UNIT-4

Transaction :-

- A transaction is a set of logically related operation. It contains a group of task.
- A transaction is a program unit whose execution change the contents of a database.

Database before transaction → After transaction opⁿ. → Database after transaction (consistent state)

Transaction Property :- (ACID Property)

There are 4 properties :-

(1) Atomicity :- (All or None) It states that all the operations of the transaction take place at once if not, the transaction is aborted.

e.g: A = ₹.2000 , B = ₹.3000 ⇒ A+B = 5000 (consistent)

T₁ :
R (A, a)
a - 500
Write (A, a)
R (B, b)
b = b + 500
Write (B, b)

after sucessfull completion
A = 2000 - 500 = 1500
B = 3000 + 500 = 3500
] = 5000

(2) Consistency:- Correctness.

It ensures that the integrity constraints are maintained so that the database is consistent before & after the transaction.

e.g: Total Sum (A,B) before transaction = Sum(A,B) after transaction

(3) Isolation:- It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.

T_1	T_2
R(A)	
a-500	
W(A)	R(A)
	R(B)
R(B)	print(A,B)
B=B+500	
W(B)	

A = 2000 /-
B = 3000 /-

a-500 = 1500

A = 1500
B = 3000
A+B = 4500/-

first T1 complete then T2 complete.

(4) Durability:- All updates done by a transaction must be permanent.

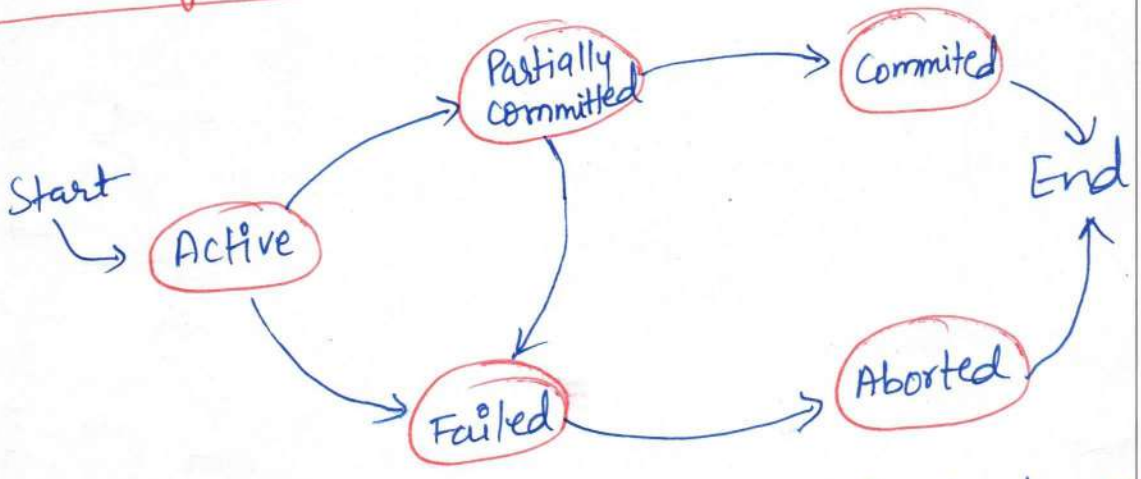
→ It ensures that the commit action of a transaction on its termination will be reflected in the database.

Operation on Transaction :- There are 2 operations :-

(1) Read (X) :- Read opⁿ is used to read the value of X from the database & stores it in a buffer memory.

(2) Write (X) :- write opⁿ is used to write the value back to the database from the buffer.

States of Transaction :-



(1) Active State :- It is the first stage of every transaction. In this state, the transaction is being executed.

(2) Partially committed :- In this state, the transaction executes its final operation, but the data is still not saved to the d/b.

(3) Committed :- A state, if all operations executes successfully. All the effects are permanently saved on the d/b system.

(4) Failed state :- If any of the checks made by the database recovery system fails. Then, the transaction said to be in failed state.

(5) Aborted :- If ~~any of the~~ transaction fails in the middle of the transaction then before executing the transaction all the executed transactions are roll back to its consistent state. After abort, there are two operations to choose
- Re-start the transaction
- Kill the transaction.

