OnRail - Transaction Schedule

Transactions:

Transaction 1:

```
BEGIN TRANSACTION;

SELECT (MAX(partner_id)%100)+1 FROM ORDER_ITEMS;

INSERT INTO ORDER_ITEMS (customer_id, product_id, quantity_added, partner_id)
VALUES (1, 2, 3, 3);

SELECT *
FROM INVENTORY i
WHERE i.product_id = 2;

UPDATE INVENTORY i
SET i.quantity_in_stock = i.quantity_in_stock - 3
WHERE i.product_id = 2;

INSERT INTO ORDER_PAYMENT (customer_id, partner_id, payment_mode, shipping_address, order_value, order_date, status)
VALUES (1, 3, "UPI", "---", 100, 2022-04-24, 0);

DELETE FROM CART c
WHERE c.customer_id = 1;

COMMIT;
```

Transaction 2:

```
BEGIN TRANSACTION;

UPDATE INVENTORY i
SET i.quantity_in_stock = i.quantity_in_stock - 5
WHERE i.product_id = 2;

COMMIT;
```

Note:

In the schedules mentioned below, the notations are defined as follows:

- `quantity_added_A` : Quantity Added of Product A
- `price_A` : Price of Product A
- `inventory_A` : Quantity of Product A in Stock (in Inventory)

Non-Conflict Serialisable

A schedule is considered non-conflict serialisable if it cannot be transformed into a serial schedule due to transaction conflicts. In other words, transactions in a non-conflict serialisable schedule have non-conflicting

operations between them, but conflicting operations exist within the same transaction. This makes it impossible to create a serial schedule by swapping operations. The Schedule for such a scenario is given as follows:

Transaction 1	Transaction 2
R(quantity_added_A)	
R(price_A)	
`R(inventory_A)`	
cost = (quantity_added_A) * (price_A)	
`inventory_A = inventory_A - quantity_A_added`	
	`R(inventory_A)`
`W(inventory_A)`	
	`inventory_A = inventory_A- 5`
	`W(inventory_A)`
order_items = (A, customer_ID, cost)	
W(order_items)	
COMMIT	
	COMMIT

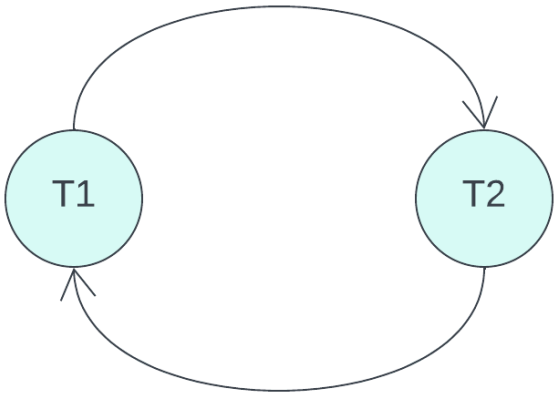
The above example is a non-conflict serialisable schedule because of the marked steps. `R(inventory_A)` and `W(inventory_A)` are conflicting and thus cannot be swapped.

Similarly, `W(inventory_A)` and `W(inventory_A)` are conflicting and thus cannot be swapped in the schedule.

This causes a **WW Conflict** because the value of the quantity of stock written by Transaction 1 is overwritten by Transaction 2 because it reads the uncommitted value.

Precedence Graph:

First, we identify all transactions in the schedule to build a precedence graph. Then, for each conflicting pair of operations between different transactions, draw an arrow from the transaction that performed the earlier operation to the transaction that performed the later operation. Following these steps, we arrive at the given precedence graph:



We see that the graph has a cycle between the two transactions. We need a direct acyclic graph for the schedule to be serialisable. Thus, the given schedule cannot be transformed into a serial schedule and is thus a **Non-Conflict Serialisable Schedule.**

Conflict Serialisable

A conflict serialisable schedule is given as follows:

Transaction 1	Transaction 2
R(quantity_A_added)	
R(price_A)	
`R(inventory_A)`	
cost = quantity_A_added*price_A	
`inventory_A = inventory_A - quantity_A_added`	
`W(inventory_A)`	
	`R(inventory_A)`
order_items = (A, customer_ID)	
W(order_items)	
	`inventory_A = inventory_A - 5`
Commit	
	`W(inventory_A)`
	Commit

The table given above causes a **WR** conflict. Reading the quantity from the inventory before Transaction 1 commits and terminates raises the possibility of a **Dirty Read** if Transaction 1 terminates with some error, causing the transaction to rollback.

Precedence Graph:

First, we identify all transactions in the schedule to build a precedence graph. Then, for each conflicting pair of operations between different transactions, draw an arrow from the transaction that performed the earlier operation to the transaction that performed the later operation. Following these steps, we arrive at the given precedence graph:

Since we get a directed acyclic graph, we can say that the schedule is **Conflict Serialisable.** The conflict can be resolved by performing non-conflicting swaps between the steps. This helps us obtain a Conflict Equivalent.

The following are non-conflicting swaps possible for two distinct data points, A and B:

- W(A) & R(B)
- W(A) & W(B)
- R(A) & W(B)
- R(A) & R(B)
- R(A) & R(A)
- R(B) & R(B)

After performing non-conflicting swaps, we obtain a **Conflict Equivalent** as follows:

Transaction 1	Transaction 2
R(quantity_A_added)	
R(price_A)	
`R(inventory_A)`	
cost = quantity_A_added*price_A	
`inventory_A = inventory_A - quantity_A_added`	
`W(inventory_A)`	
order_items = (A, customer_ID)	
W(order_items)	
Commit	
	`R(inventory_A)`
	`W(inventory_A)`
	`inventory_A = inventory_A - 5`
	Commit

Since Transaction 1 commits before Transaction 2 starts, we get a **Serial Schedule** with the conflict resolved through swapping non-conflicting steps in the originally Conflict Serialisable Schedule. We can also resolve this conflict by using **locks** to prevent race conditions and only read values when the other transaction has been committed/terminated.