Transaction Concurrency UNIT-IV

Transaction: Transaction refers to a set of operations used to perform a logical unit of work. A transaction generally represe a change in the database

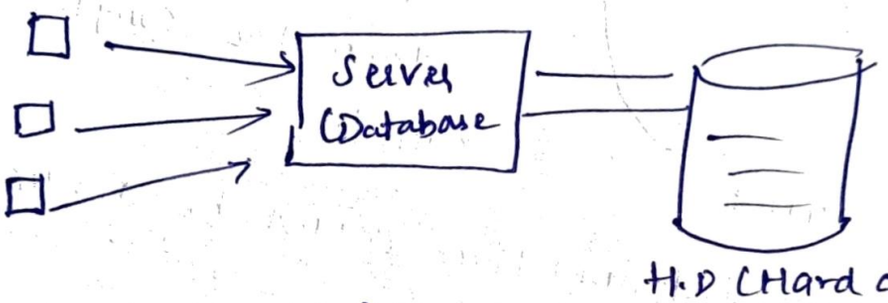
Database transactions have 3 operations:- Read, write & comm

Read operation is used for accessing of database.

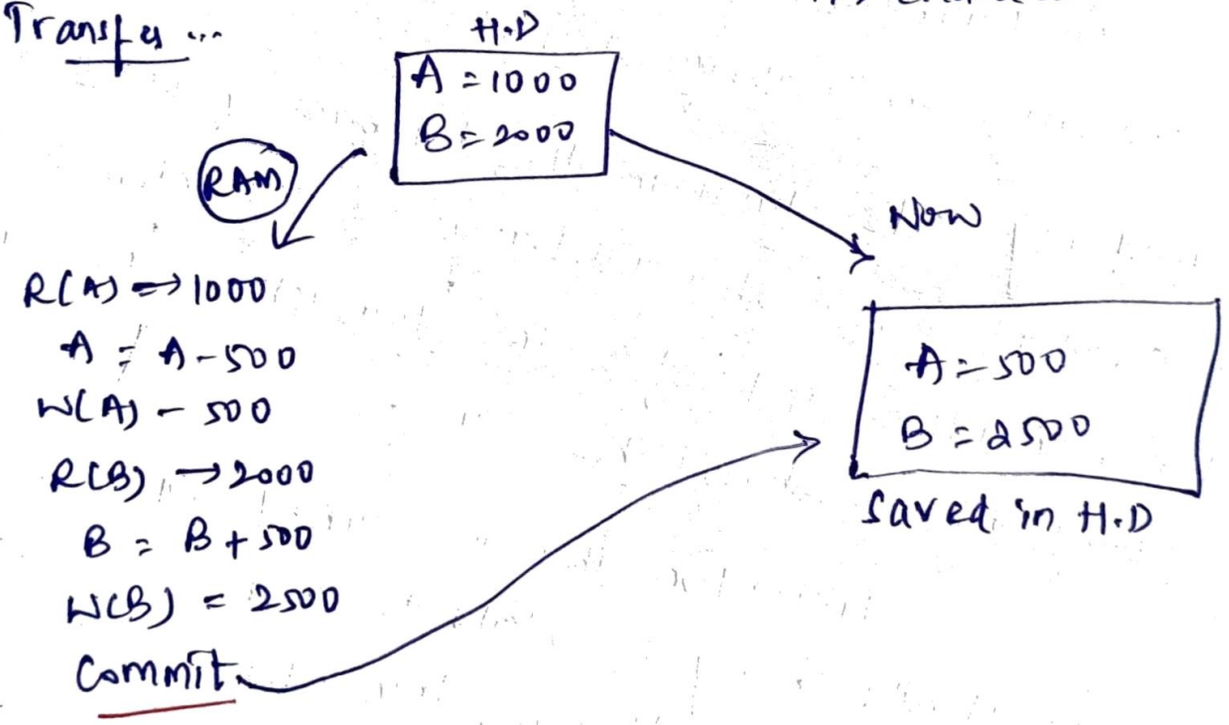
Write operation is used for modifying/change the database (user mode)

When we read or access any data from HDD (Hard Disk) then it comes to RAM (where we perform operations)

Commit: Whatever changes we make, those are saved permanently in Hard disk. once we commit.



Transfer ...



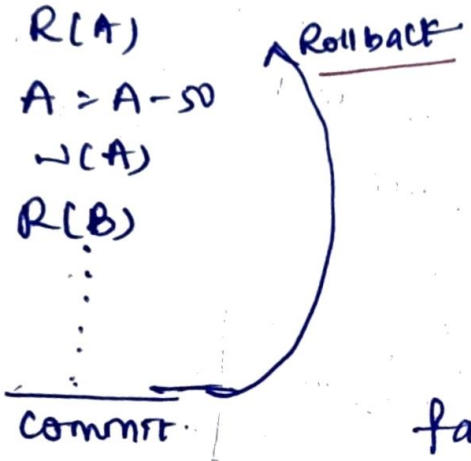
ACID properties of a transaction:-

A → Atomicity
C → Consistency
I → Isolation
D → Durability

} At backend

Atomicity: Atomicity ensures that the transaction is either executed completely or not executed at all. Incomplete transaction consequences are not valid.

Ex:



If a transaction has 1000 operations, and if due to any failure if it stops by 999th operation, then the complete transaction is failed.

