Transaction Management in DBMS

Transaction :- It is also a sequence of instructions/operations (same like process, job/task, thread..)

only diff. is
All the operations of the ~~pro~~ transaction will execute completely at same time. (if some interrupts, then it won't stop there)

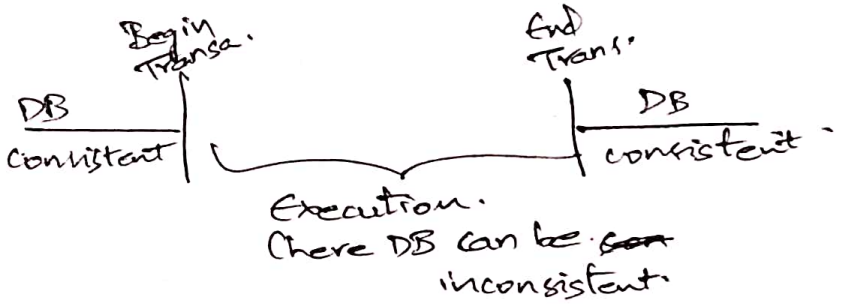
Transactions are atomic in nature (either complete totally or roll back) if ~~com~~

→ A transaction is a single logical unit of work which access DB and possibly modify the DB.

ACID Properties:- Atomicity, consistency, Isolation, Durability.

~~Assumption~~, initially database is consistent,
only Transaction interacts with DB, so it has to modify the DB such that it transforms one consistent state to another consistent state.

If it transforms to inconsistent state (no)



R(A)
A=A-10
W(A)

R(B)
B+10
W(B)

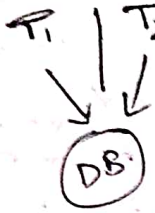
If an interrupt happens here, then it must roll back.

Atomicity :- ~~Either all the~~ No partial execution of transactions is allowed

Consistency :- ~~If~~ A transaction has to lead ~~from~~ one consistent state to another consistent state.

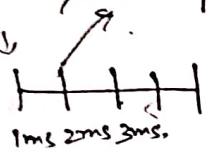
Ensure consistency.

Isolation:- Tatkal tickets, suppose transactions serially done., it takes lot of time, not possible >
So, simultaneously no. of transactions done.
Concurrency.
Race condition arises and leading to inconsistency



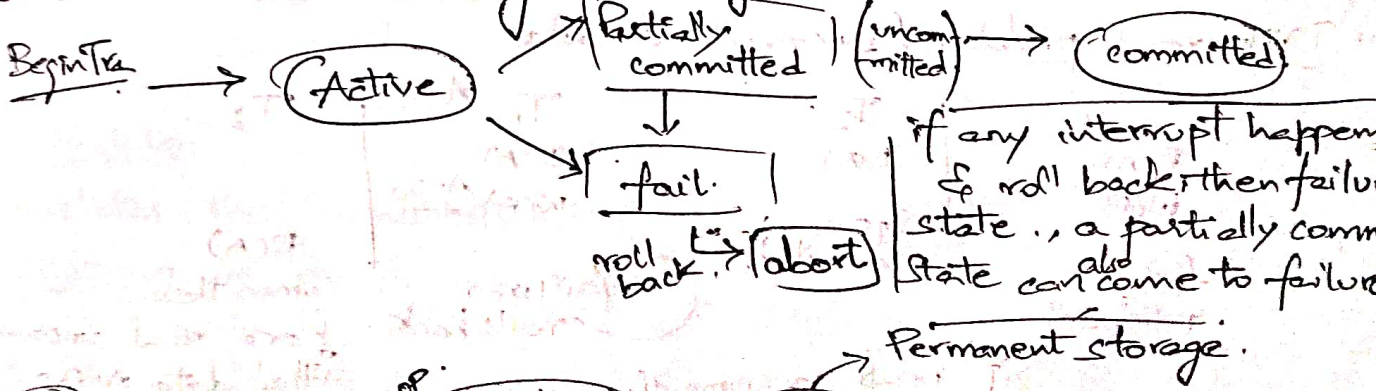
logical Isolation:- Many ~~simult~~ transactions simultaneously done.,
But, its like only one is executing for them.
but for T_1, it thinks, only its executing
Same for T_2.
Intermediate results of individuals transactions must not be available to each other.

