TRANSACTIONS

Ayush Chauhan
2020188

Conflicting Transactions:

1. Write-Read Conflict:

Suppose, an administrator is updating the contact details of a delivery agent while the delivery agent is already allotted to a customer. In this case, the Customer will have access to old contact details, causing miscommunication between the delivery partner and the customer.

SQL statements for the two transactions are as follows:

- **Transaction 1 (t1)**

Updation of Delivery Agent's Contact Details (phone number) by Admin

```
BEGIN TRANSACTION;
UPDATE Delivery SET phone  = '9999999999'
WHERE DP_ID = '1';
COMMIT;
```

- **Transaction 2 (t2)**

User checking the details of a delivery guy.

```
BEGIN TRANSACTION;
SELECT * FROM Delivery WHERE DP_ID = '1';
COMMIT;
```

Serialization Table with conflict:

Time	Transaction	SQL Statement	Operation
T1	BEGIN		
T1	UPDATE	BEGIN TRANSACTION; UPDATE Delivery SET phone = '999999999'	WRITE on DP_ID = 1
T2	BEGIN		

T2	SELECT	BEGIN TRANSACTION; SELECT * FROM Delivery where DP_IP = 1; COMMIT;	READ on DP_ID = 1
T1	COMMIT		
T2	COMMIT		
END			

As we can see, T1 has a WRITE operation, whereas T2 has a READ operation, As T2 is reading the non-updated values while T1 is updating them, This has caused a WRITE-READ Conflict.

Convert the conflict to non - conflicting transaction:

We must make sure that the transactions are serialized or sequenced in a way that prevents conflicting actions from happening simultaneously in order to prevent read-write conflicts.
Using a serializable isolation level and making sure that all transactions are scheduled in a way that maintains serializability are two ways to accomplish this. This can be accomplished by employing a two-phase locking protocol, in which each transaction is required to obtain each lock that is required before carrying out any write operations and to release each lock after the transaction is finished.

1. In order to update rows, Transaction 1 (T1) first takes an exclusive lock on those rows, which prohibits any other transactions from accessing or changing those rows until the lock is released.
2. In order for subsequent transactions to read the same rows but not update them until the lock is released, Transaction 2 (T2) first acquires a shared lock on the rows it intends to read.
3. Since Transaction 2 (T2) has obtained a shared lock on the rows being read, T2 is able to read the data without encountering any conflicts.
4. Transaction 1 (T1) acquires an exclusive lock on the rows being changed, modifies the data, and then commits the transaction.
5. As soon as Transaction 2 (T2) completes the transaction, the shared lock it had on the rows being read is released.

2. Read Write Conflict:

Assume you have a table named products that contains data about the goods you offer and a table called cart that contains data about the goods that consumers have placed in their shopping carts. You may have one transaction that reads a products price from the products database and then computes the total cost of the items in a customer's cart in order to cause a read-write conflict. Another transaction that adjusts the price of the same product in the products database could occur concurrently.

Due to the outdated price being read in the first transaction, the total price calculation would be incorrect.

SQL statements for the two transactions are as follows:

- **Transaction 1 (t1)**

Reads the price of a product from the products table and calculates the total price of items in a cart.

```
BEGIN TRANSACTION;
SELECT price
FROM products
WHERE product_id = 1';
UPDATE cart
SET total_price = total_price + (SELECT price FROM products WHERE
product_id = 1)
WHERE cart_id = 1;
COMMIT;
```

- **Transaction 2 (t2)**

Updation of the price of the SAME product in the products table.

```
 BEGIN TRANSACTION;
UPDATE products SET price = 1050
WHERE product_id = 1;
COMMIT;
```

Serialization Table with conflict:

TIME	TRANSACTION	SQL STATEMENT	OPERATIION
T1	BEGIN		
T1	UPDATE	BEGIN TRANSACTION; UPDATE cart SET total_price = total_price + (SELECT price FROM products WHERE Product_id = 1); COMMIT;	READ on Product_id = 1
T2	BEGIN		
T2	UPDATE	BEGIN TRANSACTION; UPDATE products SET price = 1050 WHERE Product_ID = 1 COMMIT;	WRITE on Product_id = 1
T1	COMMIT		
T2	COMMIT		

Convert the conflict to non - conflicting transaction:

For this, we can modify the second transaction that updates the price of the product in the products table to acquire an exclusive lock on the row for that product before updating the price. This ensures that no other transaction can read or write to that row until the lock is released, preventing any read-write conflicts.

Then, we can modify the first transaction that calculates the total price of the items in the customer's cart to acquire shared locks on all rows in the cart_items table that correspond to the products in the cart. This ensures that no other transaction can modify the cart_items table until the locks are released, preventing any write-read conflicts.

By using this approach, both transactions can execute in a serialized manner, ensuring that no conflicting actions occur simultaneously and preventing any read-write conflicts

Non-Conflicting Transactions:

Insert a new product:

```
BEGIN TRANSACTION T1;
INSERT into product (Product_ID, Category_ID, Name, Price, Quantity)
Values (1, 1, Coke, 40, 88);
COMMIT;
```

Update name of a product:

```
BEGIN TRANSACTION T2;
update products
set
        Name = 'MSI Gaming'
where
        Product_ID = 2;
COMMIT;
```

Updating the prices of all products in a category

```
BEGIN TRANSACTION T3;
update products
set
        price = price*2
where
        Category_ID = 4;
COMMIT;
```

Place an order

```
BEGIN TRANSACTION T4;
INSERT into orders (Order_ID, Cust_ID, Payment_ID, Cart_ID)
Values (5, 2, 3, 1)
COMMIT;
```