DBMS

ACID properties

① Atomicity : In Transcations all instructions must execute compelely if at least one instructions fail to execute then all instructions will rollback.
DBMS component : Transcation mgmt is responsible for atomicity.

{ Transation mgmt comp.

ya toh all ins. execute successfully otherwise koi bhi nhi hogi.

---

② Isolation : If one transaction T_1 execute in isolation and T_1 execute with others no. of Transcuction $T_2 .. T_n$. the result in the both is case is same. then we can say that Transactions are in isolations.

e.g Ticket T_1 ek ticket ko book karne se , baki tickets ka koi farak nhi padta hai

{ component : concurrency comp.

---

③ Durability : agar koi transaction se changes hote hai , toh vo db mai persist karenge . assa nhi ho skata ki hardware failure se vo changes undo ho jaye. { recovery mgmt component hota cause of durability }

Consistency

$DB_1 \xrightarrow{T} DB_2$

Transaction se pahle database consistent tha, after transaction bhi database hai vo bhi consistent hona chahiye.

When a transactions follow ACID property then DB will remain consist even after performing the transaction.

Advantages of Concurrency

(A) Waiting time ↓ (B) Response time ↓ (C) Resource utilization ↑ (D) Efficiency ↑

To achieve these advantages we must follow concurrency transactions, this requires the transaction mgmt. Due to this we have some problems which we need to tackle.

Commit state — [when change done by Trans. in local buffer move to original database.]

Dirty Read problem

10	T_1	T_2	11
10	R(A)		
11	W(A)	R(A) → 11	
	⋮	⋮	
	fail ↑	C →	
	rollback		

dirty read is a problem becoz T_2 read uncommited value from T_1 (local buffer) and commit into DB.

Issue: If T_1 face some failure, then T_1 will rollback due to this now, DB become inconsistant (as T_2 already commit it).

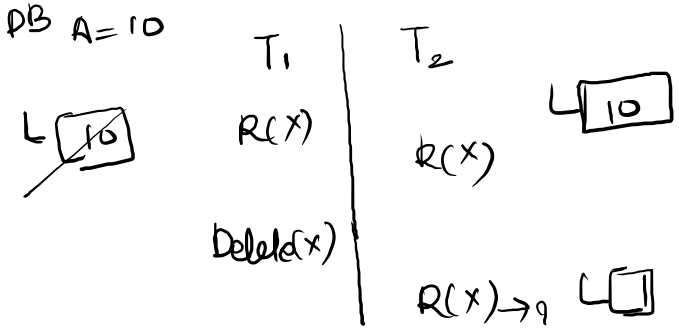
solution: T_2 only commit after, T_1 is commit. so that T_2 know the latest value of DB.

Unrepeatable read problem :