A failed transaction cannot be resumed. A failed transaction will always restart.

Consistency: The data in the database must always be in a consistent state. Consistency exactly refers that before the transaction and after the transaction, the sum of ~~money~~ amount should be same

A transaction that is carried out on a consistent data should bring the data to another consistent state. after execution. However, transaction need not maintain

Consistency at intermediate stages.

ex:

A = 2000
B = 3000

sum = 5000

Before transaction

T_1

R(A) = 2000
A = A - 1000
W(A) = 1000
R(B) = 3000
B = B + 1000
W(B) = 4000
Commit; ⟶

A = 1000
B = 4000

sum = 5000

After transaction

sum is same. It means it is consistent.

Isolation:

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

→ parallel transaction

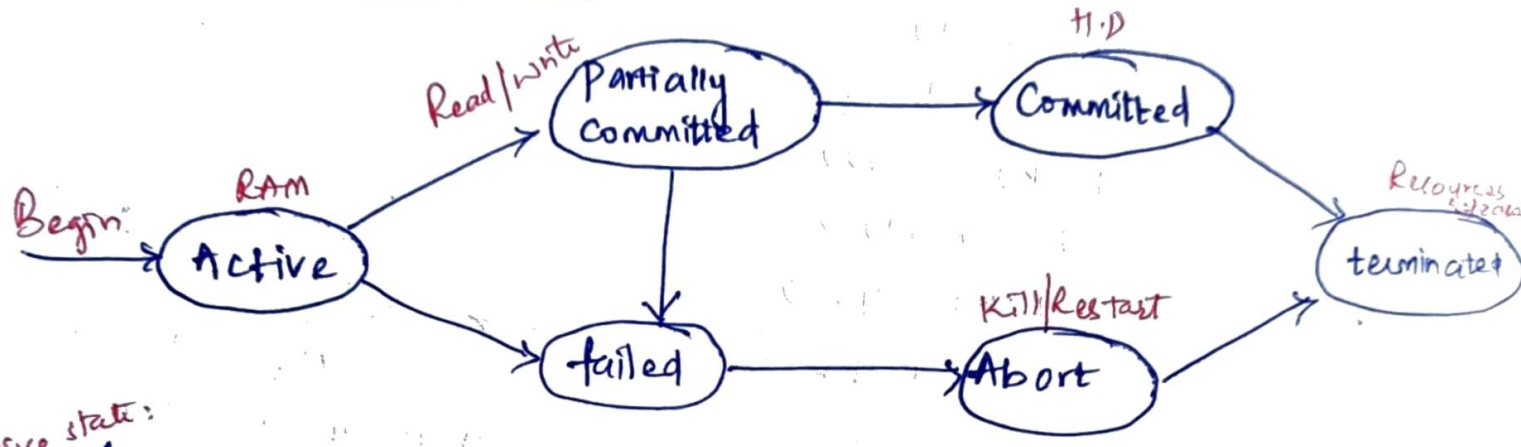
We try to convert a parallel schedule into a serial schedule (Conceptually).

All transactions must run in isolation from one another that is every transaction should be kept unaware of other transactions and execute independently. The intermediate results generated by the transactions should not be available to other transactions.

Durability:- Whatever the changes we make they must

be permanent. They must be updated for lifetime until we again update the data. That's why, we save data in HDD for durability.

States of transaction:



Active state: Initially, transaction is in passive state. And as we start executing it, it comes into active state.

Partially Committed state: A transaction is said to be in partially committed state if it finishes all operations except "Commit". If there are n operations then (n-1) are finished. Here, there may be chances for a transaction to get aborted due to hardware or software failure. In order to avoid data loss, the data is stored temporarily in RAM. This helps in recreating the updates done by the aborted transaction in case of system failure

Committed: A transaction is committed if each and every statement of a transaction is executed successfully. and the changes made are saved to hard disk.

Failed: A transaction is in failed state, if the execution of the transaction cannot be processed further due to power

failure, hardware or software failure. The transaction may fail either from active or partially committed state. On this case, the system performs the rollback operation.

After failure, a transaction is never resumed it always performs rollback. It again starts executing from beginning.

Abort: The rollback transaction enters the abort state when the system performs any one of the following operation.

(i) Restart: If the transaction was aborted due to hardware or software failure, then the transaction can be restarted, such transaction is considered as a new transaction.

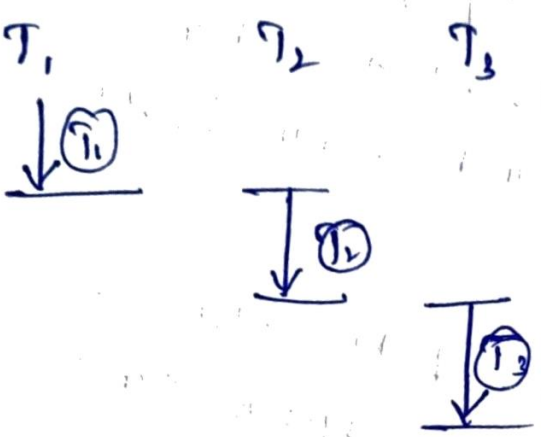
(ii) Kill: If the transaction was aborted due to some internal logical error or due to bad input then the system can kill the transaction.