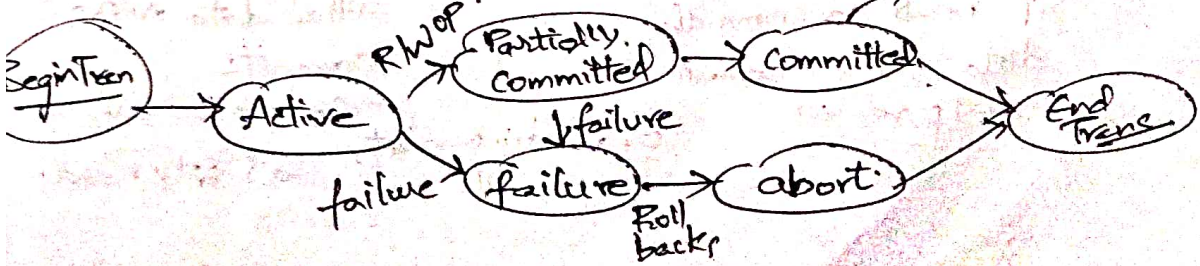
Durability:- ~~talkout~~
If a transaction commits, whatever changes it does, persist forever.
Even if system fails, then can't roll back,



How to roll back a committed Transaction (not possible).
if R(A)
W(A)
Compensating Transaction — then we need unwrite., write another Transaction.

Transaction states:-
When a transaction begins, it goes into active state.





Concurrency & problems associated with it

If, ATMs transactions are done serially, then not two persons can withdraw money at the same time. (not reliable)

So, concurrency.

Suppose T_1 T_2 T_3 T_4

Serially done (order doesn't matter) always leading to a consistent state

T_1
T_2
T_3 T_4

Serial execution of Transaction:-
The execution is not overlapped.

Non-serial ex. of Trans:- (Concurrent ex. of Trans.)
- always allow overlapped Transaction

T_1 T_2
T_1 T_3
T_2 T_4

Advantage of concurrency:-

Drawback of Serial ex.:- no effectiveness, productivity ↓ces.

(1) Response Time ↓ces. for T_2 → (T_1 T_2)
(2) Avg. waiting Time ↓ces. (T_2 must wait for T_1 in serial)
(3) Resource Utilization ↑ces
(4) Efficiency ↑ces (because of all above, performance ↑ces)

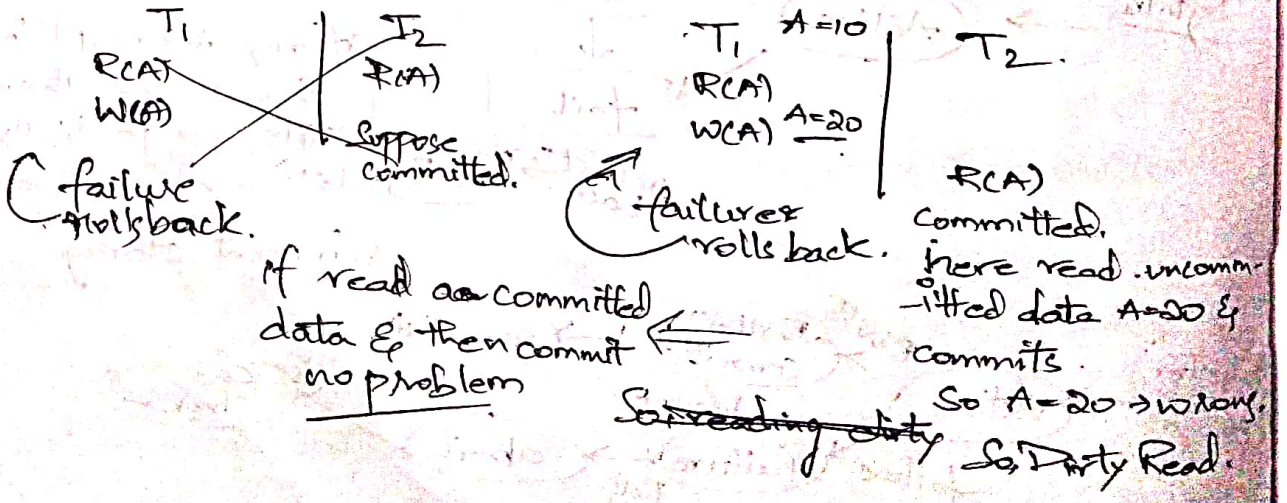
But concurrency may lead to inconsistent data.

So, a controlled concurrency is allowed.
(controlled by ACID properties).

A transaction with ACID properties is allowed.

Problems:-

1) Dirty read problem / uncommitted data



Solutions:- (1) Donot read uncommitted data.
(2) If read an uncommitted data, then donot commit, if the previous commits, then commit.

(2). Unrepeatable read problem. (due to isolation problems).

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

for T_2, it hasn't modified the data, but happened before T_1 has, so for T_2 T_1 is ghost.
So, for T_2 → it is not isolated
Suspects that some other Trans. is interfering
So, rolls back.

here in T_2, read is repeated, so, problem arised
So, donot repeate the read.

