ACID [Consistency of DB] To ensure

$DB_{11} \xrightarrow{T\text{-}SQL} DB_{12}$ [Both DB should be Consistance]

1] Atomicity

- No adha adhura work → Full Transaction or no transaction.
- No Partial execution of instruction in Transaction
- If there is a failure → rollback → undo all
- Full/every instruction should be executed.

* Transaction management Components handles Atomicity

2] Isolation

Logical isolation is implemented. No Practical isolation. → It will be slower

↳ The result of a Transaction should not be affected when that same Transaction is Running parallely with other Transactions

OR

Transactions which are Running [in Parallel should not affect the Result] of other Transactions running along Them.

* Concurrency control Component handles Isolation

3] Durability → changes [not Temp or changeable] [should be reflected in the DB]

→ changes made by a fully executed Transaction should be reflected/persist in the DB. [It should not be Temp and can not be changed in future]

* Recovery Management Component handles Durability.

4] Consistency [depends on AID]

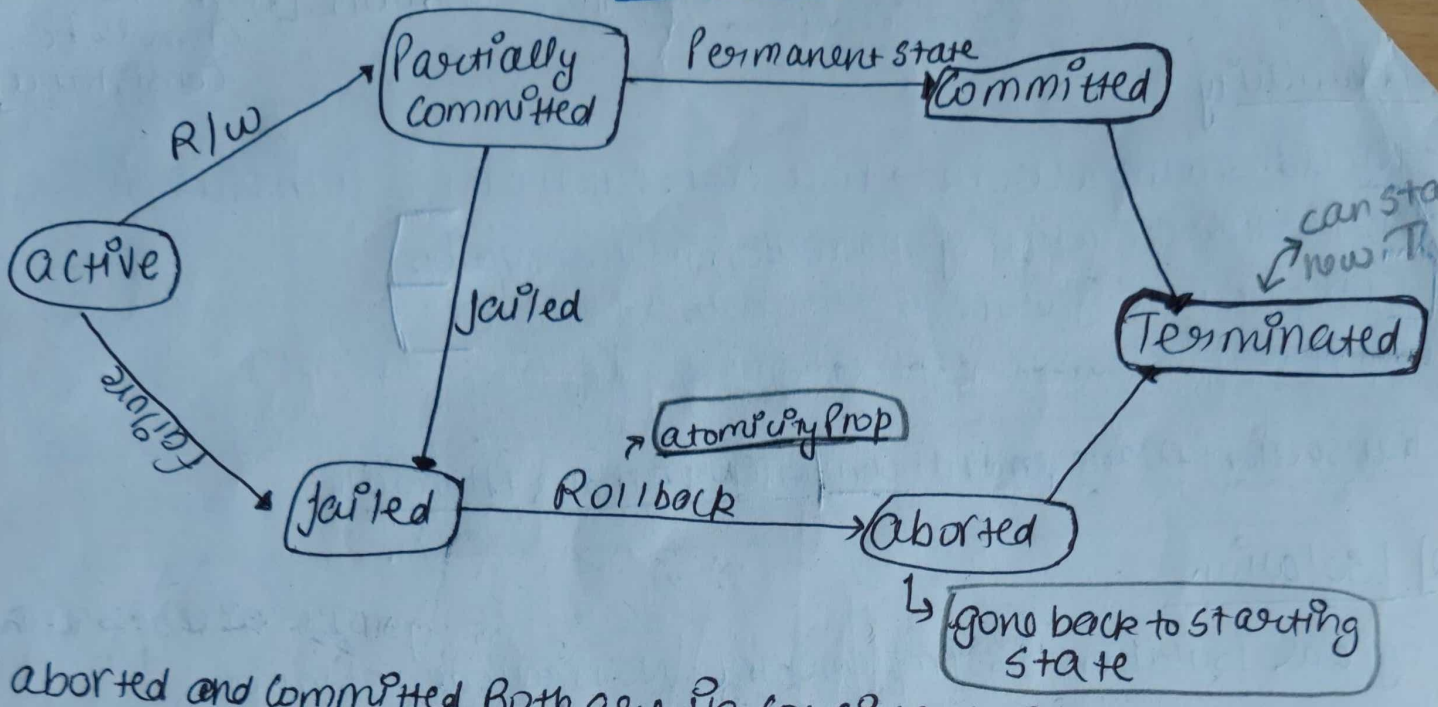
→ $DB_{11} \xrightarrow{T} DB_{12}$ [If DB_{11} → consistant then DB_{12} should be consistant]

- If AID works then consistency is auto apply/hold.

* Transaction are applied on a copy of DB → [Partially Commit]

* If every thing works fine then changes in copy → applied to org DB → [Commit]

Transistance State



* aborted and committed Both are in consistant state.

Advantages of Concurrency

1] waiting Time ↓ 2] Response Time ↓ 3] Resource utilization ↑

4] Efficiency ↑

* when in concurrent/multiple Transaction, each Transaction tries to maintain its ACID Prop then Inconsistency may be introduced in system so Instead of removing concurrency we have methods through which we can check for this Inconsistency.

