ONLINE RETAIL STORE

Group 102: Saketh Ragirolla 2021092 & Ritisha Singh 2021089

Conflicting Transactions

Conflicting transactions can be both conflict serializable and non-conflict serializable.

How to resolve the conflict?

Since schedules are concurrent to achieve recoverability and serializability, we use different methods:

1. **Locking:** Locking restricts concurrent access to data items using shared, exclusive, and deadlock detection.
2. **Timestamp ordering:** Timestamp ordering orders transactions based on their timestamps to ensure concurrency.
3. **Two-phase locking:** Two-phase locking is a concurrency control method that divides a transaction into two phases: a locking phase and an unlocking phase.
4. **Optimistic concurrency control:** Optimistic concurrency control assumes that conflicts are rare, and each transaction reads a version of the data item and writes a new version when it commits.
5. **Multi-version concurrency control:** Multi-version concurrency control creates multiple versions of a data item and assigns timestamps to them, allowing transactions to access the database concurrently without acquiring locks.

FIRST SET:

-- T1

```
START TRANSACTION;

SELECT * FROM Product WHERE prod_id = 1195;

UPDATE Product
SET
	Availability = "No"
WHERE
	Prod_Id = 1195;
COMMIT;
```

-- T2

```
START TRANSACTION;

UPDATE Product
SET
        price = 1488
WHERE
        Prod_Id = 1195;
COMMIT;


-- T3

START TRANSACTION;

INSERT INTO Product
        (Prod_ID,Supplier_ID,Category_ID,Prod_name,Brand,Price,Description,Availability)
VALUES
        (1195,165,401,"Poppy","Freshflora",1037,"freshasever,","Yes");

COMMIT;
```

NON - CONFLICT SERIALISABLE SCHEDULE (NON-SERIAL)

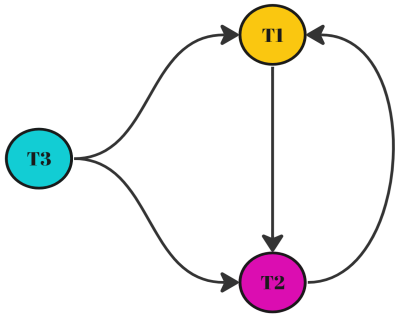
Transaction table:

Step	T1	T2	T3
1	START		
2		START	
3			START
4			Write(Product) **X**
5			COMMIT **U**
6	Read(Product) **S** **U**		
7		Write(Product) **X**	
8		COMMIT **U**	
9	Write(Product) **X**		
10	COMMIT **U**		

Operation Table:

Step	Transactions	Operation	Data Value
1	T1	START TRANSACTION;	N/ A
2	T2	START TRANSACTION;	N/ A
3	T3	START TRANSACTION;	N/ A
4	T3	INSERT INTO Product (Prod_ID,Supplier_ID,Category_ID,Prod_name,Brand,Price,Description,Availability) VALUES (1195,165,401,"Poppy","Freshflora",1037,"freshasever,","Yes");	All
5	T3	COMMIT;	N/ A
6	T1	SELECT * FROM Product WHERE prod_id = 1195;	All
7	T2	UPDATE Product SET price = 1488 WHERE Prod_Id = 1195; COMMIT;	price
8	T2	COMMIT;	N/ A
9	T1	UPDATE Product SET Availability = "No" WHERE Prod_Id = 1195; COMMIT;	Availability
10	T1	COMMIT;	N/ A

Precedence Graph:



Since, we can detect a cycle in the sub-graph, G(V', E'), where, V' ={ T1, T2}, and these nodes represent conflicting operations, hence the schedule above is not - conflict serialisable.

CONFLICT SERIALISABLE SCHEDULE (NON- SERIAL)

Transaction table:

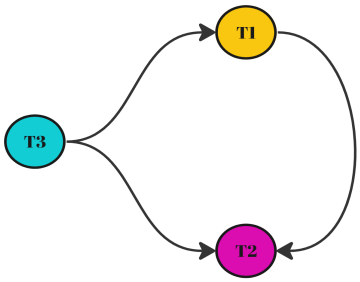
Step	T1	T2	T3
1	START		
2		START	
3			START
4			Write(Product) **X**
5			COMMIT **U**
6	Read(Product) **X**		
7	Write(Product)		
8	COMMIT **U**		
9		Write(Product) **X**	
10		COMMIT **U**	

Operation Table:

Step	Transactions	Operation	Data Value
1	T1	START TRANSACTION;	N/ A
2	T2	START TRANSACTION;	N/ A
3	T3	START TRANSACTION;	N/ A
4	T3	INSERT INTO Product (Prod_ID,Supplier_ID,Category_ID,Prod_name,Brand,Price,Description,Availability) VALUES (1195,165,401,"Poppy","Freshflora",1037,"freshaseaver,","Yes");	All

