TRANSACTION & CONCURRENCY CONTROL

Read (Q) : Accessing data item (Q)
Write (Q) : Updating the data item (Q).
Commit : Commit indicates Completion of transaction Sucessfully.

→ A transaction is a unit of program execution that accesses and possibly update various data items.
eg: Transfer Rs 500 from Account A to Account B.

1. read (A)
2. A = A - 1000
3. write (A)
4. read (B)
5. B = B + 1000
6. write (B)

→ To Preserve the integrity of data the database System must ensure:

1. Atomicity
2. Consistency
3. Isolation
4. Durability

} ACID Property

Atomicity : Either all operations of the transaction are properly reflected in the database or none.
{ Full or None }.

Reasons of Failure:
* System failure
* Software Crash
* Hardware Crash .. etc.

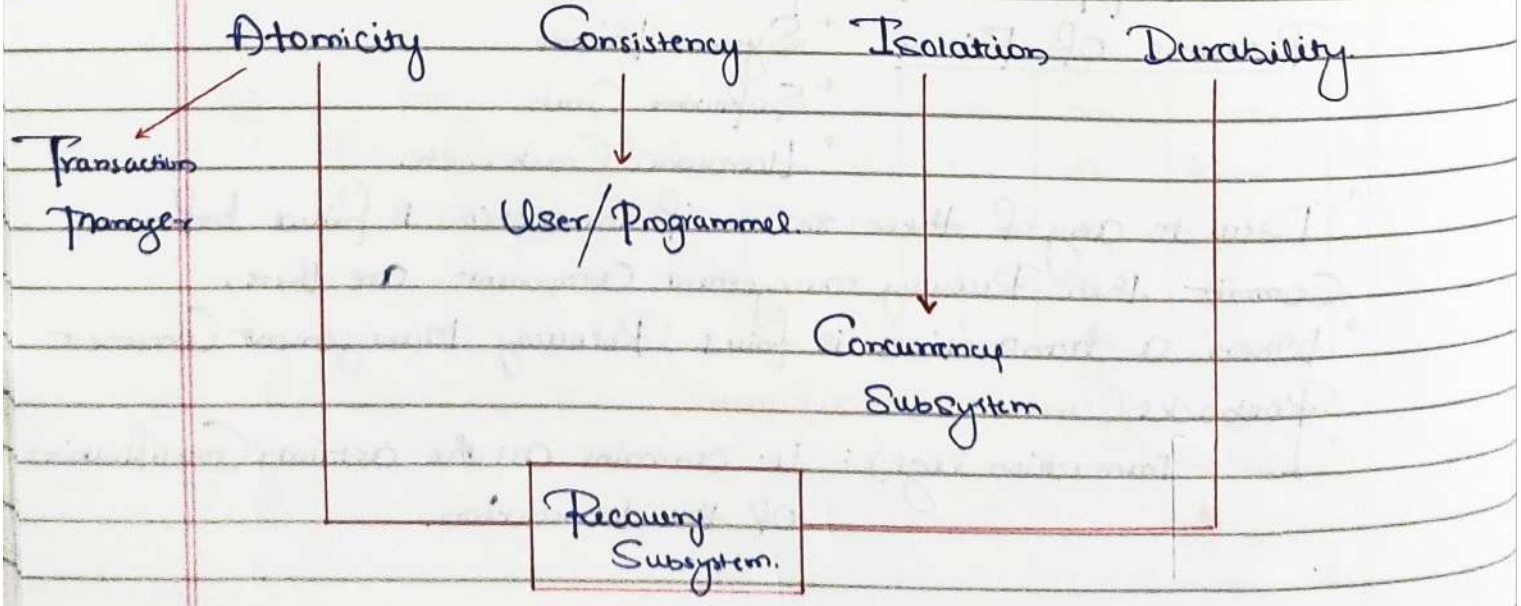
* Due to any of these reason if transaction is failed before Commit then Recovery management Component are there.
* When a transaction is failed Recovery Management Component Rollbacks (undo all modifications).
* Logs (Transaction logs) : it contains all the activity (modifications) of the transaction.

Consistency: Execution of transaction T_i in isolation preserves the consistency of the database. In other words, before and after the transaction the database must be consistent.

Isolation: Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

* When two or more transactions execute concurrently then isolation come into picture.
* Concurrent execution of two or more transaction should be equal to any special schedule.

Durability: Any change in the database must be persist for long period of time.
Database must be able to recover under any case of failure.



Transaction State:

Active: the initial state, the transaction stays in this state while it is executing.

Partially Committed: after the final statement has been executed.

Failed: after the discovery that no longer the normal execution will proceed

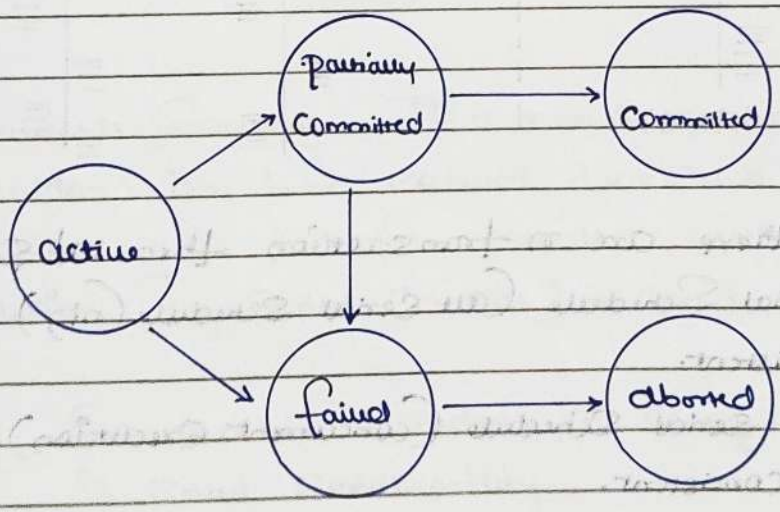
Aborted: After the transaction has been rolled back and the database restored to its state prior to the start of the transaction

Two options after it is aborted:

(i) Restart the transaction - can be done if no internal logical error.

(ii) Kill the transaction.

Committed: After successful completion.



Schedule: a sequence of instructions that specify the chronological order in which instructions of current transactions are executed.

* A schedule for a set of transactions must consist of all instructions of those transactions.

* Must preserve the order in which the instructions appear in each individual transaction.

→ A transaction that successfully completes its execution will have a commit instructions as the last statement.
* By default the transaction assumed to execute commit instruction as its last step.

→ A transactions that fails to successfully complete its execution will have an abort instruction as its last statement.

Schedule. { Time order sequence of two or more transaction }.

① Serial Schedule. - Execution of one transaction successfully, then other will start.

T_1	T_2
≡	
	≡

<T_1, T_2>

T_1	T_2
	≡
≡	

<T_2, T_1>

② Non Serial Schedule. { Concurrent Execution }
Interleaved execution of two or more transaction.

T_1	T_2
≡	
	≡
≡	
	≡

T_1	T_2
	≡
≡	
	≡
≡	

Note: * If there are n transaction then n! Serial Schedule.
* Serial Schedule (All Serial Schedule (n!)) are always consistent.
* Non-Serial Schedule (concurrent execution) may or may not be consistent.
* But we execute concurrent execution.

Q1. Why Concurrent Execution?
* To improve CPU utilization
* Enhanced throughput
* Less response, less waiting time.
* Effective Utilization of Resources.

Date 15/8/2023.

Serial Schedule:

* After commit of one transaction, begins (starts) another transaction.
* Number of possible Serial Schedules with 'n' transactions is "n!".
* The execution sequence of serial schedule always generates consistent result.
* Example: $R_1(A)$ $W_1(A)$ Commit (T_1) $R_2(A)$ $W_2(A)$ Commit (T_2)

Advantage: Serial schedule always produce correct result (integrity guranteed) as no resource sharing.

Disadvantages:
* Less degree of Concurrency.
* Throughput of system is low.
* It allows transaction to execute one after another.

Serializable Schedule: If a non Serial Schedule (concurrent execution) has been executed, that could have same effect on database as a schedule executed without any concurrent execution (any Serial Schedule) is called Serializable Schedule.

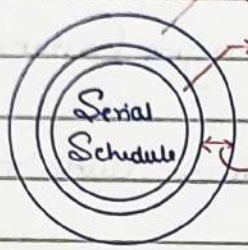
Note: Serializable Schedule are always Consistent. This process is called Serializability.

How to achieve Serializable Schedule

① Conflict Serializable.

② View Serializable

* A schedule is Serializable if it is equivalent to a Serial Schedule.



→ Concurrent Schedule
→ Serializable Schedule
→ Non-Serializable Schedule
{ isolation-fails }

Serializability:

Basic assumption: Each transaction preserves database consistency.

* Thus, serial execution of a set of transactions preserves database consistency.
* A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of
  1. Conflict Serializability.
  2. View Serializability.