A=10	T_1	T_2	
	R(A)		A=10
11		R(A)	?
	W(A)		
		R(A)	A=11

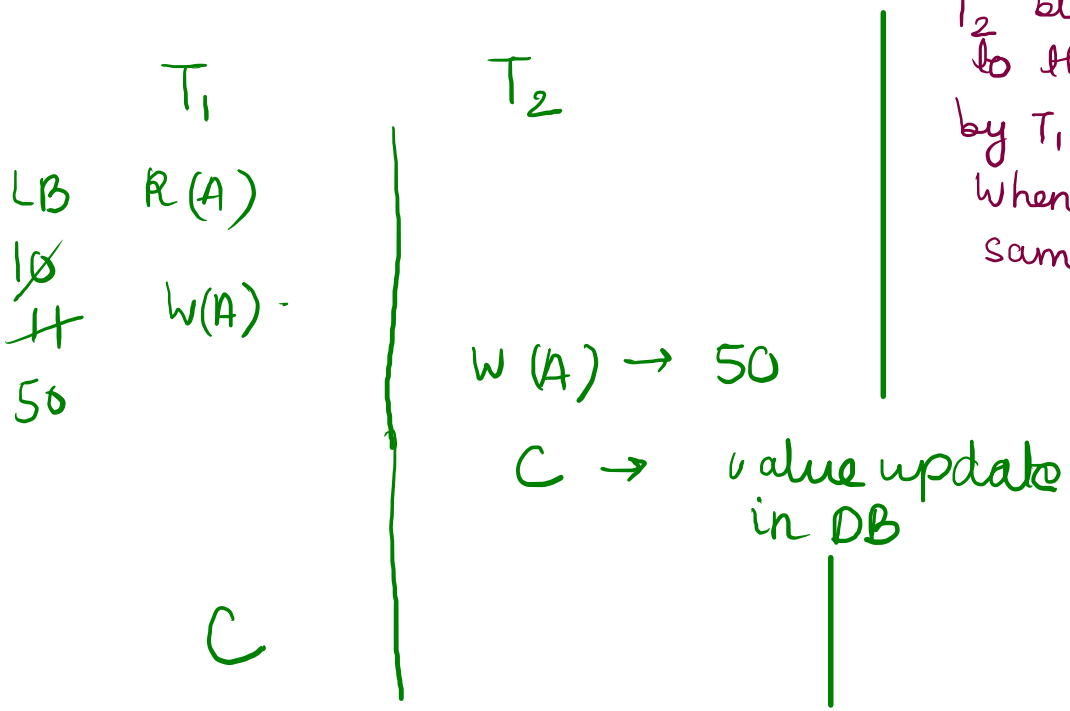
T_2 read the value of some variable at 2 different time and find the inconsist in their values.

Phantom read problem



When one transactions read values from DB and again try to read same variable value but this time variable does not exit cause phantom read problem.

lost update problem (write - write conflict)



T_2 blind write the value into DB. and due to this whatever value hold in local buffer by T_1 get override by T_2 value (50).
When T_1 do commit then T_1 re update the same T_2 value (50).

This is a problem becoz T_1(11) is lost and T_1 is unware about the value change from 11 → 50.

Conflict serializable

a non-schedule transaction can convert into serial schedule after swaping non-conflict instructions then we can say that this transaction is consistent

Conflict seriliazability

① T_1: R(X) | T_2: R(Y) ✓

reading of data → 2 different sets of datas

② T_1: R(X) ✗ | T_2: R(X)

Not allowed

③ T_1: R(X) | T_2: W(X) ✗

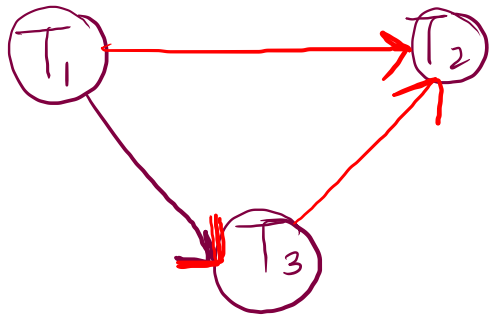
Not allowed

④ T_1: W(X) | T_2: W(X) ✓

[This shows that T_2 is depend upon T_1

how to check C.S?

