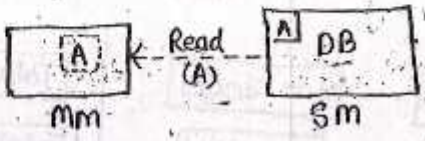
TRANSACTION & Concurrency Control

Transaction: set of logically related operations which is used to perform some "unit of works"

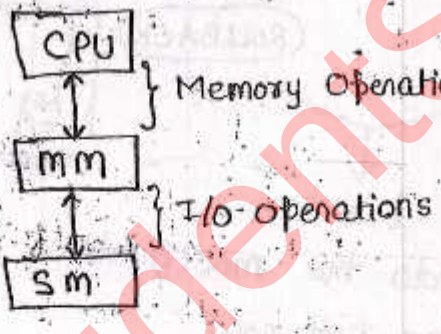
↳ "Set of logically related operations."

» **READ(A):** Accessing data item A from DB to programmed variable.



» **Write operation:** Updating the details into DB.

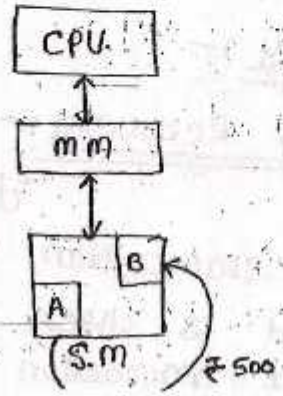
Read/Write operations and I/O operations



» Assume some transaction T, t/f Rs 500 from A/c A to A/c B.

Trans T : Transfer ₹ 500 from A to B

```
Begin Trans
    Read A
    A = A - 500
    Write A
    Read B
    B = B + 500
    Write B
Commit
```

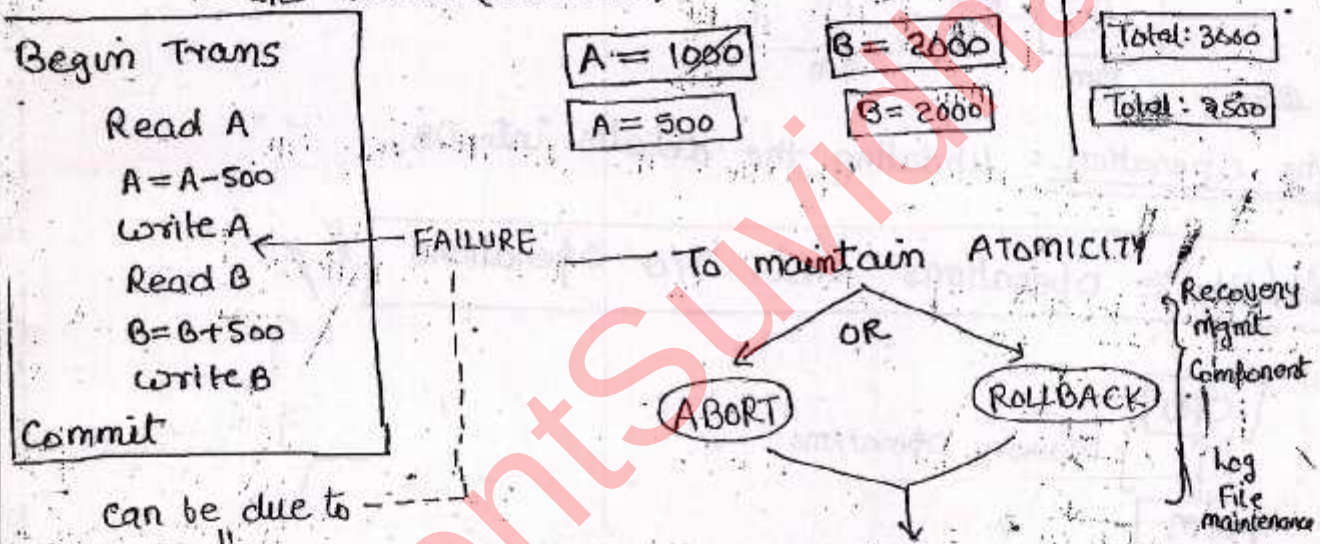


» Commit () → is a memory operation that specify successful completion of transaction.

» To maintain INTEGRITY, transaction should satisfy following conditions.

(ACID Properties)

Atomicity :- Transaction should execute all the operations successfully (commit) or none of them.
└ s/w modules (automated)

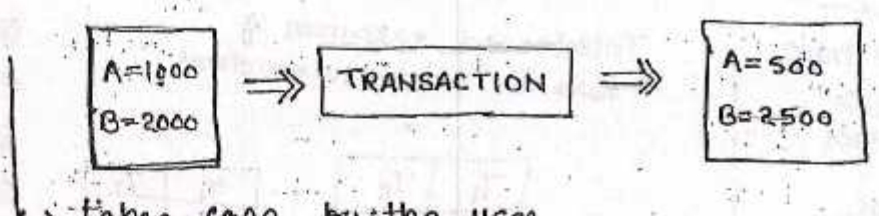


Log file is not permanent

Recovery :-

Shadow Recovery :- Whenever a transaction begins, it maintains a log file corresponding to transaction, if transaction fails, log file searched to check the previous fresh point and if transaction commit, log file deleted.

Consistency :- DB should be consistent before and after the transaction execution.



→ taken care by the user.
→ if failure occur due to some major problem (disk crash) the automated recovery may not be possible, in such cases user takes care of data consistency.
(Eg. using fresh disk etc)

Factors affecting consistency:

1. Atomicity
2. Isolation
3. Durability

ISOLATION :-

Schedule : Time order sequence of two or more trans.

Trans T_1 : T/f 500 Rs from A to B
Trans T_2 : Display total amount

```
Begin Trans:
    Read (A)
    Read (B)
    display (A+B).
Commit
```

T_1

```
R(A)
A=A-500
W(A)
R(B)
B=B+500
W(B
Commit
```

Schedule means these two transactions can execute simultaneously.

Schedule

Serial schedule

Execution of one trans. possible only after completion of other

Execution of Trans., one after other

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

or

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

T_1, T_2 Two Transactions then T_1 followed by T_2 / T_2 " " T_1

Two Serial schedule

⇓

In General:

n transaction then n! Serial schedule possible

Concurrent schedule

Interleaved execution of two or more transactions

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

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

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

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

Total no. of schedules possible with 2 trans.

$= \frac{6!}{4! \times 2!}$ (6 operations; 4 of T_1, 2 of T_2)

1 order (T_1) — 2 of order (T_2)

R(A) ___ W(A) R(B) ___ W(B) ⇒ $^6C_4$

R(A) R(A) W(A) R(B) R(B) W(B)

$^6C_4$ ✓

✗ It is not $^6C_4 \times 4! \times {}^2C_2 \times 2!$

T_1 ∵ R(A) (W(A)) R(B) W(B) — 1 order only

T_2 R(A) R(B) — 1 order only

If m and n operations of Trans T_1 and T_2 Then

How many Concurrent schedules are possible

⇒ $^{m+n}C_m$ or $^{m+n}C_n$

If m, n, and p operations of Trans T_1, T_2, T_3

$$\#\text{ Concurrent Schedules} = {}^{m+n+p}C_m * {}^{n+p}C_n * {}^{p}C_p \;\#$$

out of (m+n+p), m people made to sit

$$= {}^{m+n+p}C_m * {}^{n+p}C_n * 1 \;\#$$

out of remaining n+p, n made to sit

- There is no inconsistency if schedule executed in serial order.
- Serial order leads to less throughput.
- To avoid this problem, we go for concurrent schedule. CPU ideal time now is reduced, hence leads to better throughput.

A = 1000
B = 2000
3000

T_1	T_2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A) 500
	R(B) 2500 3000

Consistent

T_1	T_2
R(A)	
W(A)	
	R(A) 500
	R(B) 2000 2500 X
R(B)	
W(B)	

Inconsistent Concurrent Schedule

Concurrency Control Component should not allow any inconsistent schedule.

└ How??

└ How it come to know about a particular schedule consistent or inconsistent??

⇒ Serial Schedule always results in consistency. (mapped)

If any concurrent schedule reducable or is similar to any serial schedule (equivalent) then it is consistent.

If any concurrent transaction T_i equal to any serial schedule $(T_1 \to T_2)$ or $(T_2 \to T_1)$ it is said to follow ISOLATION.

ELSE, it is not a consistent concurrent schedule and may violate Isolation (produce inconsistent result).

⇒ Isolation means concurrent execution of two or more transactions should be equal to any serial schedule. Then only we say transaction is isolated.

4) DURABILITY :- Database should be able to recover under any case of failure.

» Redundancy introduced to enforce durability

» Both hw and sw responsibility " "

» RAID : Redundant Array of Independent Disk.

RAID-0 — No redundancy, only one copy | ↑% of risk | updation cost ↓

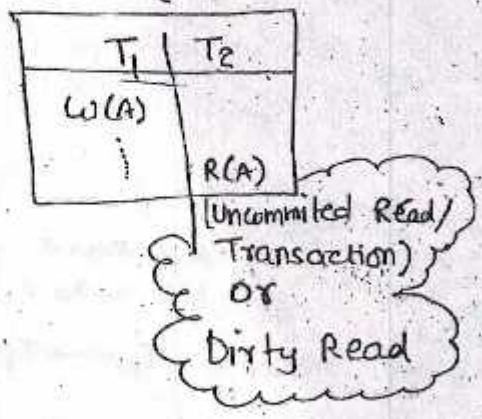
RAID-1 — Mirroring — one fail other used | update at 2 places ↑ | Cost doubles

RAID-2 —

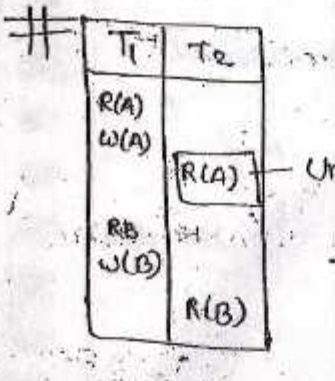
Problems Because of Concurrent Execution :-

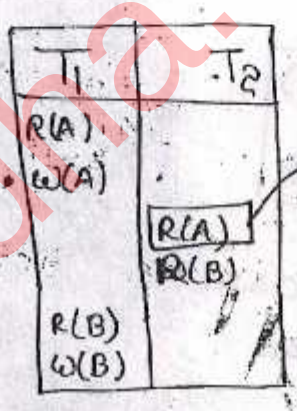
(I) WR Problem :

(Read after write operation)

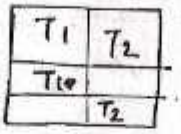


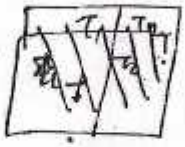
Uncommitted Read : Transaction T_2 reads data item A that is updated by Un_commited transaction.





⇒ Uncommitted read may create problem sometimes but are effective in awareness to people.

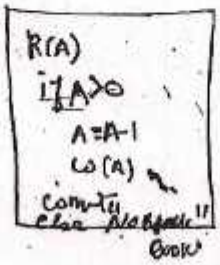




(II) RW Problem (Write After Read)

Ex :- Library DB

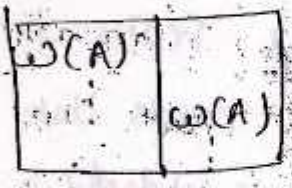
A : # of copies of DB book



(III) Write-Write Problem (Blind write problem)

Assume Two Transaction are there

Before commit [illegible] is already updated by [illegible]



T1 :

A, B are no
A B are 20
of T2 → T1

occuring concurrent

How assume
execution

In T1 followed by T2 (T1 → T2), final updation done by T2

Problems related to concurrent execution

Write Read Problem : (Read after write problem)

T_1	T_2
W(A)	
	R(A) (Uncommitted Read) or (Dirty Read)

T_1	T_2
W(A)	
commit	
	R(A) (committed Read)

Uncommitted Read may not create inconsistency

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

Consistent Schedule

Uncommitted Read ..

Transaction T_2 reads A that is updated by uncommitted transaction T_1. Uncommitted read sometimes results in inconsistency.