* No 2 Transaction occurs at same Time, The Run in Parallel in CONTEXT SWITCHING manner

Before commit → local Buffer [copy]	After Commit → from DataBase

+ DIRTY READ PROBLEM

→ Reading Uncommitted Value

→ when a Transaction Reads a value from local Buffer / Uncommitted Transaction

T_1	T_2
→ A=A+k R(A) W(A)	R(A) → Reading Uncommitted value
	C → committing it.

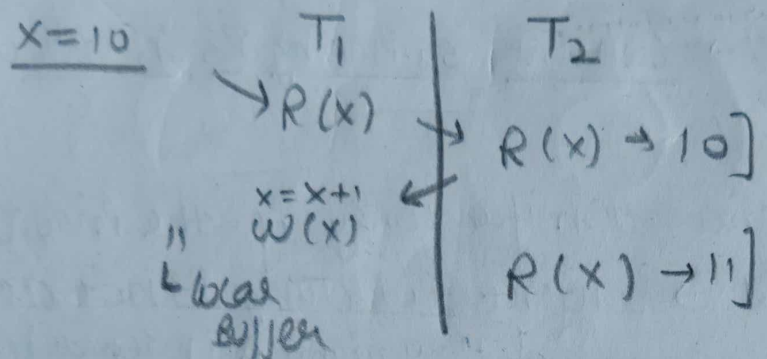
[Now any Prob in R_1 → Rollback But R_2 can't Rollback]

+ Unrepeatable Read Problem

→ When a Transaction Reads a value 2 times and gets Diff Results Because that value was changed By another Transaction

→ Reading a value ~~once~~ two times and getting Diff Answers

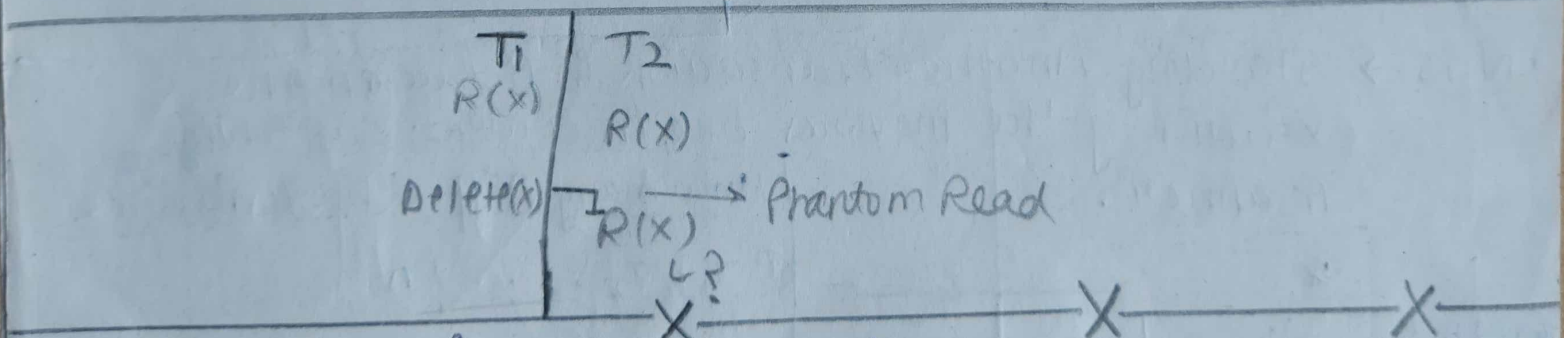
→ The Isolation Property is violated here [Clearly T_1 is affecting T_2]



+ Phantom Read Problem

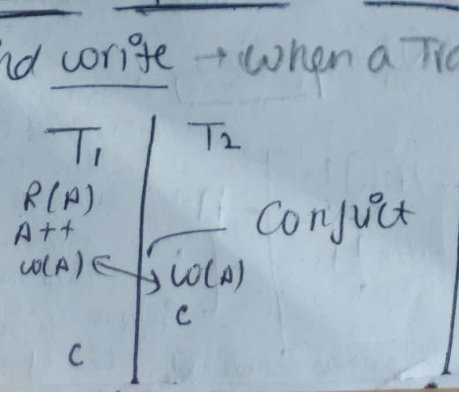
→ The Incapability To Read a Variable which was previously capable To Read Because its structure or the variable itself is Removed By another Transaction.

→ The Isolation Property is violated here



+ Lost Update Prob (write-write conflict)

Blind write → when a Transaction writes a value without Reading it



When the value updated by T_1 is lost due to Blind write of T_2 — The value which was was updated by T_1 got lost and Both Time the value written by T_2 was committed.

T_1 will commit value of T_2 again. T_1's value is lost.