Conflict Serializable:

Let us consider a schedule S in which there are two consecutive instructions Ii and Ij of transactions Ti and Tj respectively [i≠j]

Same data item. → No Conflict operation/instruction.

Ti	Tj
R(A)	W(A)
W(A)	R(A)
W(A)	W(A)

[Swapping not possible]

Ti	Tj	
R(A)	R(A)	[Swapping possible]
R(A)	W(B)	

Different data item.

Methods of Conflict Serializable:

1. Basic Concept.
2. Testing Method (precedence Graph)
3. Conflict Equal to any Serial Schedule

S: Non Serial Schedule [Given Question] S': Any Serial Schedule of S (of Given question).

$S \xrightarrow{\text{by Series of Swap of non Conflicting instructions.}} S'$

→ then the Schedule 'S' is Conflict Serializable.

* If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are Conflict equivalent.
* We say that a schedule is Conflict Serializable if it is conflict equivalent to any Serial Schedule.

Q1.

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

not possible. not possible by swapping. Swap can be done. Conflict.

S: <Given>. S': <T_2, T_1>.

→ Here "S" can't be converted into "S" <T_2, T_1> by swapping T_2 followed by T_1.
∴ S is not Conflict Serializable.

Q1.

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

Possible to commit. Swap allowed. different. no conflict.

→ Here S can be converted into S' <T_1, T_2> by swapping.
T_1 followed by T_2.
∴ S is Conflict Serializable.

Conflicting Instructions:

* Instructions I_i and I_j of transactions T_i and T_j respectively, conflict if and only if there exists some item Q accessed by the both I_i and I_j and at least one of these instructions write (Q).
  1. I_i = read (Q), I_j = read (Q) → non conflict instructions
  2. I_i = read (Q), I_j = write (Q) } they conflict.
  3. I_j = write (Q), I_j = read (Q) } they conflict.
  4. I_i = write (Q), I_j = write (Q). } they conflict.
* Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
* If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Testing for Serializability.

Precedence Graph Method:

$G(V, E)$

(vertex) V: Set of transactions.

(edge) E: Edge $T_i \rightarrow T_j$ edge occur if any one condition hold

Conflict operations:

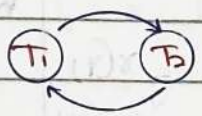
T_i	T_j	
R(A)	W(A)	} then edge draw.
W(A)	R(A)	} then edge draw.
W(A)	W(A)	} then edge draw.

C.N.C

Cycle Not Confli-

Note: If precedence graph contains cycle (any one cycle) then schedule is not conflict serializable.

Testing for Conflict Serializability

* Consider some schedule of a set of transactions $T_1, T_2 \ldots T_n$
* Precedence Graph: a direct graph where the vertices are the transactions (names).
* We draw an arc from T_i to T_j if the two transaction conflict and T_i accessed the data item on which the conflict arise earlier.
* We may label the arc by the item that was accessed
* eg: 

A schedule is Conflict-Serializable if and only if its precedence graph is acyclic.

Q1.

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

Acyclic

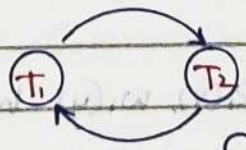
∴ Conflict Serializable

$\langle T_1, T_2 \rangle$

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

Q2.

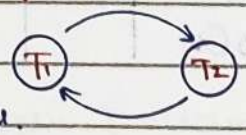
T_1	T_2
R(A)	
W(A)	
	R(B)
	W(B)
R(B)	
W(B)	
	R(A)
	W(A)



Cycle

∴ Not Conflict Serializable.

∴ Not Conflict Serializable



Cyclic.

Date 17/8/2023

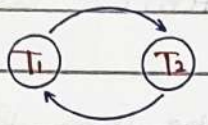
Q3. Consider the following Schedules involving two transactions. Which of the following is true?

S_1: $r_1(x), r_1(y), r_2(x), r_2(y), w_2(y), w_1(x)$.

S_2: $r_1(x), r_2(x), r_2(y), w_2(y), r_1(y), w_1(y)$.

Ans

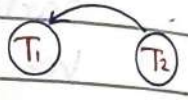
T_1	T_2
$r(x)$	
$r(y)$	
	$r(x)$
	$r(y)$
	$w(y)$
$w(x)$	



Cycle

∴ Not Conflict Serializable. (S_1)

T_1	T_2
$r(x)$	
	$r(x)$
	$r(y)$
	$w(y)$
$r(y)$	
$w(x)$	



$\langle T_2, T_1 \rangle$

∴ S_2 Conflict Serializable

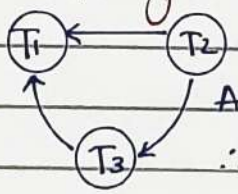
⟹ S_1 not conflict and S_2 is Conflict Serializable.

Q4. Consider the following four schedules due to three transactions (indicated by subscript) using read and write on data item x, denoted by $r(x)$ and $w(x)$ respectively. Which of them is conflict Serializable?

Ans

T_1	T_2	T_3
	$r(x)$	
	$w(x)$	
		$r(x)$
$r(x)$		
$w(x)$		

Not Cyclic



A Cyclic

∴ Conflict Serializable

$\langle T_2, T_3, T_1 \rangle$

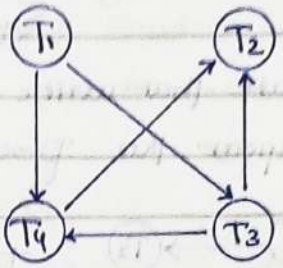
⟹ $r_2(x), w_2(x), r_3(x), r_1(x), w_1(x)$.

Q5. Let $R_i(z)$ and $W_i(z)$ denote read and write operations on data element z by transaction T_i, respectively. Consider the schedule S with four transactions.

S: $R_4(x), R_2(x), R_3(x), R_1(y), W_1(y), W_2(x), W_3(x), R_4(y)$

Which of the following serial schedules is conflict equivalent to S?

T_1	T_2	T_3	T_4
			$R(x)$
	$R(x)$		
		$R(x)$	
$R(y)$			
$W(y)$			
	$W(x)$		
		$W(y)$	
			$R(y)$



$\Longrightarrow \langle T_1, T_3, T_4, T_2 \rangle$

Method-II

S: $R_4(x), R_2(x), R_3(x), R_1(y), W_1(y), W_2(x), W_3(y), R_4(y)$.

Data item (x): $R_4(x) - W_2(x) : T_4 \rightarrow T_2$
$R_3(x) - W_2(x) : T_3 \rightarrow T_2$

Data item (y): $R_1(y) - W_3(y) : T_1 \rightarrow T_3$
$W_1(y) - W_3(y) : T_1 \rightarrow T_3$
$W_1(y) - R_4(y) : T_1 \rightarrow T_4$
$W_3(y) - R_4(y) : T_3 \rightarrow T_4$

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

Serializability Order

If schedule is conflict serializable (Acyclic precedence graph) then serializability order indicate (tell) that non serial schedule is equivalent to which serial schedule.

eg: 1.) 

Serializability order.
$\langle T_1, T_2 \rangle$

2.) 

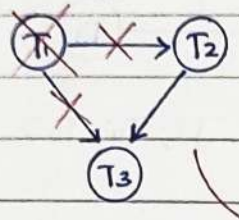
$\langle T_2, T_1 \rangle$ is equivalent to serial schedule T_2 followed by T_1

Topological Sorting: (Process)

(no incoming edge)

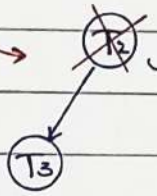
Start from the vertex which having In degree: 0 then delete that vertex and all connecting edge from that vertex and repeat this process until complete graph (all vertex) traversed.

eg:



Indegree = 0 (no incoming edge)

T_1

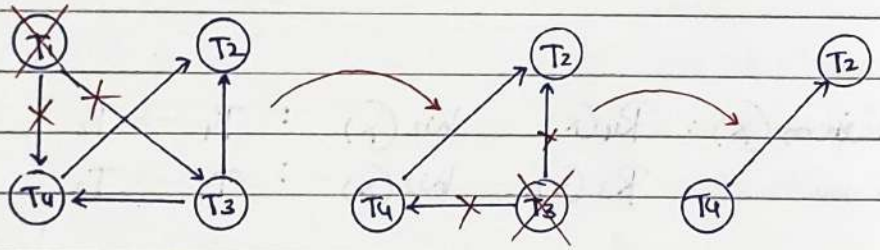


indegree = 0

$\Rightarrow \langle T_1, T_2, T_3 \rangle$

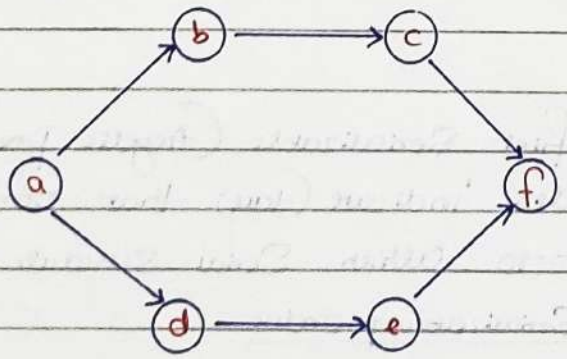
Serializability Order.

eg:



Serializability Order: $\langle T_1, T_3, T_4, T_2 \rangle$

Homework: How many topological Sorting order possible.