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

Inconsistent Schedule

Read Write Problem

Write after Read.

T_1	T_2
R(A)	
	W(A)

If serially executed no inconsistent result.

Ex Library Database

A: # of copies of DBMS books

Trans T_1

```
R(A)
if (A > 0)
    A = A - 1
    W(A)
    commit
else
    No Books
```

A = ~~10~~ 9

T_1	T_2
R(A)	
if (A>0)	
{ A=A-1	
	R(A)
	if (A>0)
	{ A = A+1
	W(A)
	Commit
W(A)	}

ie Before commit of T_1, T_2 perform write operations.

Write Write Problem

T_1	T_2
W(A)	
	W(A)

Transaction T_2 updating data item(A) that is already updated by uncommitted Trans T_1.

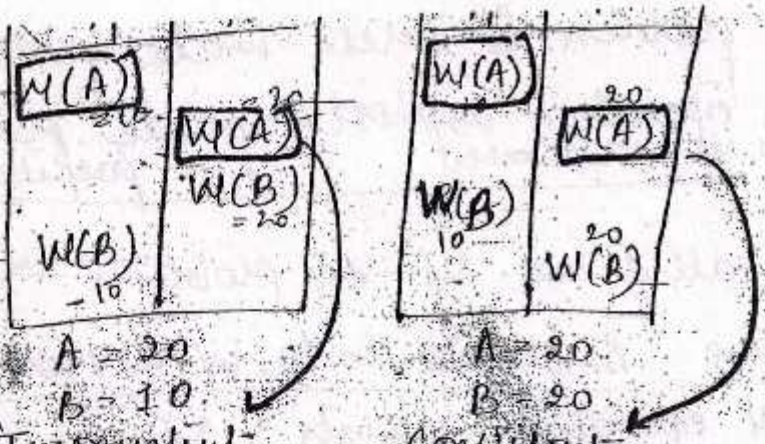
$\Rightarrow T_1$: Set A, B as 10 (T_1: W(A), W(B))

T_2: Set A, B as 20 (T_2: W(A), W(B))

$T_1 : T_2 \Rightarrow$ A = B = 20. T_1 followed by T_2

$T_2 : T_1 \Rightarrow$ A = B = 10 T_2 " " " T_1

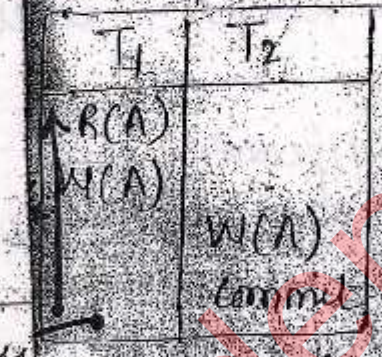
W-W may or may not suffer from consistency



Schedule

- Inconsistent
  - RW (or)
  - WR (or)
  - WW problem.
- Consistent
  - No problem

WW problem (Not equal to any serial Schedule)



(1) RW Problem, WR Problem, WW Problem } Concurrent Sch not equal to any serial execution

(2) Other than above problems we have lost-update problem

A = 0

No Concurrent execution Problem because it is $T_1 \to T_2$ serial Schedule

T_1 log
- $A_{old} = 0$
- $A_{New} = 10$

T_2 log
- $A_{old} = 10$
- $A_{New} = 20$

A = 20

If failure occurred → Rollback occurs

Because of T_1 failure, updation of T_2 is lost.

Problem because

Every lost update problem is write-write problem.
Simultaneous write-write operation is lost update problem and schedule Non-serializable possible...

Simultaneous operation
W-W is called as W-W problem → NO

Two conditions required to be W-W problem

1) WW operation exists &&
2) 'S' Schedule not equal to any Serial schedule.

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

← lost update possible but not W-W problem

A = 20
B = 20
(Consistent)

→ W W operation ✓
→ & not equal to serial schedule result ✓

⇓

Will be W-W problem if Schedule gives inconsistent result.

** Classification of Schedule

Schedule is said to be correct only if it satisfies

(1) Recoverability (&&)
(2) Serializability

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

which is a Consistent Serial Schedule

Consistent or Inconsistent

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

consistent

When we say

RW Problem
WR "
WW "

} exists — it means

Concurrent execution of transactions, not equal to serial execution.

(1) Lost Update Problem :-

A=0 — Initially

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

T_1 log
Aold = 0
Anew = 10

T_2 log
Aold = 0
Anew = 20

→ T_2 commit then T_2 log lost.
If T_1 commits

rollback

if failure

⟹ Transaction T_2 lost updation becoz if T_1 failure.

⟹ Problem occurs becoz simultaneous write-write operations.

Classification of Schedules:

Basis
- (1) Recoverability &&
- (2) Serializability

} any correct schedule should always follow two conditions.

↳ If Protocols must satisfy both recoverability && serializability.

Recoverability :-

IRRECOVERABLE: Rollbacking of committed transaction is irrecoverable.

↳ It is not possible to recover a committed transaction. Because once a T commit, its log no more exists.

↳ Why rollback of transaction required???

DEPENDENCY :-

(1)

T_1	T_2
W(A)	
⋮	R(A)
Commit	

→ Updated value of A, read before commit. So it is Uncommitted Read.
Hence T_2 depend on T_1

(2)

T_1	T_2
W(A)	
Commit	
	R(A)

→ A value read after commit, no dependency there b/w two transacks. (No dependency)

(3)

T_1	T_2
W(A)	
Rollback	
	R(A)

→ A value modified [A=0 to 10], but is Rollbacked soon to the back. ~~read~~ older value i.e (A=0) only (No dep)

(4)

T_1	T_2
W(A)	
	W(A)

→ No modification depend on other
→ A modified by T_1 then updated by T_2 No problem. (No dep)

(5)

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

⇒ Here it is not uncommitted read, uncommited read, specify modification by one transaction and reading by another. (No dep)

(6)

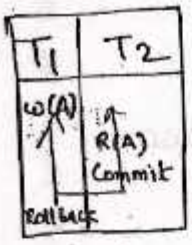
T_1	T_2
W(A)	W(B)
	R(A)
R(B)	

⇒ T_1 depends on T_2
T_2 depends on T_1
} Dependency.

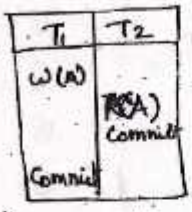
Possibility to get Irrecoverability :-

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

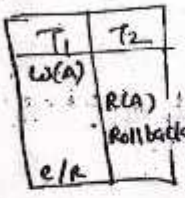
Commit Rollback



IRRECOVERABLE



IRRECOVERABLE



Recoverable

⟹

How to get Recoverable schedule :-

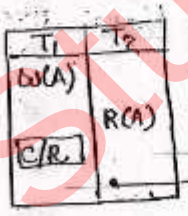
Recoverable Schedule :-

(Rollbacking of Uncommitted Transaction).

[Restriction to put on schedule, to make it Recoverable only.

If T_2 depend on T_1, commit of T_2 should be delayed until commit/Rollback of T_1.

↳ If T_1 rollbacked / commit → T_2 also can be rolled back / commit.



⟹ " If Tran T_2 reads dataitem A that is updated by T_1 and T_2 commit operation should be delayed until C/R of T_1." Then T_1 is RECOVERABLE

NOTE:

» Recoverable schedule may / may not be free from

D

- RW Problem
- WR "
- WW "
- Lost Update Problem

Illustration:

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

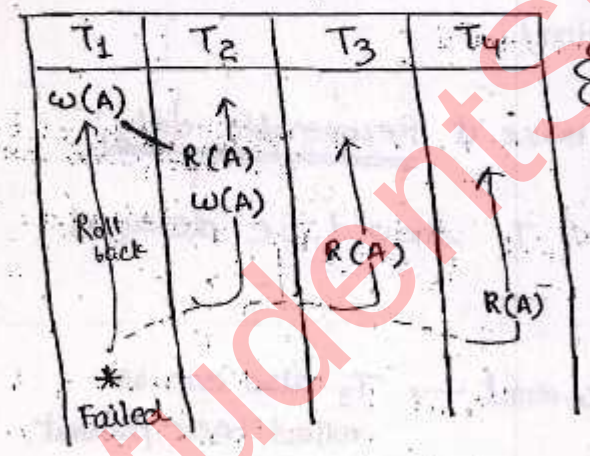
⇒ In this scenario, T_2 depends on T_1 i.e Uncommitted read.

Commit of T_2 delayed, till commit/Rollback of T_1. Thus schedule is **RECOVERABLE**

BUT

It suffer from **WR PROBLEM**

② Cascading rollbacks are possible:-



only T_1 failed, but due to dependency among transactions all T_1, T_2, T_3, T_4, .. rollbacked.

Whatever the work done by other transaction is lost. Hence a wastage of CPU execution time and I/O cost.

⇒ Recoverable schedule may cause cascading rollbacks.

Cascadeless Rollbacking Recoverable schedule :-

(Recoverable schedule && No cascading Rollbacks)

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

Trans. T_2 should not be allowed to read until modified value committed / Rollbacked.

⇒ Not allowing any uncommitted Reads

⇒ If one Trans (T_1) updates some value by, that value should not be read (allowed to read) by other transaction until former transaction commit / rollback.

⇒ "If Trans. T_1 updates a data item A, then other transaction T_2 is not allowed to read the data item until commit / rollback of T_1"

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

✓ Cascadeless Rec. Sch.

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

✓ → Cascadeless Rollback Recoverable schedule.

NOTE :-

① Cascadeless Rollback Recoverable schedule is "free" from

Write Read (WR) Problem.

But RW, WW, Lost update problem may be possible.

- RW, WW → due to non-existence of equivalent serial schedule for Concurrent schedule → These problems controlled by
- Lost update → Concern of Recoverability

Ex:

T_1	T_2
w(A)	
	R(A)

X

But

T_1	T_2
w(A)	
C/R	:
	R(A)

✓

or

T_1	T_2
w(A)	
	w(A)

✓

↳ Whenever WW possible ∃ possibility of Lost Update problem.

⇓

To avoid that ⟹

⇓

i.e

T_1	T_2
w(A)	
C/R	
	w(A)/R(A)

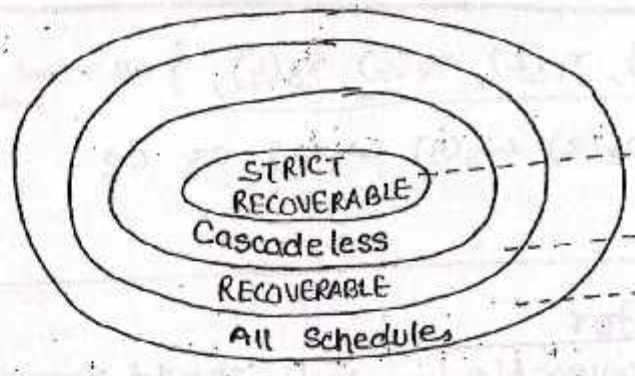
If some Trans. T perform write on dataitem A, then other Trans. T' not allowed to perform read of A or write of A (w(A)/R(A)) untill Commit or Rollback of Trans. T.

↳ STRICT RECOVERABLE SCHEDULE.

- Removal of Lost Update problem leads to removal of WW problem also.
- In STRICT RECOVERABLE SCHEDULE, ∄ are
  - → Recoverable Schedule
  - → ∃ No cascading Rollbacks
  - → No WR Problem
  - → ∃ No Lost Update Problem
  - → ∃ no WW problem

BUT RW Problem is still possible.

Classification of Schedules



Q: Which is TRUE ??

a) Irrecoverable
b) Recoverable but not cascadeless
c) Cascadeless but not strict
d) strict schedule

(1) S1 : $r_1(x)\ r_2(z)\ r_3(z)\ r_3(x)\ r_3(y)\ w_1(x)\ w_3(y)\ r_2(y)$
$w_2(z)\ w_2(y)\ c_1, c_2, c_3$

S1 : Schedule
c_1 : Commit 1
c_2 : Commit 2
c_3 : Commit 3

⇒ 3 Trans. involved.

