DATABASE TRANSACTIONS

Conflicting Transactions

Conflicting transactions arise when multiple parties attempt actions that contradict each other, leading to disputes. Resolving them requires careful analysis and negotiation for a mutually agreeable solution, vital for maintaining integrity and trust.

Conflicting Transactions 1:

Conflicting transaction scenario can be created while adding items to the cart by two customers simultaneously. We can simulate a situation where both customers try to add the same product to their carts at the same time, potentially leading to inconsistencies or conflicts in the data. Let's assume Customer A and Customer B both try to add the same product to their carts simultaneously. Here's how the conflicting transaction could occur:

Customer A’s Transaction

```
-- For Customer A's Transaction:
START TRANSACTION;

-- Check if the product exists and is available
SELECT * FROM Product WHERE Product_ID = <product_id> FOR UPDATE;

-- Check if there's enough quantity of the product available
SELECT Quantity FROM Product WHERE Product_ID = <product_id> FOR UPDATE;

-- If the product is available and there's enough quantity, add it to the cart
INSERT INTO cart (Customer_ID, Product_ID, Supplier_ID, quantity)
VALUES (<customer_A_id>, <product_id>, <supplier_id>, <quantity>);

COMMIT;
```

Customer B's Transaction

```
-- For Customer B's Transaction
START TRANSACTION;

-- Check if the product exists and is available
SELECT * FROM Product WHERE Product_ID = <product_id> FOR UPDATE;

-- Check if there's enough quantity of the product available
SELECT Quantity FROM Product WHERE Product_ID = <product_id> FOR UPDATE;

-- If the product is available and there's enough quantity, add it to the cart
INSERT INTO cart (Customer_ID, Product_ID, Supplier_ID, quantity)
VALUES (<customer_B_id>, <product_id>, <supplier_id>, <quantity>);

COMMIT;
```

In this scenario, both Customer A and Customer B are attempting to add the same product to their carts concurrently. However, since both transactions are isolated with the '`START TRANSACTION`' and '`COMMIT`' statements, and they both lock the rows they are accessing with '`FOR UPDATE`', only one of the transactions can proceed while the other will wait until the first transaction commits or rolls back.

This situation can lead to a conflict if, for example, Customer A's transaction commits first, reducing the available quantity of the product before Customer B's transaction commits. In such a case, Customer B's transaction would fail due to insufficient quantity, or the database would have to handle the conflict resolution based on the transaction isolation level and concurrency control mechanisms in place.

Conflict-Serializable Schedule

Making a Conflict-Serializable Schedule for showing the above conflicting transactions involves showing the sequence of events during the conflicting transaction

Let's denote:

- T1: Customer A's transaction
- T2: Customer B's transaction

And the actions as follows:

- R(X): Read operation on data item X
- W(X): Write operation on data item X
- C: Committing all the all changed data items after the transactions

Here's the Conflict-Serializable Schedule table:

Action	Tíansaction
R(X)	T1
R(X)	T2
W(X)	T1
W(X)	T2
C	T1
C	T2

Conflict-Serializable Schedule with Locks

Creating a Conflict-Serializable Schedule table showing the sequence of events during the conflicting transaction, we'll outline the actions performed by Customer A and Customer B along with their corresponding locks and unlocks.

Let's denote:

- T1: Customer A's transaction
- T2: Customer B's transaction

And the actions as follows:

- R(X): Read operation on data item X
- W(X): Write operation on data item X
- Lock(X): Acquire lock on data item X
- Unlock(X): Release lock on data item X

- C: Committing all the all changed data items after the transactions

The schedule should ensure conflict serializability, meaning that the outcome of the transactions should be the same as if they were executed serially in some order.

Here's the Conflict-Serializable Schedule table:

Action	Tíansaction	Lock/Unlock
Lock(X)	T1	Lock
R(X)	T1	Read
Lock(X)	T2	Wait (blocked)
R(X)	T2	Wait (blocked)
W(X)	T1	Write
W(X)	T2	Wait (blocked)
C	T1	Commit
Unlock(X)	T1	Unlock
C	T2	Commit
Unlock(X)	T2	Unlock

Conflicting Transactions 2:

Another conflicting transaction scenario can be created where two suppliers attempt to change the name of the same product simultaneously, we need to ensure that both transactions are altering the same data item without proper synchronization. Here's how it could look:

Customer A's Transaction

```
START TRANSACTION;
UPDATE Product
SET Name = 'New Product Name A'
WHERE Product_ID = <product_id> AND Supplier_ID = <supplier_a_id>;
-- Hold the transaction without committing
```

Customer B's Transaction

```
START TRANSACTION;
UPDATE Product
SET Name = 'New Product Name B'
WHERE Product_ID = <product_id> AND Supplier_ID = <supplier_b_id>;
-- Hold the transaction without committing
```

In this scenario:

- Both Supplier A and Supplier B are attempting to update the name of the same product with different values ('<new_name_A>' and '<new_name_B>' respectively) concurrently.
- Without proper synchronization or locking mechanisms, this could lead to conflicts where one update overwrites the changes made by the other, resulting in inconsistent data.

To prevent such conflicts, you would typically use locking mechanisms or concurrency control techniques such as transaction isolation levels to ensure that only one transaction can modify the product's name at a time, or you might implement a conflict resolution strategy to handle concurrent updates gracefully.