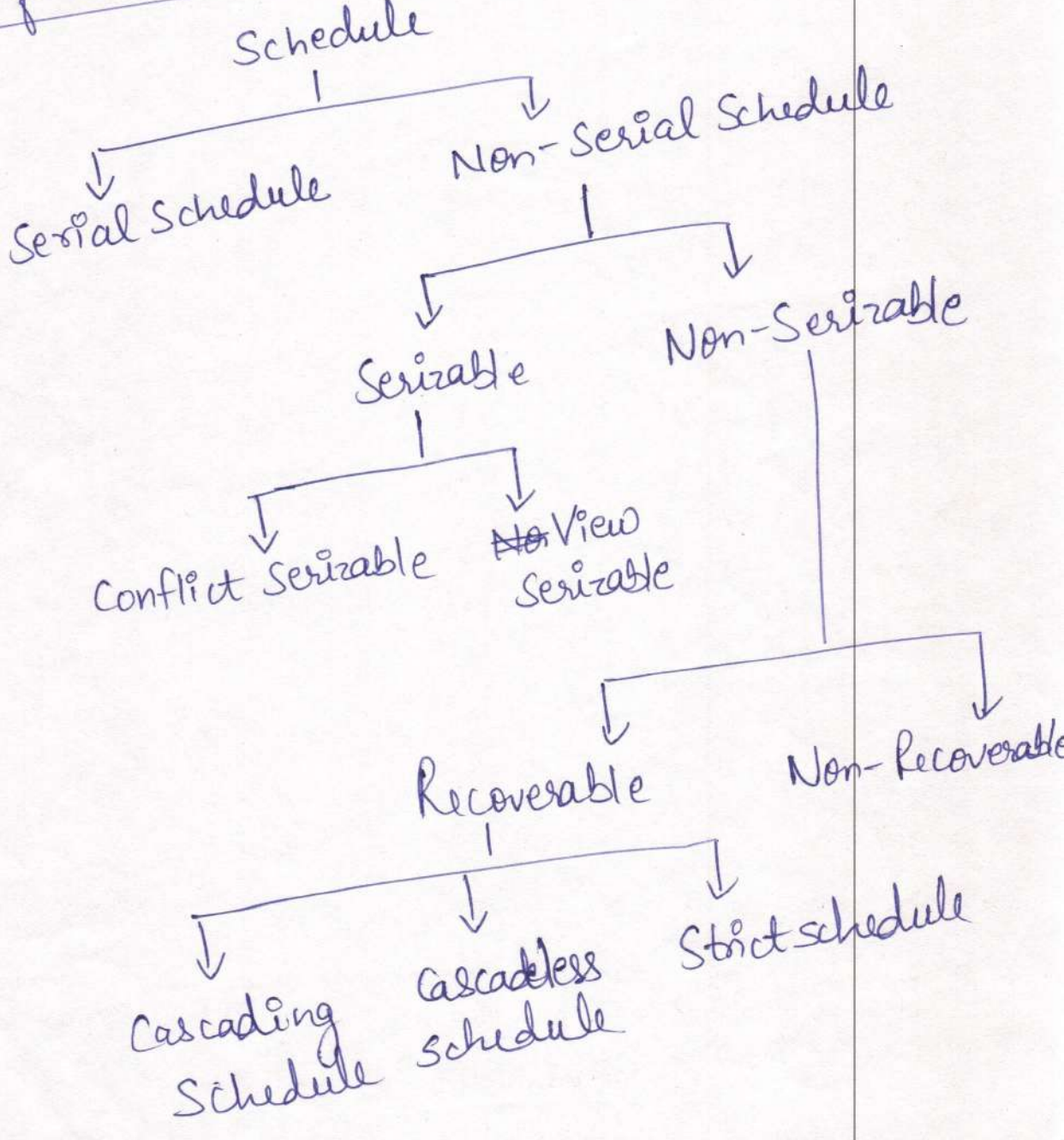
* Schedule :- When several transactions are executing concurrently then the order of execution of various instructions is known as Schedule.

Schedule:-

⇒ A Schedule 'S' of 'n' transactions $T_1, T_2, \ldots T_n$ is an ordering of operations of the transactions.

⇒ When several transactions are executing concurrently then the order of execution of various instructions is known as **Schedule.**

Types of schedule:-



(1) Serial Schedules:-

→ All transactions execute serially one after the other. When one transaction executes no other transaction is allowed to execute.

→ Serial schedules are always consistent.

e.g:

T1	T2
R(A)	
W(A)	
R(B)	
W(B)	
commit	
	R(A)
	W(B)
	commit

$T_1 \rightarrow T_2$

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

$T_2 \rightarrow T_1$

(2) Non-Serial Schedules:-

→ Multiple transactions executes concurrently.

→ Operations of all the transactions are interleaved or mixed with each other

→ It does not ensures consistency always.

e.g:

T1	T2
R(A)	
W(B)	
	R(A)
R(B)	
W(B)	
commit	
	R(B)
	commit

Serializable Schedule:- A schedule 'S' of 'n' transactions is serializable if it is equivalent to some serial schedule of same 'n' transactions.

(i) Conflict Serizable:-

→ If a given non-serial schedule can be converted into a serial schedule by swapping its non-conflicting operations, then it is called Conflict Serizable.

Conflict Operations:-

→ Both the opⁿ belong to different transactions.
→ Both the opⁿ are on same data item
→ At least one of the two opⁿs is a write opⁿ.

e.g:

T1	T2
R1(A)	
W1(A)	
	R2(A)
R1(B)	

Here, W1(A) & R2(A) are conflict opⁿs.

Testing Serializability:-

* We used Precedence Graph to test. It is directed graph.

Steps:-

(1) Create a node for each transaction.

(2) A directed edge, $T_i \rightarrow T_j$, if T_j reads a value of an item written by T_i.

(3) Directed edge $T_i \rightarrow T_j$, if T_j writes a value into item if it has been read by T_i.

(4) Directed edge, $T_i \rightarrow T_j$, if T_j write after T_i write.

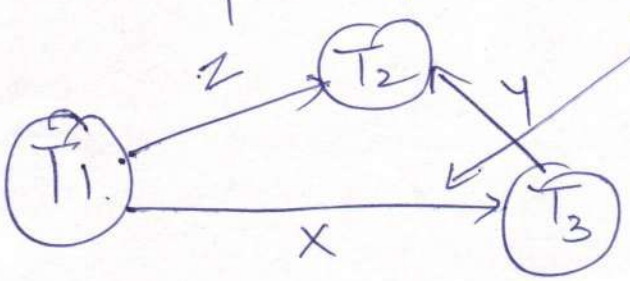
Note: A schedule is conflict serializable if precedence graph have "no cycle."