T_1	T_2	T_3
R(X) ①		
	R(Y)	
		R(Y) ③
	W(Y) ③	
W(X) ②		
	R(X) ④	
		W(X) ①, ④
	W(X) ②	

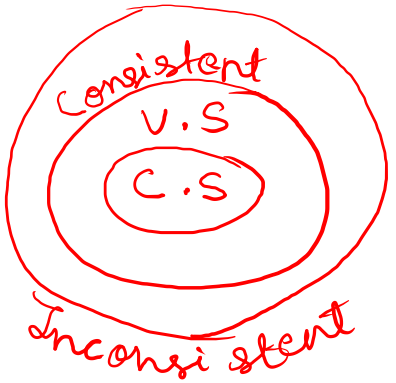


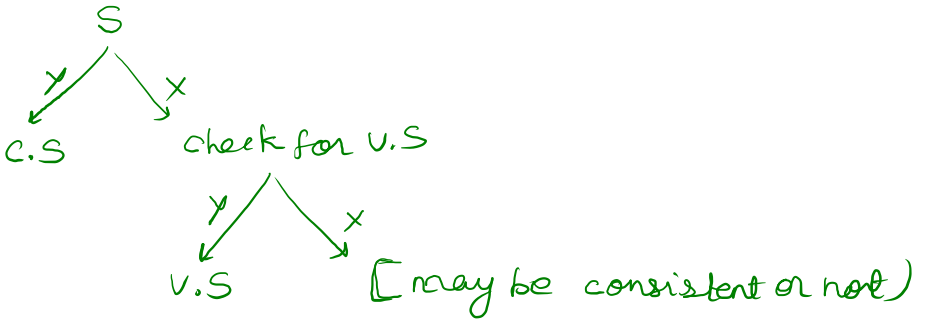
edge defines T_2, T_3 depend on T_1
T_1 must execute before T_2 & T_3

If in a graph there is a cycle then S is not conflict serializable (not consistent)

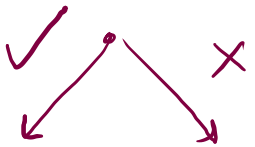
In eg. there is no cycle, This S is conflict ser.

order : $T_1 \rightarrow T_2 \rightarrow T_3$





how to check V.S?



blind write

[V.S X]

V.S [may be]

Recoverable Schedule

Those schedule which can be rollback or undo there changes.

★ case I

	T_1	T_2
10	R(A)	
11	W(A)	
		R(A) 11
		W(A)
		C → DB
	C	

(dirty Read) → Irrecoverable Schedule

★ case II

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

$T_2 \rightarrow T_1$

recoverable schedule

becoz, when there is any failure in T_1, And T_1 is rollback. Then T_2 will also be rollback.

Time stamp Ordering Protocol:-

→ Basic idea of Time stamping is to decide the order between the transactions before that enters into the system. so that in case of conflict during execution, we can resolve the conflict using ordering.

→ the reason we call time-stamp not stamp because, for stamping we the value of system clock. (as it will always be unique. can never repeat itself)

→ Two ideas of time stamping T_i T_j

→ Time stamp with transaction:- with each transaction t_i, we associate a time-stamp denoted by $TS(T_i)$,

→ It is the value of the system clock when a transaction enters into the system. so if a new transaction T_j enters after T_i then, $TS(T_i) < TS(T_j)$. always unique, will remain fixed through the execution.

→ also determine serializability order if $TS(T_i) < TS(T_j)$ then, sytem ensure that in the resultant C.S.S T_i will execute first before T_j.

→ Time stamp with data item:- for each data item Q, protocol maintains two time-stamps.

W-time stamp(Q): is the largest time-stamp of any transaction that executed write(Q) succesfully.

R-time-stamp(Q): is the largest time-stamp of any transaction that executed read(Q) succesfully.

6.22 #TimesStampingProtocol #Transaction in detail in Hindi

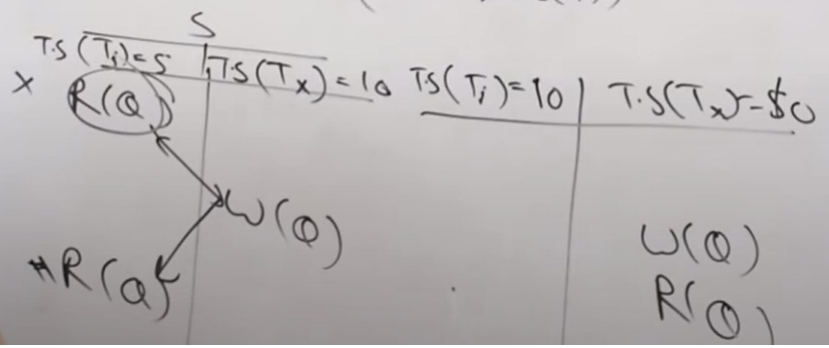
→ if $T.S(T_i) < W.T.S(Q)$, means T_i needs to read a value of Q that was already overwritten
(5) (10)
Hence request must be rejected & T_i must rollback.