T_1	T_2	T_3
R(x)		
	R(z)	
R(z)		
		R(x)
		R(y)
w(x)		
		w(y) → ∃ Uncommitted Read
	R(y)	
	w(z)	
	w(y)	
Commit		
	Commit	
		Commit

Hence it is **Irrecoverable**

(ii) S2: $r_1(x), r_2(3), r_3(x), r_1(z), r_2(y), r_3(y),$ } all read no problem

$w_1(x)$ C1 $w_2(z)$ $w_3(y)$ $w_2(y)$ C3 C2

T_1 Recoverable T_2

$w(A)$	
	$R(A)$

→ For Recoverable → T_1 should Commit before T_2

→ ∄ no Write Read sequence, so it is
- ✓ — Recoverable and
- ✓ — Cascadeless

If WW sequence (write after write) also not there then it will become STRICT Recoverable (Here it is not)

→ For Cascadeless → Updation then Commit then Read

T_1	T_2
$w(A)$ C/R	
	$R(A)$

∄ this is not followed then it is not Cascadeless.

S2: → Recoverable ✓
Cascadeless ✗
STRICT ✗

(iii) S3: $r_3(x)$, $r_2(x)$, ~~$w_3(y)$, $w_2(y)$, $r_2(y)$, c_1, c_2~~
$w_3(x)$, $r_2(x)$, $w_1(y)$, $r_2(y)$, $w_2(x)$, c_3, c_1, c_2
↳ Recoverable ✓
↳ Cascadless ✗

(iii) S4: $r_1(x)$, $r_2(x)$, $w_1(y)$, $w_2(y)$, $r_2(y)$, c_1, c_2
↳ Rec ✓
↳ Cas ✓
↳ Strict ✗

Schedule

Classification Based on Serializability

Serial Schedule	Serializable Schedule
* No Concurrency	* Concurrency allowed * Concurrent schedule should be equivalent to any serial schedule execution
* Every Serial Schedule are correct/Consistent	* All serializable schedules are consistent * No [RW, WR, WW] problems not possible. but cascading rollback (irrecoverably) and lost update may be possible.

NOTE

⇒ If a schedule is BOTH

RECOVERABLE and SERIALIZABLE, then.

⇒ No WR, RW, WW Problem

⇒ No Irreversibility

BUT

⇒ Lost Update
⇒ Cascading Rollback } are possible

⇒ STRICT RECOVERABLE + SERIALIZABLE

GOAL OF CONCURRENCY CONTROL.

GOAL OF CONCURRENCY CONTROL TECHNIQUE

STRICT RECOVERABLE AND SERIALIZABLITY

only suffer from.

RW problem — which is ultimately overcommed by Serializability.

6th NOV 2011

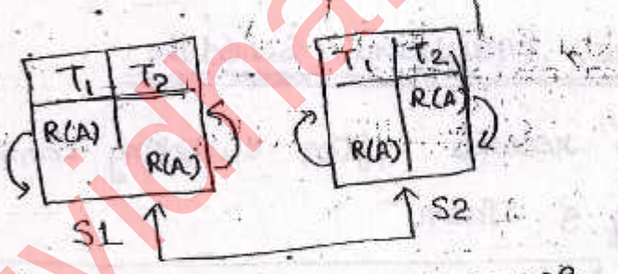
Schedules based on SERIALIZABILITY

① Conflict Serializable Schedule :-

⇒ Conflict pair :- On swaping operations of schedule equivalent. Then it is called non-conflict pair.

S1 : T_1: R[A] T_2: R[A]

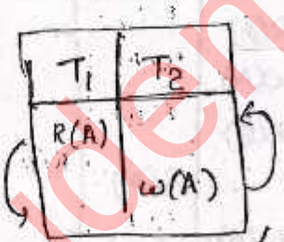
S1 ≡ S2



Both schedules equal so we call them NON-CONFLICT PAIR.

Initial Read ⇒ Read operation from Database.

S1 :



S1 ≠ S2

—— CONFLICT Pair.

Conflict Pairs:-

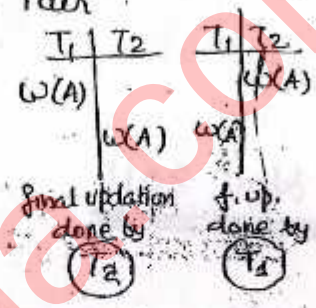
(1) T_1 :

(2) T_1 :

(3) T_1 :

(4) T_1 : $W(A)$ $T2$: $W(A)$ ——— Conflict Pair

(5) T_1 : $R(A)/W(A)$ $T2$: $R(B)/W(B)$



Pair of operations is said to be conflict only if

(1) Atleast one write operation and

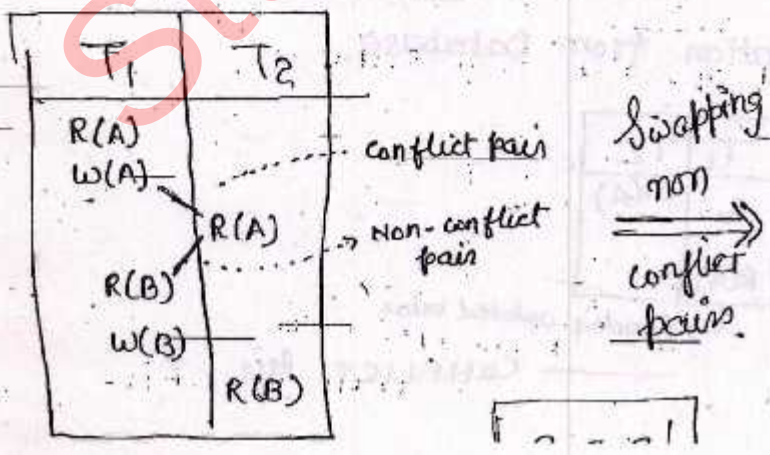
(2) Both operation (Read/write) should be on same data item. and

(3) Both operation should not be from same transaction.

Conflict Equivalent Schedule:-

S' results after swapping consecutive non-conflict pairs of S. Then

S' and S are conflict equivalent #



Q: 11 Pg 43 Which one conflict equi. to which ???

S1:

$R_2(A)$
$W_2(A)$
$R_3(C)$
$W_2(B)$
$W_3(A)$
$W_3(C)$
$R_1(A)$
$R_1(B)$
$W_1(A)$
$W_1(B)$

swapping non conflict pair

S2:

$R_3(C)$
$R_2(A)$
$W_2(A)$
$W_2(B)$
$W_3(A)$
$R_1(A)$
$R_1(B)$
$W_1(A)$
$W_1(B)$
$W_3(C)$

swapping non conflict pair

S3:

$R_2(A)$
$R_3(C)$
$W_3(A)$
$W_2(A)$
$W_2(B)$
$W_3(C)$
$R_1(A)$
$R_1(B)$
$W_1(A)$
$W_1(B)$

By swapping non conflict pairs of S1, we get S2
so S1 ≡ S2 or S1 = S2

S2:

$R_3(C)$
$R_2(A)$
$W_2(A)$
$W_2(B)$
$W_3(A)$
$R_1(A)$
$R_1(B)$
$W_1(A)$
$W_1(B)$
$W_3(C)$

non conflict pairs swapped

S3:

$R_2(A)$
$R_3(C)$
$W_3(A)$
$W_2(A)$
$W_2(B)$
$W_3(C)$
$R_1(A)$
$R_1(B)$
$W_1(A)$
$W_1(B)$

conflict pairs swapped. Hence $S_2 \neq S_3$

S1

T_2	T_3
$W_2(A)$	
$W_2(B)$	
	$W_3(A)$

final updation (A) by T_3

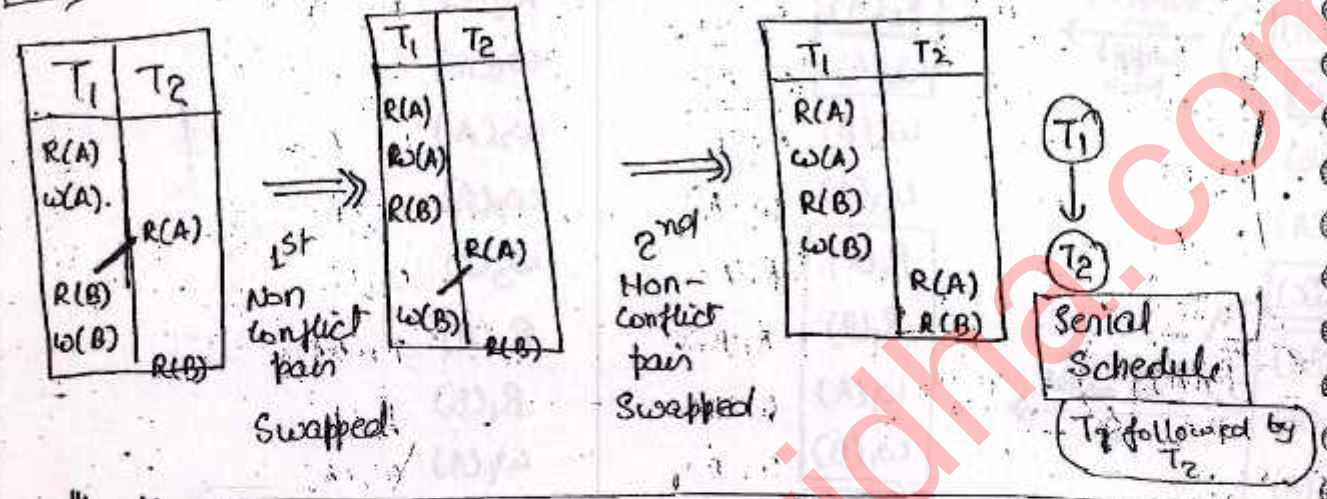
S2

T_2	T_3
	$W_3(A)$
$W_2(A)$	
$W_1(B)$	

Final Updation (A) by T_2

Conflict pair → Conflict Schedule

⇒ Given schedule is Conflict serializable schedule if ∃ conflict equivalent serial schedule of given schedule.



Aise (non-conflict) pairs ko swap karte-2 agar hme koi serial schedule mil jae then we say that given schedule is.

Conflict-serializable

cycle?

To check Conflict Serializability

Precedence Graph

Graph : (V, E)

Vertices : Transactions in the schedule.

Edges : $T_i \longrightarrow T_j$ $i \neq j$

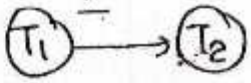
only if ∃ conflict pair with T_i preceded then T_j.

RW	T_i : R(A)	T_j : W(A)
WR	T_i : W(A)	T_j : R(A)
...	...	T_j : W(A)

If the Precedence Graph Contain
No Cycle
→ then it is conflict serializable

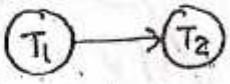
#
1.

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



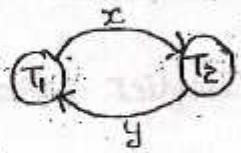
#
2.

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



3.