Ques:

T_1	T_2	T_3
R(X)		
	R(Z)	
R(Z)		
		R(X)
		R(Y)
		W(X)
	R(Y)	
	W(Z)	
	W(Y)	

R	W
W	R
W	W

→ conflict op^n.



No cycle

So, cycle present

∴ Not Conflict Serizable.

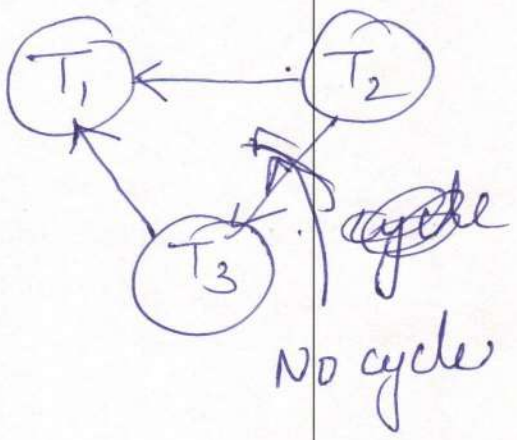
Q.

T_1	T_2	T_3
R(X)		
		R(Y)
		(R(X)
	R(Y)	
	R(Z)	
		W(Y)
	W(Z)	
R(Z)		
W(X)		
W(Z)		

Check conflict serizability Diagram

R W
W R
W W

Soln:



No cycle

Soln:

R(X), W(X) $T_3 \rightarrow T_1$

R(Z), W(Z) $T_2 \rightarrow T_1$

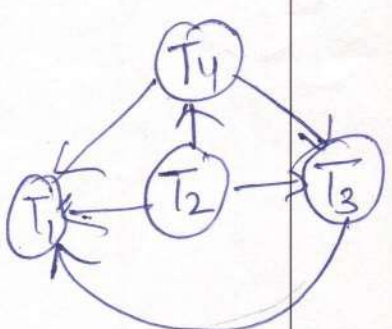
W(Z), R(Z) $T_2 \rightarrow T_1$ (Already draw) > No need to draw again

W(Z), W(Z) $T_2 \rightarrow T_1$ (" ") >

∴ This is ~~non~~ conflict serizable.

Q.30

T_1	T_2	T_3	T_4
	R(X)		
		W(X)	
		commit	
W(X)			
commit			
	W(Y)		
	R(Z)		
	commit		
			R(X)
			R(Y)
			commit



∴ No cycle
∴ S is conflict seriable.

Q. Check whether the given schedule S is conflict serizable or not –

S: $R_1(A), R_2(A), R_1(B), R_2(B), R_3(B), W_1(A), W_2(B)$

Soln:

T_1	T_2	T_3
$R_1(A)$		
	$R_2(A)$	
$R_1(B)$		
	$R_2(B)$	
		$R_3(B)$
$W_1(A)$		
	$W_2(B)$	

WR
RW
WW

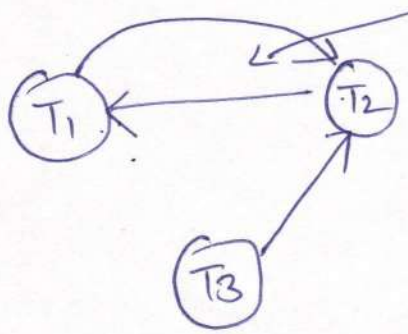
Now,

$T_2 \rightarrow T_1$ $(R_2(A), W_1(A))$

$T_1 \rightarrow T_2$ $(R_1(B), W_2(B))$

~~$T_2 \rightarrow T_1$ $(R_2(B), W_1(A))$ → Ignore already exist.~~

$T_3 \rightarrow T_2$ $(R_3(B), W_2(B))$



∴ S is not conflict serizable.

View Serializability :-

→ A schedule will be view serizable if it is view equivalent to a serial schedule.

→ Every conflict serizable is also view serizable schedule but not vice-versa.

① View Equivalent :- Two schedule S_1 & S_2 are view equivalent if they met the foll. conditions :—

(1) Initial Read :- An inital read of both schedule is same.

eg: S_1 :

T_1	T_2
R(A)	
	W(A)

S_2 :

T_1	T_2
	W(A)
R(A)	

(2) Update Read :- In schedule S_1 if T_i is reading 'A' which is updated by T_j then in S_2 also.

S_1 :

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

↑ Read T_3 but updated by T_2.

S_2 :

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

↑ Read T_3 but updated by T_1

∴ So, S_1 & S_2 are not view serizable.

③ Final Write:- Final write must be same b/w both the schedule.

e.g.: S_1:

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

S_2:

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

Same

e.g.: Check the foll. schedule is View serizable or not?

