DBMS Project- Group 155

Lakshya Agrawal
Kartikey Dhaka

ONLINE RETAIL STORE

SCOPE

The project implements a database management system based on an online retail store. In the application, sellers could add the things they want to sell via the application, and different customers would be able to buy the things they are interested in. This project tries to implement relationships between different entities and stakeholders(listed below) to ensure an efficient flow between them. The application would allow customers to browse through a wide variety of products, add the products they are interested in into the cart and then proceed to checkout. Customer signup details, Seller signup details, product information, order details and payment details are some of the different types of data which will be stored in our database.

STAKEHOLDERS

1) Customers(People who will buy the products)
2) Sellers(retailers who are selling products)
3) Delivery agents

Workflow

The admin manages the customer, seller, delivery agent, product and product category entities.
The customer can add products to the cart, update the product's quantity, and delete any product not required from the cart.
Each cart contains various cart_items(productID and quantity)
Once the customer is satisfied that he/she has added all the required products to the cart, he places an order.
To place an order, he has to pay for the order. The details of the payment is stored in the entity payment.
After payment for the order, a delivery agent is assigned corresponding to each order.

Entities and Attributes

1)customer

customer(customerID,first_name,last_name,email,phoneNumber)

Attribute	Data type	Constraints/Extra
customerID	INT	NOT NULL, AUTO_INCREMENT, Unique, PRIMARY KEY
first_name	varchar(20)	NOT NULL
last_name	varchar(20)	
email	varchar(30)	NOT NULL, Unique
phoneNumber	varchar(30)	NOT NULL, Unique

2)seller

seller(sellerID, first_name,last_name,phoneNumber)

Attribute	Data type	Constraints/Extra
sellerID	INT	NOT NULL, AUTO_INCREMENT, Unique, PRIMARY KEY
first_name	varchar(40)	NOT NULL
last_name	varchar(40)	
phoneNumber	varchar(30)	NOT NULL Unique

3)product

product(productID,productName,productPrice,ProductStock)

Attribute	Data type	Constraints/Extra
productID	INT	NOT NULL, AUTO_INCREMENT, Unique, PRIMARY KEY
productName	varchar(40)	NOT NULL
productPrice	varchar(40)	NOT NULL CHECK(productPrice>0)
productStock	INT	NOT NULL CHECK(productStock>=0)

4)product_category

product_category(categoryID,categoryName)

Attribute	Data type	Constraints/Extra
categoryID	INT	NOT NULL, AUTO_INCREMENT, Unique, PRIMARY KEY
categoryName	varchar(40)	NOT NULL Unique

5)cart

cart(customerID,discount,total_cost)

Attribute	Data type	Constraints/Extra
customerID	INT	NOT NULL Foreign key references customer, PRIMARY KEY
discount	INT	CHECK(discount>=0)
total_cost	INT	NOT NULL CHECK(productPrice>0) Default 0

6)order_detail

order_detail(orderID,totalCost,orderStatus,customerID,deliveryID)

Attribute	Data type	Constraints/Extra
orderID	INT	NOT NULL, AUTO_INCREMENT, Unique, PRIMARY KEY
totalCost	INT	NOT NULL CHECK(totalCost>0)
orderStatus	varchar(30)	
customerID	INT	NOT NULL Foreign key references customer
deliveryID	INT	NOT NULL Foreign key references deliveryagent

7)deliveryagent

deliveryagent(deliveryID,first_name,last_name,phoneNumber)

Attribute	Data type	Constraints/Extra
deliveryID	INT	NOT NULL, AUTO_INCREMENT,

		Unique, PRIMARY KEY
first_name	varchar(40)	NOT NULL
last_name	varchar(40)	
phoneNumber	varchar(30)	NOT NULL Unique

8)payment

payment(transactionID,modePayment,dateTransaction,orderID)

