TRANSACTIONS

Sanmay Sood (2021095)
Sanskar Ranjan (2021096)

Non-conflicting transactions:

1. <u>Add a new product</u>

```
BEGIN TRANSACTION;
INSERT INTO product
(product_id,prodduct_name,product_price,product_stock,product_rating)  VALUES
(203,'Munch',20,38,4.1);
COMMIT TRANSACTION;
```

2. <u>Change Password</u>

```
BEGIN TRANSACTION;
UPDATE customers SET pass = 'Sanmay7788' WHERE customer_id = 145;
COMMIT TRANSACTION;
```

3. <u>Remove product from cart</u>

```
BEGIN TRANSACTION;
DELETE FROM cart WHERE product_id = 50;
COMMIT TRANSACTION;
```

4. View details of a customer

```
BEGIN TRANSACTION;
SELECT * FROM  customers WHERE customer_id = 115;
COMMIT TRANSACTION;
```

The above transactions are non-conflicting because they all operate on different tables or rows within the same table.

The first transaction inserts a new record into the product table with a unique product ID, product name, price, stock, and rating.

The second transaction updates a single row in the customers table, changing the password for the customer with customer_id of 145.

The third transaction deletes a row from the cart table, removing product with product_id of 50 from the cart.

The fourth transaction displays the details of a customer with customer_id of 115.

Since these statements operate on different tables or different rows within the same table, they can be executed concurrently without causing conflicts, hence they are non conflicting transactions.

Conflicting transactions

<u>Example 1:</u>

This transaction reduces the order_total by giving a discount of 5% on order with order_id = 15 and 27

TRANSACTION T1

```
BEGIN TRANSACTION;

UPDATE orders
SET order_total = order_total - 0.05*order_total
WHERE order_id = 15;

UPDATE orders
SET order_total = order_total - 0.05*order_total
WHERE order_id = 27;

COMMIT TRANSACTION;
```

This transaction adds 10 to the order total with order_id = 15 and 27.

TRANSACTION T2

```
BEGIN TRANSACTION;

UPDATE orders
SET order_total = order_total + 10
WHERE order_id = 15;

UPDATE orders
```

SET order_total = order_total + 10
WHERE order_id = 27;

COMMIT TRANSACTION;

This is an example of **write-write conflict** (which means overwriting uncommitted data).

Here's an example of a schedule that is **not conflict serialisable:**

Schedule S1

Transaction 1	**Transaction 2**
BEGIN TRANSACTION;	

UPDATE orders
SET order_total = order_total - 0.05*order_total
WHERE order_id = 15;	

	BEGIN TRANSACTION;

	UPDATE orders
	SET order_total = order_total + 10
	WHERE order_id = 27;

UPDATE orders
SET order_total = order_total - 0.05*order_total
WHERE order_id = 27;	

COMMIT TRANSACTION;	

	UPDATE orders
	SET order_total = order_total + 10
	WHERE order_id = 15;

	COMMIT TRANSACTION;

Schedule S2 - **Serial Schedule**

Transaction 1	**Transaction 2**
BEGIN TRANSACTION;	

UPDATE orders
SET order_total = order_total - 0.05*order_total
WHERE order_id = 15;	

UPDATE orders
SET order_total = order_total - 0.05*order_total
WHERE order_id = 27;	

COMMIT TRANSACTION;	
	BEGIN TRANSACTION;

	UPDATE orders
	SET order_total = order_total + 10
	WHERE order_id = 27;

	UPDATE orders
	SET order_total = order_total + 10
	WHERE order_id = 15;

	COMMIT TRANSACTION;

The above schedule S1 is not conflict equivalent to any of the possible serial schedules.
Here is an example of a **serial schedule S2 that is not conflict equivalent to S1.**
Since S1 is not conflict equivalent to a serial schedule, it is **not conflict serialisable.**
Schedule S1 is inconsistent.

Here's an example of a schedule that is **conflict serialisable:**

Schedule S1

Transaction 1	Transaction 2
BEGIN TRANSACTION;	

UPDATE orders
SET order_total = order_total - 0.05*order_total
WHERE order_id = 15;	

	BEGIN TRANSACTION;

	UPDATE orders
	SET order_total = order_total + 10
	WHERE order_id = 15;

UPDATE orders
SET order_total = order_total - 0.05*order_total
WHERE order_id = 27;	

COMMIT TRANSACTION;	

	UPDATE orders
	SET order_total = order_total + 10
	WHERE order_id = 27;

	COMMIT TRANSACTION;

The above schedule is a concurrent schedule S1. It is conflict equivalent to a serial schedule S2 given below. So schedule **S1 is a conflict serializable schedule**, which means that it maintains **data consistency.**

