Module - 5

Transaction Management

→ It is a program unit whose execution may change the content of database.
like read, write, update or delete

DB before transaction ⟶ DB after transaction { consistent state }

A	B	
(300)	(400)	100 from A → B
Read (A)	Read (B)	Befor: 700
A = A − 100	B = B + 100	After: 700
[200] Write (A)	Write (B) [500]	

Transaction satisfy the ACID properties.

ACID properties

① Atomicity:

→ It states that the transaction must be atomic
i.e. all will be executed or none.
→ If any part of transaction fails, everything will be rollback

Abort: Changes to DB are not visible

Commit: Change are visible.

Ex:

T_1 & T_2 tran. 100%. From A to B

A : 500 B : 200

T_1	T_2
Read(A)	Read(B)
A = A − 100	B = B + 100
write(A)	write(B)
Commit	Commit

③ Consistency:

Transaction should maintain data integrity, regardless of whether it succeeds or fails.

④ Isolation:

Transaction must be isolated i.e. a transaction cannot be used by the second transaction until the first one is completed.

→ Must be independent i.e. concurrent transaction donot interfere with each other.

⑤ Durability:

→ Once a transaction is commited, its changes are permanent, even in the event of system failure.

✳ Operations of Transaction:

(i) Read(x)

→ Used to read the value of x
→ Store it in a buffer in the main memory for further processing

(ii) Write(x)

→ Used to modify the database
→ Change the state of the database as part of the transaction

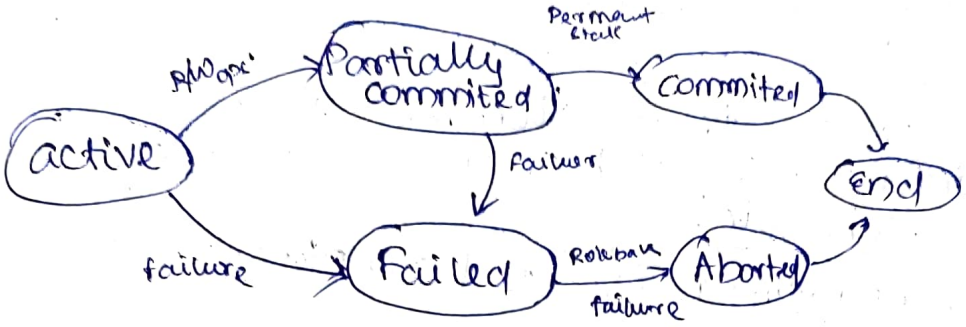
(iii) Commit

→ Used to maintain the data integrity in the database.
→ Makes all changes permanent in the database

(iv) Rollback:

→ Aborts the transaction and revert changes made to the database.
→ Ensures atomicity.

⊛ States of Transaction



(i) Active state :
→ The active state is the first state of every transaction.
→ In this state, transaction is being executed.
→ Stores in buffer (local memory)

(ii) Partial Commited :
→ If all the query execution satisfies the ACID properties then it is commited
→ If any system failures occurs it is terminated.

(iii) Commit State :
→ A transaction is said to be in a committed state if it executes all the operations successfully.

(iv) Failed state :
→ If any of the checks made by the database recovery system fails, then the transaction is said to be in failed state.

(v) Aborted State
→ In the aborted state it clears all the buffer (local memory) and make the database in its previous consistent state
Kills the transaction
Restart the transaction

→ And finally it comes to end.

Deffered Update
→ Update are applied only at commit time
→ Changes not visible
→ only redo is req. for recovery
→ Slower
→ Concurrency control are easier to implement

Immediate update
→ Updates are applied during transaction execution
→ Change may visible
→ Both redo & undo req.
→ faster
→ Requires stronger concurrency mechanism.

Transaction Schedule

→ By default, commit is executed on last step
→ Abort instruction on the last statement

→ Chronological order of execution of multiple transaction.
→ Series of one transaction to another.
→ To maintain the database consistency & integrity

① Serial Schedule:

One transaction is executed completely before starting another.