(3) Phantom read problem

T_1	T_2
R(x)	
	R(x)
delete(x)	
	R(x)

here its already deleted, so phantom problem.

write can happen, it can write for the first time.
So only phantom read problem.

(5) Lost Update problem:-

T_1	T_2
A=10 R(A)	
A=20 W(A)	
	W(A) A=30
	commits.

→ at last they all share a buffer.
So previous update A=20 is lost.

Schedules
dictates the order in which the transaction should execute.
$T_1, T_2 \ldots T_n$ Transactions → how they are scheduled.

Serial Schedule / Non-serial / Concurrent schedule

T_1, T_2, T_3 → how many possible serial schedule.
3×2 → 6 (3!) possible.

T_1	T_1	T_2	T_2	T_3	T_3
T_2	T_3	T_1	T_3	T_1	T_2
T_3	T_2	T_3	T_1	T_2	T_1

Suppose, n transactions then n! serial schedulings

S.S- $n!$ s- $T_1 \ldots T_n$.

N.S.S- may lead inconsistency.

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

How many non-serial schedules.

R(B) W(B) ✓ W(B) R(B) not possible.

(b)

R(A) | R(B) W(B) | . R(B) | W(B) R(B)

W(A) (✓) W(A) (b)

$T_1, T_2 \ldots T_n$.

$n_1 \; n_2 \ldots n_n$

$$\frac{\lfloor n_1+n_2+\ldots+n_n}{\lfloor n_1 \lfloor n_2 \lfloor n_3 \ldots \lfloor n_n} - \lfloor n$$

division to remove inner schedules.

subtract serial schedules.

$\Rightarrow$ formula for non-serial schedules $= \frac{\lfloor n_1+n_2+\ldots+n_n}{\lfloor n_1 \lfloor n_2 \ldots \lfloor n_n} - \lfloor n$.

Given a NSS schedule, is it a serializable schedule.
How to generate a N.S. schedule which is serializable.

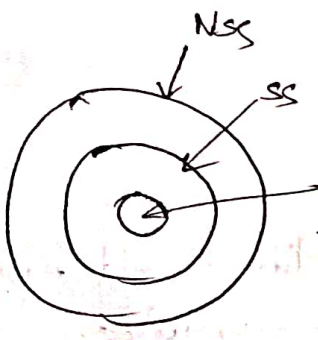
Serializability

Since, N.S. schedule may leads to inconsistency.

10K. trans^{s}.

100K trans^{s} $\rightarrow$ difficult to find which are N.S.

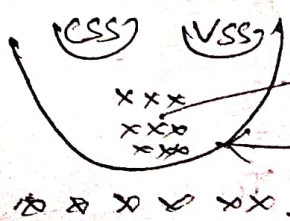
All the N.S.S which are output equivalent to one of the serial schedule then we say it is a serializable schedule.



→ Conflict serializable sch.
→ View serializable sche.

Both the serializable schedule lead to a consistent data. but converse is not true.

C.S.S & V.SS always lead to consistency
But if the given schedule is not C.S.S (or) not V.SS. it doesn't mean that it will lead to inconsistency.



this problem is an NPC problem i.e. they have no algorithm & procedure but given an answer we can verify.

Conflict serializability:- (Stricter)

Given a N.S.S, is it conflict s.s.

CSS : conflicting operations. =>

Simultaneously done.
Read Read. => Non-conflict
Read write Simultaneously => Conflicting
W , R => Conflicting
W , W => Conflicting

when done on same thing R(A) W(A) conflicting.

R(A) W(B) Non conflicting

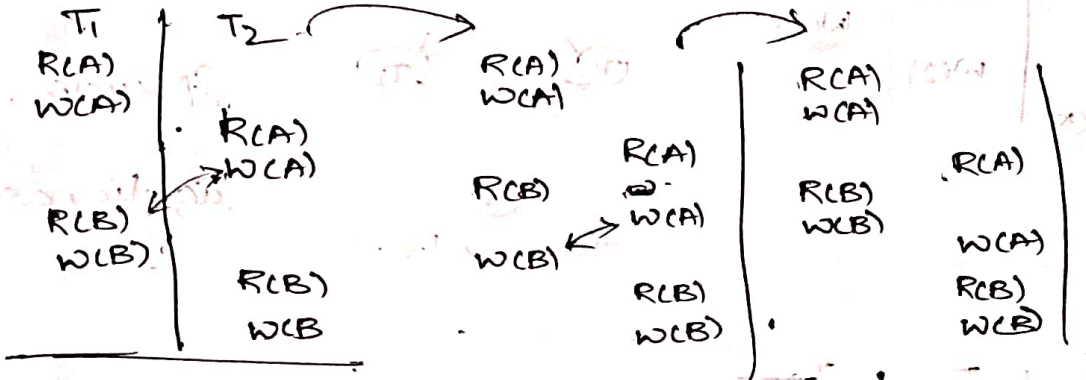
View serializability.