S: $R_2(z)\ R_2(y)\ W_2(y)\ R_3(y)\ R_2(z)\ R_1(x)\ W_1(x)\ W_3(y)\ W_2(z)\ R_2(x)\ R_1(y)\ W_1(y)\ W_2(x)$

Soln:

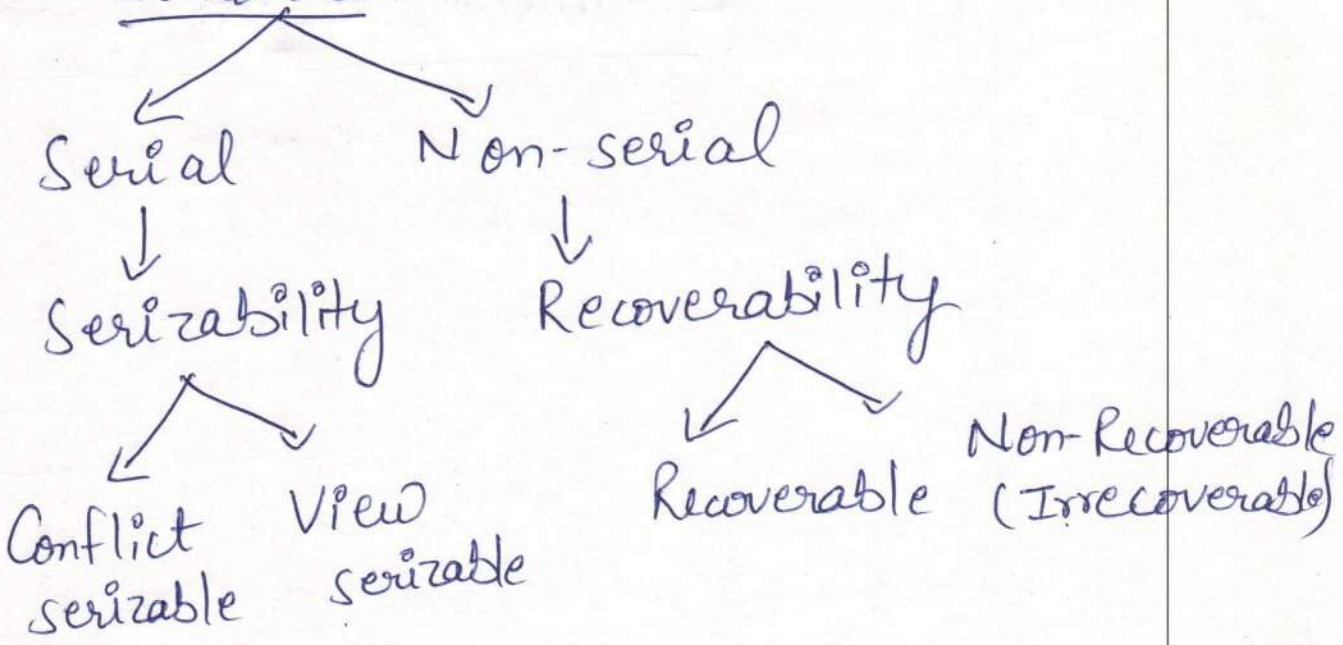
Data Item / Operation	x	y	z
Initial READ	T_1	T_2	T_2
Update READ	T_1, T_2	T_2, T_3, T_1	T_2
FINAL WRITE	T_2	T_1	T_2

order wegot: T_1, T_2 $\quad$ T_2, T_3, T_1 $\quad$ T_2

No proper order (Both order contradictory)

∴ It is not view serizable.

*. Schedules :-

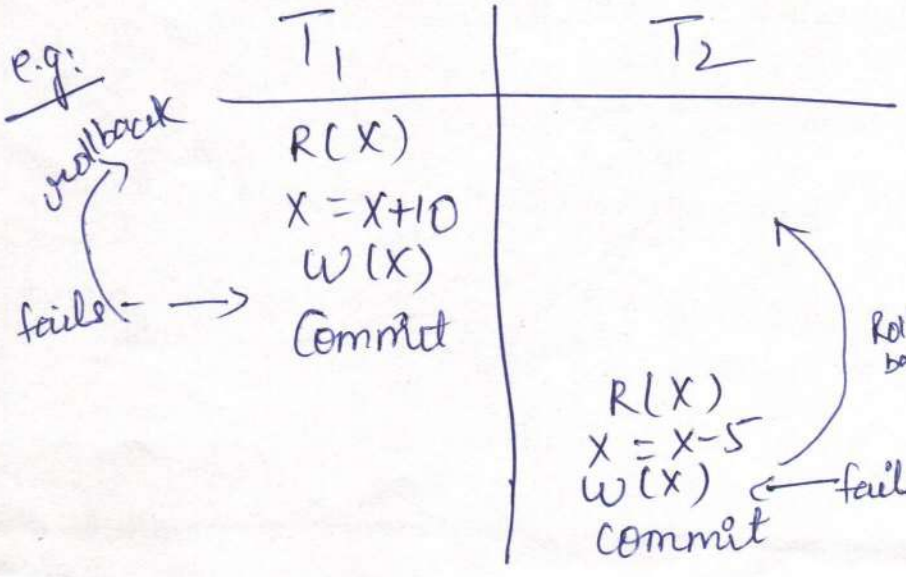


(1) **Recoverable :-** It is a schedule in which changes can be undone and the previous state can be restored.

* A committed transaction should never be rollback.