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



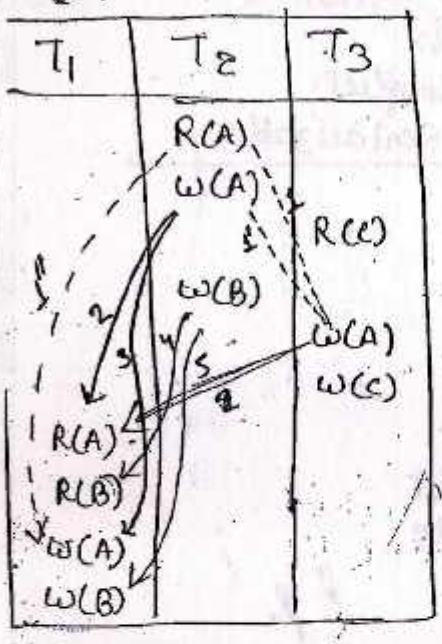
— Not conflict serializable

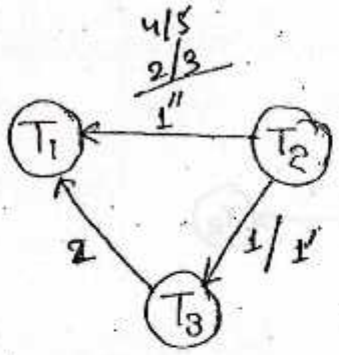
NOTE:- TESTING condition :-

- If precedence graph ACYCLIC, then schedule is Conflict serializable.
  - ⇒ Equivalent serial schedule is Topological sequence of acyclic precedence graph.
- If precedence graph cyclic, then schedule not **NOT Conflict serializable**

Q11

S1

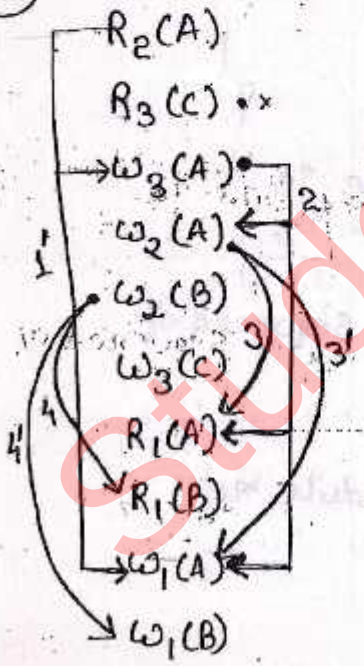


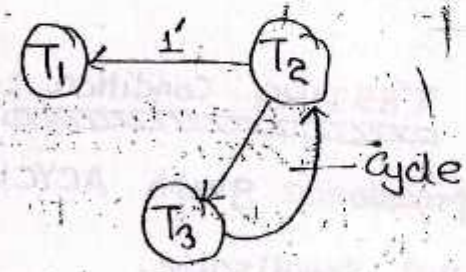


Acyclic graph

↳ It is Conflict Serializable

S3





Not Conflict Serializable Schedule

Topological Sequence

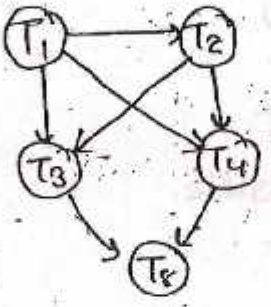
(Traversing Acyclic Graph)

If no vertex with indegree = 0 then there exis CYCLE in GRAPH

[1] Visit Vertex (v) indegree "0" and delete vertex 'v' and outdegree edges of "v"

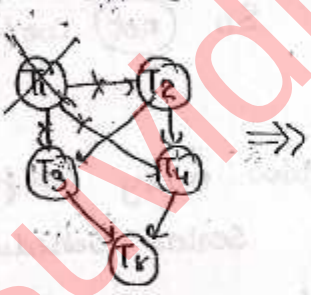
[2] Repeat (1), till graph becomes empty.

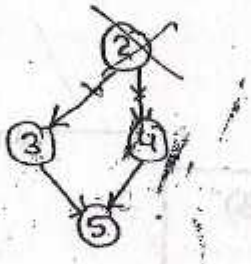
Ex

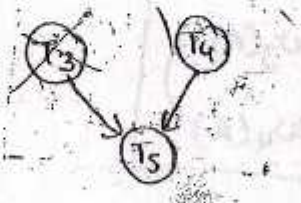


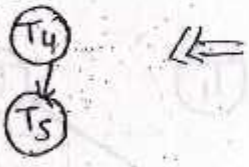
Start with indeg(0) = T1

Delete T1 & outdegree of T1

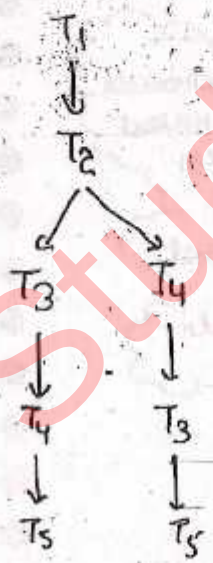








Schedules



Two schedule possible

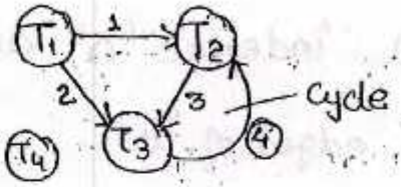
(1) $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_5$

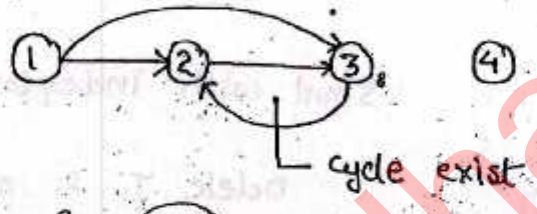
(2) $T_1 \rightarrow T_2 \rightarrow T_4 \rightarrow T_3 \rightarrow T_5$

Q- Check whether conflict serializable or not ??

S

$R_4(A)$
$R_2(A)$
$R_3(A)$
$w_1(B)$
$w_2(B)$
$w_3(B)$
$w_2(B)$

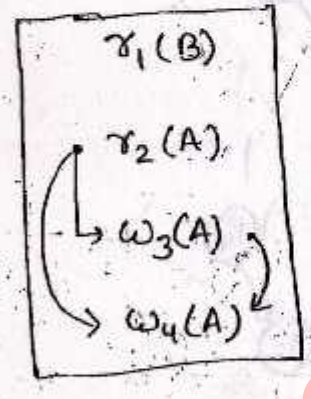


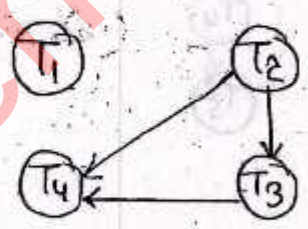


So (not) conflict serializable

Q: S ⟶ Q: How many conflict equivalent serial schedule possible ??

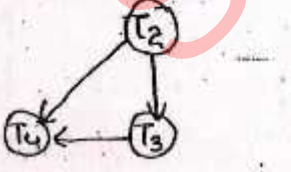
(# graph cyclic = 0 conflict schedule)

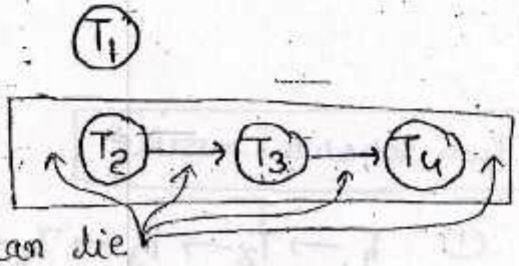




This schedule is equivalent to four conflict equivalent serial schedule.

Sequence = ?





So possible

T1 can die

Sequences are:

(1) $T_1\ T_2\ T_3\ T_4$
(2) $T_2\ T_1\ T_3\ T_4$
(3) $T_2\ T_3\ T_1\ T_4$

#

	T_1	T_2
① →	X(A)	
	R(A)	
	W(A)	
X(B)	U(A)	S(A)
		R(A)
	~~U(B)~~	S(B) ← denied already locked.
		R(B)
	R(B)	
	W(B)	

Not allowed by 2PL

Explaintion

Q

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

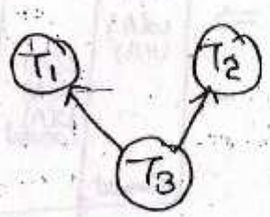
Allowed to execute by 2PL

Q Allowed by 2PL or not??

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

X → not allowed

This schedule not allowed by 2PL.



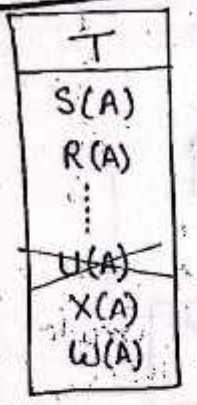
Conflict Serializable even then not 2PL

T_3 (indeg=0) → T_1 → T_2

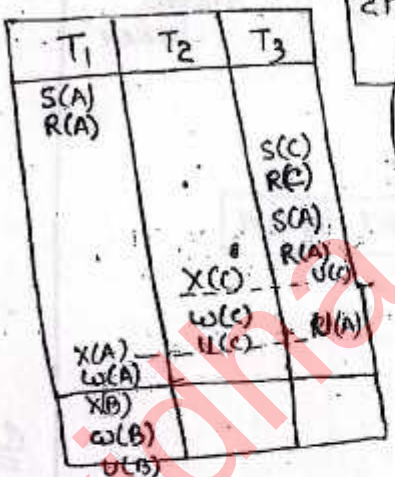
$T_3 \rightarrow T_2 \rightarrow T_1$

Lock Upgrading Techniques:

Shared Lock can be upgraded to Exclusive Lock by requesting 'X' lock without unlock of shared Lock.



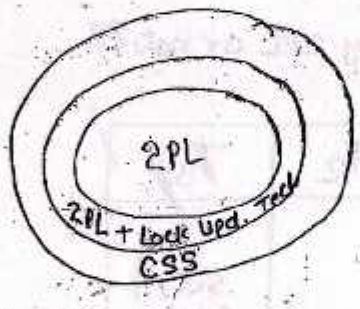
E.g



2PL + updating Lock Technique

↓ allowed

higher degree of conc. than 2PL.



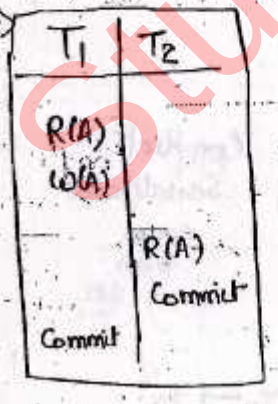
NOTE

S/X && 2PL

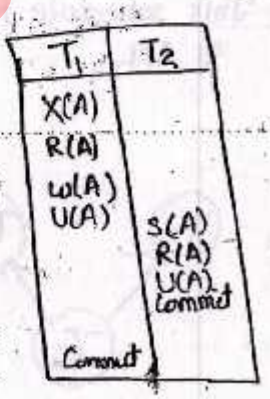
↳ always ensures CSS

↳ may not be free from irrecoverability.

X



⇒



Allowed by 2PL

↳ so definitely serializable

But is IRRECOVERABLE

∵ T_2 commit before T_1

T_1 updating data item A. Hence to be recoverable, it should commit before T_2. But here T_2 is committing before T_1. Hence Irrecoverable

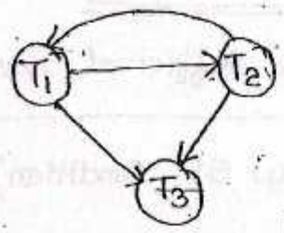
Imp

Conflict serializable = ??

Q

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

Sol^n



→ Precedence graph contain cycle. So

⇓

Schedule Not Conflict SS

⇓ equivalent Serial schedule

Q

	T_1	T_2	T_3
S_1	R(A)		
	W(A)		
		W(A)	
			W(A)

⇒ For S_1, equal serial schedule $T_1 \rightarrow T_2 \rightarrow T_3$

For S ∃ S1 which is serial schedule
So S is SERIALIZABLE

NOTE

⇒ In Both S and S1

- R(A) done from database ..... No plbm
- Final updation by T_3 in both S and S1 ..... or, no plbm

NOTE:

If the Schedule is Conflict serializable schedule, we can say serializable schedule.
But if schedule is not conflict serializable, may or may not be serializable.

```
if (Acyclic Precedence Graph)   // Only sufficient condition
{                                  Not necessary
   Conflict SS                     To prove
      ↳ Serializable
}
else   // Not necessary condition
{
   not conflict serializable schedule
      ↳ may / may not be serializable
}
```

VIEW SERIALIZABLE SCHEDULE

(Both necessary & sufficient condition).

```
if (S satisfied VIEW SS condition)
    {
      Schedule is Serializable
    }
else
    {
      Not serializable schedule
    }
```

↳ View Serializable Schedule :- View Equivalent Serial Schedule.