Conflict Equivalence:

Two schedule are said to be conflict equivalent, if all conflicting operations in both the schedule must be executed in the same order.

eg: S_1: $R_1(x), W_1(x), R_2(y), W_2(y), R_1(y)$

S_2: $R_1(x), W_1(x), R_1(y), R_2(y), W_2(y)$

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

Conflict operation:

$W(y) - R(y) : T_2 \rightarrow T_1$

or

$W_2(y) \rightarrow R_1(y)$: $T_2 \rightarrow T_1$

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

$R(y) - W(y)$: $T_1 \rightarrow T_2$

or

$R_1(y) - W_2(y)$ $T_1 \rightarrow T_2$.

→ S_1 is not conflict equivalent to S_2.

eg:2 S_1: $R_1(A)\ W_1(A)\ R_2(A)\ W_2(A)\ R_1(B)\ W_1(B)$

S_2: $R_1(A)\ W_1(A)\ R_2(A)\ R_1(B)\ W_2(A)\ W_1(B)$

T_1	T_2	
R(A)		$R(A) - W(A)$: $T_1 \rightarrow T_2$
W(A)		$W(A) - R(A)$: $T_1 \rightarrow T_2$
	R(A)	$W(A) - W(A)$: $T_1 \rightarrow T_2$
	W(A)	or
R(B)		$R_1(A) - W_2(A)$: $T_1 \rightarrow T_2$
W(B)		$W_1(A) - R_2(A)$: $T_1 \rightarrow T_2$
		$W_1(A) - W_2(A)$: $T_1 \rightarrow T_2$

R(A)		$R(A) - W(A)$: $T_1 \rightarrow T_2$
W(A)		$W(A) - R(A)$: $T_1 \rightarrow T_2$
	R(A)	$W(A) - W(A)$: $T_1 \rightarrow T_2$
R(B)		or
	W(A)	$R_1(A) - W_2(A)$: $T_1 \rightarrow T_2$
W(B)		$W_1(A) - R_2(A)$: $T_1 \rightarrow T_2$
		$W_1(A) - W_2(A)$: $T_1 \rightarrow T_2$

S_1 is Conflict equivalent to S_2

Conflict Serializable: A Schedule is said to be Conflict Serializable if it is conflict equivalent to a Serial Schedule.

Same Conflicting Operation Order in C_1 & S_1.
∴ Its $\{C_1\}$ Conflict is Conflict Serializable.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

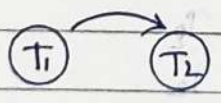
T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Important Note:

1. If S_1, S_2 Schedule are conflict equal then precedence Graph of S_1 and S_2 must be Same.
2. If S_1 and S_2 have same precedence graph then S_1 and S_2 may or may not be conflict equal.

eg: S_1:

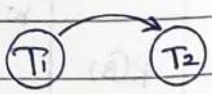
T_1	T_2
W(A)	
	R(A)



Precedence Graph is Same but not conflict equal $\{S_1 \& S_2\}$

S_2:

T_1	T_2
W(C)	
	R(C)



Serializable:

A schedule is Serializable if either it is (1.) Conflict Serializable or (2.) View Serializable or both.

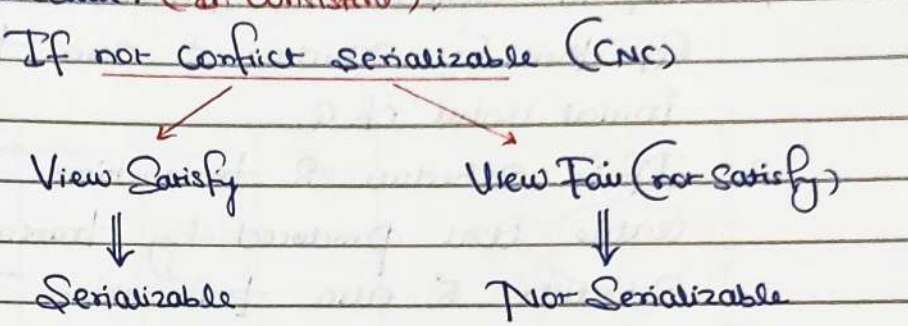
Note: * If Schedule is Conflict Serializable (Acyclic graph) then already it is View Serializable.
* If Schedule is not Conflict Serializable (CNC) then it may or

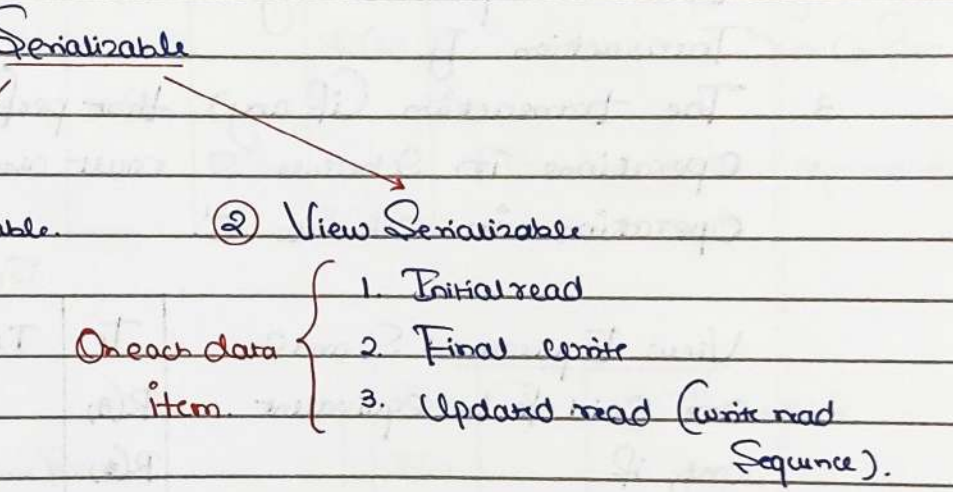
may not be Serializable.

* If Schedule is View Serializable but not Conflict then schedule is Serializable. (Consistent)

* If Schedule is not Conflict and not View then Schedule is not Serializable Schedule. (Inconsistent).

* If Conflict ✓ then already View Satisfy. ✓	* If not Conflict Serializable (CNC)





Equivalent Schedule

1. **Result Equivalent** : Two schedule are said to be result equivalent if they produce same final result for some initial value of data.

eg: A = 100

S_1	S_2
Read(A)	Read(A)
A = A + 10	A = A * 1.1
110 ← Write(A)	Write(A) → 110

* S_1 and S_2 are Result equivalent.

2. **Conflict Equivalent**

3. **Complete Schedule** : A Schedule is said to be complete Schedule if last operation of each transaction is either Commit or Abort Called complete Schedule Otherwise Partial Schedule.

View Serializability.

Let S and S' be two schedules with the same set of transactions. S and S' are view equivalent if following three conditions are met: for each data item Q:

1. If in schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q.
2. If in schedule S transaction T_i executes read(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same write(Q) operation of transaction T_j.
3. The transaction (if any) that performs the final write(Q) operations in schedule S must also perform the final write(Q) operation in schedule S'.

View Equivalent: S_1 and S_2 are said to be equivalent only if

(i) Initial reads of S_1 and S_2 should be same.

(ii) Final updations for every data item should be same in S_1 and S_2.

(iii) Write-Read sequence should also be equal. (updated reads should be same).

S_1

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

S_2

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

$S_1 \neq S_2$

eg:

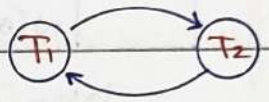
T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

Conflict Serializable $\langle T_1, T_2 \rangle$
and it is already "view Serializability".

Schedules: $\langle T_1, T_2 \rangle$ ✓
× $\langle T_2, T_1 \rangle$

eg: 2.

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



Cycle
- Not Conflict.

But it is view equivalent.

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

① Initial Read
A : T_1
B : T_1
② Final Write
A : T_2
B : T_2
③ Write-Read (Updated Read)

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

① Initial Read
A : T_1
B : T_1
② Final Write
A : T_2
B : T_2
③ Updated Read (Write-read Sequence)

A : $W_1(A) - R_2(A) : T_1 \rightarrow T_2$
B : $W_1(B) - R_2(B) : T_1 \rightarrow T_2$

A : $W_1(A) - R_2(A) : T_1 \rightarrow T_2$
B : $W_1(B) - R_2(B) : T_1 \rightarrow T_2$

it's view Serializable $\langle T_1, T_2 \rangle$

eg:3

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

Dummy. $\langle T_1, T_2 \rangle$ (fair as final write).
$\times \langle T_2, T_1 \rangle$ (fair as initial read)

① Initial Read
A: T_1, T_2
B: T_2
② Final Write
A: T_1
B: T_2

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

① Initial Read
B: T_1
A: Only T_1 not T_2
② Final Write
A: T_2 } fair.

