Transaction – Update an Employee's Salary

Start Transaction;] → Begin (T)
Select * from Employee where ID = 1 FOR UPDATE;] → Read (A)
update Employee SET Salary = Salary + 2000] → write (A)
where ID = 1;
Commit;] → Commit (T)

T_1	T_2
Begin (T_1)	
Read (A)	
A → A + 2000	
write (A)	
Commit;	
	Begin (T_2)
	Read (B)
	B → B + 1000
	write (B)
	Commit;

Non-Conflict Serializable Schedule

T_1	T_2
Begin (T_1)	
Read (A)	
A → A + 2000	
write (A)	
	Begin (T_2)
	Read (A)
	A → A + 1000
	write (A)
commit (T_1)	
	Commit (T_2)

Conflict Serializable Schedule
{W-R Conflict}

In Case of Non-Conflict Serializable schedule – Transaction T_1 is increasing the employee's Salary by 2000, and after commiting the transaction (T_1) into database, transaction T_2 is running and updating/Increasing the employee's Salary by increasing it by 1000. Here both transactions are working on two different employee's (Employee ID) hence both are independent of each other. Even if any one of the transactions failed or aborted, then it will not affect the other transactions. Therefore, no need to do anything further as it is already a valid schedule.

In Case of Conflict Serializable Schedule – Both transactions T_1 and T_2 are trying to update the Salary of Same Customer Concurrently, and if any one of the transactio (mostly T_1) failed/aborted then there will be Some kind of dirty reads as there are R-W, W-R Conflicts which leads to the inconsistency of the Database.

* To resolve these conflictions, we can use locking methods

By using Two phase Locking Protocol (2PL) [Strict-2PL].

In S2PL, a transaction acquires all locks as it needs before proceeding with any updates and hold all locks until it completes and commits.

T_1	T_2
Begin (T_1)	
X(A)	
Read (A)	
A → A+2000	
write (A)	
	Begin (T_2)
	X(A)
	Read (A)
	A → A + 1000
	write (A)
Commit (T_1)	
	Commit (T_2)
U(A)	
	U(A)

After applying exclusive locks on ~~the~~ (A), the transaction is now isolated and other transactions can't run simultaneously until the lock is released from (A). This will make T_2 to wait until T_1 commits/completes.

X(A) → Exclusive Lock
U(A) → Lock released

By using the S2PL method, we have converted a conflicting serializable schedule to a non-conflicting serializable schedule

Transaction - Update customer's membership from "Normal" to "Prime" or "Elite".

```
Begin Transaction;                          ] → starting a transaction
Select wallet, membership From customer     ] → Read (A)
Where ID = 10 For update;                   ] → X-lock   Read (B)
Update Customer SET membership = 'Elite',   ] → write (A)
wallet = wallet - 300 where ID = 10;                     write (B)
Commit;                                     ] → commiting the changes.
```

T_1	T_2
Begin (T_1)	
Read (A)	
Read (B)	
	Begin (T_2)
	Read (A)
	Read (B)
write (A)	
write (B)	
	write (A)
	write (B)
commit (T_1)	
	commit (T_2)

Conflict serializable schedule

{w-w, R-w Conflicts}

T_1	T_2
Begin (T_1)	
Read (A)	
Read (B)	
write (A)	
write (B)	
commit (T_1)	
	Begin (T_2)
	Read (A)
	Read (B)
	write (A)
	write (B)
	commit (T_2)

Non-Conflict Serializable schedule

Here in Non-Conflict schedule, T_1 is updating the membership of customer from 'Normal' to 'Prime' and reducing the wallet amount by '200'. After the commiting of T_1, T_2 Transaction is starting hence there will be no conflict and the data consistency will be maintained.
Therefore no need to do anything further as it is already a valid schedule.
Even if any one of the transactions failed or aborted, then it will not affect the other transaction.

In Case of Conflict Serializable Schedule – Both Transactions

T_1 and T_2 are trying to change the membership status of Customer (Same Customer) Concurrently and if any one of the transactions (mostly T_1) failed / aborted then there will be some kind of dirty reads as there are R-W, W-R conflicts which leads to the inconsistency of the Database.

To Solve these conflictions, we can use locking methods

By using Two phase Locking Protocol (2PL) [Strict-2PL]

In S 2PL, a transaction acquires all locks it needs before proceeding with any updates and holds all locks until it completes and commits.

T_1	T_2
Begin(T_1)	
X(A)	
X(B)	
R(A)	
R(B)	
	Begin(T_1)
	X(A)
	X(B)
	R(A)
	R(B)
W(A)	
W(B)	
Commit(T_1)	
	W(A)
	W(B)
	Commit(T_2)
U(A)	
U(B)	
	U(A)
	U(B)

After applying exclusive locks on A and B, the transaction is now isolated and other transactions can't run simultaneously until the locks are released from A and B. This will make T_2 to wait until T_1 commits / completes.

X(A) → Exclusive lock

U(A) → Unlocking / lock released