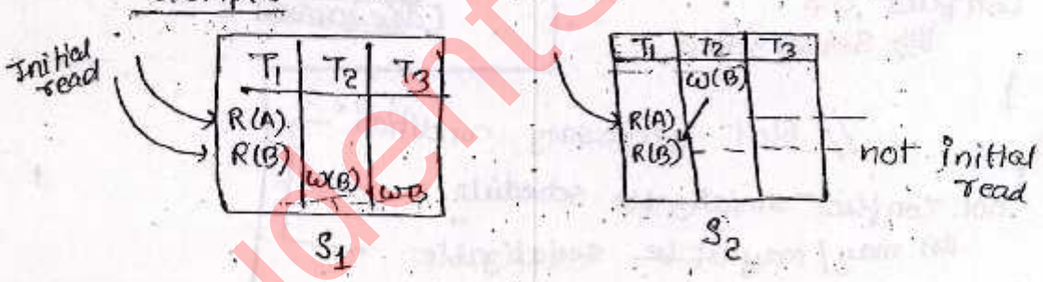
View Equivalent:

S1 and S2 said to be View Equivalent

Only if

[1] Initial reads of S1 and S2 should be same.

Example



$S_1 \neq S_2$

[2] $W_i(A)$ $R_j(A)$

Write - Read sequence should also be equal.
(Updated Reads should be same)

T_i	T_j
W(A)	
	R(A)

T_i	T_j
W(A)	
	R(A)

Example

S_1

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

Updated Read
$T_2 \rightarrow T_3$

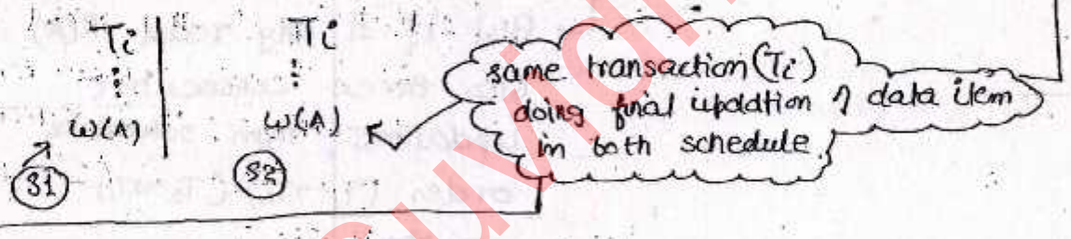
S_2

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

Updated read
$T_1 \rightarrow T_3$

$S_1 \neq S_2$

[3] Final Updations for every data item should be same in S1 and S2.



Example

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

S_1

Final Updation
A — T_3
B — T_3

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

S_2

Final Updation
A — T_2
B — T_3

$S_1 \neq S_2$

⇒ When all above three conditions satisfied, S schedule is said to be VIEW SERIALIZABLE.

#

T_1	T_2	T_3
$w(A)$		
.	$w(A)$	
.		$w(A)$

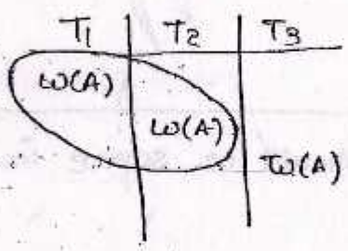
(S1)

T_1	T_2	T_3
	$w(A)$	
$w(A)$		
		$w(A)$

(S2)

» S1 and S2 are NON-CONFLICT Equivalent solutions.

» S1 and S2 are VIEW SERIALIZABLE



↳ says that if final updation done by T_3 order of T_1 & T_2 doesnot matter.

↳ But if ∃ any read R(A) b/w three consecutive updations, then schedule order $(T_1 \to T_2)$ $(T_2 \to T_1)$ matters.

» If ∃ no read b/w three consecutive w(A) from three transactions, first two updations can be in any order.

Q: Schedule serializable or not?

a) $T_1 \rightarrow T_2 \rightarrow T_3$

b) $T_2 \rightarrow T_1 \rightarrow T_3$

c) $T_2 \rightarrow T_3 \rightarrow T_1$

d) Non-serializable (all the above)

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

	CSS	
a	X	X ✓
b	X	X
c	X	
d	✓ X	All of above

↳ View serializable

↳ Possible if view serializable

Q ⟹ S Conflict. serializable as :- check CSS

⟹ S serializable as :- check for View SS

VIEW

(1) Final updation :-

A : T_1 T_2 T_3 ⟹ updation on A by all but finally by T_3

B : T_1 --- only 1 trans. updating, so order not an issue here.

⟹ On its equivalent serial schedule T_3 must execute at last.

$(T_1, T_2) \rightarrow T_3$

(a) $T_1 \rightarrow T_2 \rightarrow T_3$ or $T_2 \rightarrow T_1 \rightarrow T_3$

till here.

[2] Initial Read — Write Operations

A: T_2 — T_1 T_2 T_3

B: T_2 — T_1

↳ A and B both initially read by T_2. — means T_2 should execute before other two ~~write~~ operations.

(b) $T_2 \rightarrow (T_1, T_3)$ ✓ — ok till now

$T_1 \rightarrow T_2$ ✗

— To preserve Initial Read, T_2 should execute before other Trans.

$T_2 \rightarrow T_1$.

→ To preserve ~~Write~~ Read followed by ~~Read~~ write: (WR)

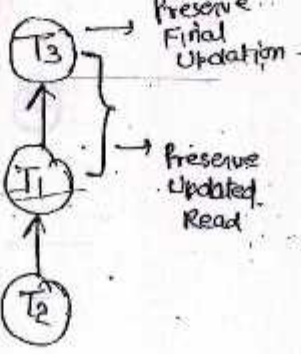
↳ T_3 Reading value Updated by T_1

(c) i.e $T_1 \rightarrow T_3$

⇒ From above three conditions, we have

a) $(T_1, T_2) \rightarrow T_3$
$(T_2 - T_1) \rightarrow T_3$
b) $T_2 \rightarrow (T_1, T_3)$
c) $T_1 \rightarrow T_3$

T_3 should be final



⇒ If some give $T_1 \rightarrow T_3$ and other give $T_3 \rightarrow T_1$ } Schedule is Non Serializable Schedule

Q Pg 90 (Q41)

(i) Final Updation

D_1 : T_2
D_2 : T_1
D_3 : T_3
} write

D_1 : T1 , (T2) ⇒ $T_1 \rightarrow T_2$
D_2 : T2 , T3 , (T1) ⇒ $(T_2 T_3) \rightarrow T_1$
D_3 :

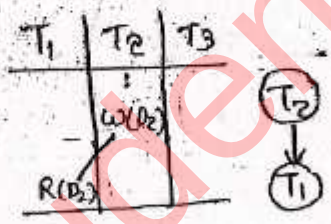
Both can't be true at the right time

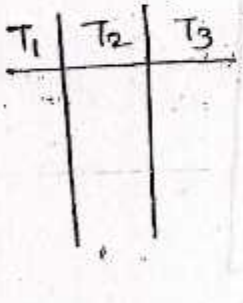
Non-Serializable

(ii) Initial Reads

D_1 : T_1
D_2 : T_2
D_3 : T_2

(iii) WR (updated Read)





Pg 92 Q:9

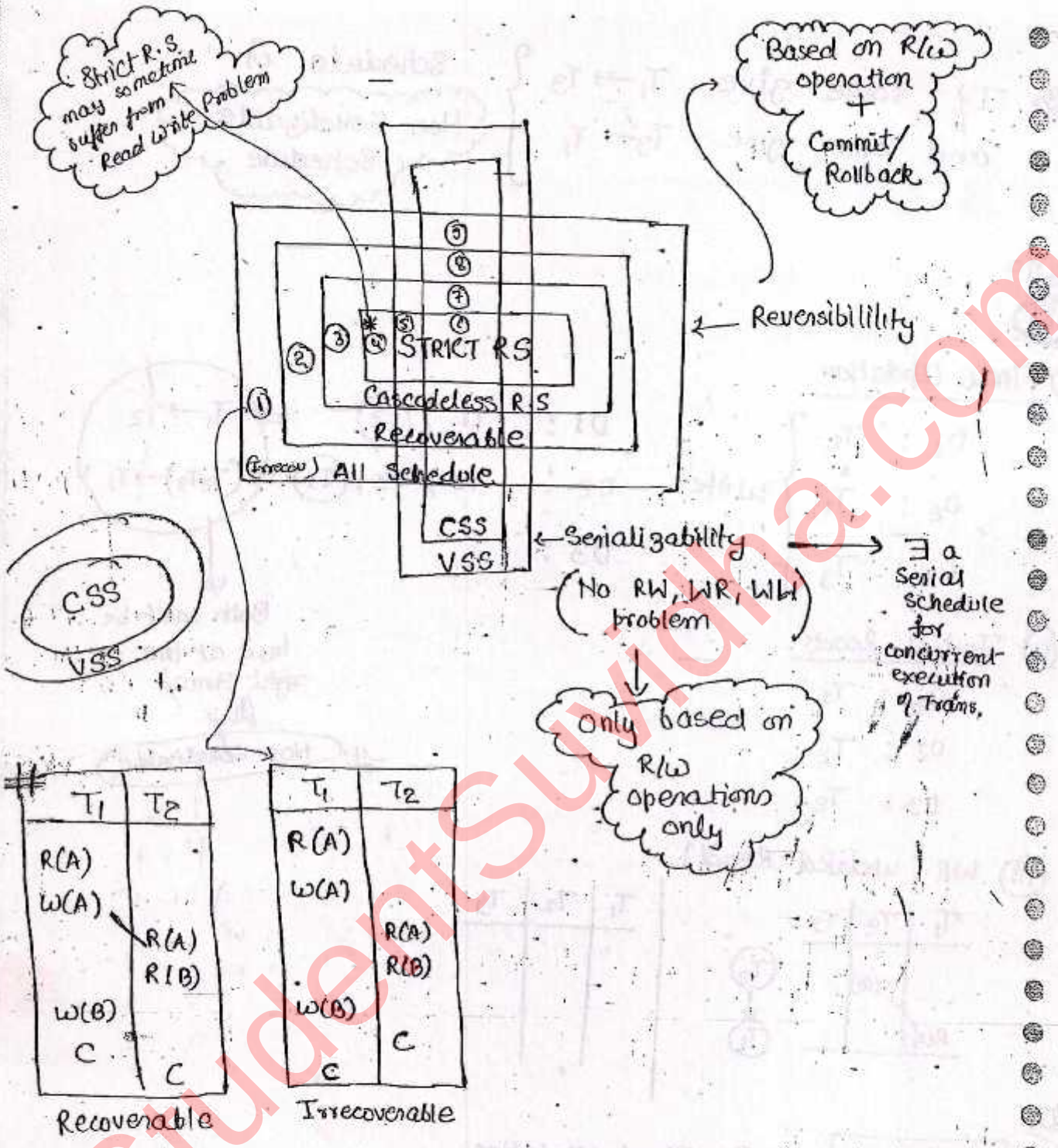
T_1	T_2
R(A)	
	R(A)
	W(A)
	R(B)
W(A)	
R(B)	
W(B)	
	W(B)

(i) Final Updation

Write operation

A : (T1) T2 ⇒ $T_2 \rightarrow T_1$
B : T1 (T2) ⇒ $T_1 \rightarrow T_2$

Both not possible simultaneously . Thus Non-Serializable



Q:- Write a schedule which is Recoverable but not Serializable

(1) Recoverable not serializable
(2) Cascadeless
(3)
(4)
(5)
(6)

(6) CSS & Strict
(7) CSS & CL but Strict
(8)
(9)

7/11/11

Classification

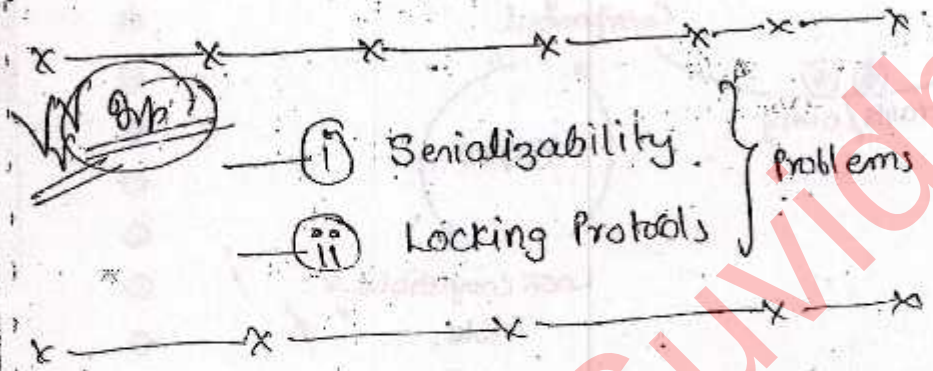
Recoverability | Serializability

Problems

⇒ A schedule is free from all problems (RW, WR, WW, Irrecoverability) if it is STRICT RECOVERABLE and Serializable.