• Serial Schedule and Non-Serial Schedule

└ Set of Transactions – context switching Transactions

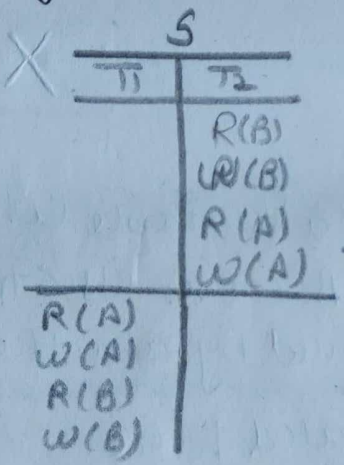
T_1	T_2
R(A)	R(B)
W(A)	W(B)
R(B)	R(A)
W(B)	W(A)
n_1	n_2

① → Schedule → $n_1 + n_2$ ✓

② → No change in sequential order

③ → Content switching is Allowed

• SS → Starting Another Transaction if and only if the Previous one is fully finished → content switching is not allowed.



How many Diff Schedules Possible

① $T_1\ T_2 \ldots T_n$

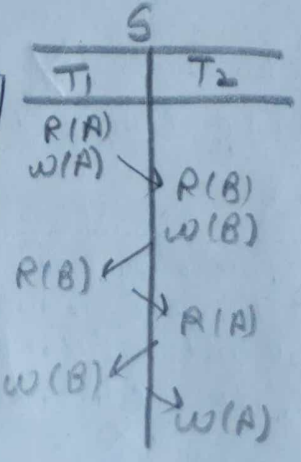
$n \times (n-1) \times (n-2) \ldots 1$

$= n!$

× ×

• NSS → Starting Another Transaction in Between the execution of the previous one in content switching Manner.

☆ $\dfrac{(n_1 + n_2 + \ldots + n_n)!}{(n_1! \cdot n_2! \cdot \ldots \cdot n_n!)} - n!$ ☆



How many Diff Schedules Possible

① $T_1 \times T_2 \ldots T_n$

n_1 n_2 n_n → Instructions of S

Content switch is allowed

$(n_1 + n_2 + \ldots n_n)!$ ~ Arrange → we have consider same cases where the seq order is broken

$\dfrac{(n_1 + n_2 + \ldots + n_n)!}{(n_1! \cdot n_2! \cdot \ldots \cdot n_n!)} - n!$

Subtract serial ones.

Total schedules

Conflict Serializable

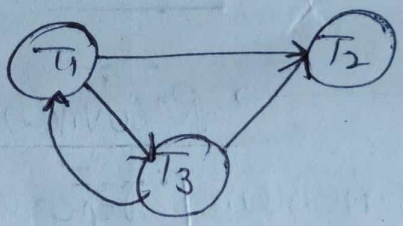
- Converting / showing that a Non-serial sch into a serial sch. Then that non-serial sch is known as Conflict serializable sch.
- we achieve this by swapping Instructions. Specifically swapping non-conflict Instructions

└ Conflict instructions ─ Belong to 2 Diff Transaction
                        ─ works on same Variable/value
                        └ one of it has write Instruction

└ R → W | W → R | W → W

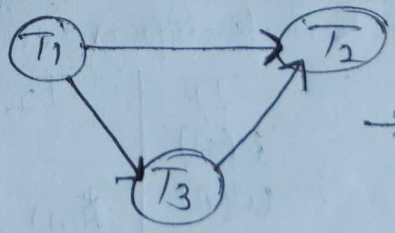
Practice Prob

T_1	T_2	T_3
R(x)		R(z)
	R(y)	W(z)
R(y)	W(y)	
	W(z)	W(x)
W(x)		



- If there is a conflict → draw a edge
- If cycle obtained → No conflict seriable sch

T_1	T_2	T_3
R(x)	R(y)	R(y)
	W(y)	
W(x)		W(x)
	R(x)	
	W(x)	



→ order ? of conflict seriable

out of 3! = 6

First instruction → No incoming edge → T_1

Last instruction → No outgoing edge → T_2

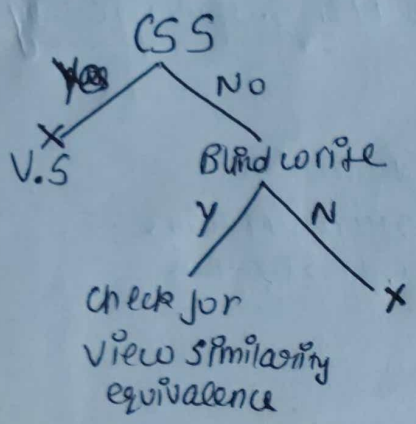
$T_1 \rightarrow T_2$, T_3 → $T_1\ T_3\ T_2$, $T_3\ T_1\ T_2$ (can be any where), $T_1\ T_2\ T_3$

