1) Properties of transactions :-

In DBMS, a transaction must satisfy 4 key properties known as ACID properties :-

i) Atomicity (All or nothing) :-

A transaction is treated as single unit,

* Either all operations are completed or none are applied.
* If any part fails, the entire transaction rolled back.

Ex :-

Bank transfer (₹1000)

- Debit from A ✓
- Credit to B ✗ → Entire transaction is undone.

ii) Consistency (Valid state) :-

A transaction must take database from one valid state to another valid state.

* It should follow all rules, constraints & integrity conditions.

Ex :- If total balance before transaction = ₹10000

After transaction → must still be ₹ 10000 (no loss)

iii) Isolation (No interference) :-

Multiple transactions can run simultaneously, but they should not affect each other.

Ex :- Two users booking same seat → System ensures only one succeeds.

iv) Durability (Permanent Changes) :-

Once a transaction is successfully committed, changes are permenant.

- Even if system crashes, data is not lost.

Ex :-

After money transfer confirmation, data remains saved even after power failure.

2) Concurrency Control Importance:-

Concurrency control ensures that when multiple transactions execute simultaneously, the database remains:

- Consistent
- Accurate.
- Isolated
- Free from conflicts.

Without concurrency control, simultaneous operations can lead to wrong or inconsistent data.

Problems in concurrency processing:-

i) The Lost Update Problem (WW conflict):-

The lost update problem occurs when two transactions update the same data item, and one update is overwritten by the other resulting in loss of data.

ii) The Temporary update (or dirty read) Problem (WR conflict):-

The Dirty Read problem occurs when one transaction reads data that has been updated but not yet committed by another transaction. If that transaction later rollsback, the first transaction has used invalid (dirty) data.

iii) The incorrect summary problem (RW conflict):-

Incorrect summary problem occurs when a transaction calculates aggregate values while another transaction updates some of the data, resulting in inconsistent results.

3) Serial and Non-serial Schedule with examples:-

Serializable Schedule:-

The serializability of a schedule is used to find non-serial schedules that allow transaction to execute concurrently without interfering with one another. A non-serial schedule will be serializable if its result is equal to result of its transactions executed serially.

Non-Serializable Schedule:-

A non-serial schedule which is not serializable is called as a non-serializable schedule. A non-serializable schedule is not guaranteed to produce the same effect as produced by some serial schedule on any consistent database.

T_1	T_2
Read(A)	
Write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

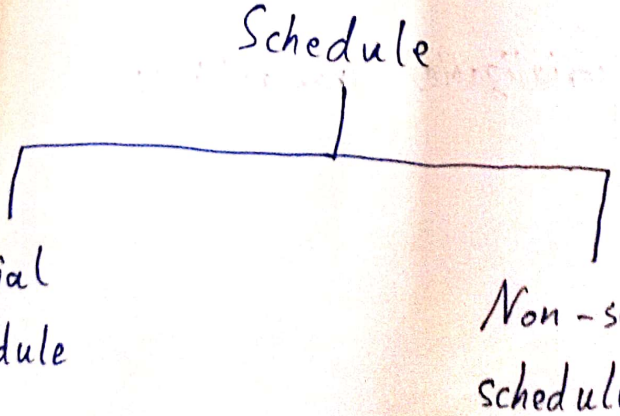
Schedule S1

Non-serial schedule

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule, S2

Serial Schedule



4) Serializability with examples :-

Serializability is a concept in non concurrency control that ensures a schedule produces the same result as some serial execution.

- Even if transactions run concurrently, the final result should be correct as if they ran one by one.
- There are 2 types of serializability,
  - → Conflict Serializability.
  - → View Serializability.
- It guarantees that even multi-user environments, the database remains consistent and reliable.