Terminated: Here, we deallocate all resources which are used for transaction once the transaction is executed successfully. Resources are also deallocated when the transaction is failed/aborted.

SCHEDULE: Schedule is a series of actions that represent sequence of execution. Therefore, a schedule for any set of transaction comprises of instructions that are executed by those transaction. The execution order of the instructions within a transaction is determined by a schedule, (i.e. it specifies whether write instruction should appear before

or after the read instruction).

Serial schedule: Serial schedule is a schedule wherein th transactions are executed one after the another sequential The no. of serial schedules generated for a given sche depends on the number of transaction (ie, if there are K transactions, then K! serial schedules are generated).



T_1	T_2
Read(x)	
Write(x)	
Read(y)	
Write(y)	
	Read(x)
	write(x)
	Read(y)
	write(y)

Serial schedule

Advantage:

Consistent because there is no interfearance while executing.

Disadvantage:

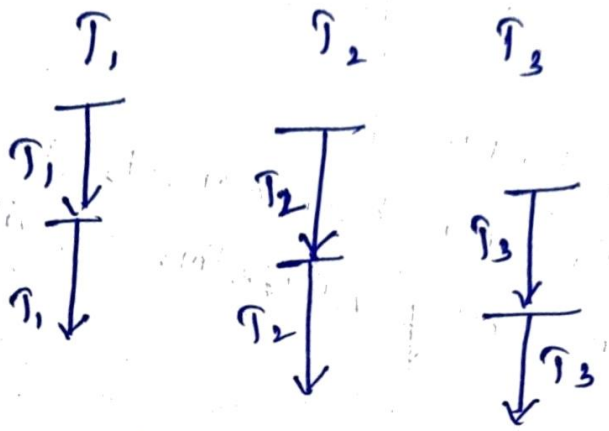
Only one transaction is performed at one time. Other transactions have to wait until it gets finished / failed.

Parallel schedule: If multiple transactions are executed concurrently, then the schedule is a parallel schedule. In concurrent execution, the Operating system initially executes few instructions of first transaction and then CPU performs context switching and executes the instruction of second transaction. Later it switches back to first transaction

and executes the remaining instructions. and so on. If several transactions are executed concurrently, then CPU time is shared synchronously between all these transaction

Advantage: Performance is improved (throughput ie, no. of transactions executed per unit time is increased)

Disadvantage: It is not always true that concurrent executions leads to consistent state ie, they may be schedu that may lead to incorrect result.

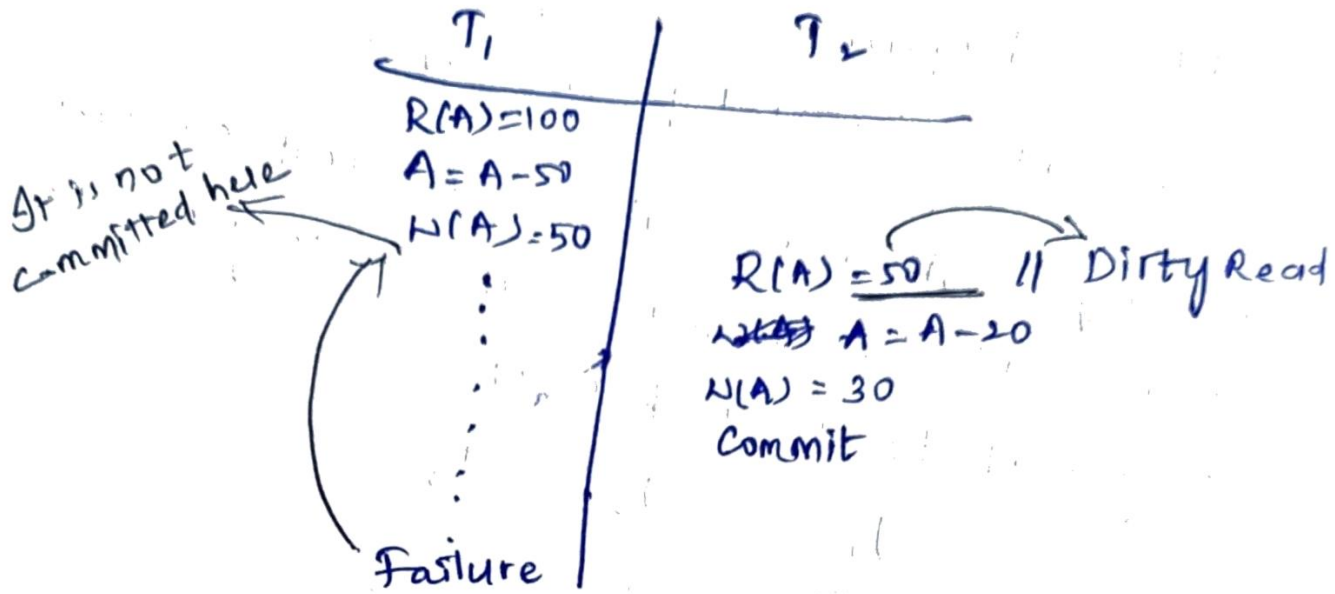


Parallel schedule

Types of Problems in Concurrency:-

1) Dirty Read
2) Incorrect summary
3) Lost Update
4) Unrepeatable read.
5) Phantom read

1) Dirty Read problem: This problem occurs when a transaction updates an item and fails. But the updated item is used by another transaction before the item is changed or reverted back to its last value.