Attribute	Data type	Constraints/Extra
transactionID	INT	NOT NULL, AUTO_INCREMENT, Unique, PRIMARY KEY
modePayment	varchar(30)	NOT NULL
dateTransaction	date	NOT NULL
orderID	INT	NOT NULL Foreign key references order_detail

9)cart_item

cart_item(customerID,productID,quantity,cost)

Attribute	Data type	Constraints/Extra
customerID	INT	NOT NULL, PRIMARY KEY Foreign key references customer
productID	INT	NOT NULL, PRIMARY KEY Foreign key references product
quantity	INT	NOT NULL,

		PRIMARY KEY CHECK(quantity>=0)
cost	INT	DEFAULT 0 CHECK(cost>=0)

10)sells (Many to Many relationship between Seller and Product)

sells(sellerID,productID)

Attribute	Data type	Constraints/Extra
sellerID	INT	NOT NULL, PRIMARY KEY Foreign key references seller
productID	INT	NOT NULL PRIMARY KEY Foreign key references product

11)belongs(Many to Many relationship between product and product Category)

belongs(productID,categoryID)

Attribute	Data type	Constraints/Extra
categoryID	INT	NOT NULL, PRIMARY KEY Foreign key references category
productID	INT	NOT NULL PRIMARY KEY Foreign key references product

WEAK ENTITY

Cart is a weak entity in this database. Cart is identified by another entity ,here customer. A cart not identified by a customer doesnt make sense.
The attributes of cart as well as the primary key of customer uniquely define this weak entity.

INDEXES (attributes)

Indexes are created for faster selection of data.
Some of the indexes made upto Deadline-3 are:
(The primary key of all the entity are already indexed attributes)
1)

```
CREATE INDEX first_name_idx
ON customer(first_name);
```

This index helps in faster searching of customer by its first_name

2)

```
CREATE INDEX first_name_idx
ON deliveryagent(first_name);
```

This index helps in faster searching of deliveryagent by its first_name

3)

```
CREATE INDEX first_name_idx
ON seller(first_name);
```

This index helps in faster searching of seller by its first_name

4)

```
CREATE INDEX product_name_idx
ON product(productName);
SHOW indexes from product;  //Displays all the indexes for the entity product
```

This index helps in faster searching of product by its productName

Cardinality

The cardinality is mentioned in the ER diagram itself. Some important ones are:
1)one to one relation between order and payment

2)one to many between delivery agent and order
(1 agent can have multiple orders to deliver)

3)A product can belong to multiple product categories.

4)A product category can consist of multiple products

5)A cart can contain multiple cart_items

Total/Partial participation

1)Every customer must have a cart (total participation)

2)Each order made must correspond to a customer

3)Each order must have a delivery agent assigned

Data generation and creation of database in mysql:

create database online_store // (creates a new database named online_store)

use online_store   //(To start using the created database)

```
CREATE TABLE seller
(
   sellerID INT NOT NULL unique auto_increment,
   first_name varchar(40) not null,
   last_name varchar(40),
   phoneNumber varchar(30) not null,
   primary key(sellerID)
) //Creating a table for seller (taken as an example to explain)
```

The online bulk data generator used is Mockeroo. Around 3100 data entries are fed in the database.
Once the csv file has been generated for each corresponding table, it is then imported into the mysql online_store database.entity table.

Query to alter the auto_increment number

```
alter table order_detail auto_increment=302;
```

QUERIES

Query to Update Cost (in cart_item)

```
UPDATE cart_item
SET cost = (SELECT productPrice FROM product WHERE product.productID =
cart_item.productID) * quantity;
```

Explanation: We retrieve the column productPrice with the help of productID given as an attribute of cart_item. Then we simply perform cost= productPrice Quantity of that product*

Query to Update total cost (in cart)

```
UPDATE cart
SET Total_Cost = (SELECT SUM(cost) FROM cart_item WHERE cart_item.customerID =
cart.customerID) * (1 - (discount / 100));
```

*Explanation: We sum the costs of all the cart_items belonging to a particular customer. Then we simply perform total_cost= (sum of costs) - (sum of costs)*discount/100*

