ONLINE RETAIL STORE

Group 50

Samanyu Kamra 2021487

Shriya Verma 2021490

A. CONFLICTING TRANSACTIONS :

1. READ-WRITE CONFLICT:

Let's say you have a table called products that stores information about the products you sell and another table called cart_items that stores information about the items in customers' shopping carts. To create a read-write conflict, you could have one transaction that reads the price of a product from the products table and then calculates the total price of the items in a customer's cart. At the same time, you could have another transaction that updates the price of the same product in the products table. This would cause the first transaction to have an incorrect total price calculation since it read an outdated price.

TIME	STEP	TRANSACTION	SQL STATEMENT	OPERATIONS
T1	1	TRANSACTION 1	BEGIN;	BEGIN
T1	2	TRANSACTION 1	SELECT price FROM products WHERE product_id = “405”	READ(products: 405)
T2	1	TRANSACTION 2	BEGIN;	BEGIN
T1	3	TRANSACTION 1	UPDATE products SET price = 50 WHERE product_id = ‘405';	WRITE(products: 405)

T1	4	TRANSACTION 1	SELECT SUM(quantity*price) FROM Cart_items	READ(Cart_items : 405), READ(products : 405)
T2	3	TRANSACTION 2	COMMIT;	COMMIT;
T1	5	TRANSACTION 1	COMMIT;	WRITE(Cart_items : 405) WRITE(Products : 405) COMMIT

T1
BEGIN TRANSACTION;

```
SELECT price
FROM products
WHERE product_id = 'P01';
```

Calculate the total price of cart items

```
UPDATE cart
SET total_price = total_price + (SELECT price FROM
products WHERE product_id = 'P01')
WHERE cart_id = 'C01';
COMMIT;
```

T2
BEGIN TRANSACTION;

```
UPDATE products
SET price = 200
WHERE product_id = 'P01';
COMMIT;
```

T1: Reads the price of a product from the products table and calculates the total price of items in a cart.
T2: Updation of the price of the SAME product in the products table

T1(R P01) ---> T2(W P01)

Convert the conflict to non - conflicting transaction:
To avoid read-write conflicts, we need to ensure that the transactions are serialized or ordered in a way that prevents conflicting actions from occurring simultaneously.

One way to achieve this is by using serializable isolation level and ensuring that all transactions are scheduled in a way that maintains serializability. This can be done by using a two-phase **locking** protocol, where each transaction must acquire all necessary locks before executing any write operation, and then release all locks once the transaction is complete.

To apply this approach to the scenario you described, 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.

Step wise:

1. Since Transaction 1 already acquired shared locks on the rows in the Cart_Items table, Transaction 2 must wait until Transaction 1 releases the locks before it can acquire the exclusive lock on Product B.
2. Transaction 1 calculates the total price of the items in the customer's cart using the current prices in the Products table.
3. Transaction 1 releases the shared locks on the rows in the Cart_Items table.
4. Transaction 2 acquires the exclusive lock on the row for Product B and updates the price.
5. Transaction 2 releases the exclusive lock on the row for Product B.
6. Both transactions are committed.
7. Using this approach ensures that Transaction 1 completes before Transaction 2 updates the price of Product B, preventing any read-write conflicts.

To avoid deadlocks:
Alternatively, we can also use optimistic concurrency control by allowing both transactions to proceed with their operations without locking any resources, but checking for conflicts before committing the changes. If a conflict is detected, one of the transactions will be rolled back, and

the other can proceed with its changes. However, this approach may result in more rollbacks and retries, which can affect performance.

2. WRITE-READ CONFLICT:

Here, 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 STATEMENT:

- **Transaction 1 (T1):**

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

```
BEGIN TRANSACTION;
UPDATE Delivery SET phone_num = '9195551234'
WHERE agent_id = '19024';
 COMMIT;
```

- **Transaction 2 (T2):**

```
BEGIN TRANSACTION;
SELECT * FROM Delivery WHERE agent_id = '19024';
COMMIT;
```

Serialization Table :

TIME	TRANSACTION	SQL STATEMENT	OPERATIONS
T1	BEGIN		
T1	UPDATE	BEGIN TRANSACTION; UPDATE Delivery SET phone_num =	WRITE(Agent_id = 19024)

		'9195551234' WHERE agent_id = '19024';COMMIT;	
T2	BEGIN		
T2	SELECT	BEGIN TRANSACTION; SELECT * FROM Delivery WHERE agent_id = '19024'; COMMIT;	READ(Agent_id = 19024)
T1	COMMIT		
T2	COMMIT		

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:

1. Transaction 1 (T1) begins by acquiring an exclusive lock on the rows it is going to update, which prevents any other transactions from accessing or modifying the same rows until the lock is released.
2. Transaction 2 (T2) begins by acquiring a shared lock on the rows it is going to read, which allows other transactions to read the same rows but prevents them from updating them until the lock is released.
3. Transaction 2 (T2) reads the data without any conflicts because it has acquired a shared lock on the rows being read.
4. Transaction 1 (T1) updates the data and commits the transaction, releasing the exclusive lock it had acquired on the rows being updated.

5. Transaction 2 (T2) commits the transaction, releasing the shared lock it had acquired on the rows being read.

RESOLVING THE CONFLICT :

In order to resolve this conflict, we can either :

1. ABORT one transaction.
2. DELAY a transaction until the other one is complete.

Conflicts are usually resolved via the following methods :
1. Locking: Prevents multiple transactions from accessing the same data at the same time.
2. Timestamping: Ordering of transactions, the earlier time stamped transaction is prioritized.
3. Conflict Detection and Resolution: This can be practiced using specialized algorithms.
4. Optimistic Concurrency Control: Assumes conflicts are rare and checks if other users have made changes while one user has; if changes are found, the user is notified that changes were not saved and has to do the process again.

B. NON-CONFLICTING TRANSACTIONS :

1. **Insert a new delivery agent :**

```
BEGIN TRANSACTION T1;
INSERT INTO Agents (agent_id, first_name, last_name, password, phone_number, email, average_rating)
VALUES ('D001', 'Jane', 'Smith', 'password', '123-456-7890', 'janesmith@email.com', 4.5);

COMMIT TRANSACTION T1;
```

2. **Insert a new offer**

```
BEGIN TRANSACTION T2;

INSERT INTO Offers (offer_id, discount_percentage, promo_code, min_orderValue,

max_discountValue)

VALUES ('O001', 10, 'OFF10', 5000, 1000);

COMMIT TRANSACTION T2;
```

3. **Apply a discount to the price of the cart and update the order price**

```
START TRANSACTION;
UPDATE _order
JOIN cart ON _order.cart_id = cart.cart_id
JOIN offer ON cart.offer_id = offer.offer_id
SET _order.amount = cart.total_price * (1 - offer.percentagediscount/100)
WHERE _order.order_id = 15091;

COMMIT;
```

4. **Updating the prices of all products in a category**

```
BEGIN;
UPDATE product
SET price = price * 1.1
WHERE product.category_id = (
SELECT category.category_id
FROM category
WHERE category.category_name = 'Dairy'
);
COMMIT;
```