Serializable Transactions

Transaction T1: Update medicine price and quantity
Transaction T2: Buying medicine, Update medicine quantity

Conflict-Serializable Schedule:

Transaction 1:
START TRANSACTION;
UPDATE Medicine SET Price = 10, -- Write(A)
Quantity = Quantity - 1 WHERE MedicineID = 123; -- Write(B)
COMMIT;

Transaction 2:
START TRANSACTION;
SELECT Quantity INTO quantity FROM Medicine WHERE MedicineID = 123; -- Read(B)
IF quantity <= 0 THEN
ROLLBACK;
ELSE
INSERT INTO Order_Items (Order_ID, Medicine_ID, Quantity, Price)
VALUES
(Order_ID, MedicineID, quantity, Price);
UPDATE Medicine SET Quantity = quantity - 1 WHERE MedicineID = 123;
-- Write(B)
COMMIT;
END IF;

Explanation:
Transaction 1 writes to data item 'A' (Price) and data item 'B' (Quantity) by decreasing the quantity of medicine with ID 123 and updating its price.

Meanwhile, Transaction 2 reads the quantity of medicine with ID 123 (data item 'B') and then writes to data item 'B' (decreasing the quantity of medicine with ID 123 by 1) and then inserts a new order in the Order_Items table (data item 'C').
Since both transactions access data item 'B' in a conflicting way (Transaction 1 writes to it and Transaction 2 reads from it and then writes to it), the schedule is not conflict serializable.

To solve the conflict in the transaction schedule, you can use locks to ensure that only one transaction can access a data item at a time. There are two types of locks that can be used: shared locks and exclusive locks.

Shared locks allow multiple transactions to read a data item simultaneously, but they prevent any transaction from writing to it until all shared locks have been released. Exclusive locks, on the other hand, allow only one transaction to read or write to a data item, and they prevent all other transactions from accessing it until the lock has been released.

In the given transaction schedule, Transaction 2 can use a shared lock to read the quantity of medicine with ID 123 from the Medicine table, while Transaction 1 can use an exclusive lock to update both Price and Quantity of the same medicine. This will prevent Transaction 2 from accessing and modifying the data item 'B' while Transaction 1 is updating it.

The updated transactions with locks would look like this:

Transaction 1:
START TRANSACTION;
UPDATE Medicine SET Price = 10, Quantity = Quantity - 1 WHERE MedicineID = 123 FOR UPDATE; -- Use exclusive lock
COMMIT;

Transaction 2:
START TRANSACTION;
SELECT Quantity INTO quantity FROM Medicine WHERE MedicineID = 123 FOR SHARE; -- Use shared lock
IF quantity <= 0 THEN
ROLLBACK;
ELSE
INSERT INTO Order_Items (Order_ID, Medicine_ID, Quantity, Price)
VALUES
(Order_ID, MedicineID, quantity, Price);
UPDATE Medicine SET Quantity = quantity - 1 WHERE MedicineID = 123;
-- Use exclusive lock
COMMIT;
END IF;

By using locks, the transactions can be executed in a way that ensures their serializability and prevents conflicts between them.

Conflict-Serializable Schedule:

Transaction 1:
BEGIN TRANSACTION;
SELECT *
FROM Employee
WHERE EmployeeID = 1
FOR UPDATE;
UPDATE Employee
SET Department_ID = 2
WHERE EmployeeID = 1;
COMMIT;

Transaction 2:
BEGIN TRANSACTION;
SELECT *
FROM Employee
WHERE EmployeeID = 1
FOR UPDATE;
UPDATE Employee
SET Department_ID = 3
WHERE EmployeeID = 1;
COMMIT;
In this example, before each update, a SELECT statement with the FOR UPDATE clause is used to lock the row that is being updated. This ensures that only one transaction can access and update the row at a time, preventing conflicts and maintaining data consistency. Once the update is complete, the transaction is committed.

Non-Conflict Serializable Schedule:

T1: START TRANSACTION
T2: START TRANSACTION
T1: SELECT Price FROM Medicine WHERE MedicineID = 1 FOR UPDATE
T2: SELECT Price FROM Medicine WHERE MedicineID = 2 FOR UPDATE
T1: UPDATE Medicine SET Price = Price + 50 WHERE MedicineID = 1
T1: COMMIT
T2: UPDATE Medicine SET Price = Price - 50 WHERE MedicineID = 2

T2: COMMIT

Explanation:

- T1 and T2 are two different transactions that start simultaneously.
- T1 acquires a shared lock on the row with MedicineID = 1 for update, while T2 acquires a shared lock on the row with MedicineID = 2 for update. This way, they are not conflicting with each other, as they are operating on different rows.
- T1 updates the Price value of Medicine with MedicineID = 1 by adding 50 to it and then commits the changes.
- T2 updates the Price value of Medicine with MedicineID = 2 by subtracting 50 from it and then commits the changes.

Since both transactions operate on different rows of the Medicine table, they can proceed independently without conflicting with each other. This way, the resulting schedule is non-conflict serializable.

Transaction schedule 1:

Transaction 1:
BEGIN;
INSERT INTO Medicine (Name, Date_of_manufacture, Date_of_expiry, Manufacturer_Name, Price, Description, Side_effects, Category_ID, Manufacturer_ID)
VALUES ('Paracetamol', '2022-01-01', '2023-01-01', 'ABC Pharma', 10, 'Pain relief medicine', 'Nausea, Headache', 1, 1);
COMMIT;

Transaction 2:
BEGIN;
UPDATE Customer SET Wallet_Balance = Wallet_Balance - 10 WHERE CustomerID = 1;

```
INSERT INTO Orders (Customer_ID, Total_Bill) VALUES (1, 10);
INSERT INTO Order_Items (Order_ID, Medicine_ID, Quantity, Price)
VALUES (1, 1, 1, 10);
COMMIT;
```

Transaction Schedule 1:
This transaction schedule involves two transactions.

The first transaction inserts a new row into the Medicine table, which represents a new medicine product that was manufactured by a particular manufacturer. The transaction specifies the name, date of manufacture, date of expiry, manufacturer name, price, description, side effects, category ID, and manufacturer ID. This transaction will commit once the insert operation is successful.

The second transaction involves updating the wallet balance of a specific customer and then inserting a new order with the total bill amount. The transaction then inserts an order item with the medicine ID, quantity, and price. This transaction will commit once all the operations are successful.

Transaction schedule 2:

Transaction 1:

```
BEGIN;
UPDATE Medicine SET Price = 12 WHERE MedicineID = 1;
COMMIT;
```

Transaction 2:

```
BEGIN;
INSERT INTO Customer (Name, Phone_Number, Email_Id, Password,
Address, Customer_Type, Wallet_Balance)
```

VALUES ('John Doe', '1234567890', 'johndoe@gmail.com', 'password', '123 Main St', 'Gold', 100);
COMMIT;

Transaction Schedule 2:
This transaction schedule also involves two transactions.

The first transaction updates the price of a particular medicine in the Medicine table. This transaction will commit once the update operation is successful.

The second transaction inserts a new customer into the Customer table with the specified details, including their name, phone number, email, password, address, customer type, and wallet balance. This transaction will commit once the insert operation is successful.

Both transaction schedules are non-conflicting as they are working on different tables and not dependent on each other. Therefore, these transactions can run concurrently without interfering with each other's operations.