* A schedule 'S' is recoverable if no transaction T in S commits until all transaction T' that have written some item 'X' that T reads have committed.

e.g:

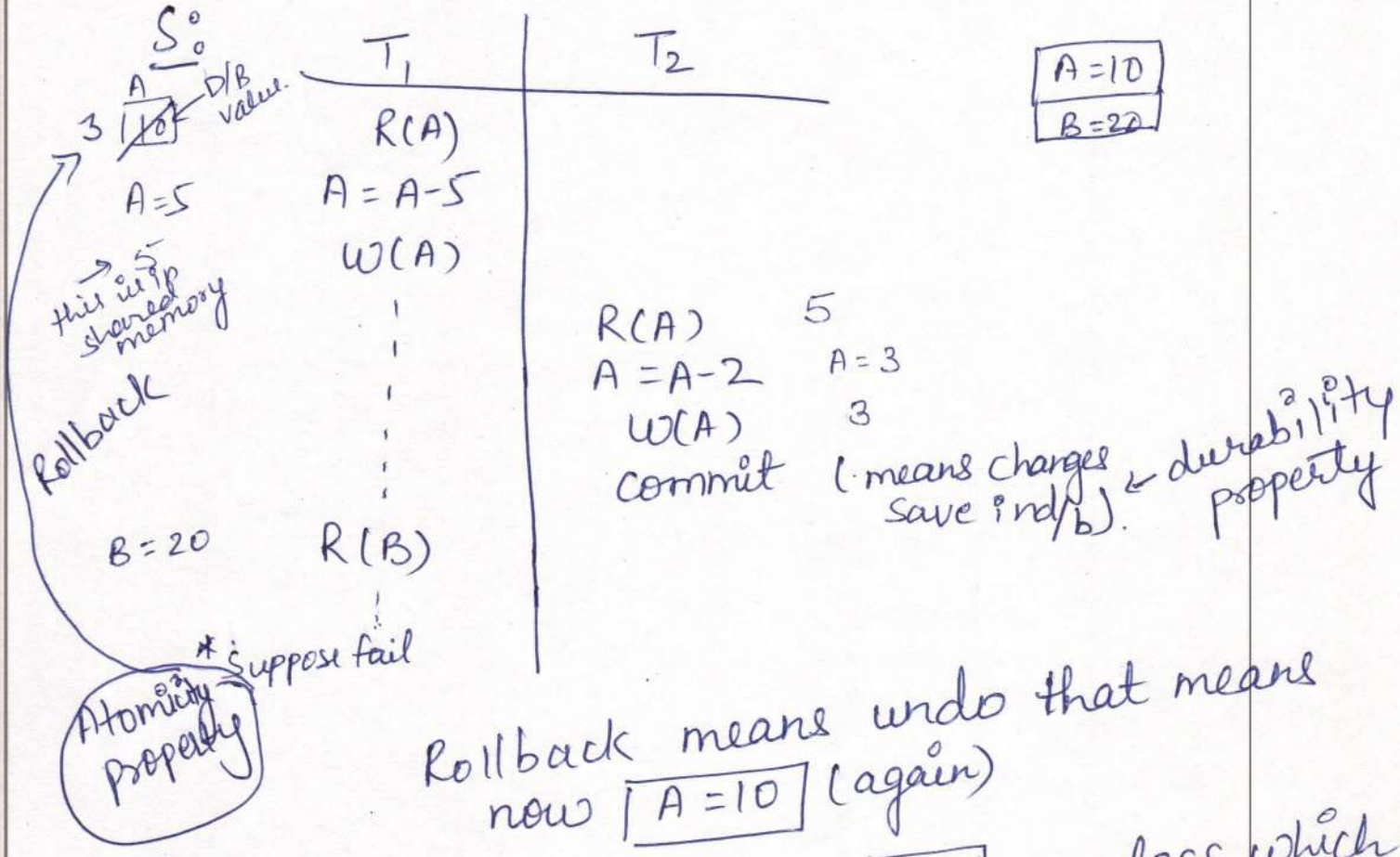


Transaction T_2 reading a committed value of X.

In case of any failure it can be recover.

(iii) Non-Recoverable (Irrecoverable):-

A schedule in which cannot be undone & the previous state cannot be stored.



Rollback means undo that means now A=10 (again)

The change value A=3 get loss which cannot be recover. This is known as irrecoverable schedule.

Ques: Consider the schedules:-

S1 : r1(X), w1(X), r1(Y), w1(Y), r2(X), w2(X)
C2, C1.

S2 : r1(X), w1(X), r2(X), r1(Y), w2(X), w1(Y)
C1, C2.

Check S_1 & S_2 is recoverable or not?

Soln:

S1

T_1	T_2
$r_1(X)$	
$w_1(X)$	⊗
$r_1(Y)$	
$w_1(Y)$	
	$r(X)$
	$w(X)$
	commit
commit	

If transaction failure.

→ we cannot recover

So. S1 is is not-recoverable

S2

T_1	T_2
$r(X)$	
$w(X)$	
	$r(X)$
$r(Y)$	
	$w(X)$
fail then T_1 will rollback $w(Y)$	
Fail	commit
	commit

T_2 will rollback

S2 is recoverable.

Note:

* All conflict serizable schedules are Recoverable,
* but not vice-versa.