- Not Conflict
- Not View Serializable.

eg:4

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

1. Initial Read A: T_1
2. Final write A: T_3
No write-Read
(no updated Read)

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

1. Initial Read : A: T_1
2. Final write A: T_3.
(No updated Read).

∴ Not Conflict but View Serializable
∴ Serializable.

eg:5.

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

Dummy
$\langle T_2, T_1, T_3 \rangle$.
① Initial Read
A: T_2, B: T_2
② Final write on
A: T_3 B: T_1
③ Updated Read
$T_1 \rightarrow T_3$.

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

① Initial Read
A: T_2, B: T_2
② Final write on
A: T_3, B: T_1
③ Updated Read.
$T_1 \rightarrow T_3$

Not Conflict but View Serializable.

Problem due to Concurrent Execution:

1. WR (Write-Read) / Uncommitted Read / Dirty Read Problem.
2. RW (Read-write) / Non/un repeatable read problem.
3. WW (Write-Write) / lost update problem.
4. Phantom Tuple Problem

Finding total number of schedule.

Concurrent Schedule { Serial Schedule + Non Serial Schedule }

→ m Transactions.

Total number of serial Schedule = m! Serial schedule.

Non serial Schedule = Total Concurrent Schedule − Serial Schedule (m!)

m: no. of transaction

eg: T_1 T_2

L_1 L_3

L_2 L_4

6 {
$L_1 L_2 L_3 L_4$
$L_3 L_4 L_1 L_2$
$L_1 L_3 L_2 L_4$ or $L_1 L_3 L_4 L_2$
$L_3 L_4 L_1 L_2$ or $L_3 L_4 L_2 L_4$

T_1 – n_1 Operation

T_2 – n_2 Operation

Total no. of Concurrent Schedule $= \frac{(n_1+n_2)!}{(n_1)!(n_2)!}$

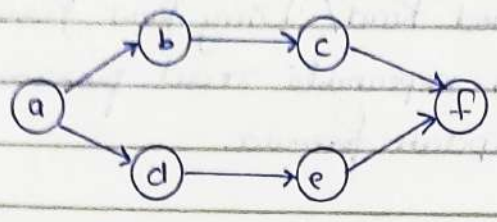
The number of Concurrent Schedule that can be formed over m transaction having $n_1 n_2 n_3 \dots n_m$ Operations respectively.

Total no. of concurrent Schedule $= \frac{(n_1+n_2+n_3+\dots n_m)!}{(n_1!)(n_2!)(n_3!)\dots(n_m!)}$

Total no. of non Serial Schedule $= \frac{(n_1+n_2+n_3+\dots n_m)!}{(n_1!)(n_2!)(n_3!)\dots(n_m!)} - m!$

Date 21/8/23

Q1. Consider the following directed graph:



The no. of different topological orderings of the vertices of graph is 6.

a b c d e f
a d e b c f
a b d c e f
a b d e c f
a d b e c f
a d b c e f

Q2. Consider the following transaction involving two bank accounts x & y.
read(x), x = x - 50, write(x), read(y), y = y + 50, write(y).
The constraint that the sum of accounts x and y should be constant is that of.

Ans Consistency.

Q3. Which of the following is not a part of the ACID properties of database transactions?

Ans Deadlock-freedom.

① Write-Read Problem / Uncommitted / Dirty Read Problem.

T_1	T_2
W(A)	
	R(A)

UnCommitted / Dirty Read: Here T_2 read the value of data item (A) that is updated (written) by uncommitted transaction T_1

② RW Problem / Non / Un Repeatable Read

T_1	T_2
R(A)	
	W(A)

eg:

T_1	T_2
A=10 Read(A)	
if(A<) {	
10-1 = 9.	
A = A-1	
Write(A) }	
	A=10 Read(A)
	if(A>0) { A = A-1, A=9. Write(A) }

9 book in Stock
+2 books issued
11. → Inconsistency

③ WW Problem / Lost update Problem:

T_1	T_2
W(A)	W(A)

T_1	T_2
A=1000 Read(A)	
A = A-100	
	Read(A) → A = 1000
	temp = A*0.1. temp = 100
	A = A + temp.
A=900 Write(A)	
	Write(A) A=1100.

Suppose the operation of transaction T_1 and T_2 in such a manner T_2 read the value of Account(A) Before T_1 update and T_2 updated the value of account just after T_1 update.

So whenever update done by T_1 is overwritten by the later transaction (so lose of updation of T_1 transaction).

Lost update Checking: If there are two write operation of different transaction and between these two write there is no read operation, then Second (later) transaction overwritten the value of first transaction is called lost update.

④ Phantom Tuple Problem.

Employee

eno.	ename	esalary
e1	A	5000
e2	B	6000
e3	C	4500
e4	D	7000
e5	E	6700

T_1	T_2
Select * From Employee where Salary > 4700	
	Insert into Employee value <e5, E, 6700>.
Select * From Employee where Salary > 4700	

eno.	ename	esal
e1	A	5000
e2	B	6000
e3	D	7000

eno.	ename	esal
e1	A	5000
e2	B	6000
e3	D	7000
e5	E	6700

→ Phantom tuple

Serializability

- Consistent
  - → Conflict Serializable
  - → View Serializable

Recoverability

- Recover fn in any case of failure.
  - → Recoverable Schedule
  - → Cascadeless Schedule
  - → Strict recoverable Schedule.

Dependency.

1.

T_1	T_2
W(A)	
⋮	R(A)
Commit	

Yes - dependency.
T_2 depends on T_1.
Uncommitted / Dirty Read.

2.

T_1	T_2
W(A)	
Commit	
	R(A)
	⋮

No dependency.

3.

T_1	T_2
W(A)	
Rollback	
↓ Undo all modifications	R(A)

No dependency.

4.

T_1	T_2
W(A)	
	W(A)

No dependency

5.

T_1	T_2
W(A)	
	W(A) } Same transaction.
	R(A) }

No dependency.
No uncommitted read.

6.

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

Yes - dependency.
T_2 - depend on T_1
T_1 - depend on T_2.

T_1	T_2
R(A)	
⋮	
W(A)	
	R(A)
	⋮
	Commit
↑ failure	

{ Irrecoverable }

Recoverable Schedule: A recoverable schedule is one, for each pair of transaction T_i and T_j such that, T_j read a data item that was previously written T_i then commit of T_i appear before commit of T_j.

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

If T_2 depends on T_i then commit of T_2 must be delay until commit/rollback of T_i.

Need to address the effect of transaction failures on concurrently running transactions.

* **Recoverable Schedule:** Transaction T_j reades a data item previously written by transaction T_i, then the commit operation of T_i appears before commit operation of T_j.
* The following schedule is not recoverable:

T_8	T_9
read(A)	
write(A)	Read(A)
	Commit
read(B)	

If T_8 should abort, T_9 would have read (and possibly shown to a user) an inconsistent database state. Hence database must ensure that schedule are recoverable.

Example.

1.

T_1	T_2
W(A)	
	R(A)
	C_2(Commit)
Rollback	

Irrecoverable

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

Recoverable.

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

Recoverable

T_1	T_2
W(A)	
C_1	R(A)
	C_2

Irrecoverable.

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

Recoverable.

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

Recoverable.

Note: Recoverable Schedule may or may not be free from WR problem, RW problem, WW problem.

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

Recoverable but RW problem.

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

Recoverable but WW problem.

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

Recoverable But WR problem.

Cascading Rollbacks: a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable).

T_{10}	T_{11}	T_{12}
read A.		
read(B)		
W(A)	read(A)	
	write(A)	read(A)
abort.		

* If T_{10} fails, T_{11} and T_{12} must also be rolled back.
* Can lead to the undoing of a significant amount of work.

Cascadeless Schedule: Cascading rollbacks cannot occur.

* For each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i and the commit operation of T_i appears before the read operation of T_j.
* Every Cascadeless Schedule is also recoverable.

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

Cascadeless Schedule.

* No WR Problem
* No Uncommitted (dirty) read
* No Cascading rollback

Note: Cascadeless Schedule may or may not be free from RW problem, WW problem.

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

Cascadeless but RW problem.

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

Cascadeless but WW problem.

1.

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

Recoverable Schedule.

2.

