Online Retail Store

Aayush Ranjan (2021003)
Aishiki Bhattacharya (2021007)

Scope:

Today, due to the extremely fast-paced lives of people, it can be exhausting and time-consuming to visit different stores at different locations. Visiting a market and choosing the right product is cumbersome and may take several hours. Also, due to the recent COVID-19 pandemic, people prefer to get products delivered to their doorstep as quickly as possible.

Therefore, to cater to the needs of the public, we provide complete management solutions to manage information on customers, sellers, delivery agents and products efficiently.

A2Z gives the admins a forum to showcase their products of various categories to customers in a hassle-free and systematic way. Customers can conveniently browse products, manage items in the cart and buy as per their preferred payment method. Delivery agents are assigned orders by the admin, which are supposed to be delivered within the stipulated time to provide an excellent experience to customers. Each of the objects is identified by a unique attribute, i.e., their primary key - Customer_ID, Admin_ID, Delivery_Agent_ID, Order_ID, Product_ID and so on.

This application will enable the admin, delivery agent and customer to interact and coordinate efficiently with each other, resulting in a pleasant experience for all.

UPDATE WALLET:

(Non-Conflict Serializable)

```
BEGIN TRANSACTION T1;
UPDATE Customer SET wallet = wallet - 100 WHERE customer_Id=1;   → W(A)
SELECT wallet from Customer where customer_Id=1;                 → R(A)
COMMIT;
BEGIN TRANSACTION T2;
UPDATE Customer SET wallet = wallet + 100 WHERE customer_Id=1;   → W(A)
SELECT wallet from Customer where customer_Id=1;                 → R(A)
COMMIT;
```

Schedule 1 (Serial Schedule)

T1	***T2***
W(A)	
R(A)	
	W(A)
	R(A)

Schedule 2 (Serial Schedule)

T1	***T2***
	W(A)
	R(A)
W(A)	
R(A)	

Schedule 3 (Non-Conflict Serializable)

T1	***T2***
W(A)	
	W(A)
R(A)	
	R(A)

As we can see above there are 2 transactions **T1** and **T2**. Each of these transactions are first updating the customer's wallet and then reading the updated value. The difference between both the transactions is that they are updating the wallet by different amounts.

The above schedule i.e schedule 3 is non-conflict serializable as in order to change the schedule into a serial schedule as in schedule 1, we have to change the order of instructions which are conflicting, R(A) and W(A). This is a RW conflict and hence they conflict. On changing the order of these 2 instructions, we will get a different output.

RETURNING AN ORDER :

(Conflict Serializable)

```
BEGIN TRANSACTION T1;

 # Update the status of the return order to "done"

  Update `return` set status='done' where order_Id=20;          → W(A)

 # Read the status of the return transaction

  Select status from `return` where order_Id=20;                → R(A)

COMMIT;


BEGIN TRANSACTION T2;

 # Update the refund amount to the total price pf the order that is being returned
```

```
  Update `return` set refund_amount=(select total_price from `order`
where order_Id=20) where order_Id=20;                          → W(B)

 # Read the refund amount of the return transaction

  Select refund_amount from `return` where order_Id=20         → R(B)

COMMIT;
```

Schedule 1 (Serial Schedule)

W(A)	
R(A)	
	W(B)
	R(B)

Schedule 2 (Serial Schedule)

	W(B)
	R(B)
W(A)	
R(A)	

Schedule 3 (Conflict Serializable)

W(A)	
	W(B)
R(A)	
	R(B)

As we can see above there are 2 transactions **T1** and **T2**. The transaction T1 is first updating the order return status to “done” and then reading the status. Transaction T2 on the other hand is first updating the return refund amount to the order value and then is reading the refund amount.

Schedule 1 and Schedule 3 are equivalent if the output obtained by executing the 1st schedule is identical to that of the 2nd schedule. The above schedule i.e schedule 3 is Conflict serializable as in order to change the schedule into a serial schedule as in schedule 1, we have to change the order of instructions W(B) and R(A) which are non conflicting instructions as they both are accessing different entities in the database. Since these are non-conflicting instructions, changing the order of these two statements will not change the output and hence the schedule 2 is conflict serializable.

ADD TO CART :

```
BEGIN TRANSACTION T1;

 # Read the current stock of Product having product_Id=1 in the inventory

  Select stock from product where product_Id=1;                    → R(A)

 # Update the stock of Product A to be 1 less than the current stock

  UPDATE product SET stock = stock - 1 WHERE product_id=1;    → W(A)

 # Read the customer's current wallet amount

  SELECT wallet FROM customers WHERE customer_id=1;                → R(B)

 # Update the customer's wallet by subtracting the price of the product from the wallet

  UPDATE customers SET wallet= wallet - (SELECT price FROM products
WHERE product_id = 1) WHERE customer_id = 1;                       → W(B)

COMMIT;


BEGIN TRANSACTION T2;

 # Read the current stock of Product having product_Id=1 in the inventory

  Select stock from product where product_Id=1;                    → R(A)

 # Update the stock of Product A to be 2 less than the current stock
```

```
  UPDATE product SET stock = stock - 2 WHERE product_id=1;    → W(A)

 # Read the customer's current wallet amount

  SELECT wallet FROM customers WHERE customer_id=1;              → R(B)

 # Update the customer's wallet by subtracting twice the price of the product from the wallet

  UPDATE customers SET wallet= wallet - (2*(SELECT price FROM products
WHERE product_id = 1)) WHERE customer_id = 1;                    → W(B)

COMMIT;
```