If T_1 fails then how can T_2 read A as 50, so it has worked on wrong value. This is nothing but dirty read problem.

2) Incorrect Summary problem:- Consider a situation where one transaction is applying the aggregate function sum() on some records. While another transaction is updating these records. The sum() function may calculate some values before the values have been updated and others after they are updated

T_1	T_2
	sum=0
	R(A)
	sum=sum+A
R(X)	
X=X-N	
W(X)	
	R(X)
	sum=sum+X
	R(Y)
	sum=sum+Y

R(Y)
Y = Y + N
W(Y)

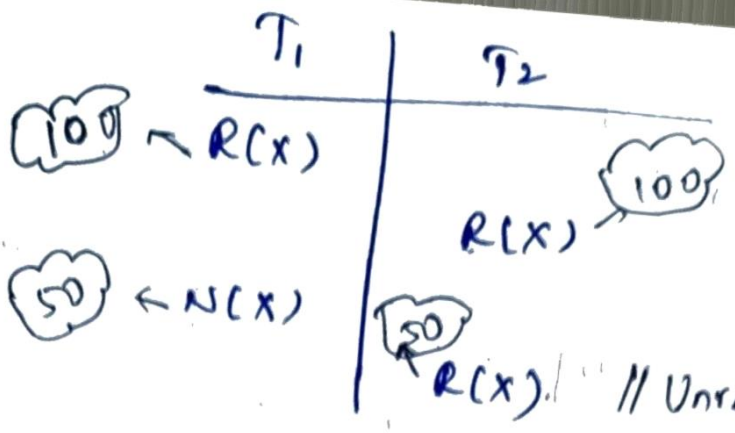
In the above example, T_2 is computing the sum of records while T_1 is updating them. Therefore, the aggregat function may calculate some values before they have bee updated and others after they have been updated

3.) Lost Update problem:- In the lost update problem, an upda done to a data item by a transaction is lost as it is ~~overwri~~ overwritten by the update done by another transaction.

T_1	T_2
R(X)	
X = X + 20	
~~W(X)~~	X = X + 30 lost update
W(X)	W(X)

In the above example, T_2 changes the value of X but it will get overwritten by the write commit by transaction 1 on X. Hence, the update done by T_2 will be lost. Basically, the write commit done by the last transaction will overwrite all previous write commits

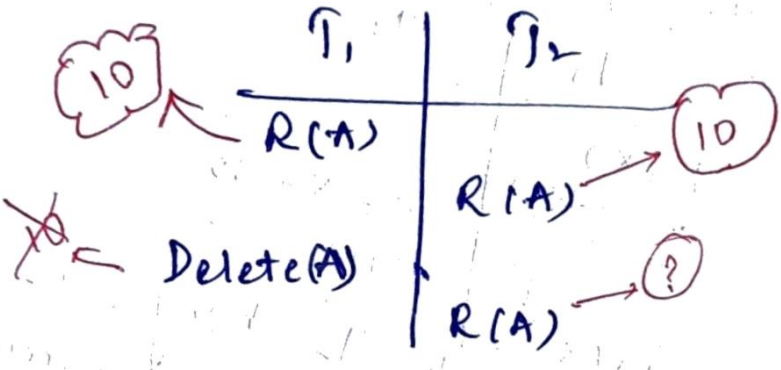
Unrepeatable Read problem:- This problem occurs when two or more read operations of the same transaction read different values of the same variable.



// Unrepeatable read

T_2 reads two different values of same variable x. This is known as unrepeatable read.

5) Phantom Read problem:- This problem occurs when a transaction reads a variable once but when it tries to read that same variable again, an error occurs saying that the variable doesn't exist.



In the above example, once T_2 reads variable A, T_1 deletes the variable A without T_2's knowledge. Thus, when T_2 tries to read A, it is not able to do it.

Conflict Equivalent Schedule:-

T_1	T_2	
R(A)	R(A)	non-conflict pair
R(A)	W(A)	conflict pairs
W(A)	R(A)	
W(A)	W(A)	

Read & write on same variable by two different transaction becomes conflict pairs

Here A & B are two diff variables so no problem.

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

} non-conflict pairs

Two transactions on same variable, then it becomes conflict pairs like R(A), W(A)
T_1 T_2

~~How to Convert~~ NOTE:- If we have adjacent non-conflict pairs, then we can swap their position. This will be useful in converting non-serializable schedule to a serializable schedule.

ex:

T_1	T_2
	R(A)
R(B)	

⟹

T_1	T_2
R(B)	
	R(A)