Ex :-

T_1	T_2	A = 50 B = 50
read(A)		150
A := A+100		
write(A)		
	read(A)	
	A := A*2	300
	write(A)	
read(B)		
B := B+100		150
write(B)		
	read(B)	
	B := B*2	300
	write(B)	

Ex :-

Serializable (conflict serializable) Transactions:

T_1 : R(A), W(A)

T_2 : R(A), W(A)

Schedule:

$R_1(A) \longrightarrow W_1(A) \longrightarrow R_2(A) \longrightarrow W_2(A)$

- T_1 completes before T_2 starts effectively.
- Conflicts;

  $W_1(A) \longrightarrow R_2(A)$,

  $W_1(A) \longrightarrow W_2(A)$

- So T_1, T_2 $(T_1 \longrightarrow T_2)$

∴ Equivalent Serial order - $T_1 \longrightarrow T_2$

Ex:-

Serializable with different data items.

Transactions:

T_1 : $R(A)$, $W(A)$

T_2 : $R(B)$, $W(B)$

Schedule:

$R_1(A) \longrightarrow R_2(B) \longrightarrow W_1(A) \longrightarrow W_2(B)$

- No conflicts
- can be rearranged in any order.

∴ Equivalent Serial order: $T_1 \rightarrow T_2$ or $T_2 \rightarrow T_1$

∴ Serializable.

5) **Conflict Serializability and Equivalent with examples :-**

It is a type of serializability where a schedule can be transformed into serial schedule by swapping non-conflicting operations.

Two operations are said to be in conflict if:

* They belong to different transactions.
* They operate on same data item.
* Atleast one operation is a write.

Ex:-

List all conflicting operations and determine dependency between transactions.

$R_2(X)$, $W_3(X)$ $(T_2 \rightarrow T_3)$

$R_2(X)$, $W_1(X)$ $(T_2 \rightarrow T_1)$

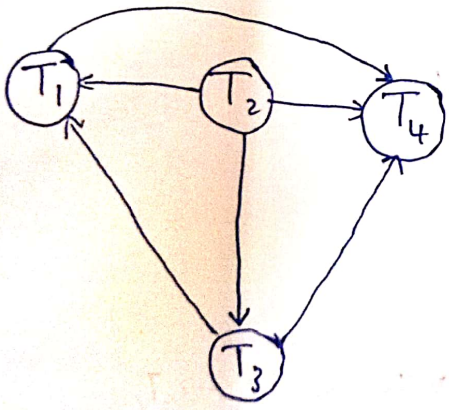
$W_3(X)$, $W_1(X)$ $(T_3 \rightarrow T_1)$

$W_3(X)$, $R_4(X)$ $(T_3 \rightarrow T_4)$

$W_1(X)$, $R_4(X)$ $(T_1 \rightarrow T_4)$

$W_2(Y)$, $R_4(Y)$ $(T_2 \rightarrow T_4)$

T_1	T_2	T_3	T_4
	R(X)		
		W(X) commit	
W(X) commit			
	W(X) R(Z) commit		
			R(X) R(Y) commit



clearly there exists no cycle in precedence graph

∴ given schedule s is conflict serializable.

Conflict Equivalent :-

Two schedules are said to be conflict equivalent when one can be transformed to another by swapping non-conflicting operations.

Ex :-

T_1	T_2
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)

Schedule s

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	

Schedule s'

R(A) R(A)
R(A) W(A)
W(A) R(A)
W(A) W(A)
} conflict pairs

R(B) R(A)
R(B) W(A)
W(B) R(A)
W(A) W(B)
} non-conflict pairs

Step 1 : check adjacent non conflict pairs as per rules.

Step 2 : swap it

consider from test:

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	

swap →

T_1	T_2
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)

swap →

T_1	T_2
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)

Hence s = s'

it is conflict serializable.

6) View Serializable and equivalent with examples:

View Serializable :-

A schedule is correct if it gives the same results as some serial (one-by-one) execution of transactions.

three conditions :-

1) Check initial read
   → if transaction reads original value, it should remain same.
2) Read from (dependency)
   → if T_2 reads value written by T_1, it must remain same.
3) Final write
   → Last transaction writing a data item ~~written by T_1~~, must ~~remain~~ be same.

i) Check for conflict serializability.
ii) Check for view serializability.