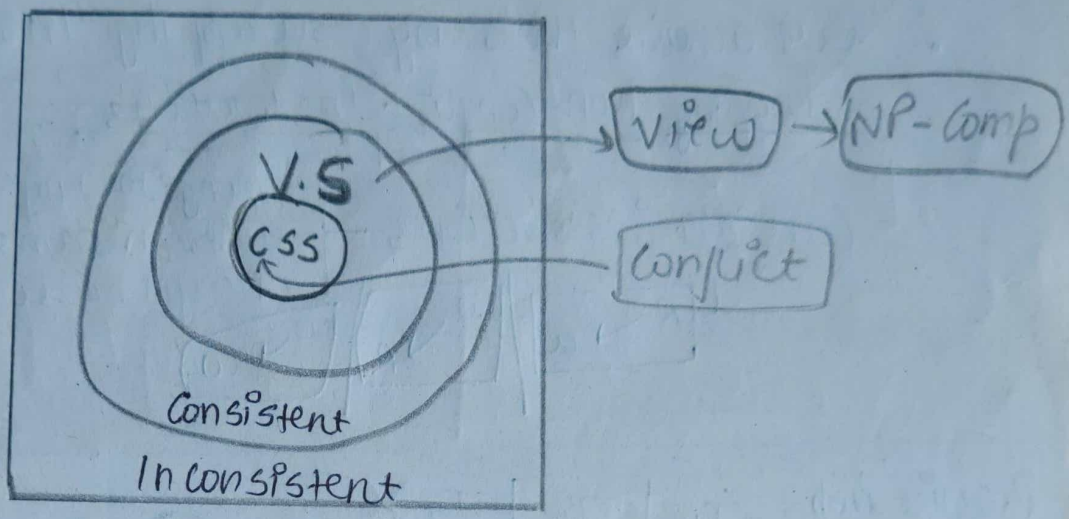
↳ $T_1 \rightarrow T_3 \rightarrow T_2$

In conflict → If there is a conflict then No need to check further for that Instruction

View Seriability

① If Blind write → Possibility of view S → Check for view Similarity (of all Transaction) → same order of initial Read, same order of final write, same order of intermediate R/w → Should match 1 out of n!

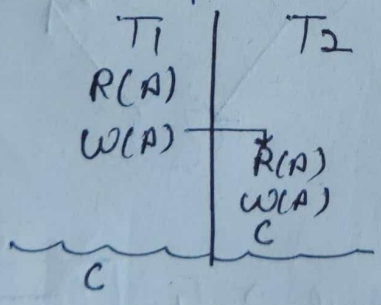




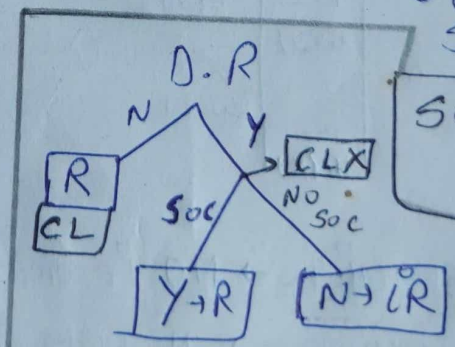
Recovrable Schedule → mandatory

① Dirty Read Problem [Dependency of one Transaction on another]

↳ when the Dependent Transaction commits Before the Independent Transaction → Problem → If any Prob happens The Independent can rollback But dependent one cannot → Non Recovrable

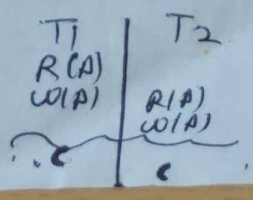


↓ work on same value



SOC → same order commit

Soln → The order in which D.R is Performed In that same order commit Should be Performed → Independent first and dependent after That → Recovrable