T_1	T_2
R(x)	
W(x)	
C	
	R(y)
	W(y)
	C

Adv:
* consistency

② Non-Serial Schedule:

→ concurrent schedule
→ parallel schedule

→ Operations from different transaction are interleaved.
→ Improve performace but risks inconsistency

T_1	T_2
R(x)	
	R(x)
W(x)	
	W(y)
C	
	C

③ Serializable Schedule:

→ A serializable schedule is a non-serial schedule that ensures the same result as a serial schedule.
→ The serializability property ensures the correctness of a non-serial schedule by verifying that its outcome is equivalent to some serial schedule.
→ It improves both resource utilization & CPU.

"Serial execution of a set of transactions preserves database consistency".
So a schedule is said to be serializable if its equivalent to serial schedule.

Testing of serializability :-

→ Serialization graph is used for testing

↓

Precedence graph

$G = (V, E)$

G → graph, V → set of vertices, E → set of edges

E contains all edges $T_i \rightarrow T_j$:

① Create a node $T_i \rightarrow T_j$ if T_i executes $W(Q)$ before T_j executes $R(Q)$

$W \rightarrow R$

② Create node $T_i \rightarrow T_j$ if T_i executes $R(Q)$ before T_j executes $W(Q)$

$R \rightarrow W$

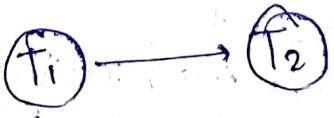
③ Create node $T_i \rightarrow T_j$ if T_i executes $W(Q)$ before T_j execute $W(Q)$

$W \rightarrow W$

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

Not serializable

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



Serializable

⊙ Concurrency :

→ It is a procedure in DBMS which help us in the management of two simultaneous process to execute without conflicts between each other.

*Types of problems in concurrency.

① Dirty read.

T_1	T_2
R(A)	
W(A)	
↓ failure	R(A)
	W(A)
	C

// Dirty read

ⓑ Incorrect Sum:

- Started when b/w two transaction when one start performe aggregate func^n.

ⓒ Lost Update:

Occurs when two transactions overwrite each other's updates without proper synchronization

T_1	T_2	
		A=10
R(A)		
A=A+5		
W(A)		
	R(A)	
	A=A-5	
	W(A)	
//A=15	C	//A=5
C		

ⓓ Unrepeatable reads:

→ Occurs when a transaction reads the same data item twice, but gets diff. results due to another transactions update.

ex:

	T_1	T_2	
(//10)	R(A)		
		A=A+10	
		W(A)	// 20
(//20)	R(A)		

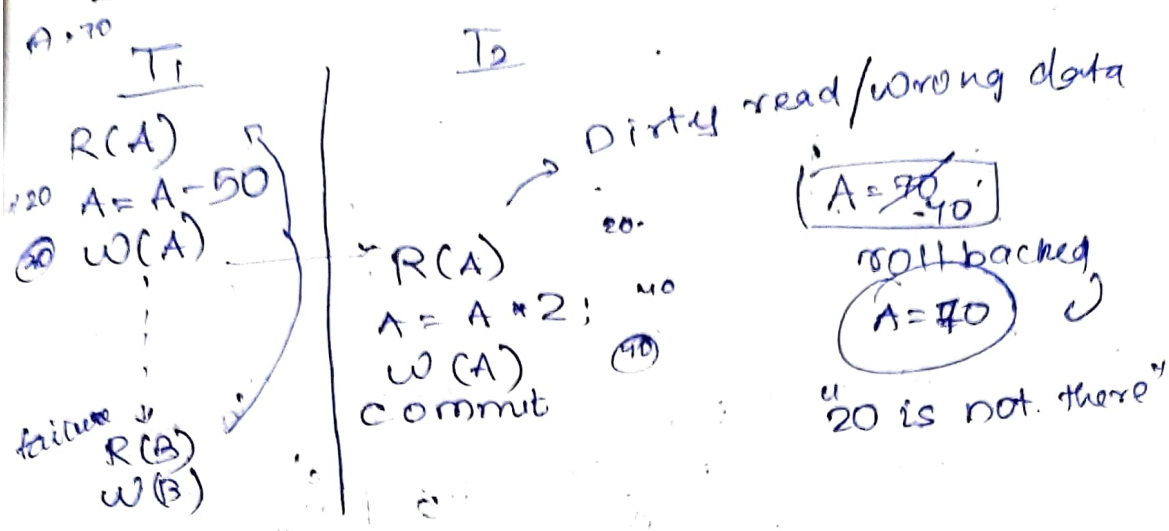
causing inconsistency.

