Non- Conflicting & Conflicting Transactions

NON Conflicting:
For Database:-
(from Mysql)

```
START TRANSACTION;
INSERT INTO Customers (customer_name, customer_email, customer_Phone_number,
customer_password, customer_DOB, customer_age)
VALUES ('Hermione Granger', 'hermione@example.com', '+1234567890', 'password123',
'1980-09-19', 43);
COMMIT;
SELECT * FROM Customers;
START TRANSACTION;
INSERT INTO Products (product_name, product_price, category_id, product_quantity)
VALUES ('Magic Wand', 29.99, 1, 50);
COMMIT;
SELECT * FROM Products;

START TRANSACTION;
INSERT INTO Orders (customer_id, order_date, order_status)
VALUES (3, '2024-04-20', 'Pending');
COMMIT;
SELECT * FROM Orders;
```

Transactions that we are using:-

we considered operations related to the checkout process, product addition to the cart, and applying discounts as the basis for constructing transactions.

Transaction 1 (T1) - Checkout Process:

- Read the quantity and product ID of each item in the cart.
- Update the stock availability of each product based on the quantity purchased.

2. Transaction 2 (T2) - Add Product to Cart:

- Retrieve the customer's cart ID.
- Insert a new entry into the cart item table for the desired product with the specified quantity.

3. Transaction 3 (T3) - Apply Discount:
    - Retrieve the percentage discount from the entered coupon code.
    - Update the total amount of the cart after applying the discount.

```
-- Transaction 1: Checkout Process

START TRANSACTION;
FOR EACH item IN customer_cart:

   SELECT item.product_id INTO productID, item.quantity INTO quantity
   FROM cart_item
   WHERE cart_id = customer.cart_id;
   UPDATE Product
   SET stock = stock - quantity
   WHERE product_id = productID;
COMMIT;

-- Transaction 2: Add Product to Cart
START TRANSACTION;
SELECT cart_id INTO current_cart_id
FROM customer
WHERE customer_id = current_customer_id;
INSERT INTO cart_item (cart_id, product_id, quantity)
VALUES (current_cart_id, desired_product_id, desired_quantity);
COMMIT;


-- Transaction 3: Apply Discount
START TRANSACTION;
SELECT percentage_discount INTO discount
FROM coupon
WHERE coupon_code = entered_coupon_code;
UPDATE cart
SET cart_total_amount = cart_total_amount * (100 - discount) / 100
WHERE cart_id = customer.cart_id;
```

COMMIT;

Step	Serial Schedule	Conflict Serializable Schedule
1	T1 Read	T1 Read
2	T1 Write	T1 Write
3		T2 Read
4	T2 Read	T2 Write
5	T2 Write	T3 Read
6	T3 Read	T3 Write
7	T3 Write	

In the table:

- "T1 Read" denotes the read operation in Transaction 1.
- "T1 Write" denotes the write operation in Transaction 1.
- Similar notations are used for the other transactions and operations.
- Blank cells indicate no operation in that step.
- Conflicts occur when a write operation in one transaction conflicts with a read or write operation in another transaction.

This table demonstrates how the transactions are serialized in both the serial and conflict serializable schedules.

Potential Conflicts:-

>> T1 and T2: There could be a conflict if T2 reads the quantity from the product table before T1 writes the updated quantity to the product table. This could result in T2 reading an outdated quantity, leading to incorrect availability calculations.

>> T2 and T3: If T2 reads the quantity from the product table and updates the cart_item table before T3 applies any discounts based on the total amount, the discount applied by T3 might not reflect the accurate total amount considering the quantity changes made by T2.

Non Conflicting

we can order the transactions T1, T2, and T3 in a sequence that does not result in conflicts. Here's a possible non-conflict serializable schedule:

1. T1: { READ(quantity) from the cart_item, WRITE(quantity) to product, COMMIT}
2. T2: { READ(quantity) from product, WRITE(quantity) to cart_item, COMMIT }
3. T3: { READ(coupon_discount) from coupon, WRITE(cart_total_amt) to cart, COMMIT}

Step	Transaction	Action	Data Item	Operation
1	T1	READ(quantity) from cart_item	quantity	Read
2	T1	WRITE(quantity) to product	quantity	Write
3	T1	COMMIT	-	Commit
4	T2	READ(quantity) from product	quantity	Read
5	T2	WRITE(quantity) to cart_item	quantity	Write
6	T2	COMMIT	-	Commit
7	T3	READ(coupon_discount) from coupon	coupon_discount	Read
8	T3	WRITE(cart_total_amt) to cart	cart_total_amt	Write
9	T3	COMMIT	-	Commit

We Also tried to show transactions (conflicting) via python as well code :

Considering a scenario where two users are trying to purchase items simultaneously, resulting in a race condition.so we tried to represent the payment method .