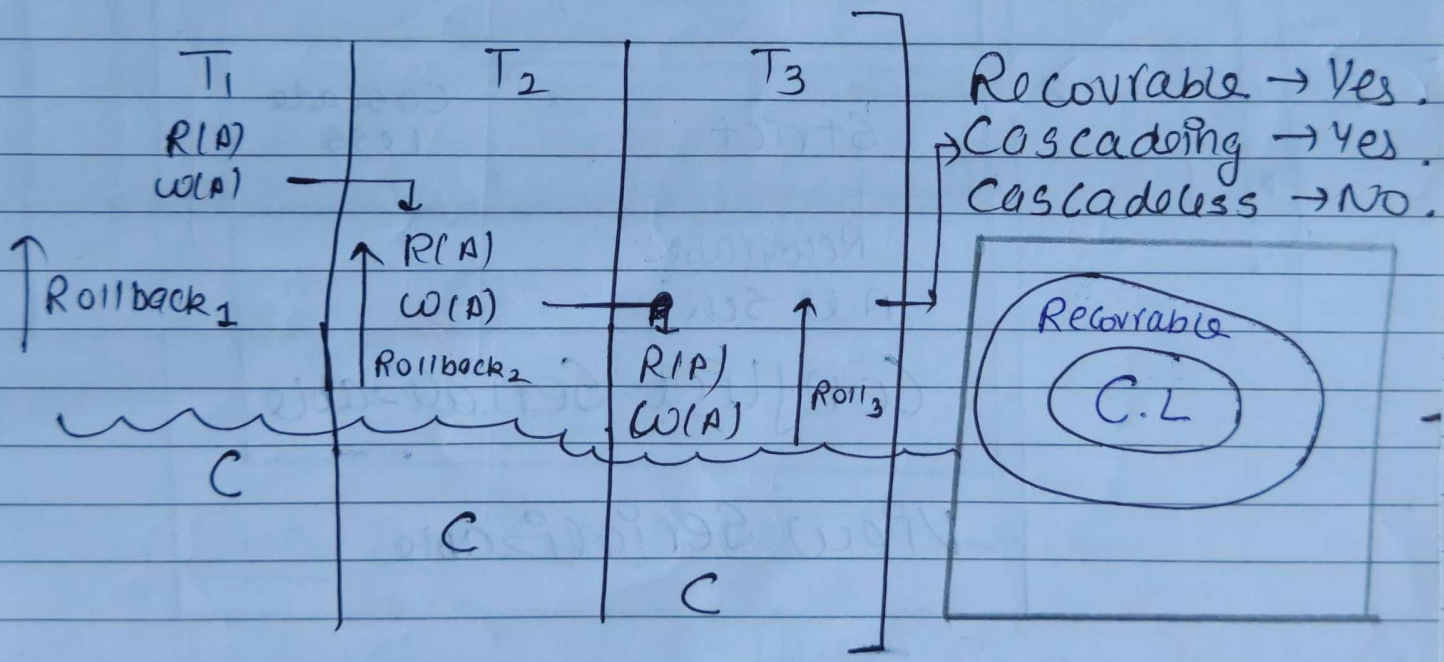


Cascadeless Schedule [Not compulsory]

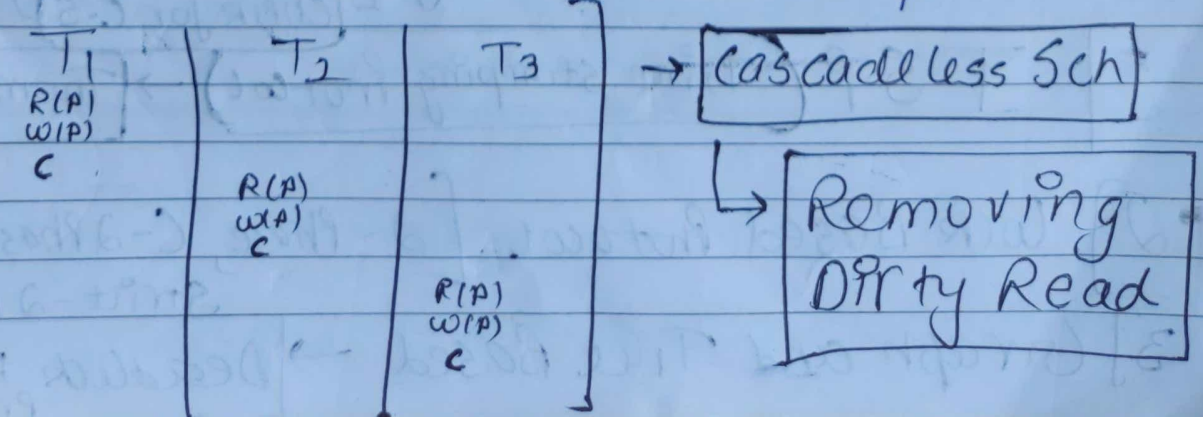
→ Cascade | Rollback / Schedule | Cascading Rollbacks

- Jab ek Transaction ko dekhkar Baki Transactions ussi order mai Rollback kare

- When the main Transaction Rollbacks due to failure then all other dependend Transaction Rollbacks in the same order of Dirty Read → efficiency ↓



If a system has No Cascading Rollbacks then it is considered as Cascadeless Rollbacks / Schdule.