ⓔ Phantom read

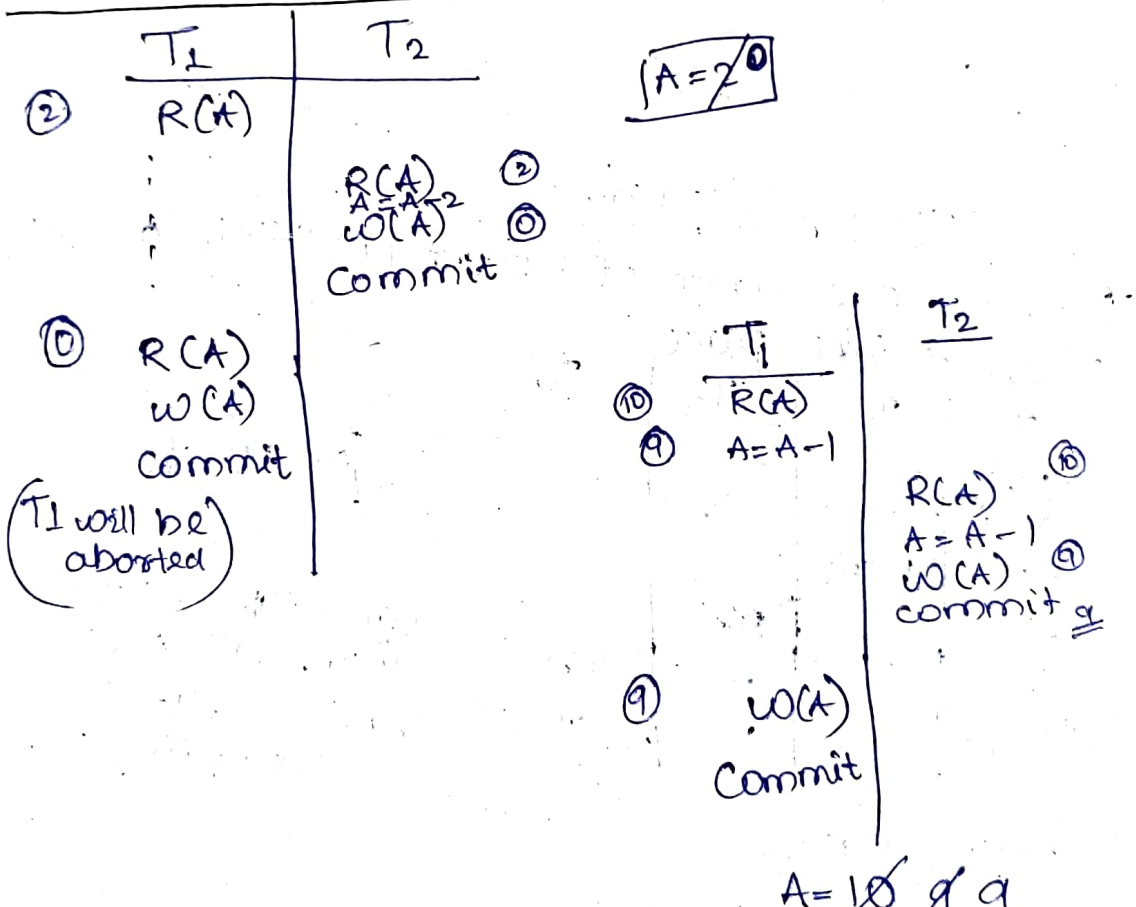
→ Occurs when a transaction retrieves different sets of rows in repeated queries due to another transaction's insertion or deletion