Check whether given schedule is conflict equivalent or not? $S \equiv S'$ check

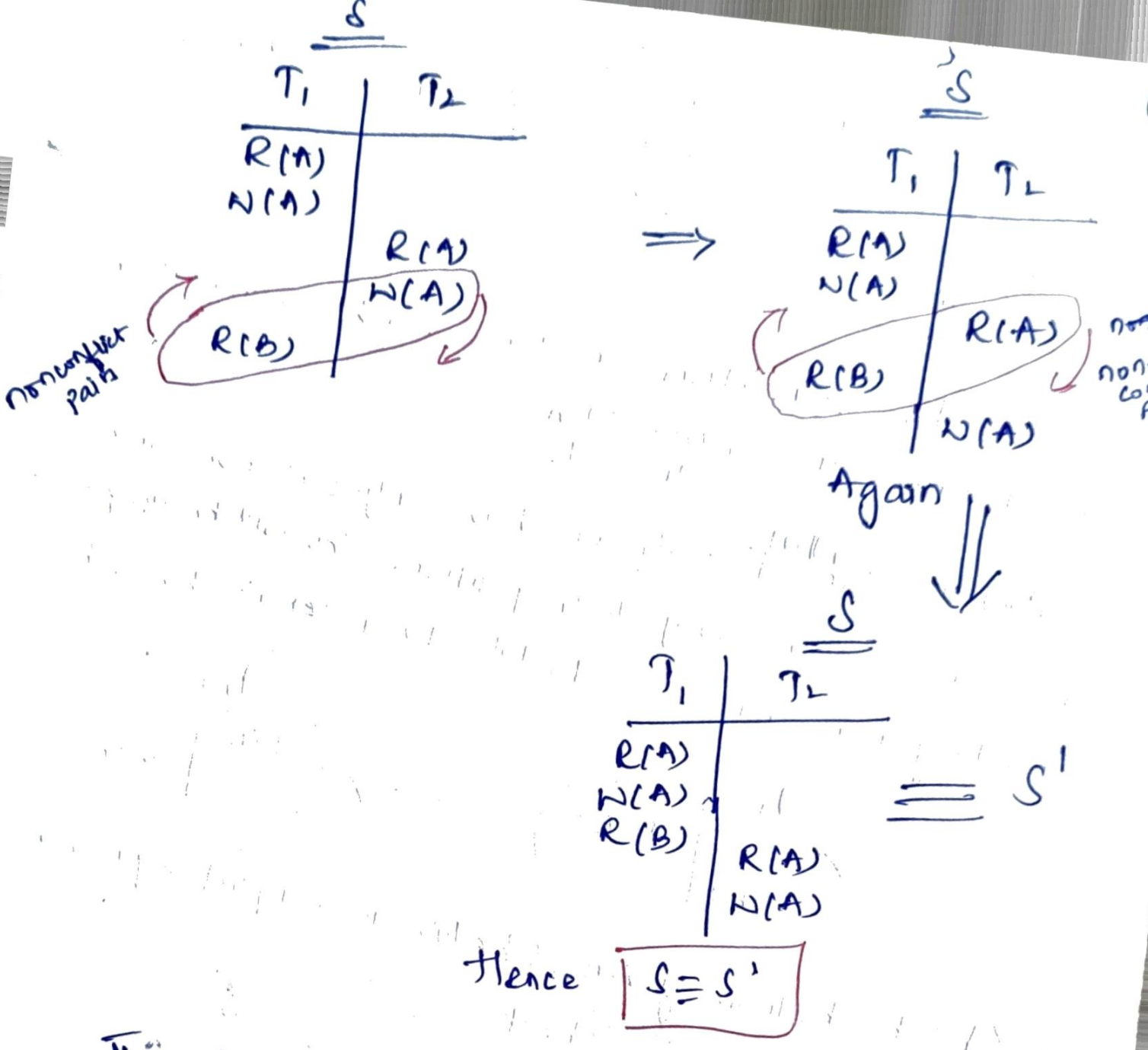
S

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

S'

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

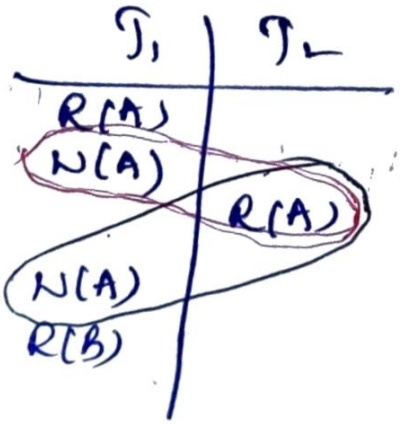
In 'S', we have adjacent non-conflict pair, so we swap them



Hence $S \equiv S'$

The schedules are conflict equivalent and clearly ~~the~~ It is a serial schedule. Like here we can say after completion of T_1 transaction, T_2 is executed. So serially they are executed.

Ex: 2



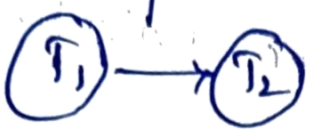
In this, example, there doesn't exist non conflict pairs, so positions cannot be changed. so this schedule cannot be converted into serial schedule.

Serializability: If a given schedule can be converte into a serial schedule then it is known as serializabi

S

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

(i)



S

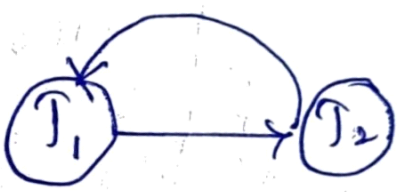
T_1	T_2
	R(A)
	W(A)
R(A)	
W(A)	

(ii)

By ~~se~~ above examples it is clear that both schedules are executing ~~bet~~ in a serial manner. In (i) T_1 is first executed then comes T_2 and in (ii) T_2 is first executed and then T_1. It means either T_1 or T_2 is executed first ~~then~~ completely then only other one starts executing.