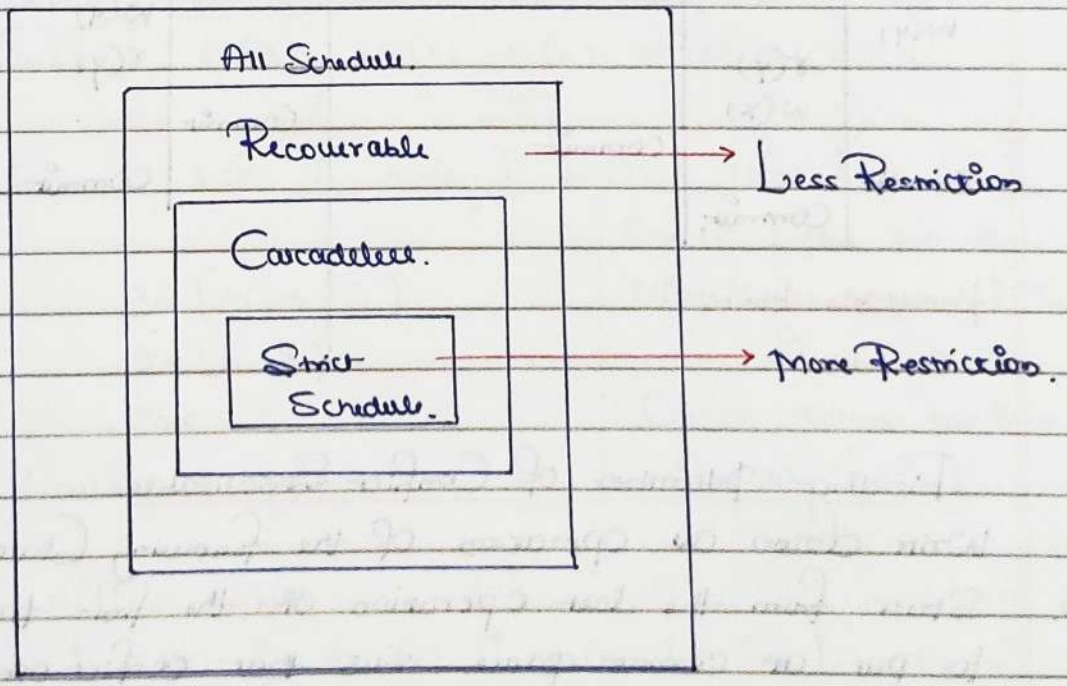
T_1	T_2
W(A)	
C/R	
	R(A)

Cascadeless Schedule.
- WW problem
- RW problem

3.

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

Strict Recoverable



Q1. $r_1(x)\ r_2(z)\ r_1(z)\ r_3(x)\ r_3(y)\ w_1(x)\ w_3(y)\ r_2(y)\ w_2(z)\ w_2(y)\ C_1\ C_2\ C_3$.

22/2/23

T_1	T_2	T_3
$r(x)$		
	$r(2)$	
$r(2)$		
		$r(x)$
		$r(y)$
$w(x)$		
		$w(y)$
	$r(y)$	
	$w(2)$	
Commit	$w(y)$	
	Commit	Commit

∴ Not Recoverable.

Q2. $r_3(x)\ r_1(x)\ w_3(x)\ r_2(x)\ w_1(y)\ r_2(y)\ w_2(x)\ C_3\ C_1\ C_2$.

T_1	T_2	T_3
		$r(x)$
$r(x)$		
		$w(x)$
	$r(x)$	
$w(y)$		
	$r(y)$	
	$w(x)$	
		Commit
Commit		
	Commit.	

Recoverable but not Cascadeless.

Q3. $r_1(x)\ r_2(x)\ w_1(y)\ w_2(y)\ r_2(y)\ C_1\ C_2$

T_1	T_2
$r(x)$	
	$r(x)$
$w(y)$	
	$w(y)$
	$r(y)$
Commit	
	Commit.

Recoverable ✓
Cascadeless ✓
Strict Recoverable ✗
- No Cascading rollback

Finding Number of Confict Serializable

1. Write down all operations of the following (later) transactions.
2. Start from the last operation of the first transaction and try to put at correct place such that confict operations order must be maintained same as transaction order.

Q1. Two transactions are given: T_1: $r(x)\ w_1(x)\ r_1(y)\ w_1(y)$.

T_2: $r_2(y)\ w_2(y)\ r_2(z)\ w_2(z)$

Where $r_i(v)$ denote a read operation by transaction on T_i on a variable v and $w_i(v)$ denote a write operation by transaction T_i on a variable v. The total no of conflict serializable schedule that can be formed by T_1 and T_2 is ____

Ans. $T_1 \rightarrow (T_2)$: following

Start from last transaction of T_1.

$w_1(y)$ ✓ $w_1(y)$ ✓ $w_1(y)$ ✓ $w_1(y)$ ✗

$\uparrow r_2(y) \uparrow w_2(y) \uparrow r_2(z) \uparrow w_2(z)$

$T_1 \rightarrow T_2 = 1$.

$T_2 \rightarrow (T_1)$

Start from last transaction of T_2.

$w_2(y)$ ✓ $w_2(y)$ ✓ $w_2(y)$ ✓ $T_1 \rightarrow T_2$ ... our choice. $w_2(y)$ ✗ $w_2(y)$ ✗

$\uparrow r_1(x) \uparrow w_1(x) \uparrow r_1(y) \uparrow w_1(y) \uparrow$

$\{r_2(z)\ w_2(z)\}$ can be placed anywhere because different data item (z), but only after $w_2(y)$

Case 1: $r_1(x)\ w_1(x)\ \boxed{w_2(y)}\ r_1(x)\ w_1(y)$

Case 2: $r_1(x)\ \boxed{w_2(y)}\ w_1(y)\ r_1(x)\ w_1(y)$

Case 3: $\boxed{w_2(y)}\ r_1(x)\ w_1(x)\ r_1(x)\ w_1(y)$

Case 1 : $r_2(y)$ ✓ $r_2(y)$ ✓ $r_2(y)$ ✓ ① ② ③

$\uparrow r_1(x) \uparrow w_1(x) \uparrow w_2(y) \uparrow r_1(y) \uparrow w_1(y) \uparrow$

$3 \times [{}^3C_1 + {}^3C_2]$

$3 \times [3+3]$

3×6

Case 1 = 18 ways

Out of 3 place put them $r_2(z)\ w_2(z)$ together $[{}^3C_1]$

or

Out of 3 places put them seperately. $[{}^3C_2]$

${}^3C_1 + {}^3C_2$

$r_2(y)$ $r_2(y)$ ① ② ③ ④

Case 2 : $\uparrow r_1(x) \uparrow w_2(y) \uparrow w_1(x) \uparrow r_1(y) \uparrow w_1(y) \uparrow$

2 ways $\Rightarrow 2 \times [{}^4C_1 + {}^4C_2]$

$2 \times [4+6]$

$= 20$ ways

Case 3 : ① ↑ $W_2(y)$ ① ↑ $R_1(x)$ ② ↑ $W_1(x)$ ③ ↑ $R_1(y)$ ④ ↑ $W_1(y)$ ⑤ ↑

$1 \times [{}^5C_1 + {}^5C_2]$

$5 + 10 = 15$ ways.

$T_2 \rightarrow T_1$:

Case 1 : 18

Case 2 : 20

Case 3 : 15

53 ways $+1 \Rightarrow$ 54 ways

$T_1 \rightarrow T_2$:

1 way

Q2. Consider the transaction T_1 and T_2 given below:

T_1 : $R_1(A)$ $R_1(B)$ $W_1(B)$

T_2 : $R_2(A)$ $R_2(B)$ $W_2(B)$

Where $R_i(A)$ denote read operation by transaction T_i on a data item (A) $W_i(B)$ denote a write operation on transaction T_i on data item (B).

The total number of conflict Serializable Schedule is ____.

Ans. $T_1 \rightarrow T_2$ ✓$W_1(B)$ ✓$W_1(B)$ $T_2 \rightarrow T_1$ ✗$W_1(B)$ ✗$W_1(B)$

↑ $R_2(A)$ ↑ $R_2(B)$ ↑ $W_2(B)$ ↑

Case I : ① ↑ $R_2(A)$ ② ↑ $W_1(B)$ $R_2(B)$ $W_2(B)$

$2C_1 + 2C_2$

$\rightarrow 2 + 1 = 3$ ways

Out of 2 place, place them together $2C_1$

Or

Out of 2 spot, place them seperately $2C_2$

Case II : ↑ $W_1(B)$ $R_2(A)$ $R_2(B)$ $W_2(B)$

1 way

$T_1 \rightarrow T_2$:

Case 1 : 3 ways

Case II : 1 way

4 ways

Because T_1 and T_2 are exactly same

$T_2 \rightarrow T_1$: 4 ways

$\therefore 4 + 4 \Rightarrow$ 8 ways

Total Conflict Serializable : 8 (Serial Schedule)

Non Serial Conflict Serializable : 8 − 2

= 6

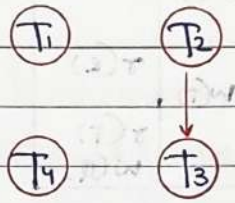
Q3 Consider the given Schedule:

S: $r_1(x)$, $r_2(y)$, $w_3(y)$, $r_4(x)$, $w_4(z)$, $w_3(y)$.

How many conflict Serializable Schedules exist for the above schedule S?

Ans.

