TRANSACTIONS AND SCHEDULING

Conflicting Transactions:

T1:

1) **Reading a specific product R(A)**:
   Select * from Product where product_id=5;

2) **Updating the quantity of a specific product W(A)**:
   Update Product set Product.cquantity=10+Product.restock_trigger where Product.product_id=5;

3) **Reading the address of a specific customer R(B):**
   Select Address from Customer where customer_id=67;

4) **Updating the address of a specific customer W(B):**
   Update Customer set Customer.Address=”XYZ” where Customer.customer_id=67;

T2:

1) **Reading a specific product R(A)**:
   Select * from Product where product_id=5;

2) **Updating the quantity of a specific product W(A)**:
   Update Product set Product.cquantity=10+Product.restock_trigger where Product.product_id=5;

3) **Reading the address of a specific customer R(B):**
   Select Address from Customer where customer_id=67;

4) **Updating the address of a specific customer W(B):**
   Update Customer set Customer.Address=”XYZ” where Customer.customer_id=67;
   Update Customer set Customer.Address=”ABC” where Customer.customer_id=67;

In the above transactions, since we're updating the same row in the product table (product_id = 5) and the customer table (customer_id = 67), the two transactions are conflicting as they involve read-write operations accessing the same memory location.

In transaction T1, after reading the product table for finding out the product with (product_id = 5) ; it updates the quantity of the product. It then reads the Customer table looking for the customer with (customer_id = 67) and updates the address of that specific customer.

In transaction T2, after reading the product table for finding out the product wih (product_id = 5); it updates the quantity of the product. It then reads the Customer table looking for the customer with (customer_id = 67) and updates the address of that specific customer **twice.**

SERIAL

T1	T2
read(A)	
write(A)	
write(B)	
Commit T1	
	read(A)
	write(A)
	write(B)
	Commit T2

Conflicting the serializable schedule by interchanging non-conflicting queries. The non-conflicting queries are read(A), write(A) with write(B).

<u>CONFLICT SERIALIZABLE</u>

T1	T2
read(A)	
write(A)	
	read(A)
	write(A)
write(B)	
Commit T1	
	write(B)
	Commit T2

Non conflicting serializable schedule by interchanging conflicting queries, those being write(B) of T1 with write(B) with T2

<u>**NON CONFLICT SERIALIZABLE**</u>

T1	T2
read(A)	
write(A)	
	read(A)
	write(A)
	write(B)
	Commit T2
write(B)	
Commit T1	

In order to solve non-conflicting transactions that lead to non-serializable outcomes, we can use concurrency control techniques such as locking, timestamp ordering, or optimistic concurrency control.

Locking: Transactions can gain locks on the data they need access to by using the locking approach. As a result, until the lock is lifted, no other transactions are able to access the same data. Transactions can be serialised with locking, preventing non-conflicting transactions from producing non-serializable results.

Timestamp Ordering: Timestamp ordering is a method in which a different timestamp is given to every transaction. Then, according to their timestamps, transactions are sorted, and each transaction can only access information that was most recently changed by a transaction with a timestamp earlier than it. Transactions can be arranged using timestamps to make sure that non-conflicting transactions do not result in non-serializable results.

Non-Conflicting Transactions:

T3:

VIEW THE CART OF A CUSTOMER R(A):

Select * from cart where customer_id=72

T4:

VIEW PRODUCTS FROM THE CATALOGUE R(B):

Select* from Product;

T5:

VIEW ORDER DETAILS OF A CUSTOMER R(C):

Select* from from Checkout where Customer_Order_info=72;

T6:

VIEW THE ITEMS CHECKEDOUT FROM THE ORDER_ID R(D):

Select * from items_contained where items_contained=53;

These queries are non conflicting among each other as they are reading the data from different memory address.