GOAL OF TRANSACTION CONTROL TECHNIQUES.

{ Lock Based Protocols
Timestamp " "

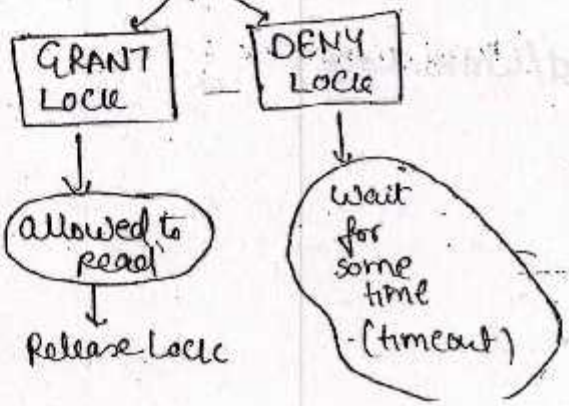


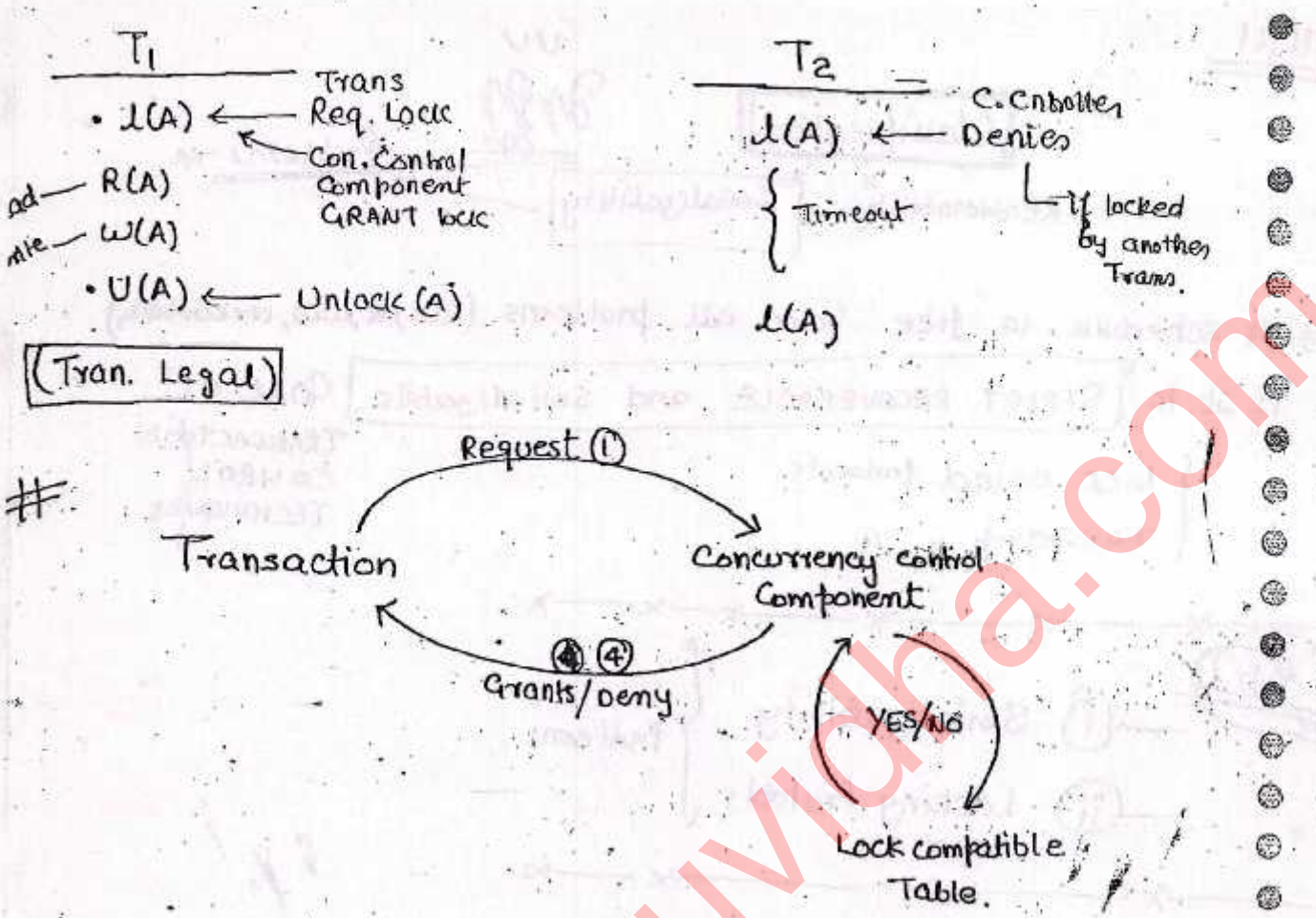
LOCK BASED PROTOCOLS :-

LOCK :- Variable used to identify status of the data item.

* Trans. should Request lock before usage of any data item.

⇒ Trans. Request Lock
↓
It is responsibility of CONCURRENCY CONTROL TECHNIQUE.





Shared / Exclusive Locking

T
S(A)
R(A)
~~W(A)~~

shared lock

T having shared Lock on data item can read it but can't write the data item.

Shared Lock [S]: ⟶ can Read but not Write

Exclusive Lock [X]: (Read/write Lock)

T
X(A)
R(A)
W(A)
⋮

⇒ Given schedule S

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

implementing using Locks

T_1 → Read + write
Require Exclusive Lock

T_2 → Only Read
shared Lock

Implementation

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

Legal Schedule

Non-serializable Schedule
(WR problem exist)

Strict Recovable + Serializable not achieved

Lock compatible Table

	S	X
S	Yes	No
X	No	No

→ Held

↓ Request

Lock compatible Table

To overcome this incapability

TWO PHASE LOCKING PROTOCOL

Trans. T not allowed to REQUEST Lock on data item if T has already performed some unlock operation.

T
L(A)
L(B)
L(C)
L(d)
⋮
U(C)
L(e)
U(A)
U(B)
U(D)

Locking PHASE (GROWING PHASE)

Lock Point

once Unlock

Not allowed to request anymore Locks.

UNLOCKING PHASE (SHRINKING PHASE)

Lock Point :- Position of last Lock operation or position of first Unlock operation.

- Transition phase from lock to unlock phase

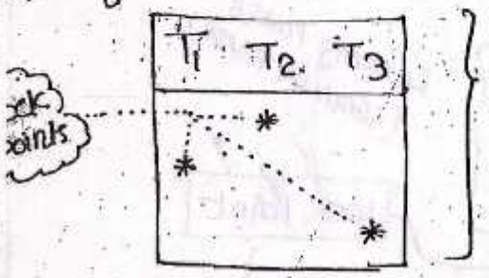
» 2PL always allowed ~~to visit~~ to Execute Serializable schedule only.
(conflict serializable schedule)

» If schedule (S) allowed to execute by 2PL then S surely conflict serializable.

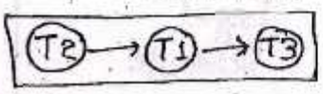
CSS
RIS

» Equivalent serial schedule of 2PL allowed schedule is based on LOCK POINTS.

» eg: assume T_1 T_2 T_3



If it is 2PL Schedule
Equivalent serial schedule
T2 : T1 : T3

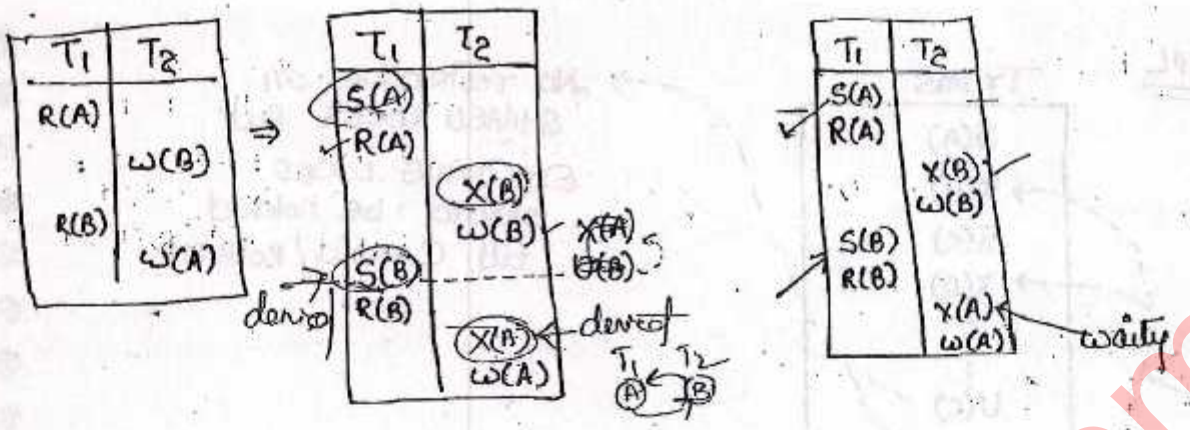


? How to test S satisfied by 2PL.
└ check whether conflict serializable

YES
Conflict Serializable
may / may not be allowed by 2PL

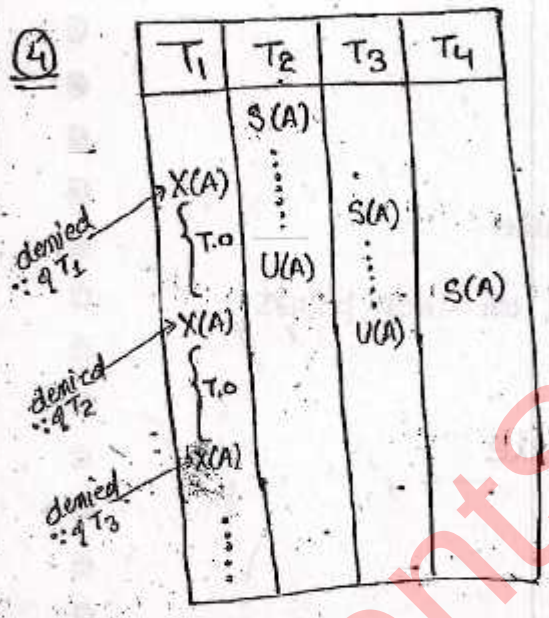
NOT conflict serializable
2PL

(3)



Note: 2PL may not be free from **DEADLOCK**

(4)