T_1	T_2	T_3	T_4
r(x)			
	r(y)		
		w(y)	
			r(x)
			w(z)
		w(y)	



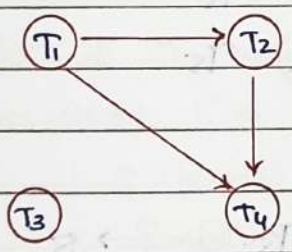
T_1 and T_4 can be placed anywhere.

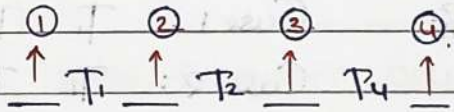
$\_\_T_2\_\_T_3.\_\_ 3! \times 2!$

= 12 ways

Topological Sorting Questions.

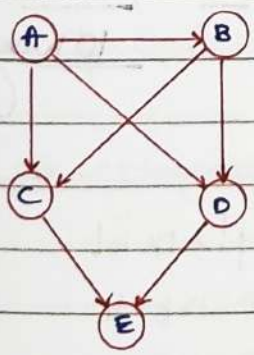
1.





Ans = 4.

2.

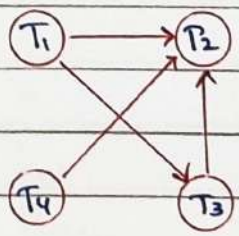


ABCDE } 2 ways.
ABDCE }

Ans = 2

3. $R_4(x)\ R_2(x)\ R_3(x)\ W_1(y)\ W_2(x)\ R_3(y)\ W_2(y)$

T_1	T_2	T_3	T_4
			R(x)
	R(x)		
		R(x)	
W(y)			
	W(x)		
		R(y)	
	W(y)		



<T1, T4, T3, T2>
<T4, T1, T3, T2>
<T1, T3, T4, T2>
} Ans

Q4. Consider the following Schedule

$S = r_1(P), r_3(S), w_1(Q), r_2(Q), r_4(Q), w_2(R), r_5(R), w_4(T), r_5(T), w_5(Q).$

How many Serial Schedules are possible which will be view equivalent?

Ans

T_1	T_2	T_3	T_4	T_5
r(P)		r(S)		
w(Q)				
	r(Q)		r(Q)	
	w(R)			r(R)
			w(T)	r(T)
				w(Q)

1. Initial Read : P : T_1 S : T_3
2. Final Write : Q : $T_5 \rightarrow T_1 \rightarrow T_5$
3. Updated Read (Write–Read Sequence)

$w_1(Q)\ r_2(Q) : T_1 \rightarrow T_2$

$w_1(Q)\ r_4(Q) : T_1 \rightarrow T_4$

$w_2(R)\ r_5(R) : T_2 \rightarrow T_5$

$w_4(T)\ r_5(T) : T_4 \rightarrow T_5.$

$T_1 \rightarrow T_5$

$T_1 \rightarrow T_2$ ($T_1 \rightarrow T_2 \rightarrow T_5$)

$T_1 \rightarrow T_4$ T_4 comes after T_1 and before T_5.

$T_2 \rightarrow T_5$

$T_4 \rightarrow T_5$

Case 1 : T_1 T_4 T_2 T_5

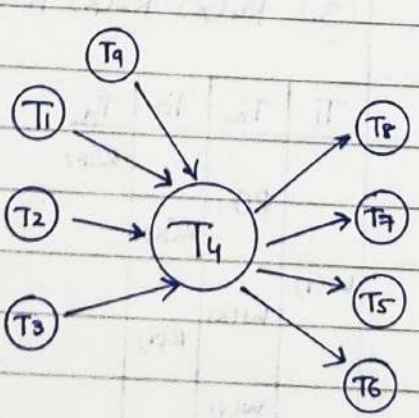
Case 2 : T_1 T_2 T_4 T_5

T_3 Can be placed anywhere.

Case I : ① T_1 ② T_4 ③ T_2 ④ T_5 ⑤ → 5

Case II : ① T_1 ② T_2 ③ T_4 ④ T_5 ⑤ → 5

10 ways. (Ans)



$4! \times 1 \times 4!$

$\Rightarrow 24 \times 24$

$\Rightarrow 576$

Date 23/2/23

Implementation of Concurrency Control.

Lock based Protocol

Request → Free → GRANTED → Lock Release / Remove
(Work finished.)
Not Free. → WAIT

Procedure:
Before using any data item, transaction should request for a lock, if lock is free (lock not taken by any data item) then it will be granted, otherwise transaction has to wait (if Busy).

Types of lock:

1. Shared lock (S) → Only read
2. Exclusive lock (X) → write (A) / (write (A), read (A)) / (read (A), write (A))

T_i \ T_j (Same data item)	S	X
S	Yes	No
X	No	No

* A lock is a mechanism to control concurrent access to a data item.
* Data items can be locked in two modes:
  1. exclusive (X) mode
  2. Shared (S) mode
* Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-compatibility matrix:

data item (A)	S	X
S	True	False
X	False	False.

* A transaction may be generated a lock on an item if the requested lock is compatible with lock already held on the item by other transactions.
* Any number of transactions can hold shared locks on an item.

* But if any transaction holds an exclusive on the item no other transaction may hold any lock on that item.

T_1	T_2
Read (A)	
A = A-100	
Write (A)	
	Read (A)
	Read (C)
Read (B)	
B = B+100	
Write (B)	

T_1	T_2	Lock Manager
Lock-X (A)		Grant-X (A, T_1)
Read (A)		
A = A-100		
Write (A)		
Unlock-X (A)		Release/Remove-X (A, T_1)
	Lock-S (A)	Grant-S (A, T_2)
	Read (A)	
	Unlock S (A)	Remove-S (A, T_2)
	Lock-S (C)	Grant-S (C, T_2)
	Read (C)	
	Unlock-S (C)	Remove-S (C, T_2)
Lock-X (B)		Grant-X (B, T_1)
Read (B)		
B = B+100		
Write (B)		
Unlock-X (B)		Remove-X (B, T_1)

* A locking Protocol is a set of rules followed by all transactions while requesting and releasing locks.
* Locking protocols enforce Serializability by restricting the set of possible Schedules.

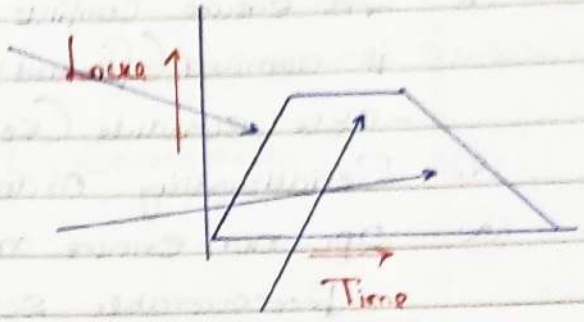
Two Phase Locking Protocol (2PL)

Lock and unlock request done in two phase

1. Growing Phase (Acquire locks)
2. Shrinking phase (Release locks)

Each transaction first finish its Growing phase then start Shrinking phase.

- A Protocol which ensures conflict Serializable Schedules.
- Phase 1: Growing phase
  - Transaction may obtain locks
  - Transaction may not release lock.
- Phase 2: Shrinking Phase
  - Transaction may release locks
  - Transaction may not obtain locks.



- The protocol assures Serializability. It can be proved that the transactions can be Serialized in the order of their lock points (ie. the point where a transaction acquired its final lock).

T_1	T_2
Read (A)	
A = A - 100	
Write (A)	
	Read (A)
	Read (C)
Read (B)	
B = B + 100	
Write (B)	

T_1	T_2	Lock Manager
Lock_X (A)		✓ Grant
Read (A)		
A = A - 100		
Write (A)	Lock_S (A)	✗ No
Lock_S (B)		
Unlock_X (A)	⟶	✓ Release_X (A, T_1)
	Lock_S (A)	✓ Yes Grant
	Read (A)	
	Lock_S (C)	✓ Yes Grant
	Unlock_S (A)	✗ Reveal.
Read (B)		
B = B + 100		
Write (B)		
Unlock_S (B)		✓

Lock Point: Position of last lock operation or first-unlock operation

Or

Position from where Shrinking phase of the transaction start.

Important Points about 2PL:

1. 2PL ensure Conflict Serializability (Serializability). If a schedule is allowed (followed) by 2PL then it ensure Conflict Serializ-able Schedule (Serializability).
2. Serializability Order is determined by Lock point.
3. 2PL not ensure recoverability.

Irrecoverable schedule followed by 2PL.

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

Irrecoverable Schedule.

T_1	T_2
X(A)	
R(A)	
W(A)	
Unlock(A)	
	S(A)
	R(A)
	Unlock S(A)
	Commit
Commit	

Irrecoverable Schedule But allowed (followed) by 2PL

(1)

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

T_1	T_2
X(A)	
W(A)	
denied by T_2 → X(B)	S(B)
	R(B)
	S(A) ← denied by T_1

∴ Deadlock.

(2)

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

T_1	T_2
S(A)	
R(A)	
denied by T_2 → S(B)	X(B)
	W(B)
	X(A) ← denied by T_1

∴ Deadlock.