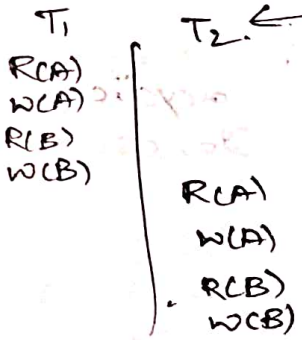
~~(Stricter)~~ flexible. (little strict)

Conflict serializability:-

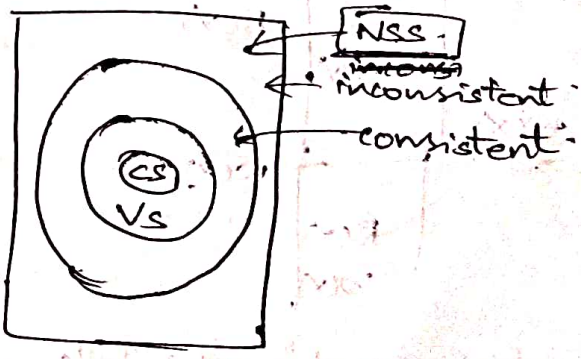
If through series of swapping ~~operations~~ of non-conflicting operations, if you can convert N.SS into one of the possible S.S then the schedule is C.S.S.

Ex.



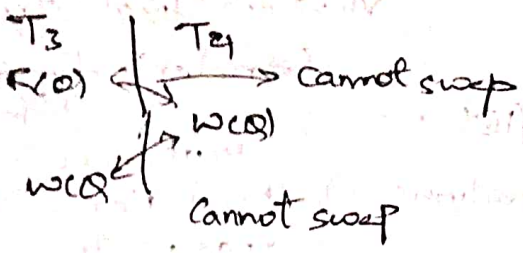


This is a serial schedule.



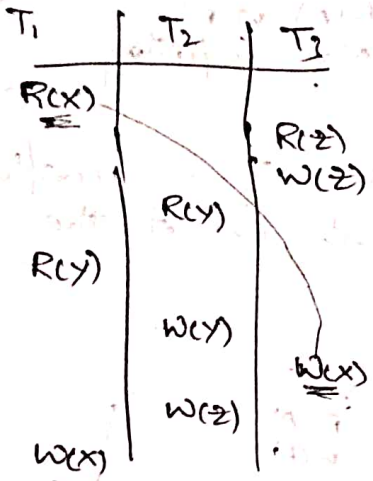
* NSS is conflict serializable iff it is conflict equivalent to any possible serial schedule.

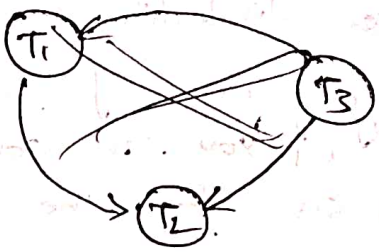
Conflict equivalent :- If through series of swapping of non-conflicting operations if you can convert NSS into one of the possible S.S. S to S'.



So, not a CSS: cannot be serializable.

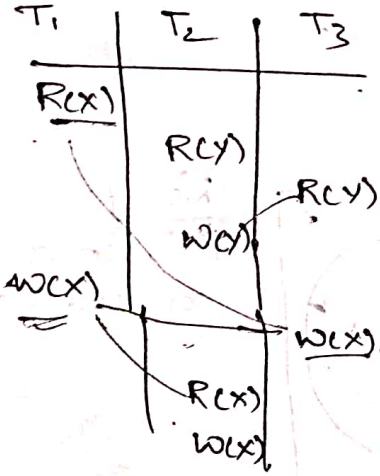
if no. of transactions r/ces, $T_1, T_2, \ldots, T_{100}$, then difficult.
Then draw a precedence graph
It is a directed graph.

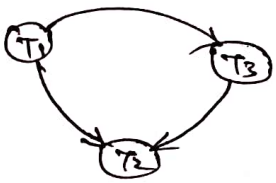




If cyclic, not CSS.
acyclic, CSS.

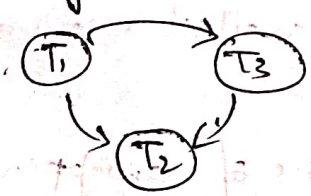
there cyclic.





acyclic
So, CSS.

Precedence graph also tells order of serializability. → Topological sorting



T_1 depends on T_1, T_2
So first T_1
& next $T_3 \rightarrow T_2$
So next T_3

T_1	T_3	T_2

View Serializability :-

it is flexible, little stricter.

try to avoid VSS.

if it is CSS, it will be VSS / All CSS are VSS.