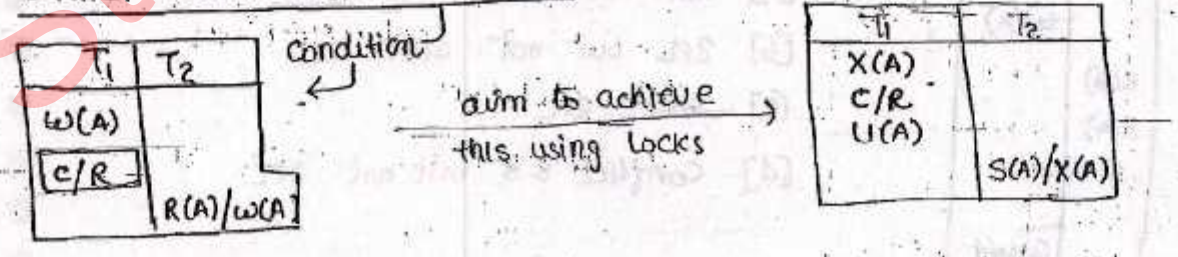


» 2PL may suffer from **STARVATION**

STRICT 2PL PROTOCOL

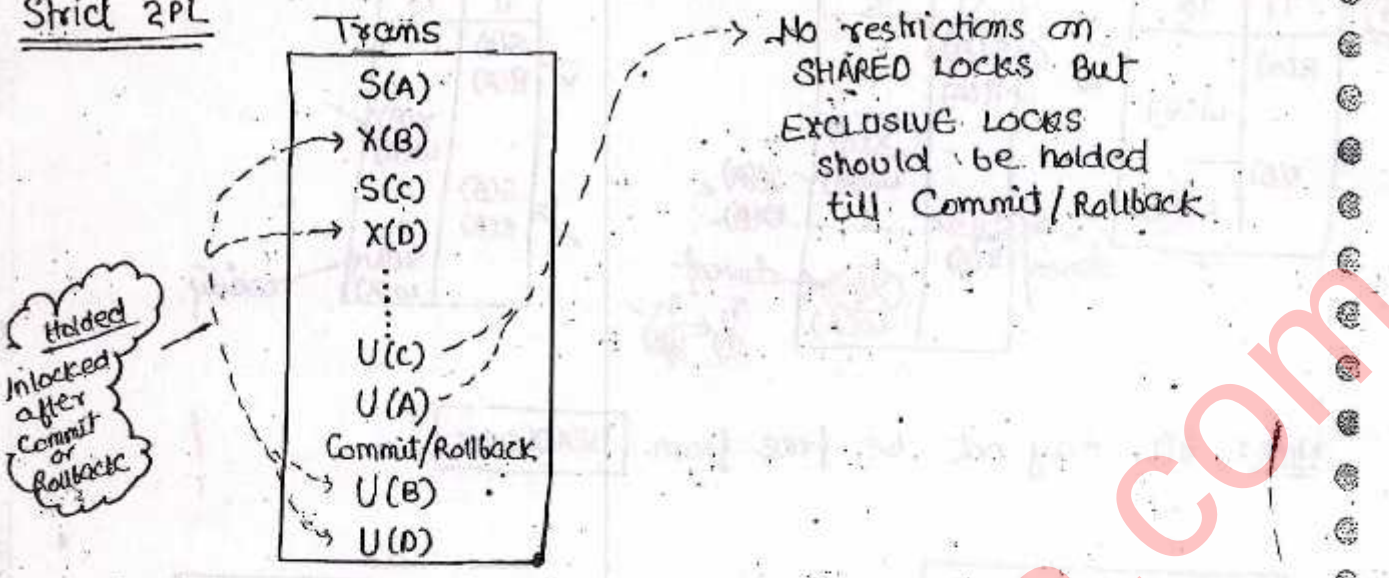
└ we are trying for strict recoverable

STRICT Recoverable Schedule



» All Exclusive Locks should be hold until commit / Rollback.

Strict 2PL



STRICT 2PL PROTOCOL

- Ensures conflict serializable schedule
- Equivalent serial schedule, based on lock points and
- Ensures Strict Recoverable schedule
- May not be free from
  - Deadlock &
  - STARVATION

Q:-

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

→ Given trans. is

[a] not in 2PL
[b] 2PL but not strict 2PL
[c] strict 2PL
[d] Conflict s s but not 2PL

Soln (2PL) ?? ⇒

T_1	T_2	T_3
S(A)		
R(A)		
U(A)		X(A)
	S(B)	W(A)
	R(B)	U(A)
	S(A)	
	R(A)	
	C	
C		
		C

2PL ✓

STRICT 2PL

T_1	T_2	T_3
S(A)		
R(A)		
U(A)		X(A) ← exclusive lock delayed till Commit / Rollback
		W(A)
	S(B)	
	R(B)	
	S(A)	
	R(A)	
	C ↑	
C		C
		U(A) ←

Strict 2PL

So not strict 2PL

conflict

(3) RIGORIOUS 2PL

2PL && every lock (shared/exclusive) should be held untill commit

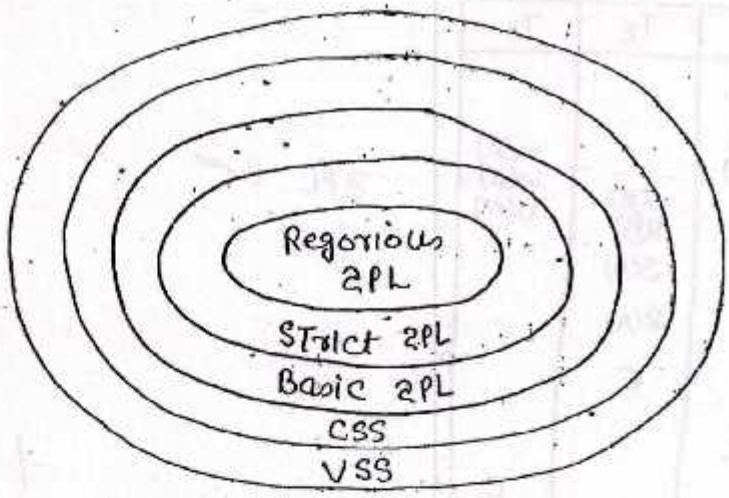
» Ensures serializability (conflict)

» Equivalent serial schedule based on the order of commit

Trans
S(A)
X(B)
X(C)
S(D)
⋮
C/R
U(B)
U(A)
U(D)
U(C)

T_1	T_2	T_3
	C	
C		
		C

⇒ Serial schedule = $T_2 \rightarrow T_1 \rightarrow T_3$

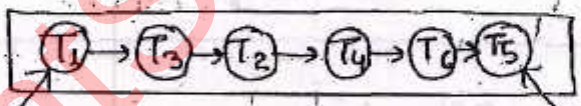


TIME STAMP BASED PROTOCOLS

Time Stamp Value :- Unique value assigned by DBMS to every Transaction in ascending order.

If Time stamp : 10 30 20 40 60 50
Trans : T_1 T_2 T_3 T_4 T_5 T_6

Then Schedule



Older Trans.

Younger Trans.

» Time stamp value can be used to
① Transaction ID
② Set the priority

» **Time stamp value of data item** :-

@ **Read_TS(A)** :- Highest Trans. time stamp value that has performed R(A) operation successfully.

10	30	20	40
T_1	T_2	T_3	T_4
R(A)			
	R(A)		

RTS (A) = 10 ——— initially

RTS (A) = 30 [max (10, 30)] ——— finally

② Write_TS(A) :- Highest Transaction Time stamp value that has performed W(A) operation successfully.

10	30	20
T_1	T_2	T_3
R(A) W(A)	R(A)	W(A)

RTS_(A) = ~~10~~ (30) → Youngest T.S for Read

WTS_(A) = ~~10~~ (40) → Youngest T.S for Write

BASIC GOALS OF TIMESTAMP ORDERING PROTOCOL

① 10 20 30 | T_1 | T_2 | T_3 |

» Concurrency allowed only if equivalent serial schedule should be based on T.S ordering

⇒ Concurrent Ex = $T_1 \xrightarrow{10} T_2 \xrightarrow{20} T_3 \xrightarrow{30}$ (serial sch)

even though $(T_3 \to T_1 \to T_2)$ $(T_2 \to T_1 \to T_3)$ ✗

↳ T.S O. Protocol not allowed

it should strictly be equal to $T_1 \to T_2 \to T_3$

Then allowed.

If not equal then Rollback.

Q IS S allowed to execute using T.S. ordering protocol

S: ——————

T_s: (T_1, T_2, T_3) : (20, 10, 30)

Sol^n If ∃ serial order $(T_2 \to T_1 \to T_3)$ for S, then S is allowed to execute using T.S ordering protocol.

→ (i) Check for CSS

→ (ii) If corresponding serial order = $T_2 \to T_1 \to T_3$ — ok.

Q S: $R_1(A)$, $R_2(B)$, $W_1(A)$, $W_1(B)$

T_s: {10, 20}

Allowed using T.S. O ??

⇒

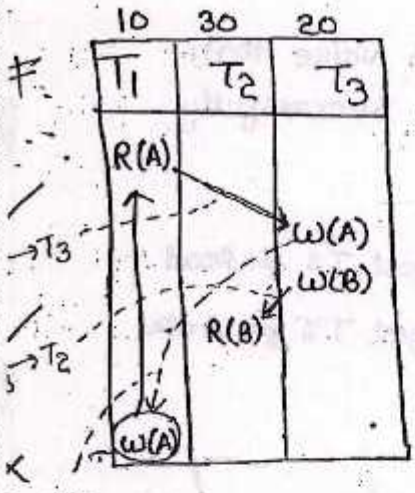
T_1	T_2
R(A)	R(B)
W(A) W(B)	

$T_1 \to T_2$

↳ Topological Seq = $T_1 \to T_2$

But we expect $T_1 \to T_2$ based on T.S value

So this Schedule not allowed to execute using T.S. O. Protocol



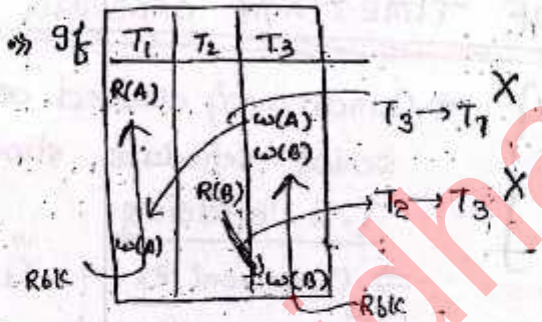
This conc. exe should be = serial schedule $(T_1 \to T_3 \to T_2)$

Allowed conflict pairs

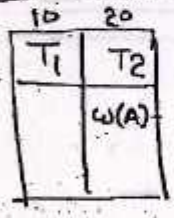
$T_1 \to T_2$
$T_3 \to T_2$
$T_1 \to T_2$

If any conflict pair other than this then T.S.O Protocol can't be applied.

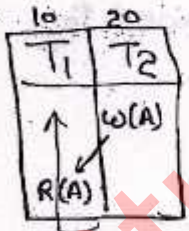
$T_3 \to T_1$ — not a allowed conflict pair Hence $W_1(A)$ causing problem so T_1 — Rollback



Possibility of Rollbacks:

a) 

$T_1 \to T_2$ --- ok (First operation is not a problem)

b) 

$T_2 \to T_1$

So T_1 Rollbacked ∵ it is (easily) violation

WTS = 20

NOTE :- If Younger Trans. updates data item. Then olden trans. not allowed to read.

T_1 : Issues R(A) :-

WTS(A) > TS(T1) ⇒ Rollback T1

(2)

10	20
T_1	T_2
	R(A)

$WTS = 0$
$TS(T1) = 20$
↳ Issue Read

$0 > 20$ false so not rollbacked.

(2)

10	20
T_1	T_2
↑ ω(A)	R(A)

»» we are getting $T_2 \rightarrow T_1$ instead of expected $T_1 \rightarrow T_2$

»» If YOUNGER TRANS. performed the R(A) then older Trans. not allowed to update A.

T1: Issue ω(A):

RTS(A) > TS(T1)

Rollback T1

(3)

10	20
T_1	T_2
↑ ω(A)	ω(A)

$T_1 \rightarrow T_2$ — expected
$T_2 \rightarrow T_1$ — actually got.
Rollback T1.

»» If Younger trans. performed write operation (updation on data item) then older Trans. Not allowed to update A.

T1: Issue ω(A):

WTS(A) > TS(1)

Rollback T1

Basic Timestamp Protocol :-

[1] Transaction T issues R(A) operation

a) if WTS(A) > TS(T) Then Rollback T

b) Otherwise allowed to execute R(A) by T

Set Read_TS(A) = max {RTS(A), TS(T)}

[2] Trans. T issues W(A) operation :-

a) if RTS(A) > TS(T) Then Rollback T

b) if WTS(A) > TS(T) Then Rollback T

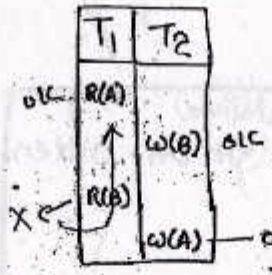
c) Otherwise execute W(A) operation by T.
Set WTS(A) = TS(T)

B.T.O. Protocol

(1) Ensures Serializability
Equivalent serial schedule should be always based on TS ordering.