4. 2PL Suffers from (may not be free from) deadlock.
5. 2PL suffers from Starvation.

	T_1	T_2	T_3	T_4
		S(A)		
denied by T_2 →	X(A)		S(A)	
		U(A)		
denied by T_3 →	X(A)			S(A)
			U(A)	
denied by T_4 →	X(A)			
✓ Granted →	X(A)			U(A)

2PL

Serializability ✓ Recourability ✗

if 2PL Applied

→ Ensures Conflict Serializable Schedule

→ Serializability

But: * not ensure recourability
* Suffer from deadlock
* Suffer from Starvation

Strict 2PL : 2PL + All exclusive lock (X lock) taken by the transaction must be held until commit/rollback.

→ X lock Release after Commit/rollback

→ It ensured : * Conflict Serializable
* recourable Schedule
* Cascadless (no cascading rollback)
* Strict recourable

→ Suffers from: deadlock and Starvation.

Rigorous 2PL : 2PL + All locks (Shared[S] and Exclusive[X]) must be held by the transaction until commit/rollback.

→ Suffers from: deadlock and Starvation.

Two phase locking protocol (Continued)

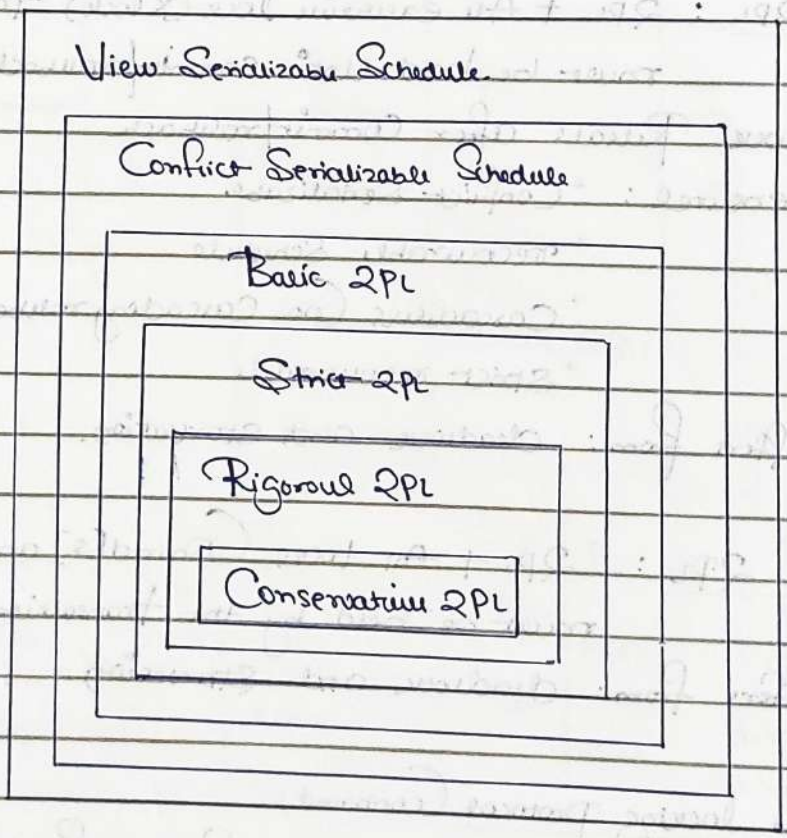
* Two phase locking does not ensure freedom from deadlocks.
* Extensions to basic two-phase locking needed to ensure recourability of freedom from cascading roll back
  1. Strict 2phase locking - a transaction must hold all its exclusive locks until commit/rollback.
     * Ensures recourability and avoid cascading roll backs.
  2. Rigorous two phase locking: a transaction must hold all locks until commit/abort.
     * Transactions can be serialized in the order in which they commit.
* Most databases implement rigorous 2-phase, but refer to it as simply 2-phase.

Conservative 2PL : Each transaction acquire all the lock in the beginning (at the start) of execution & release after commit.

It ensures:

- ✓ Conflict Serializability
- ✓ Recoverability
- ✓ Cascadless (no cascading rollback)
- ✓ Strict recoverable
- ✓ No deadlock.

But suffers from Starvation.



Q1. Consider the following database schedule with two transactions T_1 and T_2. $S = r_2(x), r_1(x), r_2(y), w_1(x), r_1(y), w_2(x), a_1, a_2$. where $r_i(z)$ denote a read operation by transaction T_i on a variable (z), $w_i(z)$ denote write operation of variable z and a_i denote or abort by transaction T_i.
Which of the following statements about schedule is true?

Ans. S does not have a cascading abort.

$r_2(x), r_1(x), r_2(y), w_1(x), r_1(y), w_2(x), a_1, a_2$

T_1	T_2
	$r(x)$
$r(x)$	
	$r(y)$
$w(x)$	
$r(y)$	$w(x)$
a_1	
	a_2

✓ Recoverable
✓ Cascadeless
× Strict Recoverable

Q2. Let S be the following schedule of operations of three transactions T_1, T_2, T_3 in relational database system.

$R_2(y), R_1(x), R_3(z), R_1(y), W_1(x), R_2(z), W_2(y), R_3(x), W_3(x)$.

Consider the statements P and Q below.

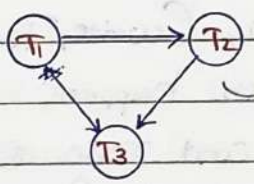
P: S is Conflict Serializable

Q: If T_3 commits before T_1 finishes, then S is recoverable

Which of the following choice is correct?

Ans.

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



Conflict Serializable

Not Recoverable

$\Rightarrow$ P is true and Q is false.

Undo and Redo

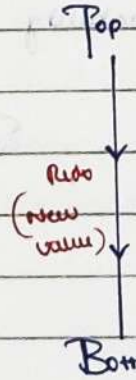
Transaction Number. (operation, data item, (OV) Old value, (NV) New value).

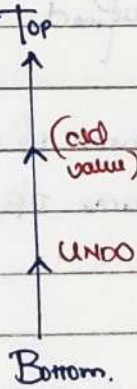
(x, 5, 7) (OV) (NV)

(x, 7, 11) (OV) (NV)

Redo: x : 5 → 7 → 11

↳ X = 11





Undo:

X : 11 → 7 → 5

X = 5

Date 24/8/23

* Those transaction commit before check point not required any Redo operation.
* Those transacrion commit after check point required redo operation.
* Those transacrion not commit require undo operation.

* If T_i commit then Redo
* If any transaction that not commit require Undo Operation.
* Log based recovery.

Q3. Consider a simple Checkpointing protocol and the following set of operations in the log.
(Start, T_4); (write, T_4, y, 2, 3); (Start, T_1);
(Commit, T_4); (write, T_1, 2, 5, 7);
{Checkpoint}
(Start, T_2), (write, T_2, x, 1, 9); (Commit; T_2)
(Start; T_3), (write, T_3, z, 7, 2);
If a crash happens now and the system tries to recover using both undo and redo operations, what are the contents of the undo and redo list?

Ans Undo : T_3, T_1 Redo : T_2.

Time Stamp Protocol

* A unique time stamp value assigned to each transaction when they arrive in the system.
* Based on this time stamp, determines the serializability order. (TSP predefined serializability order based on timestamp).

eg:
Time Stamp of T_1 : 10
Time Stamp of T_2 : 20

Serializability order : $T_1 \rightarrow T_2$
(T_1 : older transaction, T_2 : younger transaction)

Transaction	T_1	T_2	T_3	T_4	T_5	T_6
if TimeStamp of Transaction	10	30	20	40	60	50

Serializability order: $T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4 \rightarrow T_6 \rightarrow T_5$

(T_1: older transaction; T_5: larger transaction)

TimeStamp.

eg: $TS(T_1) > TS(T_2)$ Serializability Order: $T_2 \rightarrow T_1$

$TS(T_1) < TS(T_2)$ Serializability Order: $T_1 \rightarrow T_2$

* Each transaction T_i is issued a timestamp $TS(T_i)$ when it enters the System.
  * Each transaction has a unique timestamp.
  * Newer transaction have ~~transaction~~ timeStamp strictly greater than earlier ones.
  * Timestamp could be based on a logical counter.
    - Realtime may not be unique.
    - Can use (wall clock time, logical counter) to ensure.
* Time Stamp based protocols manage current execution such that time-stamp order = Serializability Order.

The timestamp Ordering (TSO) protocol

* maintains for each data Q two timestamp values:
  1. W-timestamp(Q) is the largest time-Stamp of any transaction that executed write(Q) Successfully.
  2. R-timestamp(Q) is the largest-time-stamp of any transaction that executed read(Q) Successfully.
* Imposes rules on read and write operations to ensure that:
  * any conflicting operations are executed in timeStamp order
  * Out of order ~~transaction~~ operations cause transaction rollback.