Suppose S is not CSS., (may be VSS)

↳ and S do not have a blind write then not VSS

↳ but have a blind write, then cannot escape from VSS.

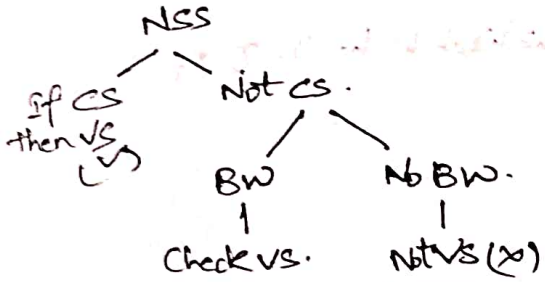
A schedule with blind write may not be VSS.

Blind write :-

if not CSS, check for blind write

No blind write, cannot be VSS.

having a blind write is a necessary cond. for VSS but not sufficient.



A NSS is VS iff it is view equivalent to one of the possible serial schedule.

View Equivalent :- check visually equivalent or not to one of the SC, so if no. of Transactions ↑ces, then becomes difficult.

	(NSS) S	(SS) S'
IR	T_1	T_1
FW	T_2	T_2
Update	$T_1 \to T_2$	$T_1 \to T_2$

S	S'
$T_1 T_2$	$T_1 T_2$
X	X

Initial Read
Final Write
Update (or Intermediate) write read

If: T_1 is doing IR in S', then S also T_1 must do IR.

If T_2 is doing FW in S', then in S also same

Suppose T_1 reads & T_2 writes ($T_1 \to T_2$) then in S also ($T_1 \to T_2$)

Blind write:

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

W(A) → in T_2, without reading, this is writing; blindly writing.

Ex:

S		
T_1	T_2	T_3
R(a)		
	W(a)	
W(a)		
		W(a)

Blind writes.

No intermediate reads

	A
IR	T_1
FW	T_3
U	a.

123
132
213
231
312
321

So T_1 T_2 T_3 (✓) viewserializable to T_1 T_2 T_3.

is it vs to 321 (No)

Ex:

T_1	T_2	T_3	T_4
R(A)			
	R(A)		
		R(A)	
			R(A)
W(B)			
	W(B)		
		W(B)	
			W(B)

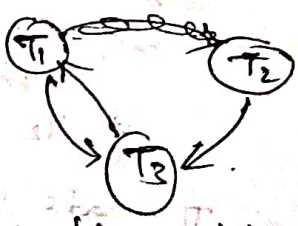
Not conflicting at all

So, ~~not~~ CSS

So, surely VSS.

Ex:-

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



cyclic Not CSS.

So may be VSS.

	A
IR	T_1
FW	T_1
IMR	α

$T_1 \; T_2 T_2 \; T_1$ (b) not possible

So, not VS

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

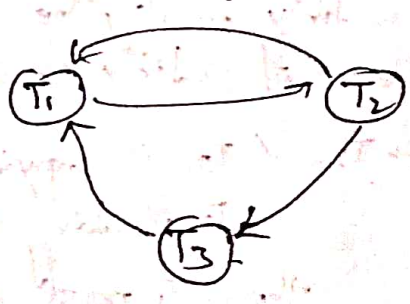
	A	B
IR	T_1	T_1
FW	T_2	T_2
IMR		

=

T_1	T_2	T_3
	R(P_3)	
	R(P_2)	
	W(P_2)	
		R(P_2)
		R(P_3)
R(P_1)		
W(P_1)		
		W(P_2)
		W(P_3)
	R(P_1)	
R(P_2)		
W(P_2)		
	W(P_1)	

	P_1	P_2	P_3
IR	T_1	T_2	T_2
FW	T_2	T_2	T_1
IMR	>		

$T_1 \; (T_2 T_3) \; T_1$ Not VS.



cyclic
Not css.

The schedule is not serializable.

Recovaribility:-

→ A schedule is recoverable, when all the transactions participating in the schedule are ~~recovable~~ recoverable.

T_1	T_2
Initial → R(A) read	
W(A)	
	R(A) → read from the buffer.
Committed.	
failure occurred	

T_1 rolls back, but T_2 can't roll back.

R(A) = 10 & here T_1 W(A) = 20 → and T_2 read 20 & committed but T_1 rolled back so, now T_2 read 20 is wrong ∴ caused inconsistency.

Recoverability is a must (necessary) Property for a schedule.

T_1	T_2
A = 10 R(A)	
A = 20 W(A)	
	R(A) A = 20
	W(A) A = 15 (A-5)
	Committed.
& failure occured	

then T_1 rolls back, & A becomes '10' then T_2 is wrong.