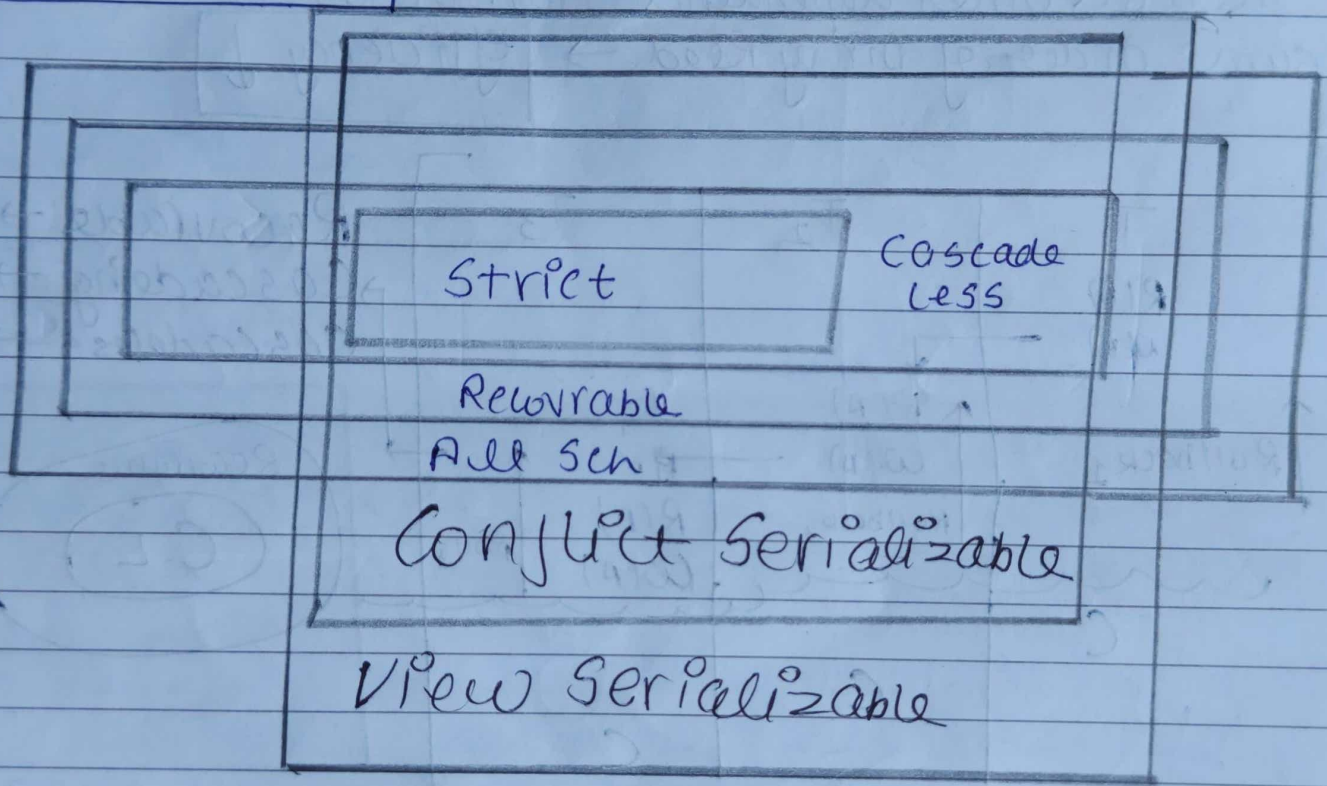


Strict Sch

→ If one Transaction is working on a value, then No other Transaction can work on that value until and unless that one Transaction's commits.

↳ No Dirty Read

→



Creating Sch using Protocol

1] TSP (Time stamping Protocol) → Thomas Write Rule
   ↳ Check for C.S

2] Lock Based Protocol [2-Phase, C-2 Phase, R-2 Phase, Strict-2 Phase]

3] Graph and Tree Based → Deadlock Handling Prevention

Time-Stamping Protocol

- Giving Tokens to each and every Transaction so that they can access the Value one at a Time Without any conjict Based on That Token
- Token = Time stamp → Time at which they arrive / Time at which They make request.

★ $T_1 \rightarrow 10:10$ | $T_2 \rightarrow 4:04$ | $T_3 \rightarrow 8:05$

- T_i → Transaction Arrives
- $T.S(T_i)$ → Time stamp given at its arrival
- T_j → $T.S(j)$ → new Transaction

Always → $T.S(T_i) < T.S(T_j)$ ∴ Time moves forward → unique and fixed

(Senior ↵, Junior ↵)

↳ ensures conjict serializable as T_i will execute Before T_j

T.S of Data Item (Value)
↳ Q

$T.S(Q)$ →
- $W.T.S(Q)$ → Latest Time stamp of ANY TRANSACTION That executed write(Q) successfully.
- $R.T.S(Q)$ → Latest Time stamp of ANY TRANSACTION That executed Read(Q) successfully.

★ Now T_i required for Read(Q) ★

↳ as Read-Read = No conjict → so check for write only.

it should have read the value first

- T.S(Ti) < WTS(Q) → | T1 R(Q) ↳ R(Q) | T2 W(Q) |
  - ↳ reject and Ti should rollback
  - ↳ Ti is trying to read a value of Q that was already overwritten

Junior → allow
Senior → Not allow

- T.S(Ti) ≥ W.T.S(Q)
  - ↳ R.T.S(Q) = |max(RTS(Q), T.S(Ti))|
    - ↳ latest one

——X——X——X——X——X——X——

Ti req for write(Q) → check for R and W Both.

- T.S(Ti) < RTS(Q)
  - ↳ Value of Q Ti is producing was needed previously and the system assumed that the value would never be produced; if we change it now then RTS would be conflicted (swap of conflict instruction).
  - ↳ Reject and Rollback.

T_1	T_2
~~R(Q)~~	W(Q)

→ even if we do it then final value still remains of T_2

- $T.S(T_i) < WTS(Q)$
  - → Phele Karna Chaiye tha
  - → T_i is attempting to write an absolute value of Q
    - → Reject and Rollback.

Thomas write Rule

- → In this case $[T.S(T_i) < WTS(Q)]$ where we face the "case of absolute write"
- → It does not matter if we Rollback or not Because at the End latest Transaction will Update the value so it is Better to ignore the write Operation rather Than Rollback.

• Better Concurrency	• Provides V.S but not sure of C.S

Properties

- ensures Conflict Serializability
- ensures V.S
- Possibility of Dirty Read → No restriction on commit
  - → chance of irrecovrable and Cascade Rollbacks are possible.
- either allow or Reject
  - → no possibility of Deadlock.

Lock Based Protocol

- To Prevent conflict → A Transaction locks a value → works on it and Then Unlocks it to be available to other Transaction
  - This Method decreases (↓) Concurrency So we lock the value in 2 ways

(1) Shared lock
- lock-S(Q)
- Can only Perform Read Operation
- Any other Transaction can obtain same lock on same Data Item at same Time

(2) Exclusive lock
- lock-X(Q)
- For R/W operations
- Any Other (T) can not Obtain either Shared or Exclusive lock.