Query to get the details of all the customers in ranking of orders placed(Customer corresponding to highest orders placed comes first)

```
SELECT customer.customerID,customer.first_name, COUNT(order_detail.orderID) AS
total_orders
FROM customer
LEFT JOIN order_detail ON customer.customerID = order_detail.customerID
GROUP BY customer.customerID
ORDER BY total_orders DESC
```

(Left join so that all the cutomerID which cannot be matched in both table→Customer which didnt order also appears)

Search for names in a table based on the first letter(here ‘Mushroom’)

```
SELECT * FROM product WHERE productName LIKE 'Mushroom%';
```

Query to get the details of all the customers who have spent more than a specific amount of money.(For giving extra discount to the customer)

```
SELECT customer.customerID, customer.first_name, SUM(order_detail.totalCost) as
Total_Spent
FROM customer
JOIN order_detail ON customer.customerID = order_detail.customerID
GROUP BY customer.customerID
HAVING SUM(order_detail.totalCost) > 5000;
```

Query to get the specific details of the orders that were placed within a specific date range(here year 2022)

```
SELECT order_detail.orderID, payment.DateTransaction, order_detail.totalCost,
customer.first_name
FROM order_detail
JOIN payment ON order_detail.orderID = payment.orderID
JOIN customer ON order_detail.customerID = customer.customerID
WHERE payment.DateTransaction BETWEEN '2022-01-01' AND '2022-12-31';
```

Query to get the details of the customers who have not placed any orders yet.

```
SELECT customer.customerID, customer.first_name
FROM customer
LEFT JOIN order_detail ON customer.customerID = order_detail.customerID
WHERE order_detail.orderID IS NULL;
```

Query for Admin to a new product

```
INSERT INTO product
VALUES (‘productID’,’productName’,’productPrice’,’productStock’);
```

```
INSERT INTO sells
VALUES('sellerID','productID');

INSERT INTO belongs
VALUES('productID','categoryID')
```

Query to get to know the what products are in Cart(along with other information)/Summary

(Using subQueries/nested Select statement)

```
SELECT
  cart_item.productID,
  (SELECT productName FROM product WHERE productID = cart_item.productID) AS
productName, cart_item.quantity,
  (SELECT productPrice FROM product WHERE productID = cart_item.productID) AS
productPrice,cart_item.cost

FROM cart, cart_item

WHERE
  cart.customerID = cart_item.customerID AND cart.customerID = 10;
```

Explanation:Provides the summary of the cart,i,e, it tells the productId as well as the productName and its cost column entries to the user.

Reduce the quantity of a product put in the cart

```
UPDATE cart_item
SET quantity=quantity-1
WHERE customerID=2 AND productID=236 AND quantity>1;

DELETE FROM cart_item
WHERE customerID=2 AND productID=236 AND quantity=1;
```

```
UPDATE cart_item
SET cost = (SELECT productPrice FROM product WHERE product.productID =
cart_item.productID) * quantity
WHERE customerID=2;

UPDATE cart
SET Total_Cost = (SELECT SUM(cost) FROM cart_item WHERE cart_item.customerID =
cart.customerID) * (1 - (discount / 100))
WHERE customerID=2;
```

Explanation: If the product's quantity is already 1, then it is removed from the cart_item table, else the quantity of the product is reduced by 1 at a time. Now since the quantity of a product is reduced, we recalculate the cart total_cost as well as the cart_item cost(in case quantity >1)

Query to buy the cart:

```
INSERT INTO order_detail (customerID,deliveryID,orderStatus ,totalCost)
VALUES (3, 4, 'dispatching',
       (SELECT total_Cost
        FROM cart
        WHERE cart.customerID = 3)
       );

INSERT INTO payment (modePayment,dateTransaction,orderID)
VALUES ('UPI', CURDATE(), last_insert_id());

UPDATE product
JOIN cart_item ON product.productID = cart_item.productID
SET product.productStock = product.productStock - cart_item.quantity
WHERE cart_item.customerID = 3;

DELETE FROM cart_item WHERE customerID = 3;

UPDATE cart
SET Total_Cost = (SELECT SUM(cost) FROM cart_item WHERE cart_item.customerID =
cart.customerID) * (1 - (discount / 100));
```

Explanation:A customer can place an order from the cart he/she has. Initially a order is placed followed with payment . Then some products have been bought by the customer, the stock of that product is reduced accordingly. Then the cart_items are deleted since they have been bought. Finally, the cart total cost is updated.