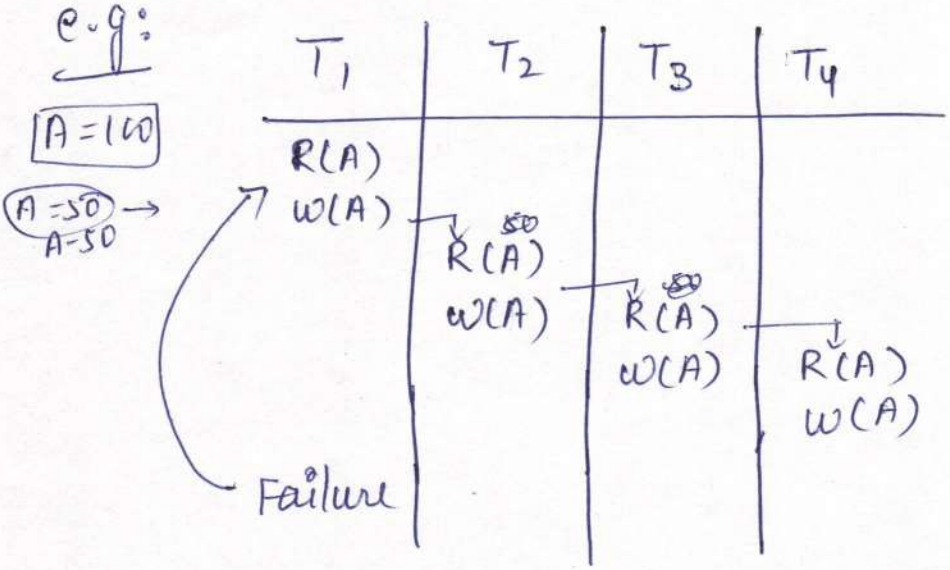
* Types of Recoverable schedules:—

1. Cascading schedules (Cascading Rollback)
2. Cascadeless schedules
3. Strict Schedules

(1) Cascading (cascading Rollback) Schedules:-

* A single transaction failure leads to a series of transaction roll back is called Cascading Rollback schedule.

e.g:



Here,
T_2 dependson T_1
T_3 >> >> T_2
T_4 >> >> T_3
So, failure of T_1 cause T_2 to rollback and so on.

Disadvanges:-

* Wastage CPU time.
* Degrade system Performance.

(2) Cascadeless Schedule:-

If in a ~~transaction~~ schedule, a transaction not allowed to read a data item until the last transaction has written it is committed or aborted, then such schedule is called as Cascadeless chedule.

* It always allows only committed READ op's.
* It avoids, cascading rollback & thus save CPU time.

e.g.:

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

→ It is cascadeless schedule b/z T_2 & T_3 read committed value of A item.

Note:

(3) Strict Schedule:- If in a schedule, a transaction is neither allowed to read or write a data item until the last transaction has written is committed or aborted(fail), then such schedule is known as Strict Schedule.

* It means allows only committed Read & Write operation.

e.g.:

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

This is strict schedule.

Note: All strict schedules are cascadeless but not vice-versa

Transaction failure, Recovery & Log :-

* Transaction failure :- A transaction has to abort when it fails to execute or when it reaches a point from where it can't go further. This is called Transaction failure.

Reasons for a transaction failure are -

(i) Logical error :- where a transaction cannot complete because it has some code error.

(ii) System error :- where the D/B system itself terminates an active transaction b/z DBMS is not able to execute it or has to stop because of some system condition. For eg:- in case of deadlock.

(iii) System failure / Crash :- H/w malfunction, etc.

(iv) Disk failure :- A disk block losses its contents during data transfer operation.

* Recovery from Transaction failure :-

⇒ The purpose of database recovery is to bring the database into last consistent state, which existed before the failure.

* There are types of techniques of recovery —

(1) Log based Recovery :—

=> Log : A log is a sequence of records, which maintains the records of actions performed by a transaction.
=> It is maintained during the normal transaction processing. It contains the enough information to recover from failure.
=> Log is written before any update is made to the d/b.

There are two techniques :—
(a) Deferred Database Modification
(b) Immediate " "

(a). Deferred (late) D/B Modification —

All logs are written on to the stable storage and the d/b is updated when a transaction commits. It ensures atomicity property, identify Transaction

T_1
R(A)
A = A + 100
W(A)
R(B)
B = B + 200
W(B)
Commit

A = 100
B = 200

Log

<T_1, Start>
<T_1, A, 200> ← new value
<T_1, B, 400>
<T_1, commit>

* For Recovery perform Redo Opn.

after commit

* If any failure occurs, then Redo operation perform.

* If failure occur, before commit then rollback perform. Old values store.

(b) Immediate database modification:-

⇒ Each logs allows an actual d/b modification. That is, the database is modified immediately after every operation.

e.g:

T_1	
	R(A)
A = ~~100~~ 200	A = A+100
B = ~~200~~ 400	W(A)
	R(B)
	B = B+200
	commit

Log of T1

<T1, Start>

<T1, A, 100, 200> (old value, new value)

<T1, B, 200, 400>

<T1, commit>

→ In Immediate we can't wait commit to change.

→ Start & commit present in T_1 log then save update value while perform recovery. (Redo Op^n).

→ But if failure occur before commit, only start present in log then "Undo" Operation. (old value from log store in d/b).