	T_1	T_2	
//20	R(x)		
		R(x)	// 10
//0	Delete(x)		
		R(x)	//0

Write Read Conflict :-



Read Write Conflict :-



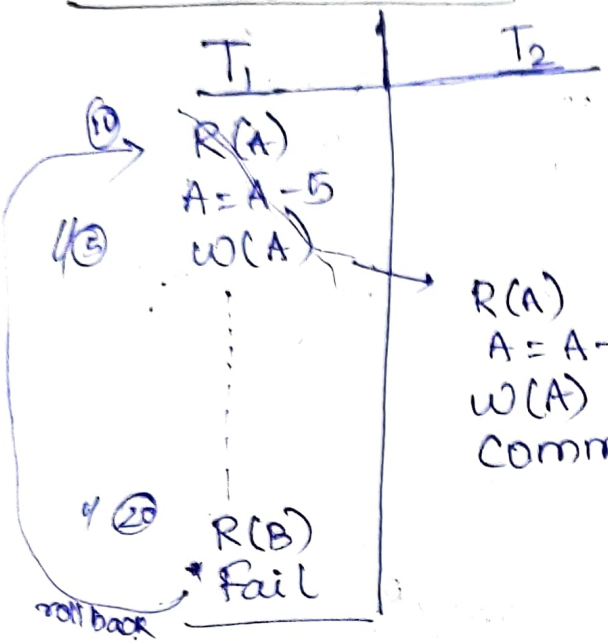
Conflict Pairs :-

Two are reduce. but only 1 is reduced.

R(A)	R(A)	← Non conflict
R(A)	W(A)	} Conflict
W(A)	R(A)	
W(A)	W(A)	
R(B)	R(A)	} Non-conflict
W(B)	R(A)	
R(B)	W(B)	
W(A)	W(B)	

Recoverability

Non-Recoverable:



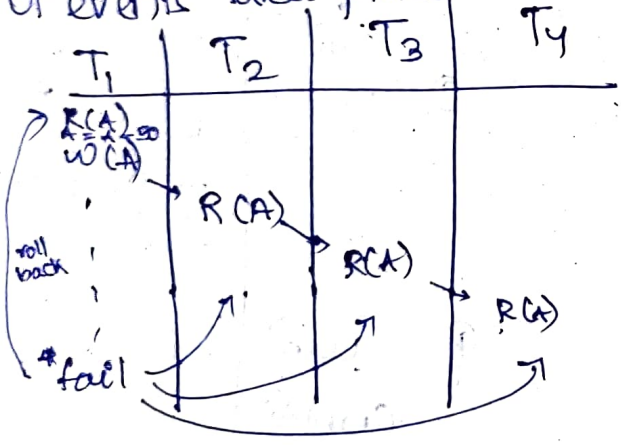
A=10 → 3 → 10

B=20

Irrecorable schedule (To change hua tha usko recover nhi kiya jaskta)

Cascading Schedule

- Due to occurence of 1 event multiple occuring of events takes place.



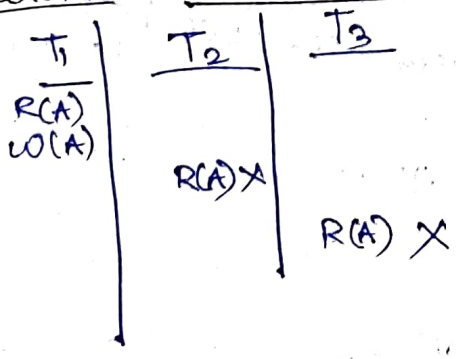
A= 100 → 50 → 100

We will tell to abort all these

This is called cascading

DisAdv: Failure

Cascadless Schedule

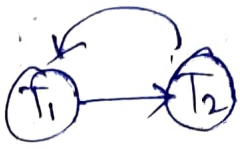


Not allow for read until T_1 is committed.