If only T_2 executed T_2 R(A) = 10, W(A) = 10-5 = 5.

∴ Avoid dirty read.

always read committed data.

If there is no dirty read, the schedule is always recoverable.

If there is a dirty read, the order of commit must correspond to order of dirty read. then recoverable else not recoverable.

$T_1 \rightarrow T_2$ (T_2 is copying from T_1)

∴ T_2 is doing dirty read

then T_1 must commit first then T_2 can commit.

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A) ↑ both can roll back.
failure	
C	
failure	T_1 can't but T_2 must roll back.
C	

Recoverable:- ~~A~~ can roll back

$T_1 \rightarrow T_2 \rightarrow T_3$.

Committment order:-

T_1 T_2 T_3

C_1 C_2 C_3.

Cascadelessness

$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_5$.

Cascading rollbacks

if T_1 failure, then all ~~must~~ T_2, T_3, T_4, T_5 must roll back.
One causes many to fail.

Cascading abort:- The abort of one transaction forces the abort of another transaction to prevent the second transaction from reading invalid (uncommitted) data.

If no dirty read, then cascadelessness(✓)
If there is a dirty read. → always cascading rollbacks.

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

always read committed data, so reading updated copy so, no need to roll back.

All ~~cascading~~ cascadelessness schedules are recoverable but not vice versa.

Strict schedule:- very close to a serial schedule.
No dirty read No dirty write.
always work on committed data.

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

W(A) A=15
(A-5)

Committed.

if thinks it is writing 20 but (no) 15

So this is dirty write.

If there is a dirty write, always a change of lost update problem, here T_1 lost its update.

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

here no dirty write because T_1 committed.

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

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

This is a strict schedule.

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

No dirty read

So, cascadelessness

It is not a strict schedule.

This is a dirty write (lost update problem)
(writing uncommitted data)

So, first commit, then write.

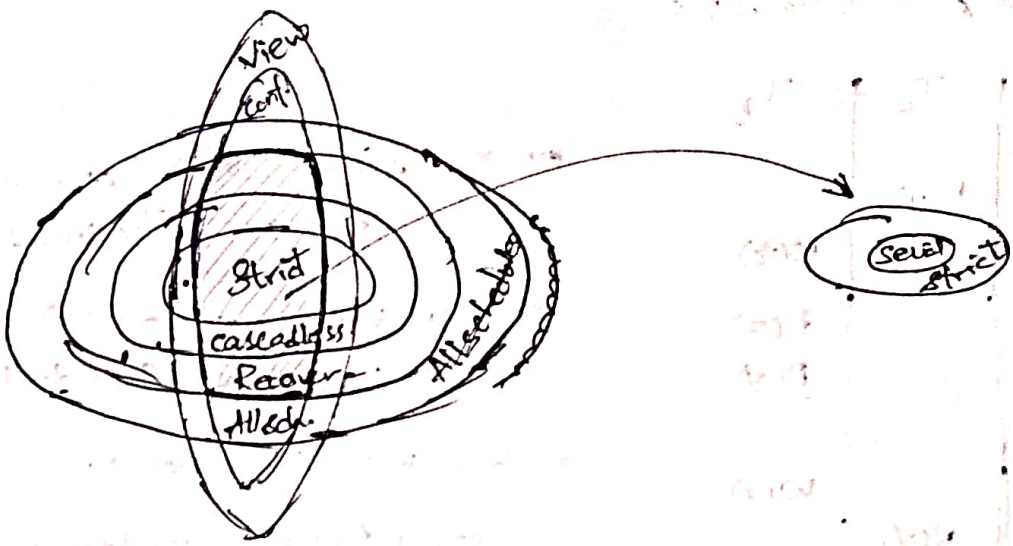
All serial schedules are strict schedules but not vice verse.

T_1	T_2	T_3
$R_1(X)$		
	$R_2(X)$	
$R_1(Z)$		
		$R_3(Z)$
		$R_3(Y)$
$W_1(X)$		
C_1		
		$W_3(Y)$
	$R_2(Y)$	C_3
	$W_2(Z)$	
	$W_2(Y)$	
	C_2	

It is non-serial, but strict schedule (No dirty read & no dirty write)

Relation among all type of schedules

We cannot work on irrecoverable & view schedules.



1) S: $R_2(B)$ $W_2(A)$ $R_1(A)$
$R_3(A)$ $W_1(B)$ $W_2(B)$ W_3B.

If is not conflicting serializable.
But there is blind write.

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

→ first writing so T_1 is not initial read.

	A	B
IR	No initial read	T_2
FW	T_2	T_3
F/M R.		X

(2).