* Diff variable → everything can work → No Restriction

ON SAME DATA ITEM

	S	X
S	T	F
X	F	F

T_i	T_j
L.S(A)	L.S(A) ✓
L.S(A)	L.X(A) ✗ [L.X → L.S]
L.X(A)	L.X(A) ✗

PROP

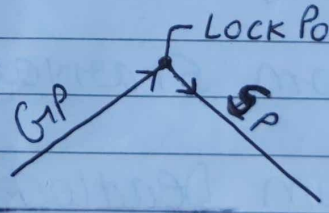
↳ If we perform unlocking → Inconsistency
↳ chance of Dirty Read

↳ If we Do not unlock → concurrency will be poor.

- we req Transaction to follow some set of Rules for lock and unlocking. ↓

2-PHASE LOCKING (2PL)

- works like this →



- Growing Phase → Locks are acquired and no locks are released.
- Shrinking Phase → Locks are released and no locks are acquired
- Lock Point → Point B/w GP and SP → Checks the order of flow of Transaction.

sum = 4500
mh = 100

100 200 300 400 500 600 700 800 9

4500

mid =

22500

T_1	T_2
L.S(A)	
GP	L.S(A)
L.X(B)	
.	L.X(D)
Unlock(A)	Unlock(A)
Unlock(B)	Unlock(D)

GP

lock point

S.P

T_1	T_2
L.S(A)	Not Allow
	L.X(A)
	[Block T_2 until A is unlocked]

Drawbacks

Dirty Read

1. May not free from - Irrecovability → Coscading Rollbacks
2. Not free from Starvation
3. Not free from Deadlocks

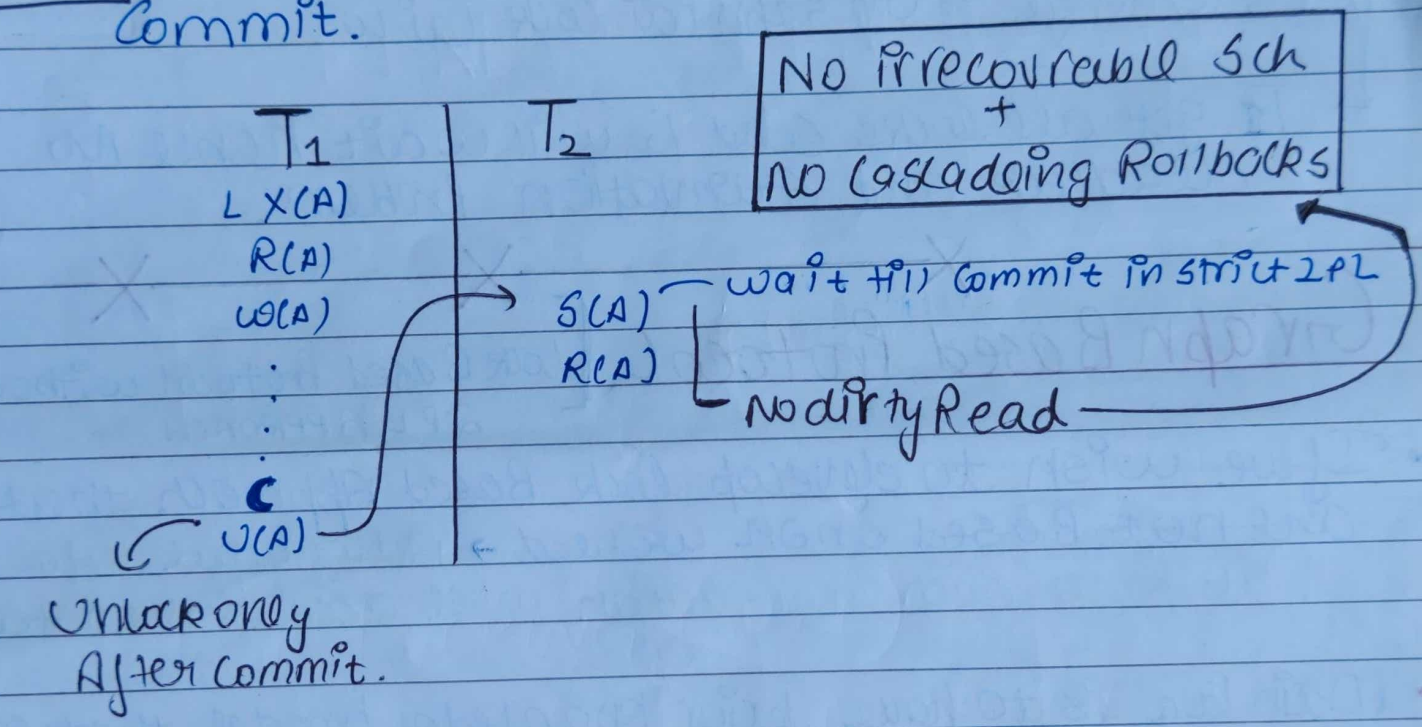
T_1	T_2
L.X(A)	
	L.X(B)
L.X(B) [wait till B is unlocked]	
	L.X(A) [wait till A is unlocked]