(2) Timestamp Ordering protocol free from deadlock.

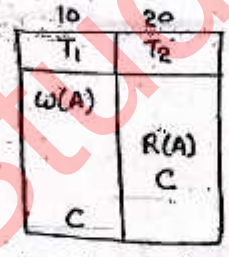
Ex



→ No possibility of cycle.

(3) Not free from Irrecoverability (Irrecoverble schedule)

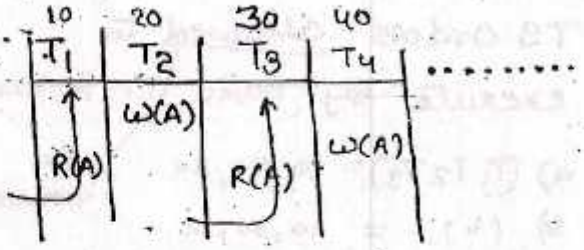
Ex



→ S allowed to execute using B.T.O protocol Thus it is a conflict serializable schedule.
But it is irrecoverable ∵ T1 performing updation getting committed after T2.

(4) may not be free from STARVATION

Ex



8th Nov 2011

BASIC Time Stamp Protocol :-

(1) ensures serializability

(2) Deadlock free

(3) Starvation may be possible

⇒ (4) Irrecoverable may be possible

STRICT TO Protocol

(1) ensure serializably

(2) Deadlock free

(3) starvation may or possible

(4) Always result Strict Recoverable schedule.

STRICT Time Stamp Ordering Protocol :

10	20
T1	T2
W(A) C/R	R(A)/W(A)

Strict Recoverability

if Trans (T2) R(A)/W(A)

WTS(A) < TS(T2)

R(A)/W(A) of T2 should be delayed until C/R (T1

⇒ STSO Protocol :

- Follow Basic TO Protocol

&&

- Trans T2 issues R(A)/W(A) operations :-

if WTS(A) < TS(T2) Then R(A)/W(A) of T2 should be delayed until Trans (T1) that has performed W(A) commit/Rollbacks.

:- S:

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

Which of the following TS orders allowed to execute by Basic T.O Protocol? (no rollbacks)

a) $(T_1\ T_2\ T_3) = 10, 20, 30$
b) (") = 10, 30, 20
c) (") = 30, 10, 20
d) (") = 30, 20, 10

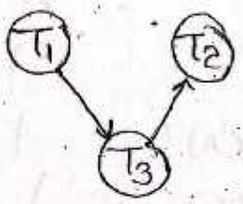
Sol^m Equivalent Serial schedule acc. to options.

(a) $T_1 \rightarrow T_2 \rightarrow T_3$
(b) $T_1 \rightarrow T_3 \rightarrow T_2$
(c) $T_3 \rightarrow T_1 \rightarrow T_2$
(d) $T_3 \rightarrow T_2 \rightarrow T_1$

allowed conflict pairs

① $T_1 \rightarrow T_2$ ② $T_2 \rightarrow T_3$ ③ $T_1 \rightarrow T_3$
② $T_1 \rightarrow T_3$, $T_3 \rightarrow T_2$ $T_1 \rightarrow T_2$
③ $T_3 \rightarrow T_1$ $T_1 \rightarrow T_2$ $T_3 \rightarrow T_2$
④ $T_3 \rightarrow T_2$ $T_2 \rightarrow T_1$ $T_3 \rightarrow T_1$

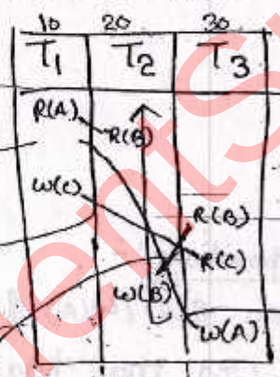
Now



Conflict equivalent to = $T_1 \rightarrow T_3 \rightarrow T_2$ (equivalent serial schedule)

Q: Schedule Given. Given Time stamp. Which trans. Rollbacks???

(a) Sequence



no conflict

1st conflict pair $T_1 \rightarrow T_3$ allowed

2nd conflict pair $T_3 \rightarrow T_2$ X

allowable Conflict Pair
$T_1 \rightarrow T_2$
$T_2 \rightarrow T_3$
$T_1 \rightarrow T_3$

any trans. violating it is Rollbacked.

$T_1 \rightarrow T_3 \rightarrow$ allowed → no mean

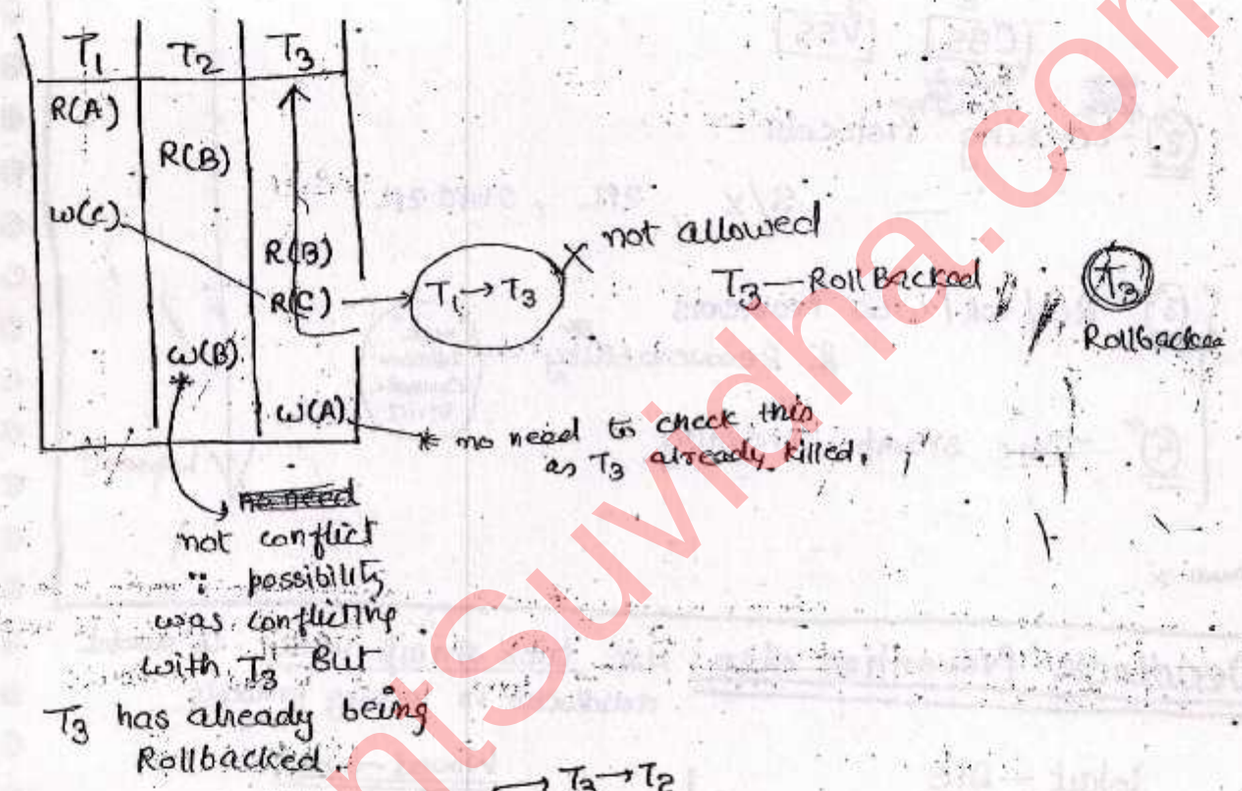
kill

Once a trans. is Rollbacked (Killed) { here T_2 Rollbacked then there is no need to consider its operations anymore.

(ie T_2 afterwards $W(A)$ comes, we can neglect that its mean nothing ∵ T_2 has already being rollbacked)

(c) $T_3 \rightarrow T_1 \rightarrow T_2$

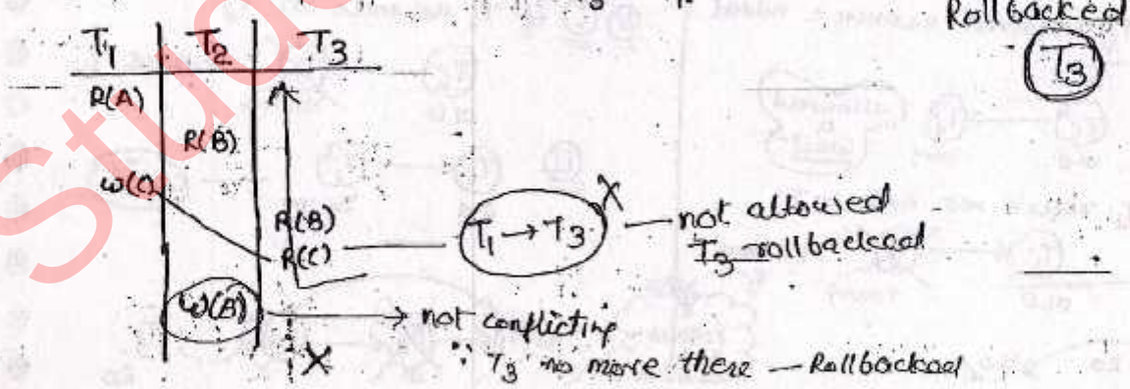
$T_1 \rightarrow T_2$
$T_3 \rightarrow T_1$
$T_3 \rightarrow T_2$

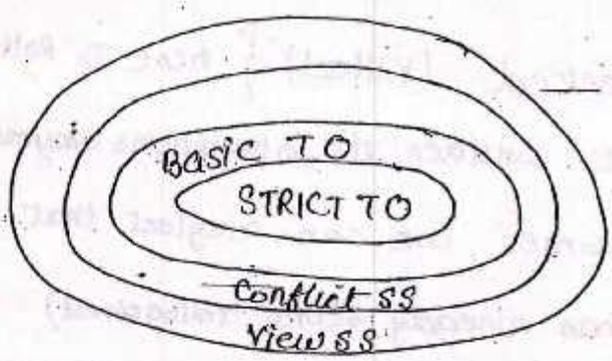


(d) $T_3 \rightarrow T_2 \rightarrow T_1$

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

Rollbacked
T3





In GATE 90% of question asked in Trans asked from these two topics

1. Serializability
- CSS (Precedence graph + conflict pairs)
- VSS

2. Locking Protocols
S/X , 2PL , Strict 2PL

3. RW / WR / WW Problems & Recoverability
(Check Irrecov, Cascadle, Strict)

4. Time stamp ordering

(mail theory ques)

Priority
- Highest
- lowest

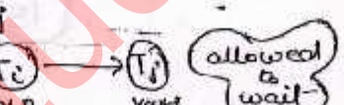
— Deadlock

Deadlock Prevention algo: use time stamp value to avoid deadlocks in locking protocols.

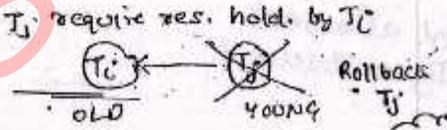
Wait – Die

(a) Trans in Ascending order of TS value TS(T_i) < TS(T_j)

(b) (i) if T_i required resource holded by T_j



(ii) if T_j require res. hold. by T_i

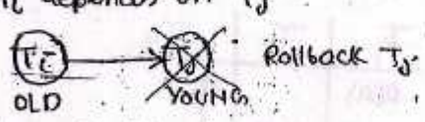


Wound – Wait

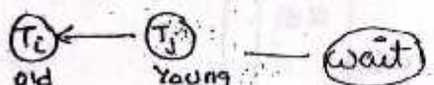
(opp. to wait-die)

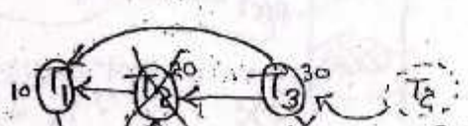
(a) TS(T_i) < TS(T_j)

(b) (i) if T_i depends on T_j



(ii)





Both are **deadlock** and **starvation** free protocols.

taken care by
Restarting with same timestamp