Schedule S2 - **Serial Schedule**

Transaction 1	Transaction 2
BEGIN TRANSACTION;	

UPDATE orders
SET order_total = order_total - 0.05*order_total
WHERE order_id = 15;	

UPDATE orders
SET order_total = order_total - 0.05*order_total
WHERE order_id = 27;	

COMMIT TRANSACTION;	
	BEGIN TRANSACTION;

	UPDATE orders
	SET order_total = order_total + 10
	WHERE order_id = 15;

	UPDATE orders
	SET order_total = order_total + 10
	WHERE order_id = 27;

	COMMIT TRANSACTION;

Example 2:

TRANSACTION T1

BEGIN TRANSACTION;

SELECT product_stock FROM product WHERE product_id = 21;
UPDATE product SET product_stock = 9 WHERE product_id = 21;
SELECT product_stock FROM product WHERE product_id = 55;
UPDATE product SET product_stock = 6 WHERE product_id = 55;

COMMIT TRANSACTION;

TRANSACTION T2

BEGIN TRANSACTION;

SELECT product_stock FROM product WHERE product_id = 21;
SELECT product_stock FROM product WHERE product_id = 55;

COMMIT TRANSACTION;

This is an example of a **write-read conflict**, where a transaction reads a value that has been modified by another transaction, but has not yet been committed**.** This is also called **dirty read or uncommitted read.**

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

Schedule S1

Transaction 1	**Transaction 2**
BEGIN TRANSACTION	
SELECT product_stock FROM product WHERE product_id = 21;	
UPDATE product SET product_stock = 9 WHERE product_id = 21;	
	BEGIN TRANSACTION
	SELECT product_stock FROM product WHERE product_id = 21;
SELECT product_stock FROM product WHERE product_id = 55;	
UPDATE product SET product_stock = 6 WHERE product_id = 55;	
COMMIT TRANSACTION	
	SELECT product_stock FROM product WHERE product_id = 55;
	COMMIT TRANSACTION

The above schedule is a concurrent schedule S1.

It is **conflict equivalent to a serial schedule S2** given below.

So schedule **S1 is a conflict serializable schedule**, which means that it maintains data **consistency.**

Schedule S2 - **Serial Schedule**

Transaction 1	Transaction 2
BEGIN TRANSACTION	
SELECT product_stock FROM product WHERE product_id = 21;	
UPDATE product SET product_stock = 9 WHERE product_id = 21;	
SELECT product_stock FROM product WHERE product_id = 55;	
UPDATE product SET product_stock = 6 WHERE product_id = 55;	
COMMIT TRANSACTION	
	BEGIN TRANSACTION
	SELECT product_stock FROM product WHERE product_id = 21;
	SELECT product_stock FROM product WHERE product_id = 55;
	COMMIT TRANSACTION

Here's an example of a schedule that is not conflict serialisable:

Schedule S1

Transaction 1	Transaction 2
BEGIN TRANSACTION	
SELECT product_stock FROM product WHERE product_id = 21;	
UPDATE product SET product_stock = 9 WHERE product_id = 21;	

	BEGIN TRANSACTION
	SELECT product_stock FROM product WHERE product_id = 21;
	SELECT product_stock FROM product WHERE product_id = 55;
	COMMIT TRANSACTION
SELECT product_stock FROM product WHERE product_id = 55;	
UPDATE product SET product_stock = 6 WHERE product_id = 55;	
COMMIT TRANSACTION	

The above schedule S1 is not conflict equivalent to any of the possible serial schedules.
Here is an example of a serial schedule S2 that is not conflict equivalent to S1.
Since S1 is not conflcit equivalent to a serial schedule, it is not conflict serialisable.
Schedule S1 is inconsistent.

Schedule S2 - **Serial Schedule**

Transaction 1	**Transaction 2**
BEGIN TRANSACTION	
SELECT product_stock FROM product WHERE product_id = 21;	
UPDATE product SET product_stock = 9 WHERE product_id = 21;	
SELECT product_stock FROM product WHERE product_id = 55;	
UPDATE product SET product_stock = 6 WHERE product_id = 55;	
COMMIT TRANSACTION	
	BEGIN TRANSACTION

	SELECT product_stock FROM product WHERE product_id = 21;
	SELECT product_stock FROM product WHERE product_id = 55;
	COMMIT TRANSACTION

HOW TO RESOLVE THE CONFLICTS :

1. Locking: Prevents multiple transactions from accessing the same data at the same time.

2. Timestamping: Ordering of transactions, the earlier timestamped 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.