Query to search for names in a table based on the first letter(here ‘M’)(For filtering purpose)

```
SELECT * FROM customer WHERE first_name LIKE 'M%';
```

Query to get the details of all the products that are currently in stock, are higher than a particular quantity and their available quantities.

```
SELECT product.productID, product.productName, product.productStock
FROM product
WHERE product.productStock > 130;
```

TRIGGER

TRIGGER 1

→Updates the cost column of the table when the productID and productCost is Inserted

```
DELIMITER $$
CREATE TRIGGER update_cart_item_price
BEFORE INSERT ON cart_item
FOR EACH ROW
BEGIN
IF NEW.quantity < 0 THEN
SET NEW.quantity = 0;
END IF;
SET NEW.cost = NEW.quantity * (SELECT productPrice FROM Product WHERE productID =
NEW.productID);
END$$
DELIMITER ;
```

TRIGGER 2

→Updates the Total Cost column of the table when the corresponding cart_item is inserted

```
DELIMITER $$
CREATE TRIGGER update_cart
AFTER INSERT ON cart_item
FOR EACH ROW
BEGIN
IF NOT EXISTS (SELECT * FROM cart WHERE customerID = NEW.customerID) THEN
INSERT INTO cart (customerID, total_Cost, discount) VALUES (NEW.customerID, NEW.cost,
0);
ELSE
-- If a cart already exists, update the totalCost
UPDATE cart
SET total_Cost = (SELECT SUM(cost) FROM cart_item WHERE cart_item.customerID =
NEW.customerID)*(1-(discount/100))
WHERE customerID=NEW.customerID;
END IF;
END$$
DELIMITER ;
```

TRIGGER 3

→ Extra 10% discount if the cart price is above 10k

```
DELIMITER $$
CREATE TRIGGER increase_discount
BEFORE UPDATE ON cart
FOR EACH ROW
BEGIN
IF NEW.total_Cost +(NEW.total_Cost*(OLD.discount/100)) > 10000 THEN
SET NEW.discount = OLD.discount+10;
SET NEW.total_Cost=
(NEW.total_Cost+(NEW.total_Cost*(OLD.discount/100)))*(1-(NEW.discount/100));
END IF;
END$$
DELIMITER ;
```

OLAP:

1)Query to get the total number of orders and revenue generated by each delivery agent over the years(Slicing OLAP Query)

```
SELECT deliveryagent.deliveryID,deliveryagent.first_name, COUNT(DISTINCT
order_detail.orderID) as total_orders, SUM(order_detail.totalCost) as total_revenue
FROM deliveryagent
JOIN order_detail ON deliveryagent.deliveryID = order_detail.deliveryID
JOIN payment ON order_detail.orderID = payment.orderID
WHERE YEAR(payment.dateTransaction) = YEAR(payment.dateTransaction)
GROUP BY deliveryagent.deliveryID
ORDER BY total_revenue DESC
```

//---

2)Query to get the details of all the customers in ranking of orders placed(Customer corresponding to highest orders placed comes first)
(Drill down OLAP query, as it drills down on the customer dimension by ranking the customers based on the number of orders they have place)(identify which customers are the most valuable or active in terms of placing orders.)

```
SELECT customer.customerID,customer.first_name, COUNT(order_detail.orderID) AS
total_orders
FROM customer
LEFT JOIN order_detail ON customer.customerID = order_detail.customerID
GROUP BY customer.customerID
ORDER BY total_orders DESC
```