(code added):-
In this if customer tries to purchase the item which was available when he added it to cart but in the meantime if another customer orders the same item then (if committed before other customers(payment made)) the item will go out of stock and the person who had that item in their cart won't be able to purchase it/ make payment.
& considering a situation if something goes wrong while ordering (by the 2nd customer) The transaction will rollback & that item remains available for other
(1st)customer.

```python
def insert_payment(customer_id):
    try:
        conn = mysql.connector.connect(host='localhost', username='root', password='Ayush@mysql1', database='retailstore2')
        print("Enter payment method (Cash On Delivery, Net Banking, UPI, EMI):")
        payment_method = input()
        payment_date_and_time = time.strftime('%Y-%m-%d %H:%M:%S')
        payment_id = generate_payment_id(conn)

        # Begin transaction
        conn.autocommit(False)
        conn.set_isolation_level(0)  # Set isolation level

        try:
            with conn.cursor() as cursor:
                cursor.execute("START TRANSACTION")

                # Simulate a delay to create a race condition
                time.sleep(2)

                if not is_cart_empty(conn, customer_id):  # Check if the cart is not empty
                    query = "INSERT INTO payment (paymentID, paymentMethod, paymentDateTime) VALUES (%s, %s, %s)"
                    cursor.execute(query, (payment_id, payment_method, payment_date_and_time))
                    if cursor.rowcount > 0:
                        print("Payment details inserted successfully!")
                        insert_order(conn, customer_id, payment_id, payment_method)
                        print("Generated PaymentID:", payment_id)
                        cursor.execute("COMMIT")  # Commit the transaction
                    else:
                        print("Failed to insert payment details!")
                        cursor.execute("ROLLBACK")  # Rollback the transaction
                else:
                    print("Your cart is empty. Please add products before proceeding with the payment.")
                    cursor.execute("ROLLBACK")  # Rollback the transaction
        except Exception as e:
            print("Error:", e)
```

```python
def generate_payment_id():
    return str(uuid.uuid4())

def is_cart_empty(customer_id):
    try:
        conn = mysql.connector.connect(host='localhost', username='root', password='Ayush@mysql1', database='retailstore2')
        with conn.cursor() as cursor:
            query = "SELECT COUNT(*) FROM cart WHERE customer_id = %s"
            cursor.execute(query, (customer_id,))
            result = cursor.fetchone()
            if result and result[0] > 0:
                return False
            else:
                return True
    except Exception as e:
        print("Error:", e)
        return True  # Assuming cart is empty if an error occurs
    finally:
        conn.close()  # Close the database connection

def insert_order(customer_id, payment_id):
    try:
        conn = mysql.connector.connect(host='localhost', username='root', password='Ayush@mysql1', database='retailstore2')
        with conn.cursor() as cursor:
            query = "INSERT INTO orders (customer_id, payment_id) VALUES (%s, %s)"
            cursor.execute(query, (customer_id, payment_id))
            conn.commit()
            print("Order details inserted successfully!")
    except Exception as e:
        print("Error:", e)
        conn.rollback()  # Rollback the transaction if an exception occurs
    finally:
        conn.close()  # Close the database connection
```

```
elif login_choice == '2':
    email = input("Enter your Email: ")
    password = input("Enter your password: ")
    if customer_login(email, password):
        username = get_customer_name(email)
        customer_id = get_customer_id(email)
        while True:
            print("\nWelcome ", username, "!")
            print("1. Place Order")
            print("2. My Orders")
            print("3. Logout")
            print("4. Make Payment")  # Option to make payment
            customer_choice = input("Enter Your Choice: ")
            if customer_choice == '1':
                list_products()
                product_id = int(input("Enter product ID: "))
                quantity = int(input("Enter quantity: "))
                place_order(customer_id, product_id, quantity)
            elif customer_choice == '2':
                my_orders(customer_id)
            elif customer_choice == '3':
                print("Logging out...")
                print(".")
                print(".")
                print(".")
                print(" ")
```

```
        except Exception as e:
            print("Error:", e)
            conn.rollback()  # Rollback the transaction if an exception occurs
        finally:
            conn.autocommit(True)  # Enable auto-commit mode
            conn.close()  # Close the database connection
    except Exception as e:
        print("Error:", e)
```