* Concurrency Problems in DBMS :—

=> When several multiple transactions execute concurrently in an unrestricted manner, then it might lead such problems. Such problem are called as concurrency problem. These are —

Concurrency Problems in Transactions
- ① Dirty Read Problem
- ② Unrepeatable Read Problem
- ③ Lost Update Problem
- ④ Phantom Read Problem.

① Dirty Read Problem :—

→ Reading the data written by an uncommitted transaction is called as dirty read.

→ This leads to inconsistency and higher chances might rollback.

e.g:

T_1	T_2
R(A)	
W(A)	
↓	R(A) // dirty read
↓	W(A)
↓	Commit
Failure	

rollback

When T_1 rollback then T_2 read values stands to be incorrect. ∴ database become inconsistent.

(2) Unrepeatable Read Problem:

The problem occurs when a transaction gets to read unrepeated i.e. different values of same variable in its different read opn even when it has not updated its value.

e.g:

T1	T2
	R(X) [X=10]
W(X) [X=15] update it	
	R(X) [X=15] // Unrepeated Read

=> T2 wonders how the value of X got changed b/z acc. to it, it is running in isolation.

(3) Lost Update Problem:- The problem occurs when multiple transaction executes concurrently & updates from one or more transactions get lost.

e.g:

T1	T2
A=10	
A=15 — R(A)	
W(A)	W(A) commit → // blind write A=50
!	
!	
Commit	

→ It is known as write-write conflict.
Acc. to T_1 value of [A=15] & acc. to T_2 value of [A=50].

(4) Phantom Read Problem —

This problem occurs when a transaction read some variable from the buffer & when it reads the same variable later, it finds that the variable does not exist.

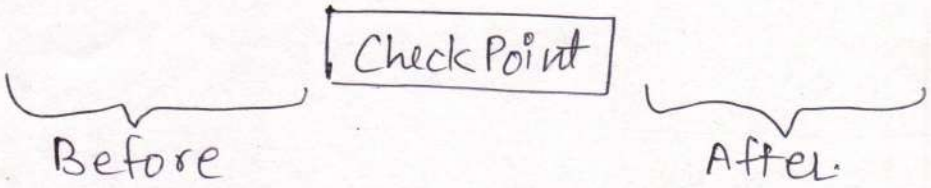
e.g.:

T_1	T_2
R(X)	
	R(X)
Delete(X)	
	Read(X) ← T_2 tries to reading X but does not find it.

* Concurrency Control Protocols helps to prevent the occurence of above probles & maintains the consistency of the database.

-!- Checkpoints:-

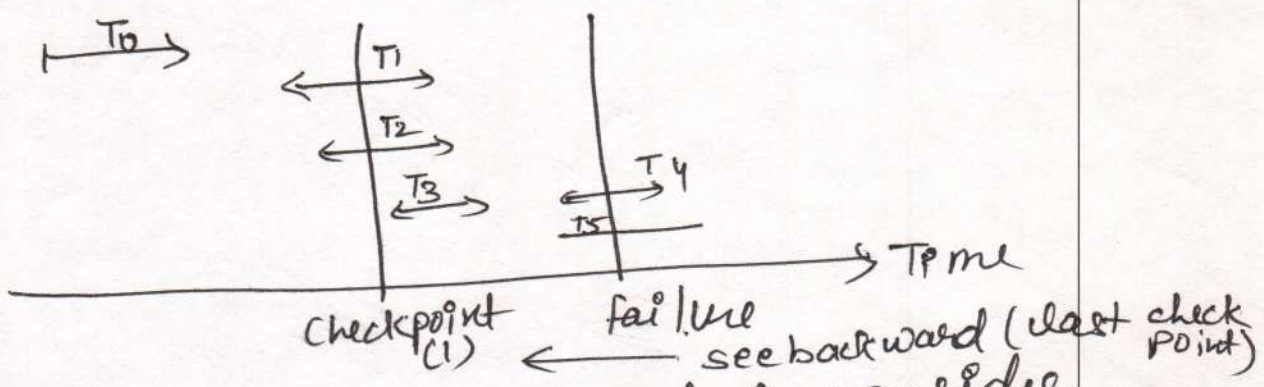
It is a mechanism where all the previous logs are removed from the system & stored permanently in storage disk.



Need?

* When more than one transactions are being executed in parallel, the logs are interleaved. It is difficult to backtrack all logs at the time of recovery. So the solution is "Checkpoint".

=> Checkpoint declare a point before which the DBMS was in consistent state & all the transactions were committed.



=> During recovery we need to consider only the most rececent transaction T_i that started before checkpoint & transactions that started after T_i.

=> When the system with concurrent transactions crashes, the recovery system reads the logs backwards from the end to the last checkpoint.

It maintains two lists —

(i) Re-do list :- If the recovery System sees a log with $< T_n$, start $>$ & $< T_n$, commit $>$ or just $< T_n$, commit $>$, it puts the transaction in the re-do list.

e.g:- T_1, T_2, T_3

(ii) Undo-list :- If the recovery system sees a log with $< T_n$, start $>$ but no commit or abort, it puts the transaction in undo list.

e.g:- T_4, T_5

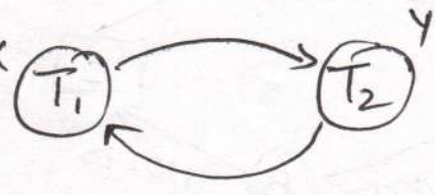
Advantages :-

(1) It saves time.