5	T3	COMMIT;	N/ A
6	T1	SELECT * FROM Product WHERE prod_id = 1195;	All
7	T1	UPDATE Product SET Availability = "No" WHERE Prod_Id = 1195; COMMIT;	Availability
8	T1	COMMIT;	N/ A
9	T2	UPDATE Product SET price = 1488 WHERE Prod_Id = 1195; COMMIT;	price
10	T2	COMMIT;	N/ A

Precedence Graph:



Since the precedence graph is acyclic, hence the schedule is conflict serialisable. (T3→ T1→ T2)

SECOND SET:

-- T1

START TRANSACTION;

INSERT INTO Cart (Cart_ID,Customer_ID,Total_cost,Deal_ID,No_of_items)

```
VALUES
  (800,597,6500,491,12);

UPDATE Offer SET Discount = Discount + 400 WHERE Deal_Id = 492;


COMMIT;



-- T2

START TRANSACTION;

SELECT total_cost INTO @cart_cost FROM Cart WHERE cart_id = 800;

UPDATE Cart SET deal_id =
  (SELECT Deal_id FROM Offer WHERE discount =
    (SELECT MAX(discount) FROM Offer WHERE min_requirement <= @cart_cost)
  AND min_requirement <= @cart_cost
  ORDER BY min_requirement DESC LIMIT 1)
WHERE cart_id = 800;

COMMIT;

-- T3

START TRANSACTION;

SELECT * FROM Cart WHERE Cart_id = 800;

SELECT total_cost INTO @cart_cost FROM Cart WHERE cart_id = 800;

SELECT Discount INTO @discount FROM CART, OFFER WHERE Cart.Deal_id = Offer.Deal_id AND
Cart.Cart_id = 800;

-- If the total number of items in the cart is greater than or equal to 10, apply the maximum discount to the
total cost
UPDATE Cart SET total_cost = @cart_cost - @discount WHERE cart_id =800;


COMMIT;

-- T4

START TRANSACTION;

SELECT * FROM Offer WHERE Deal_id = 492;

COMMIT;
```

NON - CONFLICT SERIALISABLE SCHEDULE (NON-SERIAL)

Transaction table:

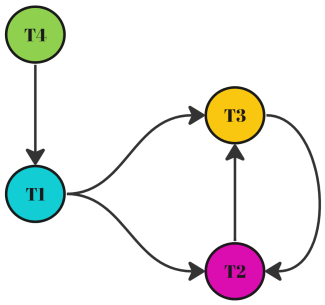
Step	T1	T2	T3	T4
1	START			
2		START		
3			START	
4				START
5	Write (Cart) **X** **U**			
6		Read (Cart) **S** **U**		
7				Read (Offer) **S**
8				COMMIT **U**
9			Read (Cart) **X**	
10			Read (Cart)	
11			Read (Cart)	
12			Write(Cart)	
13			COMMIT **U**	
14	Write (Offer) **X**			
15	COMMIT **U**			
16		Write (Cart) **X**		
17		COMMIT **U**		

Operations Table:

Step	Transactions	Operation	Datavalue
1	T1	START TRANSACTION;	N/ A
2	T2	START TRANSACTION;	N/ A

3	T3	START TRANSACTION;	N/ A
4	T4	START TRANSACTION;	N/ A
5	T1	INSERT INTO Cart (Cart_ID,Customer_ID,Total_cost,Deal_ID,No_of_items) VALUES (800,597,6500,491,12);	All
6	T2	SELECT total_cost INTO @cart_cost FROM Cart WHERE cart_id = 800;	total_cost
7	T4	SELECT * FROM Offer WHERE Deal_id = 492;	All
8	T4	COMMIT;	N/ A
9	T3	SELECT * FROM Cart WHERE Cart_id = 800;	All
10	T3	SELECT total_cost INTO @cart_cost FROM Cart WHERE cart_id = 800;	total_cost
11	T3	SELECT Discount INTO @discount FROM CART, OFFER WHERE Cart.Deal_id = Offer.Deal_id AND Cart.Cart_id = 800;	discount
12	T3	UPDATE Cart SET total_cost = @cart_cost - @discount WHERE cart_id =800;	total_cost
13	T3	COMMIT;	N/ A
14	T1	UPDATE Offer SET Discount = Discount + 400 WHERE Deal_Id = 492;	discount
15	T1	COMMIT;	N/ A
16	T2	UPDATE Cart SET deal_id = (SELECT Deal_id FROM Offer WHERE discount = (SELECT MAX(discount) FROM Offer WHERE min_requirement <= @cart_cost) AND min_requirement <= @cart_cost ORDER BY min_requirement DESC LIMIT 1) WHERE cart_id = 800;	deal_id
17	T2	COMMIT;	N/ A

Precedence Graph:



Since, we can detect a cycle in the sub-graph, G(V', E'), where, V' ={ T2, T3}, and these nodes represent conflicting operations, hence the schedule above is not - conflict serialisable.