//--
3)Query to get how many products are in each product category (Slicing)
```
SELECT pc.categoryName, COUNT(belongs.productID) AS total_products
FROM product_category pc
LEFT JOIN belongs ON pc.categoryID = belongs.categoryID
GROUP BY pc.categoryName;
```

//--
4)Query to get the total order value,number of instances of usage, and average order value for each payment method.
```
SELECT modePayment as Mode_Of_Payment,SUM(order_detail.totalCost) as
Total_Order_Value,COUNT(order_detail.orderID) as NumberOf_Times_Used,
AVG(order_detail.totalCost) AS Average_Order_Value
FROM payment
JOIN order_detail ON order_detail.orderID=payment.orderID
GROUP BY modePayment WITH ROLLUP;
```

//--
5) Query to get the revenue Generated by Year and Month
```
SELECT YEAR(payment.dateTransaction) as Year, MONTH(payment.dateTransaction)
as Month, SUM(order_detail.totalCost) as Revenue_Generated
FROM payment
INNER JOIN order_detail ON payment.orderID = order_detail.orderID
GROUP BY Year,Month WITH ROLLUP
```
//NULL NULL represents the sum of revenue generated over the 2 years while 2022 NULL means the revenue generated in 2022

EMBEDDED SQL QUERIES:

1) Query to get the specific details of the orders that were placed within a specific date range (Between '2022-01-01' AND '2022-03-01)
```
mycursor = db.cursor()
mycursor.execute('''SELECT order_detail.orderID, payment.DateTransaction,
order_detail.totalCost,
customer.first_name
FROM order_detail
JOIN payment ON order_detail.orderID = payment.orderID
JOIN customer ON order_detail.customerID = customer.customerID
WHERE payment.DateTransaction BETWEEN '2022-01-01' AND '2022-03-01';''')
results = mycursor.fetchall()
for x in results:
x = str(x)
x = x.replace("datetime.date", "")
```

```
x = x.replace("'", "")
print(x[1:len(x) - 1])
```

2) Query to get the details of all the customers who have spent more than a specific amount of money.(For giving extra discount to the customer)

```
mycursor = db.cursor()
mycursor.execute('''SELECT customer.customerID, customer.first_name,
SUM(order_detail.totalCost) as
Total_Spent
FROM customer
JOIN order_detail ON customer.customerID = order_detail.customerID
GROUP BY customer.customerID
HAVING SUM(order_detail.totalCost) > 10000;''')
results = mycursor.fetchall()
for x in results:
x = str(x)
x = x.replace("Decimal", "")
x = x.replace("(", "")
x = x.replace(")", "")
x = x.replace("'", "")
print(x)
```

3) Query to get to know the what products are in Cart(Summary).
For customerID=10

```
mycursor = db.cursor()
mycursor.execute('''SELECT
cart_item.productID,
(SELECT productName FROM product WHERE productID = cart_item.productID) AS
productName, cart_item.quantity,
(SELECT productPrice FROM product WHERE productID = cart_item.productID) AS
productPrice,cart_item.cost
FROM cart, cart_item
WHERE
cart.customerID = cart_item.customerID AND cart.customerID = 10;''')
results = mycursor.fetchall()
for x in results:
x = str(x)
x = x.replace("Decimal", "")
x = x.replace("(", "")
x = x.replace(")", "")
x = x.replace("'", "")
print(x)
```

4)Display the products available:

```
cursor = db.cursor()
cursor.execute('SELECT * FROM product')
results = cursor.fetchall()
```

Frontend made using CSS+HTML and backend using Flask ,MySQL(using mysql.connector())

User Guide

Welcome to our online retail store! This user guide will provide you with all the information you need to navigate our website and make the most of your shopping experience.

Login Page:

Customer:

To access the online retail store, you will need to create an account or log in if you already have one. To create a new account, click on the "Register" button. You will need to provide your personal information to create your account.
If you already have an account, click on the "Login" button and enter your name and phone number to login in.