But can write so it is a problem

Serializability:

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



Parrallel (loops)

↓clone convert

serializable

Serializability

- Conflict
- View

For 3 transaction serializable are:

$T_1 \to T_2 \to T_3$

$T_1 \to T_3 \to T_2$

$T_2 \to T_3 \to T_1$

$T_2 \to T_1 \to T_3$

$T_3 \to T_2 \to T_1$

$T_3 \to T_1 \to T_2$

No. of ways to make it

$3! = 6$

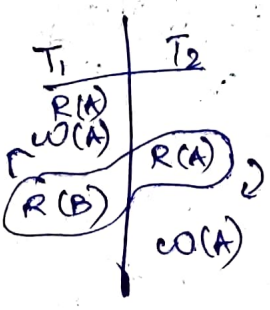
⊙ Conflict Equivalent

If a schedule S can be transformed into S' by a series of swapping of non-conflict instruction

Conflict: R(A) W(A)
W(A) R(A)
W(A) W(A)

S

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



S'

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

Adjacent non-conflict pair
Swap their pos?

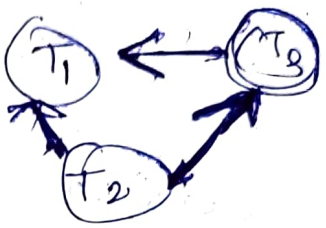
$S \to S'$ serializable

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

→ Conflict pairs
* Cannot swap

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

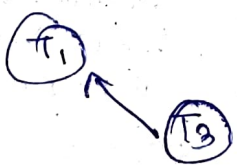
check conflict pairs on other two transactions



W(Y) → {R(Y), W(Y)}

No loop exist
∴ Conflict Serializable

Find In degree = 0.
↳ T_2 (Remove it)



Any

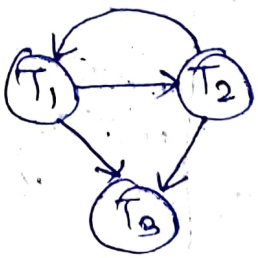
$T_2 \rightarrow T_3 \rightarrow T_1$

Indegree = 0
↳ T_3 (Remove)

⊛ View Serializability

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

(S)



Not a conflict serializability as cycle occurs.

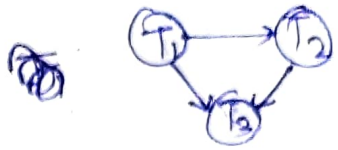
To overcome it we use view ser.

S'

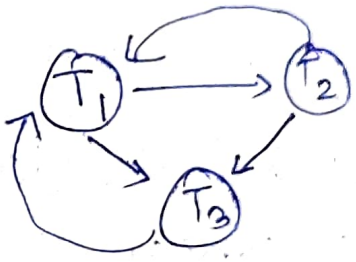
T_1	T_2	T_3
R(A)		
W(A)		
	W(A)	
		W(A)

$S \equiv S'$

$T_1 \rightarrow T_2 \rightarrow T_3$



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



Non conflict serializable

Concurrent Execution

If two transaction are running concurrently, the OS may execute T_1 for a little while then perform a context switch, execute T_2 for some time and then switch back to T_1 and so on.

→ Thus the CPU time is shared among all the transactions

Advantage :-

(i) Multiple transaction are allowed to run concurrently in the system.
(ii) Increase processor & disk utilization
(iii) Reduce avg response time
(iv) Concurrency control scheme

*Concurrency Control

→ Procedure in DBMS for managing simultaneous operations of transactions which execute without conflicting with each other.

→ Used to address conflict pairs.

Problems

- Dirty read
- Incorrect summary problem
- Lost Update
- Unrepeatable reads
- Phantom reads

⊙ Concurrency control schemes :-

① Lock Based Protocols

→ Mechanism to control concurrent access to a data item by various transaction is called Lock.