CONFLICT SERIALISABLE SCHEDULE (NON SERIAL)

Transaction table:

Step	T1	T2	T3	T4
1	START			
2		START		
3			START	
4				START
5	Write (Cart) **X U**			
6		Read (Cart) **X**		
7		Write (Cart)		
8		COMMIT **U**		
9				Read (Offer) **S**
10				COMMIT **U**
11			Read (Cart) **X**	
12			Read (Cart)	
13			Read (Cart)	
14			Write(Cart)	
15			COMMIT **U**	
16	Write (Offer) **X**			
17	COMMIT **U**			

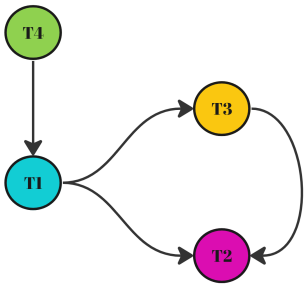
Operations Table:

Step	Transactions	Operation	Datavalue
1	T1	START TRANSACTION;	N/ A

2	T2	START TRANSACTION;	N/ A
3	T3	START TRANSACTION;	N/ A
4	T4	START TRANSACTION;	N/ A
5	T1	INSERT INTO Cart (Cart_ID,Customer_ID,Total_cost,Deal_ID,No_of_items) VALUES (800,597,6500,491,12);	All
6	T2	SELECT total_cost INTO @cart_cost FROM Cart WHERE cart_id = 800;	total_cost
7	T2	UPDATE Cart SET deal_id = (SELECT Deal_id FROM Offer WHERE discount = (SELECT MAX(discount) FROM Offer WHERE min_requirement <= @cart_cost) AND min_requirement <= @cart_cost ORDER BY min_requirement DESC LIMIT 1) WHERE cart_id = 800;	deal_id
8	T2	COMMIT;	N/ A
9	T4	SELECT * FROM Offer WHERE Deal_id = 492;	All
10	T4	COMMIT;	N/ A
11	T3	SELECT * FROM Cart WHERE Cart_id = 800;	All
12	T3	SELECT total_cost INTO @cart_cost FROM Cart WHERE cart_id = 800;	total_cost
13	T3	SELECT Discount INTO @discount FROM CART, OFFER WHERE Cart.Deal_id = Offer.Deal_id AND Cart.Cart_id = 800;	discount
14	T3	UPDATE Cart SET total_cost = @cart_cost - @discount WHERE cart_id =800;	total_cost
15	T3	COMMIT;	N/ A
16	T1	UPDATE Offer SET Discount = Discount + 400 WHERE Deal_Id = 492;	discount
17	T1	COMMIT;	N/ A

Precedence Graph:

Since the precedence graph is acyclic, hence the schedule is conflict serialisable. (T4→ T1→ T3→ T2)



To resolve the conflict using locking, we can use either of the **LOCKING** methods:

Shared-Exclusive Lock:
If a Transaction demands Read - Write access, give Exclusive lock.
If Read only, provide shared lock.

	GRANT (S , X)	
Request (S, X)	Yes (S, S)	NO (X, S)
	NO (X, S	NO (X, X)

2 Phase locking:

Better than prior, since ensures serializability.
Has 2 phases: The growing and Shrinking Phase, Either lock are only acquired or totally released.
If a transaction is holding the lock on one row, and the other transaction demands the same, then for all pairs in the compatibility table other than (S, S), the other transaction gets blocked until the prior transaction is done with its shrinking phase. Hence, evidently, making all the schedules run in a serial manner, thereby preventing conflicts faced while concurrent running.

Non -Conflicting Transactions

The Transactions that do not access the same data items or they access the same data items in a way that does not cause interference or inconsistency.

First Set:

```
-- T1

START TRANSACTION;

INSERT INTO Customer
	(Customer_ID,Cust_name,Cust_pass,Cust_mobile,Cust_email,Cust_dob,Cust_city,Cust_country)
VALUES
	(650,"Shriya
Verma","IOK33KMC5CA","9612317102","ultrices.iaculis@google.couk","09/10/03","Delhi","India");

COMMIT;

-- T2

START TRANSACTION;

UPDATE Product
SET
	Availability = "Yes"
WHERE
	Prod_Id = 1192;
COMMIT;
```

Second Set:

```
-- T1

START TRANSACTION;

UPDATE Membership
SET Membership = 'Elite'
WHERE Customer_Id = 610;

COMMIT;

-- T2

START TRANSACTION;

SELECT Discount INTO @discount FROM Offer WHERE Deal_id = 490;
UPDATE Offer SET discount = @discount + 200 WHERE Deal_id = 490;

COMMIT;
```