if $T.S(T_i) \geq W.T.S(Q)$, operation can be allowed and R.TS(Q) will be
(10) (5)
max (RTS(Q), T.S(T_i))
for Write(Q):

if $T.S(T_i) < RTS(Q)$, means value of Q that T_i is producing was needed previously and the system assumed that the value would never be produced. Hence reject and Rollback

if $T.S(T_i) < WTS(Q)$, T_i is attempting to write an obsolete value of Q. reject and Rollback

otherwise ok, $WTS(Q) = max(WTS(Q), TS(T_i))$



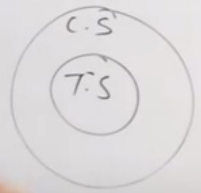
Properties of Time Stamping Protocol:

→ It ensure conflict serializability

→ It ensure view serializability

→ possibility of dirty read no restriction on commit, irrecoverable schedules and cascading rollbacks are possible.

→ Here either we allow or reject so no idea of deadlock.

→ 

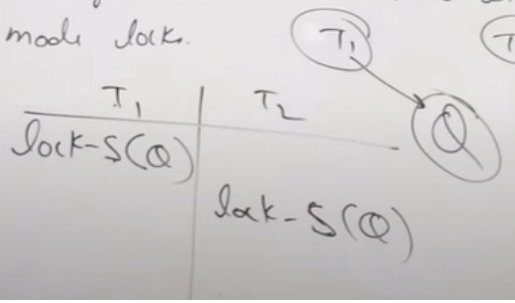
→ suffer from starvation, relatively slow

achieve isolation we first obtain a lock on a data item then performed a desired operation and then unlock it.

→ To provide better concurrency along with isolation we use different modes of locks.

Shared mode:- denoted by lock-S(Q). transaction can perform Read operation, any other transaction can also obtain same lock on same data item at same time. (So called shared)

Exclusive mode:- denoted by lock-X(Q), transaction can perform both Read/ write operations, any other transaction can not obtained either shared/Exclusive mode lock.



T_1	T_2
lock-S(Q)	
	lock-S(Q)

Compatability table

	shared	exclusive
shared		
exclusive		

Two-Phase locking Protocol (2PL) :- / Basic 2PL

→ This protocol requires that each transaction in a schedule will be two phased Growing Phase and shrinking phase.

→ In growing phase transaction can only obtain locks but can not release any lock.

→ In shrinking phase transaction can only release locks but can not obtain any lock

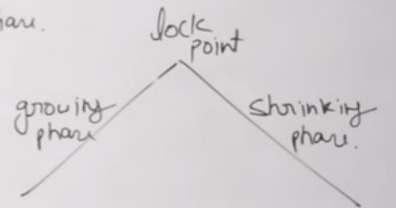
→ transaction can perform read/write operation both in growing/shrinking phase.

→ Ensure, C.S/V.S, the order of Serializability is the order in which transaction reaches lock point.

→ May generate irrecoverable schedules and cascading rollbacks.

→ Do not ensure freedom from deadlock.

$T_2 \rightarrow T_3 \rightarrow T_1 \rightarrow T_4$



S

T_1	T_2
LockX(A)	
R(A)	
W(A)	
UNlock(A)	
	Lock(C)

T_1	T_2
lock-S(A)	
Read(A)	
	lock-X(B)
	Read(B)
	Write(B)
lock(B)	
	lockX(A)

DeadLock Prevention:-

→ To ensure no hold & wait, each transaction locks all the data item before it begins execution eg C-2PL.

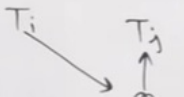
→ To ensure no cyclic wait, impose an ordering of all data item, and requires that a transaction lock data item, only in the sequence consistent with ordering. e.g tree protocol.

→ It is difficult to predict what data items are required.

→ data item utilization will be low.

→ ordering of data item may be difficult, can time taking to follow.

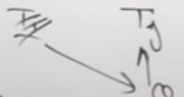
wait-die: (Non-preemptive)



if $T.S(T_i) < T.S(T_j)$ then T_i must wait (T_i is older)

if $T.S(T_i) > T.S(T_j)$ then T_i rollback (die)

Wound-wait: (Preemptive)



if $T.S(T_i) > T.S(T_j)$ then T_i can wait (T_i is younger)

if $T.S(T_i) < T.S(T_j)$ the T_j rollback

Lock-timeouts:

S Lin