→ Data items can be locked in two modes :

- Shared (S) mode : ⇒ It can only be read
- Exclusive (X) mode : ⇒ It can be both reads write as well

→ A transaction can proceed with its execution only after the request is granted.

Compatability function

Grant ↓ \ Request →	S	X
S	Yes	No
X	No	No

S - S
R - R ✓
R - W ✗
W - R ✗
R/W - W/R ✗

T_1	T_2
S(A)	X(A)
R(A)	R(A)
U(A)	W(A)
	U(A)

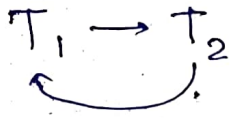
→ We can observe that if any transaction holds exclusive on the item then no other transaction can hold any lock on the same item until the first transaction doesnot release its X-lock on the item.

T_1	T_2
lock-X(B)	lock-S(A)
read(B)	read(A)
B = B - 50	unlock(A)
write(B)	lock-S(B)
unlock(B)	read(B)
	unlock(B)

⊛ Problems in Lock-Based protocols :

1) May not sufficient to produce only serializable schedule

T_1	T_2
X(A)	
R(A)	
W(A)	
U(A)	
	S(A)
	R(A)
	U(A)
X(B)	
R(B)	
W(B)	
U(B)	



Not serializable

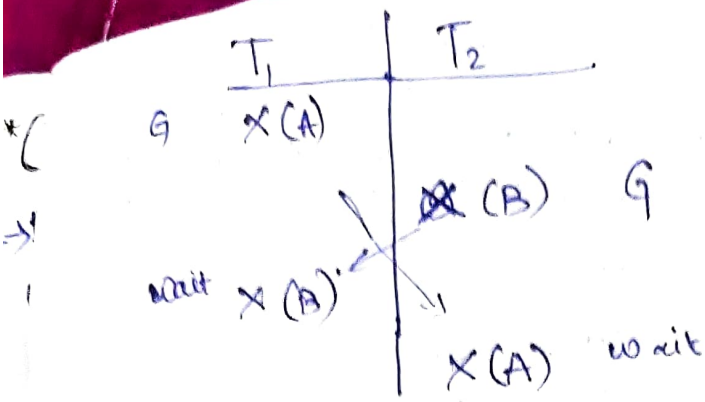
2) May not free from Irrecoverability

T_1	T_2
X(A)	
R(A)	
W(A)	
U(A)	
!	
!	S(A)
!	R(A)
failure	commit

} cannot be rollback

3) May not free from deadlock

"Deadlock" occurs when two or more transactions are waiting for each other to release locks on resources, none of them proceed as a result of causing Indefinite halt.



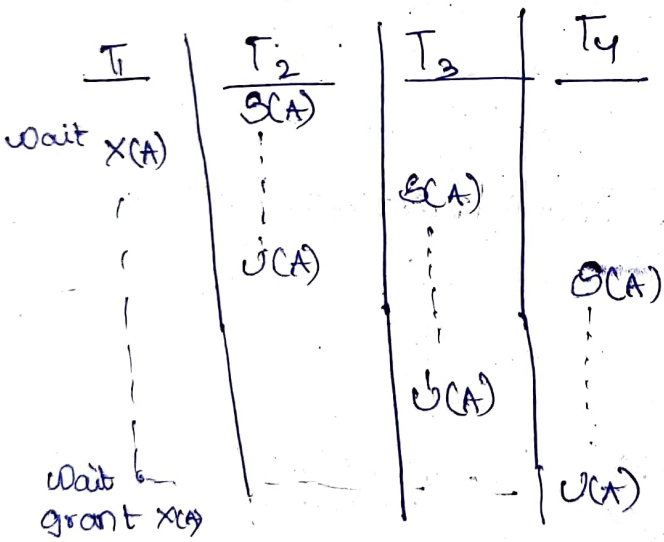
T_1 has resource A and wait for request Res. of

T_2 has resource B & wait for requ Res of A

Neither T_1 & T_2 can proceed; as both are waiting for other to release a resource. This create a circular wait, leading to deadlock

(ii) May not free from Starvation

- Starvation: Occurs when a transaction waits till higher-priority transaction keep acquiring the resources it needs



⊛ Two phase locking protocols:-

Works in two phases:-

(i) Growing Phase: Tran. can obtain locks but cannot release locks.