Now, now parallel schedules execute,

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



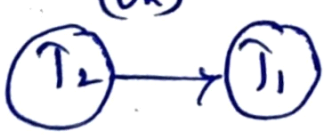
→ Convert it into serial schedule ~~(but it can be done) and conflict pairs are not there)~~

There are 2 ways

(1) 

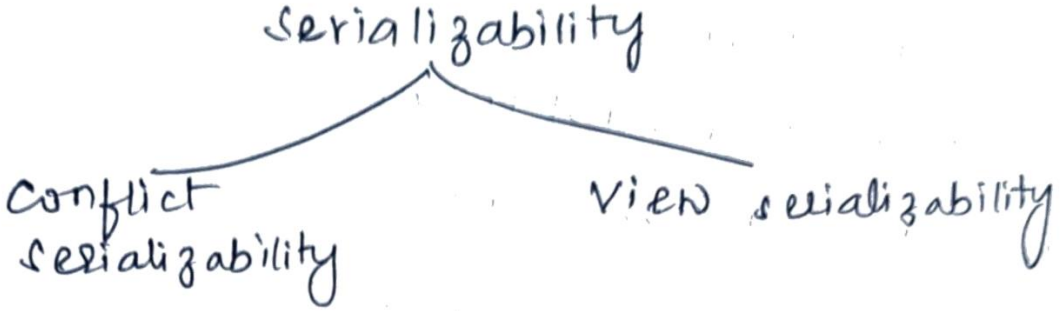
T_1 first then T_2

(OR)

(2) 

T_2 first then T_1

To check serializability, check whether there exist a serial schedule to a parallel schedule or not. (14)



Conflict serializability:

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)		

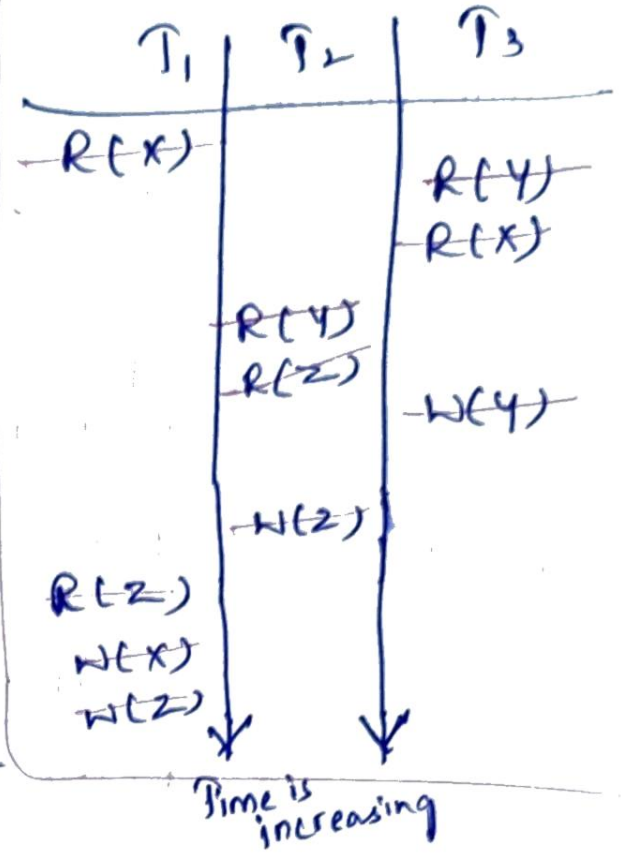
Step1: Draw precedence graph

Step2: Check conflict pairs in other transactions and draw an edge.

R-W
W-R
W-W

Conflict pairs on same data

Rough diagram

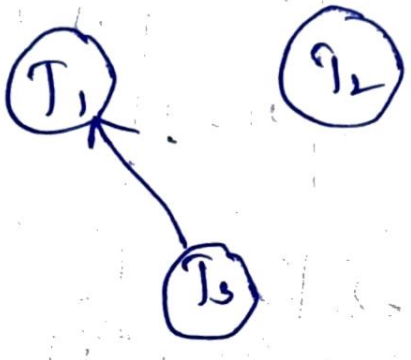


starting from T_1 ie, R(X) check its conflict pair W(X) is present in either T_2 or T_3 till end. If it is not present then cancel & move further

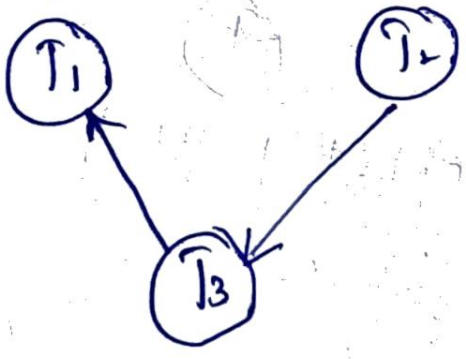
Step3: Now next T_3 R(Y) check its conflict pair W(Y) is present in T_1 or T_2

step4: Now R(X) check its conflict pair W(X) present in T_1 or T_2.

