ShipShop : An Online Retail Store

Group 144 : Kartik Gupta(2021056) | Harsh Kumar Pal(2021047)

Non-Conflicting Transactions:

```
BEGIN TRANSACTION;
SELECT Product.Name, Product.Price, Category.Name AS Category,
Product.Stock
FROM Product
INNER JOIN Category ON Product.Cat_id = Category.id
WHERE Product.Stock<=30;
COMMIT;
(#Displays Products with stock less than 30 for re-stocking purposes.)
```

```
BEGIN TRANSACTION;
SELECT Product.Name, Product.Price, Category.Name AS Category,
Product.Stock
FROM Product
INNER JOIN Category ON Product.Cat_id = Category.id
WHERE Category.Name = "Clothing";
COMMIT;
(#Displays all products in a specific category)
```

```
BEGIN TRANSACTION;
SELECT Shipping_Agent.FirstName, COUNT(Orders.id) AS
Orders_Assigned FROM Orders
LEFT JOIN Shipping_Agent ON Orders.S_id = Shipping_Agent.id
GROUP BY FirstName;
COMMIT;
(#Displays Shipping Agent Name, ID, and the number of orders that they have at the
moment.)
```

```
BEGIN TRANSACTION;
SELECT id, FirstName, Rating FROM Shipping_Agent
Where Rating>4.00
ORDER BY Rating DESC;
COMMIT;
(#Displays Shipping agents with 4-star rating or greater in descending order)
```

The above 4 transactions are non-conflicting because they are all read-only transactions, i.e., they only read data from the database and do not modify it. Since they do not modify any data, there is no possibility of conflicting with other transactions that may modify the same data. Therefore, these transactions can be executed concurrently without any issues.

Conflicting Transactions:

```
BEGIN TRANSACTION;

SELECT Stock FROM Product WHERE id = 6001; : READ(A)
UPDATE Product SET Stock = 30 WHERE id = 6001; : WRITE(A)
SELECT Stock FROM Product WHERE id = 6005; :READ(B)
UPDATE Product SET Stock = 20 WHERE id = 6005; :WRITE(B)

COMMIT;
```

```
BEGIN TRANSACTION;
SELECT Stock FROM Product WHERE id = 6001; :READ(A)
SELECT Stock FROM Product WHERE id = 6005; :READ(B)
COMMIT;
```

In this scenario, one transaction reads data that has been altered by another transaction but hasn't yet been committed, resulting in a **write-read conflict.** This is commonly known as an uncommitted read or a dirty read.

Here's an example of a **conflict serializable schedule.**

S1

T1	T2
READ(A)	
WRITE(A)	
	READ(A)
READ(B)	
WRITE(B)	
COMMIT	
	READ(B)
	COMMIT

S2 (Serial Schedule)

T1	T2
READ(A)	
WRITE(A)	
READ(B)	
WRITE(B)	
COMMIT	
	READ(A)
	READ(B)
	COMMIT

The above schedule is a concurrent schedule S1.

It is a conflict **equivalent to a serial schedule (S2)**.

So schedule **S1 is a conflict serializable schedule**, which means that it is **consistent.**

Non - conflicting serialisable schedule:

S1

T1	T2
READ(A)	
WRITE(A)	
	READ(A)
	READ(B)
	COMMIT
READ(B)	
WRITE(B)	
COMMIT	

S2 (Serial Schedule)

T1	T2
READ(A)	
WRITE(A)	
READ(B)	
WRITE(B)	

COMMIT	
	READ(A)
	READ(B)
	COMMIT

The above schedule S1 is non-conflict equivalent to any of the possible serial schedules.

Since S1 is **non-conflict equivalent to a serial schedule**, it is not conflict serialisable.
Schedule S1 is **inconsistent**.