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

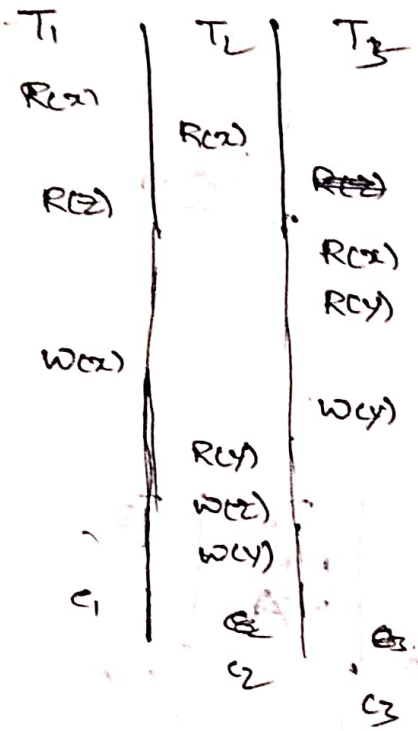
	A	B	C
IR	T_1	T_3	T_2
FW	T_1	T_2	T_3
F/M R.	—	—	—

Not viewserializable.

$T_1 T_2 T_3$ | $T_2 T_3 T_1$ | $T_3 T_1 T_2$
$T_1 T_3 T_2$ | $T_2 T_1 T_3$ | $T_3 T_2 T_1$

Recoverability examples :-

(3) S: $R_1(x), R_2(x), R_1(z), R_3(x), R_3(y), W_1(x), W_3(y), R_2(y), W_2(z), W_2(y), C_1, C_2, C_3$;



on x, after $W_1(x)$, no read
so, no dirty read

on y, $W_3(y) \rightarrow R_2(y)$
so, dirty read.

on z, no dirty read.

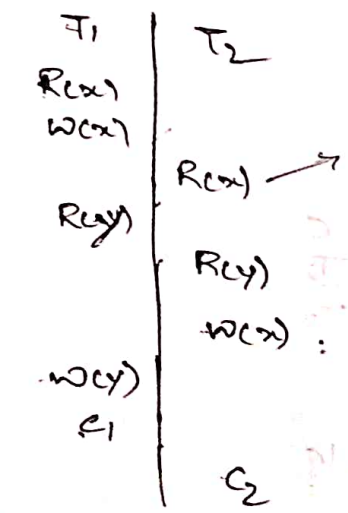
commitment order ✓,
So, recoverable

~~Ex~~ Cascading rollback (∵ dirty read)

Not a strict schedule (∵ dirty read & dirty write).

---

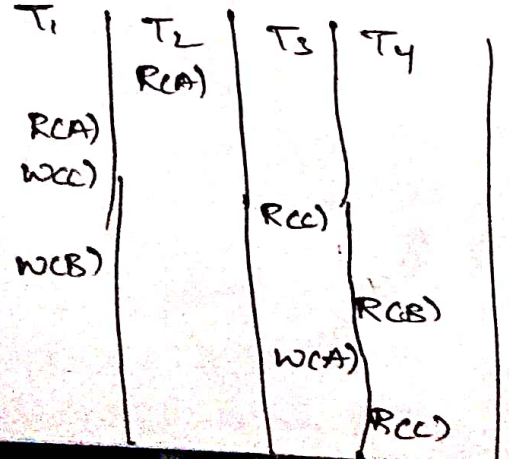
(4) S: $R_1(x), W_1(x), R_2(x), R_1(y), R_2(y), W_2(x), W_1(y), C_1, C_2$;

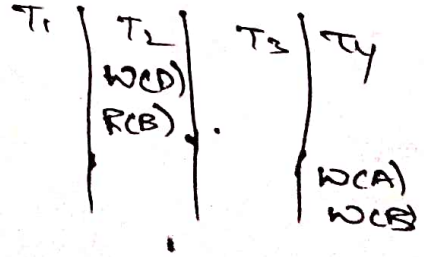


on x, dirty read.
T_2 is reading from T_1.
So, T_1 commit first (✓).
So, recoverable.
Cascading rollback (∵ dirty read)
Not a strict schedule (∵ dirty read).

---

(5) $R_2(A), R_1(A), W_1(C), R_3(C), W_1(B), R_4(B), W_3(A), R_4(C), W_2(D), R_2(B), W_4(A), W_4(B)$

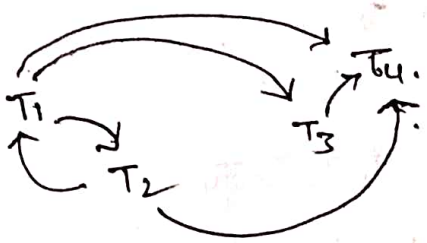




on C: $W_1(C) \rightarrow R_3(C)$ → There is a dirty read & don't have a commitment order, so can't conclude

Since there is a dirty read, so cascading roll back.