Seller:

To become a seller in our online retail store, you will need to create an account or log in if you already have one. To create a new account, click on the "Register" button. You will need to provide your personal information to create your account.
Enter your login credentials (username and phone number) and you will enter your seller page.

Main Page of Customer:

View All Products:

After logging in, you will be directed to the home page. Here, you can view all the products that we offer.

View Your Cart:

To view your shopping cart, click on the “view cart” button. Here, you can view all the items that you have added to your cart, as well as the total cost of your order.

Add Products:

To add products to your cart, simply click on the "Add to Cart". You can add products by entering their product ID and the quantity.

Buy Cart:

When you are ready to checkout, click on the “Buy Cart” button. You will be directed to a page where you can review your order and enter your shipping and billing information and your order will be placed.

Main Page of Seller:

View Products You Are Selling:

Once you have successfully logged in, you will be redirected to your seller dashboard. Click on the " View Products You Are Selling" button, here you can view all the products you are currently selling on the online retail store.

Add Products:

To add a new product, click on the "Add Product" button and fill the details of the product. Finally, click on the "Add to sell the Product" button to add the new product to your store.

Here, we have differentiated the login page for customer and seller. Customer majorly add products to his/her cart and eventually buy them, whereas, seller can add product which he/she want to sell and even view all products he/she is selling. This makes it easier for both customer and the sellers to buy and sell products respectively.

Schedules and Transactions

Updating product stock and placing an order followed by payment:
Transaction1 User A is placing an order for a product followed by payment
Transaction2User B placing an order for that same product followed by payment
Conflicting case User A and B tries to buy the same product at the same time. While B is placing an order for the product, A updates the product stock. If the stock updated by A is done before B places the order will lead to conflicting transactions. This is because for B, the product may appear in stock, whereas in reality the stock has been reduced (maybe finished). Thus, the order will not be processed for B because of A.

Example:

```
--Begin transaction 2: User B is placing an order for a product
BEGIN TRANSACTION;
SELECT productStock FROM product WHERE productID = 1; (quantity selected = 1)
IF product stock >= 1 THEN
  UPDATE product SET productStock = productStock - 1 WHERE productID=1;
  INSERT INTO order_detail (totalCost, orderStatus, customerID) VALUES ( , 'pending', 1);
INSERT INTO payment VALUES(    );
  COMMIT;
ELSE:
  -- Rollback the transaction if there is not enough stock
  ROLLBACK;
END IF;
-- Begin transaction 1: User A is updating the stock of the product
BEGIN TRANSACTION;
UPDATE product SET productStock = 0 WHERE product ID = 1; (Wants to buy everything in stock)
INSERT INTO order_detail (totalCost, orderStatus, customerID) VALUES ( , 'pending', 2);
INSERT INTO payment VALUES(    );
```

COMMIT;

In this example, User B checks the current stock of the product and attempts to place an order. However, if the stock of the product is zero, the transaction is rolled back. Meanwhile, User A is simultaneously updating the stock of the same product to zero. As a result, User B's transaction will be rolled back since there is not enough stock to fulfill the order. Payment is assumed to be done after some seconds the order is placed.

Solution:
This situation is known as a race condition, where multiple processes compete to access a shared resource at the same time, leading to inconsistent or unpredictable behavior. To avoid this, you can use a technique called concurrency control. One common approach is to use locking mechanisms, such as semaphores or mutexes, to ensure that only one process can access the shared resource at a time. In this case, you could use a lock to prevent B from updating the stock while A is placing an order, or vice versa.