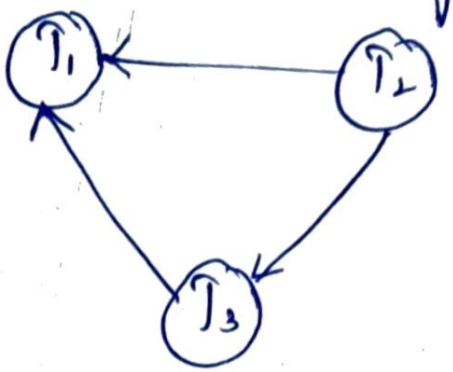
Yes W(x) is present in T_1. so, draw an edge from T_3 to T_1



steps: Now in T_2, R(Y) its conflict pair is W(Y) check whether it is present in T_1 or T_3. Yes, it is present so draw an edge from T_2 to T_3



Step 6: Now, R(z), its conflict pair is W(z), yes W(z) is present in T_1. so draw an edge from T_2 to T_1



Step 7: Now W(y), its conflict pairs are R(y) and W(y) Here W(y) have 2 conflict pairs. so check whether they are present in other transactions or not. Not there, no need to draw any edge

Step 8: Now W(z), its conflict pair is R(z) & W(z) yes it is present is in T_1 so draw one edge from T_2 to T_1. But it is already present. so leave it. No need to check the last left over R(z), W(x) & W(z) because they cant be compared to any other transactions. so just cancel them.

Step 9: So now, check whether there exists any loop or cycle formed in the precedence graph. If it is not forming any loop/cycle in the precedence graph then it is a conflict serializable schedule

↓

Serializable [equivalent schedule exist]

↓

Consistent

So there exists 6 possibilities here for the execution of transactions like: $T_1 \longrightarrow T_2 \longrightarrow T_3$

$T_1 \longrightarrow T_3 \longrightarrow T_2$

$T_2 \longrightarrow T_1 \longrightarrow T_3$

$T_2 \longrightarrow T_3 \longrightarrow T_1$

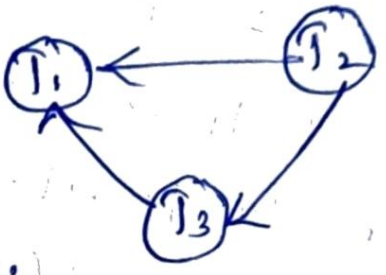
$T_3 \longrightarrow T_1 \longrightarrow T_2$

$T_3 \longrightarrow T_2 \longrightarrow T_1$

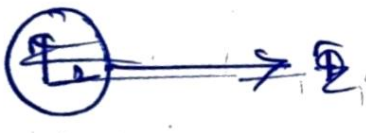
So among these, which schedule does this follow.? How can you say which one is first executed T_1 or T_2 or T_3 ?

For that check the graph, if the indegree of a vertex is '0' then it executes first.

In our example.



final precedence graph

Here T_2 indegree is '0' so T_2 executes first then T_3 then T_1. so the order of execution will be  $T_2 \longrightarrow T_3 \longrightarrow T_1$

Ex2

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

check whether it is conflict serializability or not

Precedence graph: Conflict pair R(A) & W(A) present in T_2 so draw an edge

then again in T_2 R(A) its conflict pair is W(A) it is in T_1 so draw an edge from T_2 to T_1

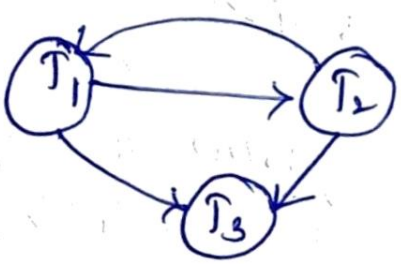
So here it is forming a loop, so we can say it is not conflict serializable

View Serializability :-

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

No loop → C.S → serial → consistent

Precedence graph :-



Here it is forming a loop so it is not a conflict serializable schedule.

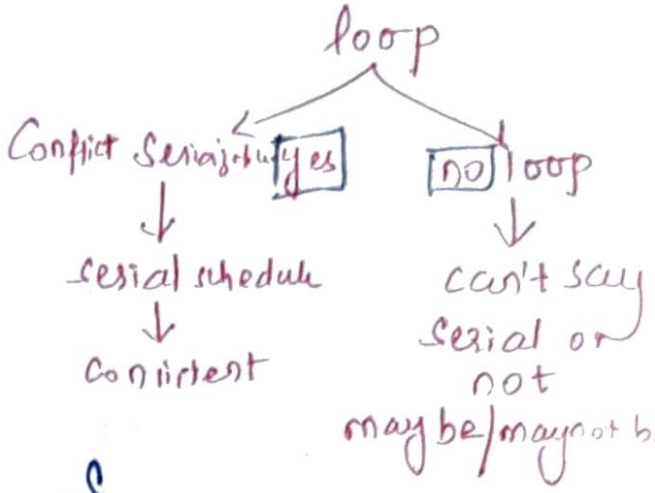
~~If it is not a conflict serializable but we can't say whether it is serializable or not.~~

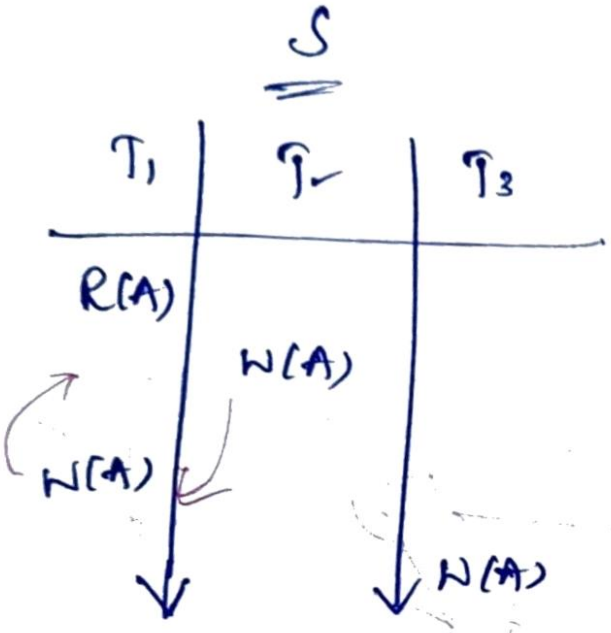
If a it is Conflict serializable then definitely it is a serial schedule and consistent also. But if it is not a conflict serializable schedule then we cannot say it is serial schedule or not.