Two types of Time Stamp:

1. Transaction Time Stamp
2. Data item - time Stamp
   - (i) Read - Time Stamp (Q) $R_{TS}(Q)$
   - (ii) Write - Time Stamp (Q) $W_{TS}(Q)$

Read - Time Stamp (Q) : denotes the higher transaction time stamp that perform Read (Q) operation Sucessfully

$(10)T_1$	$T_2^{(20)}$	T_3 (30)
R(A)		
	R(A)	
		R(A)

Initially $R_{TS} = 0$

$R_{TS} = 10$, max (0, 10)

$R_{TS} = 20$, max (10, 20)

$R_{TS} = 30$, max (20, 30)

$R_{TS}(A) = 30$

Write - Time Stamp (Q) : denotes the higher transaction time stamp that perform write (Q) operation Sucessfully.

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

$W_{TS} = 0$ initially

$W_{TS} = 10$, max (0, 10)

$W_{TS} = 20$, max (10, 20)

$W_{TS} = 30$, max (30, 30)

$W_{TS}(A) = 30$

Conflict Operation	Non Conflict Operation
Same data item	R(A) → R(A) (in different data item)
R(A) → W(A)	R(A) → W(B)
W(A) → R(A)	W(B) → R(A)
W(A) → W(A)	W(B) → W(A)

Read (Q) : T_i

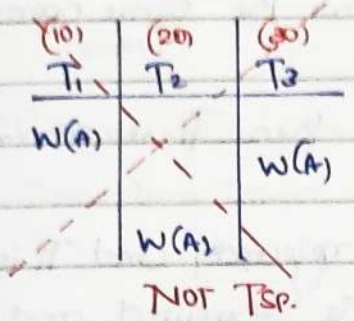
Transaction T_i wants to perform read(Q) operation

$TS(T_i) < W_{TS}(Q)$: Read operation and T_i rollback and restart with new Stamp.

Reject

T_i : Write(Q) : Transaction T_i wants to perform write(Q) operation.

$TS(T_i) < RTS(Q)$ } Write Operation & Transaction T_i rollback
$TS(T_i) < WTS(Q)$ } Reject. and restart with new timest.



$TS(T_2) < WTS(A)$
$20 < 30$
Not allowed

I T_i – Read(Q) (Transaction T_i issue R(Q) operation)

(i) If $TS(T_i) < WTS(Q)$: Read operation and T_i rollback/abort and restart with new transaction.

(ii) If $TS(T_i) \geq WTS(Q)$: Read operation is allowed.
and Set Read – $TS(Q) = \max[RTS(Q), TS(T_i)]$

II. T_i – Write(Q) (Transaction T_i issue Write(Q) operation)

(i) If $TS(T_i) < RTS(Q)$: Write operation reject and T_i rollback

(ii) If $TS(T_i) < WTS(Q)$: Write operation reject and T_i rollback.

(iii) Otherwise execute write(Q) operation.
Set Read $WTS(Q) = TS(T_i)$.

→ Suppose a transaction T_i issue a read(Q)

1. If $TS(T_i) \leq W\text{-timestamp}(Q)$, then T_i needs to read a value of Q that was already overwritten.
Hence the read operation is rejected and T_i is rolled back and restart with new timestamp.

2. If $TS(T_i) \geq W\text{-timestamp}(Q)$, then read operation is executed, and R-timestamp(Q) is set to
$\max(R\text{-timestamp}(Q), TS(T_i))$.

→ Suppose a transaction T_i issues write(Q).

(1) If $TS(T_i) < R\text{-timestamp}(Q)$, then the value of Q that T_i is producing was needed previously, and the system assumed that the value would never be produced. Hence the write operation is rejected and T_i is rolled back.

(2) If $TS(T_i) < W\text{-timestamp}(Q)$ then T_i is attempting to write an obsolete value of Q.
Hence the write operation is rejected, and T_i is rolled back.

(3) Otherwise the write operation is executed and W-timestamp(Q) is set to $TS(T_i)$.

Tsp: $TS(T_1) < TS(T_2)$

If $TS(T_1) : 10$ Order: $T_1 \rightarrow T_2$

If $TS(T_2) : 20$ T_1 followed by T_2.

Note:

Then all conflicting operations must be executed in the order T_1 followed by T_2 (same as Serializability order)

— if yes then allowed under Tsp

— if no then not allowed under Tsp.

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

$T_1 \rightarrow T_2 \rightarrow T_3$.
"Yes Tsp"

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

$T_1 \rightarrow T_2 \rightarrow T_3$.
"Yes"

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

$T_1 \rightarrow T_2 \rightarrow T_3$.
"Yes".

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

$T_1 \rightarrow T_2 \rightarrow T_3$.
"Yes".

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

$T_1 \rightarrow T_2 \rightarrow T_3$.
Not Allowed.
$TS(T_2) < WTS(A)$ not allowed
T_2 rollback & restart.

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

$T_1 \rightarrow T_2 \rightarrow T_3$
Not Tsp
T_1 Rollback

Why Restart new higher value?

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

If again restart with same time stamp.

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

again not allowed under TSP again rollback.

if higher time stamp?

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

yes allowed under TSP.

$T_2 \rightarrow T_1$

& Conflict operation

$T_2 \rightarrow T_1$.

Important Point about TSP:

1. If Schedule followed by TSP then ensure Conflict Serializable
2. TSP not ensure recoverability

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

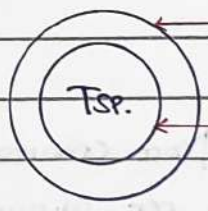
$T_1 \rightarrow T_2$

- allowed under TSP but irreplacable schedule

3. TSP free from deadlock.
4. Starvation may or may not occur

Thomas Write Rule (View Serializability)

* If S allowed by TSP then already it is allowed by Thomas write rule
* If some Schedule not allowed by TSP then it may be allowed by thomas write rule.



→ Some little modifications in TS.P

Allowed Obsolete Write. But obsolete write not allowed under TSP.

T_i : Read(Q) → Same as TSP.

$TS(T_i) < WTS(Q)$: Read Operation reject and rollback.

T_i : write(Q)

$TS(T_i) < WTS(Q)$: write operation and no rollback (Allowed Thomas) ignored.

$TS(T_i) < RTS(Q)$; Reject & T_i rollback.

Thomas Write Rule (Formal Definition)

- Modified version of the timestamp-ordering protocol in which obsolete write operations may be ignored under certain circumstances.
- When T_i attempts to write data item Q, if $TS(T_i) < W\text{-timestamp}(Q)$, then T_i is attempting to write an obsolete value of {Q}.
  - Rather than rolling back T_i as the timestamp ordering protocol would have done this {write} operation can be ignored.
- Otherwise this protocol is same as the timestamp ordering protocol.
- Thomas Write Rule allows greater potential concurrency.
  - Allows some view-serializable schedules that are not conflict serializable

(10) T_1	(20) T_2
R(A)	
	W(A)
Obsolete write → W(A)	

$TS(T_1) < WTS(A)$

$10 < 20$; not allowed under TSP

But allowed under Thomas write rule.

$T_i \Rightarrow W(A)$: ignored

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

Allowed under TWR.

→ More concurrency means this schedule is not conflict serializable, this is not allowed under TSP but allowed by Thomas Write Rule.

eg:

T_1	T_2	T_3
R(A)		
	W(A)	
W(A)		
Obsolete read.		W(A)

This is allowed under Thomas write rule.

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

Check TWR:

T_1: W(A) $TS(T_1) < WTS(A)$ — yes allowed in TWR

T_1: R(A) $\Rightarrow TS(T_1) < WTS(A)$ — not in TWR.

$T_1 \rightarrow T_2$

Not TSP, Not TWR

Time Stamp protocol: ensure Serializability, deadlock free, but Starvation possible.

Deadlock Prevention Algorithm:

(i) Wait–Die (ii) Wound–wait

Older Younger.

T_1 (10)	T_2 (20)	
R(A)		
	W(A)	Not in TSP
W(A)		But in TWR

$TS(T_1) < TS(T_2)$

$T_1 \rightarrow T_2$

Strict Time Stamp Ordering protocol = TSP + Strict Schedule.

↓

It Ensures: Serializability
Recoverability
Cascadeless (no cascading rollback)
Strict Recoverability

Q1. In which of the following lock scheme deadlock cannot occur?

Ans Conservative 2PL

Q2. Consider the following schedule: $r_1(x), r_2(y), r_2(x), w_1(z), r_1(y), w_3(y)$ $r_3(z), w_2(y), w_3(x)$.

Which of the time stamp ordering not allowed to execute schedule using Thomas Write Rule time stamping ordering protocol?

Ans $(T_1, T_2, T_3) = (10, 20, 30)$.

Q3. Consider the two statements about database transaction schedules.

I. Strict two phase locking protocol generates conflict serializable schedules that are also recoverable.

II. Timestamp ordering concurrency control protocol with Thomas write rule can generate view serializable schedules that are not conflict serializable.

Which of above statements is/are true?

Ans Both I and II.

Q4. Which of the following concurrency control protocols ensure both conflict serializability and freedom from deadlock?

1. 2-phase locking

II. Time stamp ordering.

Ans II Only.