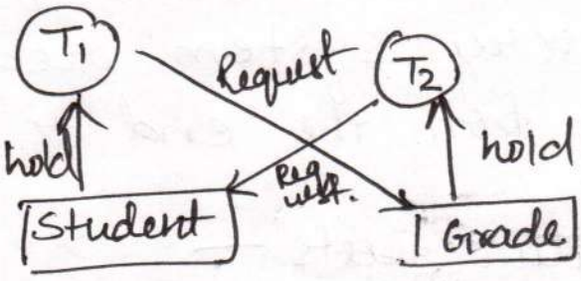
Deadlock :-

In database, a deadlock is an unwanted situation in which two or more transactions are waiting indefinitely for one another to give up locks.

e.g:-



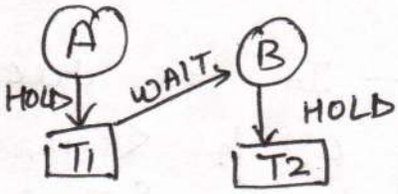
T_1 hold lock X
T_2 hold lock Y
T_1 wants to read Y but it is holding by T_2 & T_2 wants to read X but it is holding by T_1.



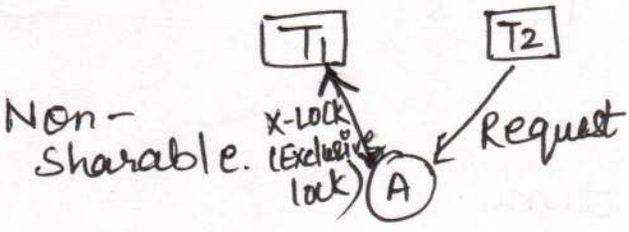
Deadlock

- Deadlock ~~Handling~~ Conditions :- There are foll. necessary conditions for deadlock —

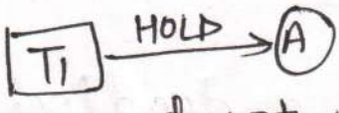
(i) Hold & Wait :-



(ii) Mutual Exclusion :-

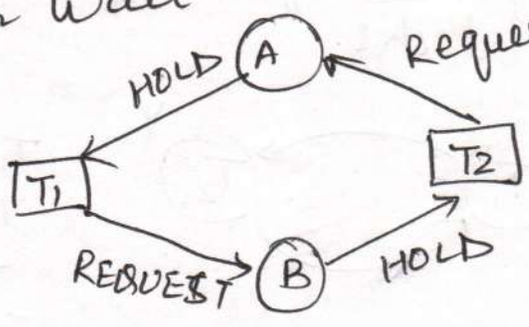


(iii) No-Preemption :- Forcefully release of resource is not allowed.



T_1 hold A and not leave after completion also.

(iv) Circular Wait



Deadlock Handling:-

There are two methods for handling deadlock —

(i) Deadlock Prevention
(ii) Deadlock detection & Recovery.

(1) **Deadlock Prevention:-** There two techniques for deadlock prevention:-

(a) Wait - Die scheme
(b) Wound Wait scheme

(a) **Wait - Die Scheme:-** In this scheme, if a transaction requests a resource that is locked by another transaction, then DBMS simply checks the timestamp of both transaction & allow the older transaction to wait until the resource is available for execution.

Suppose, T_i requests a resource held by T_j. Then,

if $T_s(T_i) < T_s(T_j)$ [T_i is older than T_j]

T_i allowed to wait otherwise T_i ~~abort~~ rollback. with same timestamp.

(b) ~~Wait~~ **Wound - Wait Scheme:-** In this scheme, if an older transaction requests for a resource held by a younger transaction, then an older transaction ~~it~~ forces a younger transaction to Kill

the transaction & release the resource. The younger transaction restarted with a minute delay with same timestamp.

If $T_s(T_i) > T_s(T_j)$ T_i = younger, T_j = older

then T_i waits

Else

T_i Rollback or abort T_j & restart it with same timestamp.

(2) Deadlock Detection:- The simplest way to detect a state of deadlock is "Wait-for Graph".

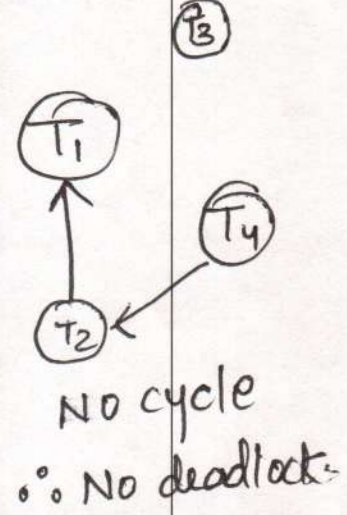
⇒ Wait-For Graph:- In this method, a graph is drawn based on the transaction & their lock on the resources. If the graph created a closed-loop or a cycle, then there is a deadlock.

$G = (V, E)$

$T_i \rightarrow T_j$ (T_i is waiting for a resource/data item hold by T_j.

e.g:-

Transaction	Data Items	Lock Mode
T_1	Q	Shared
T_2	P	Exclusive
	Q	Exclusive
T_3	Q	Shared
T_4	P	Exclusive



No cycle

∴ No deadlock