Non-conflict serializable schedule: (WR conflict) Schedule 1

T1	T2
Read product stock	
	Read product stockReads a non-zero value and decides to order
Write product stock	
Place Order (Commit)	
	## Read product stock
	Write product stock
	Place Order
OrderStatus read	
Payment is done	
	OrderStatus read
	Payment is done

This schedule is non-conflict serializable schedule since no swapping can be done to convert the schedule into a serial schedule since a consecutive conflicting instruction is occurring (Write product stockRead product Stock)WR conflictDirty Read

Solution:
One can use a concurrency control mechanism such as locking. Here one can use a read lock on the data item. This allows multiple transactions to read from the data item at the same time, but only one transaction can write to the data item while the lock is held. In this scenario, the transaction that writes to the data item will acquire a write lock and prevent any other transactions from reading or writing to the data item until the lock is released.

Conflict Serializable schedule: Schedule 2

T1	T2

Read product stock
Write product stock
Place order

Read product stock (Correct product stock read)
Write product stock (May not be able to place order if stock=0)
Place order

OrderStatus read (Pending)
Payment is done

OrderStatus read (Pending)
Payment is done

This schedule is conflict serializable since it can be converted into a serial schedule by swapping all the consecutive non-conflicting pair of instructions between T1 and T2 into:

T1 T2

Read product stock
Write product stock
Place order
OrderStatus read (Pending)
Payment is done

Read product stock (Correct product stock read)
Write product stock (May not be able to place order if stock=0)
Place order
OrderStatus read (Pending)
Payment is done

R(A)reading product stock
R(B) reading the order status
W(A)writing product stockplacing order
W(B)Payment is done

Part 2:
Conflicting transaction occurs when a customer puts something to buy in the cart and the admin/seller removes it from the product list is that the customer may not be able to complete their purchase if the item is no longer available.

```
-- When a customer adds a product to their cart, check if it's still available
INSERT INTO cart_item (customerID, productID, quantity)
VALUES (10, 1, 1);
SELECT productID FROM product WHERE id = 1;
IF record fetched THEN:
```

```
    --Continue
  ELSE:
  -- If the product is no longer available, remove it from the cart
  DELETE FROM cart_item WHERE customer id = 10 AND product id = 1;
END IF;
```

Solution:
Use real-time updates: You can use real-time updates to check and update the product availability in real-time. This can be achieved by implementing a mechanism that allows the customer's cart to refresh automatically when there is a change in the product availability. This ensures that the customer is aware of the availability of the product before they attempt to complete their purchase.

Non-conflicting Transactions:
1)Add a New Seller and Assign Products to Them

```
START TRANSACTION;

   -- Step 2: Insert the new seller details
   INSERT INTO seller (first_name, last_name, phoneNumber) VALUES ('Lakshya', 'Agrawal',
'123-456-7890');

   -- Step 3: Get the seller ID
   SELECT LAST_INSERT_ID() INTO @sellerID;

   -- Step 4: Assign products to the new seller
   INSERT INTO sells (sellerID, productID) VALUES (@sellerID,1), (@sellerID,2), (@sellerID,3);

-- Step 5: Commit the transaction
COMMIT;
```

2)Buying cart

```
START TRANSACTION;
TRY
        INSERT INTO order_detail (customerID,deliveryID,orderStatus ,totalCost)
        VALUES (13, 5, 'dispatching',
                        (SELECT total_Cost
                         FROM cart
                         WHERE cart.customerID =13 )
```

```
                );

        INSERT INTO payment (modePayment,dateTransaction,orderID)
        VALUES ('UPI', CURDATE(), last_insert_id());

        UPDATE product
        JOIN cart_item ON product.productID = cart_item.productID
        SET product.productStock = product.productStock - cart_item.quantity
        WHERE cart_item.customerID = 13;

        DELETE FROM cart_item WHERE customerID = 13;

        UPDATE cart
        SET Total_Cost =0;
COMMIT;

CATCH
        ROLLBACK;
END TRY;
```

3)Adding a new product in the database

```
START TRANSACTION;

   INSERT INTO product (productName, productPrice, productStock)
   VALUES ('Watermelon',60,35);

   SET @productID = LAST_INSERT_ID();

   INSERT INTO belongs (productID, categoryID)
   VALUES (@productID, 14);

COMMIT;
```