Since there is dirty read & dirty write, so, not a strict schedule.



So, cyclic, so, not a CSS.
May be VSS.

	A	B	C	D
IR	T_1	T_2		=
FW	T_4	T_4		T_2
IMR		$T_1 \rightarrow T_4$	$T_1 \rightarrow T_3$	
		$T_1 \rightarrow T_2$	$T_1 \rightarrow T_4$	

T_1

=

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

to make T_3 as initial read
So, $W_2(B)$ comes first

	A	B	C
IR	$T_1 T_2$	$T_3 T_2$	T_2
FW	T_1	T_2	T_3
IMR	$T_2 \rightarrow T_1$	$T_3 \rightarrow T_2$	

$T_3 \rightarrow T_2$, $T_2 \rightarrow T_3$ contradicting each other

~~T_3 T_2 T_1~~

Not VSS

=

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

	A	B
IR	✗	T_2
FW	T_2	T_3
IMR	$T_2 \rightarrow T_1$	✗
	$T_2 \rightarrow T_3$	

$T_2 \rightarrow T_1$
$T_2 \rightarrow T_3$.

$T_2 \rightarrow T_3$

$T_2 \rightarrow T_1, T_3$
$T_2 \rightarrow T_3 \rightarrow T_1$

✓ V.SS

T_1	T_2	T_3	T_4
	R(A)		
R(A)			
W(C)			
		R(C)	
W(B)			
			R(B)
		W(A)	
			R(C)
	W(B)		
	R(B)		
			W(A)
			W(B)

	A	B	C	D
IR	$T_1 T_2$	∞	∞	∞
FW	T_4	T_4	T_1	T_2
IMR	∞	$T_1 \to T_4$, $T_1 \to T_2$	$T_1 \to T_3$, $T_1 \to T_4$	∞

$T_1 \to T_4$, $T_2 \to T_4$, $T_2 \to T_4$

$T_1\ T_2\ T_3\ T_4$

It is view Serializable.

=

T_1	T_2	T_3	T_4	T_5
R(A)				
		R(D)		
W(B)				
	R(B)			
		W(B)		
			R(B)	
	W(C)			
				R(C)
			W(E)	
				R(E)
				R(A)

	A	B	C	D	E
IR	$T_1 T_5$	∞	∞	T_3	∞
FW	∞	T_3	T_2	∞	T_4
IMR	∞	$T_1 \to T_2$, $T_3 \to T_4$, $T_1 \to T_3$ (or $T_2 \to T_3$)	$T_2 \to T_5$	∞	$T_4 \to T_5$

$T_1\ T_2\ T_3\ T_4\ T_5$.

It is view Serializable.

=

T_1	T_2	T_3	T_4
	R(A)		
R(A)			
W(C)		R(C)	
W(B)			
			R(B)
		W(A)	
			R(C)
	W(D)		
	R(B)		
			W(A)
			W(B)

	A	B	C	D
IR	$T_1 T_2$	∞	∞	∞
FW	T_4	T_4	T_1	T_2
IMR	∞	$T_1 \to T_4$, $T_1 \to T_2$	$T_1 \to T_3$, $T_1 \to T_4$	∞

$T_1 T_2 \to T_4$.

$T_1\ T_2\ T_3\ T_4$.

$T_1\ T_3\ T_2\ T_4$.

Protocols that ensures to generate schedule which satisfy this property specifically CS.

$T_1, T_2 \cdots T_n$. $n = 10000, \ldots$

↓↓

Protocol. → It generates a NSS following the properties - CS; VS; R.

Protocols

deadlock.

P_1	P_2
CS	CS
R	R.
deadlock	

prefer P_2.

A lock which is dead, Permanent lock. never gets unlocked.

Program → Passive entity.
Paint → active entity.

handles 2 processes
dual core, quad core (4), octa core (8).

Every process needs some resources.

& P_1 needs a resource A, which is already given to P_2
P_2 " " " B " " " " P_3

P_1 should release A & then P_2 works & release B. & ...

This is deadlock.
(Ctrl + Alt + Del) to kill all the processes.

deadlock → Infinite Wait.

starvation (it cannot complete)

starves for CPU.

preempt (forceful)

for deadlock there must be Mutual Exclusion.
(all are Necessary & sufficient)
Circular Wait.
Wait & Hold
No Premition

kill lower priority processes, not high priority processes.

Protocols:-

1) Timestamping Protocol (2) Thomas Write Rule.
3) locking protocols 2PL (Two phase locking) — Basic 2PL, Conservation 2PL, Rigorous 2PL, Strict 2PL.
Validation Protocols.