Schedule 1(Serial Schedule)

T1	***T2***
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

Schedule 2 (Serial Schedule)

T1	***T2***
	R(A)
	W(A)
	R(B)
	W(B)
R(A)	
W(A)	

R(B)	
W(B)	

<u>Schedule 3</u> (Non-Conflict Serializable)

<u>T1</u>	***<u>T2</u>***
R(A)	
	R(A)
	W(A)
W(A)	
R(B)	
W(B)	
	R(B)
	W(B)

As we can see above, there are two transactions which are accessing the same entities, product stock and customer wallet. Schedule 3 is non-conflict serializable as in order to change the schedule 3 to a serial schedule i.e schedule 1, we have to change the order of R(A), W(A) in T2 and W(A), R(B) and W(B) in T1. As we can see, W(A) in T1 is a conflicting instruction with R(A) and W(A) in T2 and hence it can't be converted to a serial schedule with swaps of non-conflicting instructions and hence this schedule is non-conflict serializable.

<u>Schedule 4</u> (Non-Conflict Serializable)

<u>T1</u>	***<u>T2</u>***
R(A)	
W(A)	
R(B)	

	R(A)
	W(A)
	R(B)
	W(B)
W(B)	

As we can see above, there are two transactions which are accessing the same entities, product stock and customer wallet. Schedule 4 is non-conflict serializable as in order to change the schedule 2 to a serial schedule i.e schedule 1, we have to change the order of R(A), W(A), R(B) and W(B) in T2 and W(B) in T1. As we can see, W(B) in T1 is a conflicting instruction with R(B) and W(B) in T2 and hence it can't be converted to a serial schedule with swaps of non-conflicting instructions and hence this schedule is non-conflict serializable.

Schedule 5 (Conflict Serializable)

T1	***T2***
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

Schedule 1 and Schedule 5 are equivalent if the output obtained by executing the 1st schedule is identical to that of the 5rd schedule. Thus schedule 5 is a serializable schedule. Also, since we can convert the above schedule to a serial schedule by swapping the non-conflicting queries, i.e., R(A) W(A) in T2 and R(B) W(B) in T1, it is identified as a Conflict Serializable Schedule.

Schedule 6 (Conflict Serializable)

T1	***T2***
	R(A)
	W(A)
R(A)	
W(A)	
	R(B)
	W(B)
R(B)	
W(B)	

Schedule 2 and Schedule 6 are equivalent if the output obtained by executing the 2nd schedule is identical to that of the 6th schedule. Thus schedule 6 is a serializable schedule. Also, since we can convert the above schedule to a serial schedule by swapping the non-conflicting queries, i.e., R(A) W(A) in T1 and R(B) W(B) in T2, it is identified as a Conflict Serializable Schedule.

Conflicting Transactions:

1)
```
Select * from product where stock>0;

Update product set stock=stock-1 where product_id= - ;
```

2)
```
INSERT INTO category VALUES (201, 'ChocoPie', 8);

Select * from category where NO_OF_PRODUCT = 8;
```

3)
```
Update customer query set status = "resolved" where admin_id
=70;

INSERT INTO customer query VALUES (201, 76, "shjcwgbuavcacd",
"unresolved", 70);
```

Non-conflicting Transactions:

1) Working of trigger No. 2: Update customer wallet after refund

```
Update `return` set status='done' where order_id=18;

Select refund_amount from `return` where order_id=18;
```

2) Agent Menu: View assigned orders/returns

```
Select count(order_id) from delivery_agent where agent_id=18;

Select order_id from delivery_agent where agent_id=18;
```

3) Customer Menu: Add to wallet

```
Update customer set wallet=wallet+100 where customer_id=76;

Select name from customer where customer_id=76;
```

4) Place order Trigger:

```
Insert into `order` values(150, 45, 40, 32, '2022-11-15 19:59:08',
678, 21, 'Delhi');"

Select stock from product where product_id=3;
```

Non-conflict Serializable Transactions using LOCK:

`R(A)` → `Select stars from review where customer_id=76;`

`W(A)` → `Update review set stars= 3.5 where customer_id=76;`

T1	***T2***
R(A)	
	R(A)
	W(A)
	Commit
W(A)	
Commit	

T1	***T2***
EXCLUSIVE(A)	
R(A)	
	LOCK (A)
W(A)	
Commit	
	EXCLUSIVE (A)
	R(A)
	W(A)
	Commit

By using the LOCK we are ensuring that an entity accessed by T1 is not accessed by T2 at the same time. Here we have handled the RW Anomaly and made sure that the schedule is equivalent serializable.