FROM inventory WHERE product_id = 1 FOR UPDATE
	UPDATE inventory SET quantity = quantity - 1 WHERE product_id = 1
UPDATE customer SET name = 'Rajesh' WHERE customer_Id= 1;	
	UPDATE customer SET name = 'Harkesh' WHERE customer_Id= 1;

In general, a schedule is a conflict serializable if it can be transformed into an equivalent schedule where all transactions are executed in serial order without creating any conflicts between them.

If we go by the precedence, then this will be an equivalent schedule as there will be no conflict, and we can swap instructions to make a serial schedule as we are first reading quantity and then adding it, then again reading quantity and sequentially decreasing from it.

Also, we are using shared lock FOR UPDATE in the transactions to avoid any inconsistencies and finally commit the changes.

Not Conflict Serializable Schedule:

T1	T2
SELECT quantity FROM inventory WHERE product_id = 1 FOR UPDATE	
UPDATE inventory SET quantity = quantity + 1 WHERE product_id = 1	
	SELECT quantity FROM inventory WHERE product_id = 1 FOR UPDATE
	UPDATE inventory SET quantity = quantity - 1 WHERE product_id = 1
	UPDATE customer SET name = ‘Harkesh’ WHERE customer_Id= 1;
UPDATE customer SET name = ‘Rajesh’ WHERE customer_Id= 1;	

If we follow the above precedence schedule, then this is Not Conflict serializable schedule.

The reason for the conflict is that we are unable to swap instructions in the above schedule to obtain either a serial schedule <T1, T2> or <T2, T1>, which is causing conflicts and inconsistency in the database.