T_1	T_2	T_3
R(A)		
	R(A)	
		R(B)
W(A)		
	R(C)	
	R(B)	
	W(B)	
		W(C)

	A	B	C
Initial read	T_1, T_2	T_3, T_2	T_2
Update	T_1	T_2	T_3
Final update	T_1	T_2	T_3

check item by item,

A: $T_2 \rightarrow T_1$
B: $T_3 \rightarrow T_2$
C: ~~T_1~~ $T_2 \rightarrow T_3$

$(A, B)_p \rightarrow T_3 \rightarrow T_2 \rightarrow T_1$
$(BC)_p \rightarrow$ Conflicting
∴ Non-serializable.

So no order can be divided strictly are basis of VS.

We can simply check it from V.S :-

T_1	T_2	T_3
R(A)		
	R(A)	
		R(B)
W(A)		
	R(C)	
	R(B)	
	W(B)	
W(C)		

	A	B	C
Initial read	T_1, T_2	T_3, T_2	T_2
Update	T_1	T_2	T_1
Final update	T_1	T_2	T_1

A : $T_2 \rightarrow T_1$

B : $T_3 \rightarrow T_2$

C : $T_2 \rightarrow T_1$

$\{A, B\} \rightarrow T_3 \rightarrow T_2 \rightarrow T_1$

$\{B C\} \rightarrow T_3 \rightarrow T_2 \rightarrow T_1$

$\{A C\} \rightarrow T_2 \rightarrow T_1$

$\boxed{T_3 ; T_2 ; T_1}$

∴ View Serializable

View Equivalent Schedule :-

S_1 :

T_1	T_2	T_3
R(A)		
	W(A)	
W(A)		
		W(A)

Initial read : by T_1 on A

Final write : by T_3 on A

WR conflict : not present

S_2 :

T_1	T_2	T_3
R(A)		
W(A)		
	W(A)	
		W(A)

Initial read : by T_1 on A

Final write : by T_3 on A

WR conflict : not present

S_0, S_1 and S_2 are view equivalent schedules.

7) Recoverable Schedule :-

A schedule is recoverable if:

* Transaction T_j commits only after T_i commits.
* if T_j has read data written by T_i

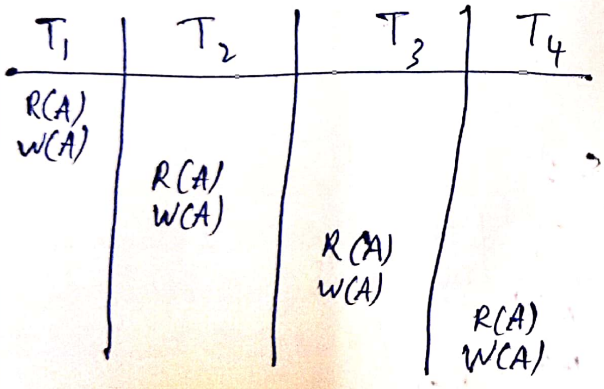
$$commit(T_i) \longrightarrow commit(T_j)$$

A recoverable schedule in DBMS ensures transactions commits only after transaction if it depends on have commited.

three types of recoverable schedules:

i) Cascading Schedule:-

It is a schedule which a transaction reads uncommited data from another transaction. If first transaction fails, then all dependent transactions must rollback.



* Failure of transactions T_1, T_2, T_3 causes the transaction T_2, T_3, T_4 to roll respectively.

∴ Such rollback is called Cascading RollBack.

ii) Cascadeless Schedule :-

It is a schedule in which a transaction reads data only after the transaction that wrote it has committed.

T_1	T_2	T_3
R(A)		
W(A)		
commit		
	R(A)	
	W(A)	
	commit	
		R(A)
		W(A)
		commit

Cascadeless Schedule allows only commited read operations

It allows uncommitted write operations

T_1	T_2
R(A)	
W(A)	
	W(A)
	//uncommitted
commit	

iii) Strict Schedule;-

A schedule is strict if no transaction can read or write a data item until transaction that last wrote it commits or aborts.

T_1	T_2
W(A)	
commit/ rollback	
	R(A)/W(A)