(ii) Shrinking Phase: Transaction can release locks but cannot obtain locks.

Ensures Serializability.

Lock Points :-

It is the exact moment in a transaction where acquiring of locks stops and releasing of acquired lock started.

Based on the order of execution

T_1	T_2
lock (A)	
	lock (A)
lock (B)	
unlock (A)	lock (B) → lock point
unlock (B)	Unlock (A)
	unlock (B)

Lock point

T_1	T_2	
X(A)		Growing phase
R(A)		
LO(A)		
	S(A)	
	R(A)	Growing phase
S(B)		
R(B)		
	X(B)	
U(A)	W(B)	Shrinking phase
U(B)		
	U(A)	Shrinking phase
	U(B)	

In T_1 X(A) is granted then S(B) is granted] OK

In T_2 S(A) is wait till the X(A) completed then it will start.

So it ensures serializability

$T_1 \rightarrow T_2$

#Drawbacks of 2PL

1) May not free from irrecoverability
   ↳ Not surity of commit.
2) Doesnot ensure free from deadlock
3) Not free from Starvation
4) Not free from Cascading Rollback.

Cascading Rollback :- When a single transaction's failure leads to the rollback of multiple dependent transactions.

T_1	T_2	T_3
X(A)		
R(A)		
W(A)		
U(A)		
:	S(A)	
:	R(A)	
failure		S(A)
		R(A)

(vi) W(B) by T_2

→ R_TS(B) > T_2
30 > 20 (True)
roll back T_2

→ (vii) W(A) by T_3

R_TS(A) > T_3
10 > 30 (False)

W_TS(A) > T_3
10 > 30 (false)

Set W_TS(A) = TS(T_3) = 30

Database Recovery :-

Failure Classification :-

(i) Transaction failure:
- Logical error (Bad input, overflow, resource limit)
- System error (deadlock, starvation)

(ii) System Crash
(iii) Disk failure

#Recovery Algorithm:-
→ Techniques to ensure database consistency and transaction atomicity & durability despite failures.

Log Based Recovery:-

→ Most widely used structure for recording database modification. is the log.
→ log is a sequence of log records, recording all the update activities in the database.

It contain info about:
(i) Transaction start <T_1, start>
(ii) Write Operations <T_1, x, Oldvalue, NewValue>
(iii) Commit <T_1, commit>
(iv) Abort <T_1, Abort>

Deffered	Immediate
→ Records all modifications to log, but defers all the write op. until the transaction commits	→ Updates are applied to the database as soon as a write operation occurs
→ Only redo logs are req.	→ Both redo & Undo are req.
→ Simpler recovery process	→ Complex one
→ Perf. may be slower	→ Faster
→ Safer against inconsistency	→ More prone to inconsistencies

Undo Database recovery :-

Restores the value of all data items that are updated to their old values.

$\langle T_1 , A, OldValue, NewValue \rangle$

T_1 New value = 200 fails before commit

Undo restores Old value = 100 to A

→ Works backward from the point of failure

Redo Database recovery :-

Set the value of all data item that are updated to new values

$\langle T_1 , A , OldVal , NewVal \rangle$

T_1 NewVal = 400 , commits, but change wasn't flushed to disk due to crash

Redo writes New Value = 400 to B

→ Redo works forward from the point of failure

Strict 2PL :-

basic 2PL all exclusive locks are holds until commit/Abort

Rigorous 2PL :

Basic 2PL with all exclusive & shared locks are hold until commit/Abort

T_1	T_2
X(A)	
R(A)	
W(A)	S(A)
:	R(A)
C	
U(A)	

cascading roll back removed

cascadless

strict recoverable

Conservative 2PL

Before starting the transaction takes all the locks

resolve deadlock problem.