[T_1 → waiting for B
T_2 → waiting for A]

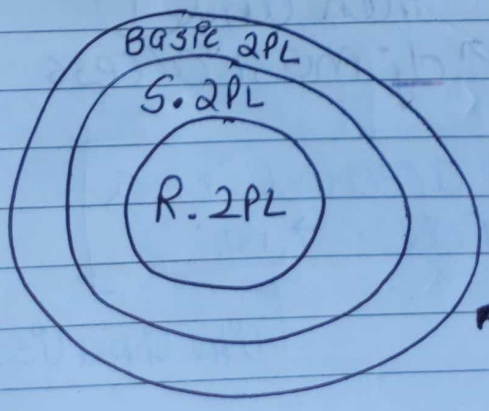
Strict 2PL

- It should satisfy the Basic 2PL and all Exclusive locks should hold until commit/abort

i.e → All Exclusive locks will be unlocked only After Commit.



Rigorous 2PL → It should statisfy the Basic 2PL. Very Restricted. All Enclusive and Shared locks should hold until commit.



Cascadeless Sch
+
Recovrable Sch

★ Dealock and Starvation still remains

[C-2PL → Starts from LOCK POINT]

To solve Deadlock and Starvation

↳ Conservative 2PL → Before operation starts,
↳ acquire All locks on data items that are going to be utilize in future Transaction.
↳ Transaction start hone se pehle → Jitne data item use honge → un sabh pai lock lai lo.
↳ T_1 gets all locks and T_2 will wait hence No deadlock and starvation problem.

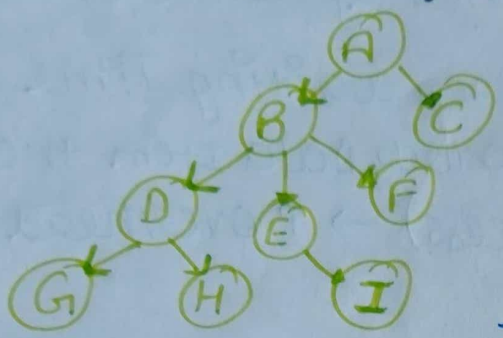
↳ No Practical Approach

Graph Based Protocol [Lock Based Protocol without 2PL APPROACH]

- If we wish to develop lock Based Approach that are not Based on 2PL we need → Additional Info that how Each Transaction will access the Data.
- Main Idea is to have prior knowledge / model that can give additional information About the ORDER IN which the database items will be accessed.
- We use PARTIAL ORDERING → on set all Data items $D = \{d_1, d_2, d_3 \ldots, d_n\}$ If $d_i \rightarrow d_j$ (PO) Then any (T) accessing Both d_i and d_j must access 'd_i Before d_j'

[Data items are accessed in a Specified Order → Po
Like linked list]

- After Po set of all data items D will be viewed as directed acyclic graph → Database graph



Initial Information needed. — (1)

- 2 Restrictions → Study Graphs that are Rooted Trees
  → only use exclusive mode locks [lock-X(A)]

---

* Tree Based Protocols [using (1) → we want to use G so have to go Through A→B→D→G] — (2)

Rule 1] Transaction T_i can lock (First lock) on any Data item.

Rule 2] Subsequently a data item Q can be locked by T_i only if the Parent of Q is currently locked by T_i

[Jabtak uska Parent locked nahi → Vo locked nhi ho sakta]

Rule 3] Data item may be unlocked at any Time

[No Growing and Shrinking Phase]

Rule [4] Data item Q that has been locked and unlocked by T_i can not be relocked by T_i

---

Properties

→ All sch that are legal under the Tree protocol are C.S and V.S

- Tree Protocol ensures freedom from Deadlock
- Do not ensure Recoverability and Cascadelessness.
- EARLY Unlocking → Shorter waiting Time → ↑ Conc
- A Transaction may have to lock Data Item that it does not access (need to access) → ↑ overhead waiting T → ↓ Conc. [using 2]
- Recoverability and Cascadelessness can be provided by Not Unlocking Before Commit

---

Deadlock Handling → Tree Protocol → Partial order

- Prevention: ensures system will never enter a Deadlock state.
  - ↳ C.2PL
- Detection & Recovery: Allow system to enter into Deadlock then Try to recover.

Prevention

- **wait-die** (non Preemptive) → T_i → T_j (↑ d)
  - $TS(T_i) < TS(T_j)$ → T_i must wait
  - $TS(T_i) \geq TS(T_j)$ → T_i rollback
    - ↳ come again with same TS
- **wound-wait** (Preemptive) → T_i - T_j (↑ d)
  - $TS(T_i) > TS(T_j)$ → T_i can wait
  - $TS(T_i) < TS(T_j)$ → T_j Rollback
- **Lock-Timeouts** (Allocated Time) → T_i - T_j (↑ d)
  - in 5 min (allocated Time) T_i does not get lock of d then for T_i → ROLLBACK
  - forcefully take d from T_j