Conflict-Serializable Schedule

To create a Conflict-Serializable Schedule table without locks for the scenario where two suppliers attempt to change the name of the same product simultaneously, we rely on the inherent serialization of transactions. Here's how it could look:

The actions as follows:

- R(X): Read operation on data item X
- W(X): Write operation on data item X
- C: Committing all the all changed data items after the transactions

Here's the Conflict-Serializable Schedule table:

Action	Tíansaction
R(X)	A
R(X)	B
W(X)	A
W(X)	B
C	A
C	B

Conflict-Serializable Schedule with Locks

Creating a Conflict-Serializable Schedule table showing the sequence of events during the conflicting transaction. Let's denote:

- T1: Customer A's transaction
- T2: Customer B's transaction

And the actions as follows:

- R(X): Read operation on data item X
- W(X): Write operation on data item X
- Lock(X): Acquire lock on data item X
- Unlock(X): Release lock on data item X
- C: Committing all the all changed data items after the transactions

Here's the Conflict-Serializable Schedule table:

Action	Tíansaction	Lock/Unlock
Lock(X)	T1	Lock
R(X)	T1	Read
Lock(X)	T2	Wait (blocked)
R(X)	T2	Wait (blocked)
W(X)	T1	Write
W(X)	T2	Wait (blocked)
C	T1	Commit
Unlock(X)	T1	Unlock
C	T2	Commit
Unlock(X)	T2	Unlock

In this schedule with locks:

- Both Supplier A and Supplier B first acquire a lock on the product before attempting to update its name.
- Supplier A acquires the lock first, updates the product's name, and then releases the lock.
- Supplier B attempts to acquire the lock but is blocked because Supplier A holds the lock. Supplier B waits until Supplier A releases the lock.
- Once Supplier A releases the lock, Supplier B acquires the lock, updates the product's name, and then releases the lock.

This schedule ensures conflict serializability by using locks to synchronize access to the shared resource (the product). It prevents conflicts where both suppliers attempt to update the product's name simultaneously, ensuring that only one transaction can modify the product's name at a time.

Non-Conflicting Transactions

Deleting items from cart:

We search for a particular product from the cart and start a transaction in order to delete it from the cart, by simply searching based on the product_id.

```
def delete_from_cart():
    product_id = request.form['product_id']
    supplier_id = request.form['supplier_id']

    cursor = mysql.connection.cursor()
    try:
        cursor.execute("START TRANSACTION")
        cursor.execute("DELETE FROM cart WHERE Customer_ID = %s AND
Product_ID = %s AND Supplier_ID = %s", (session["customer"][0], product_id,
supplier_id))
        cursor.execute("COMMIT")
        cursor.close()
        mysql.connection.commit()
    except Exception as e:
        mysql.connection.rollback()
        cursor.close()
    return redirect(url_for('checkout'))
```

Updating the Username and Password:

We make 2 transactions in order to edit the username and the password of a person, we turn off the safe mode because the username and password are not the primary keys.

```
def update_profile():

    try:

        new_email = request.form.get('email')

        new_password = request.form.get('password')

        print(new_email)

        original_email =  session["customer"]

        cursor = mysql.connection.cursor()


        if new_email and new_email != original_email:

            cursor.execute("START TRANSACTION")

            update_email_query = '''SET SQL_SAFE_UPDATES=0;

            UPDATE email NATURAL JOIN CUSTOMER SET em = %s WHERE
Customer_ID = %s;

            SET SQL_SAFE_UPDATES=1;'''

            cursor.execute(update_email_query, (str(new_email),
original_email[0]))

            cursor.execute("COMMIT")


        if new_password :

            cursor.execute("START TRANSACTION")
```

```
            update_password_query = "UPDATE customer SET password = %s
WHERE Customer_ID = %s"

            cursor.execute(update_password_query, (new_password,
original_email[0]))

            cursor.execute("COMMIT")

        cursor.close()

        mysql.connection.commit()

        flash('Profile updated successfully', 'success')

        return redirect('/profile')

    except Exception as e:

        mysql.connection.rollback()

        flash('An error occurred while updating profile', 'error')

        return redirect('/profile')
```

Deleting items from the Cart:

```
def delete_from_cart():

    product_id = request.form['product_id']

    supplier_id = request.form['supplier_id']



    cursor = mysql.connection.cursor()

    try:

        cursor.execute("START TRANSACTION")
```

```
        cursor.execute("DELETE FROM cart WHERE Customer_ID = %s AND Product_ID = %s AND Supplier_ID = %s", (session["customer"][0], product_id, supplier_id))

        cursor.execute("COMMIT")

        cursor.close()

        mysql.connection.commit()

    except Exception as e:

        mysql.connection.rollback()

        cursor.close()

    return redirect(url_for('checkout'))
```

Updating the product name:

```
def update_product_name():

    new_name = request.form.get('new_name')



    with connection.cursor() as cursor:

        cursor.execute("SET SQL_SAFE_UPDATES=0")

        cursor.execute("START TRANSACTION")

        sql_query = """

            UPDATE Product

            SET Name = %s

            WHERE Supplier_ID = %s;

        """
```

```
        cursor.execute(sql_query, (new_name, session["supplier"]))

        cursor.execute("SET SQL_SAFE_UPDATES=1")



        cursor.execute("COMMIT")

        connection.commit()

    return redirect('/supplier_dashboard')
```