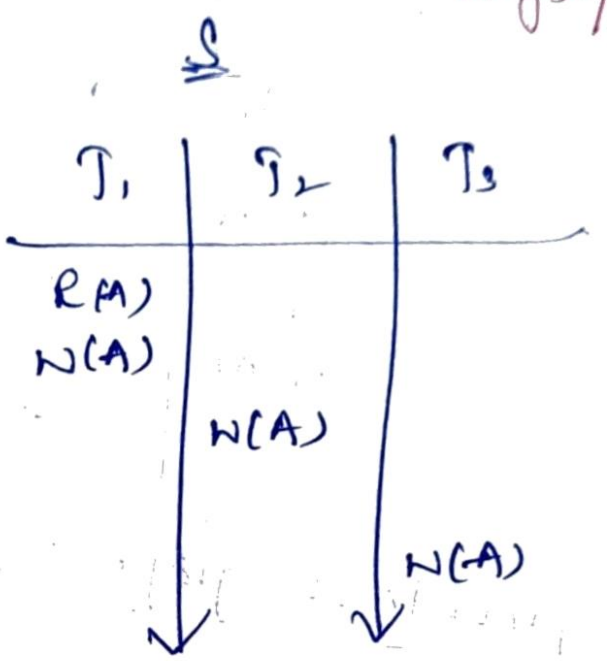
It may be serial or non-serial. Here vice versa case is not possible.

Here that's why we use view serializability.

Sometimes it doesn't seem to be a serial schedule but after some modifications in operations, it becomes serial.

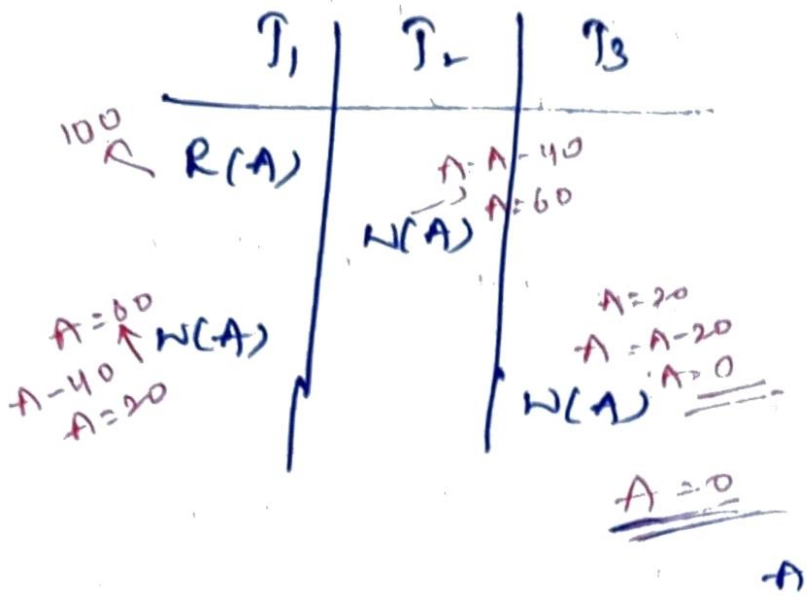


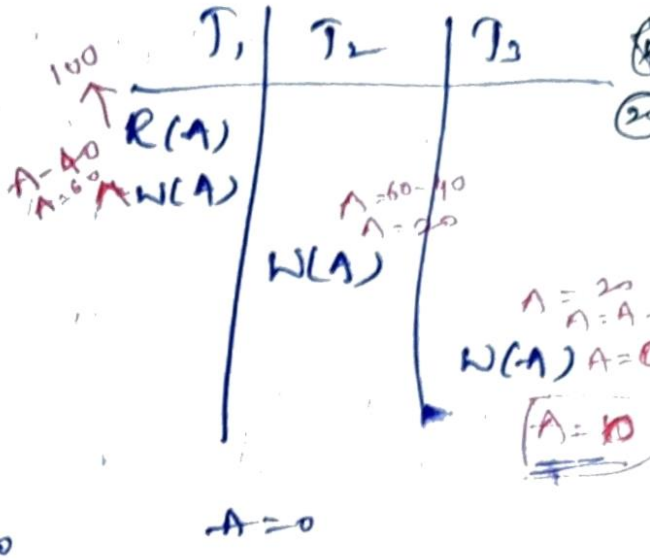




Here in second schedule, the operations W(A) are slightly changed in T_1 & T_2. Now, the second graph is a serial schedule $T_1 \rightarrow T_2 \rightarrow T_3$. Now lets see whether they are conflict serializable or not

Initially let A=100 then we perform same operations on both sides.





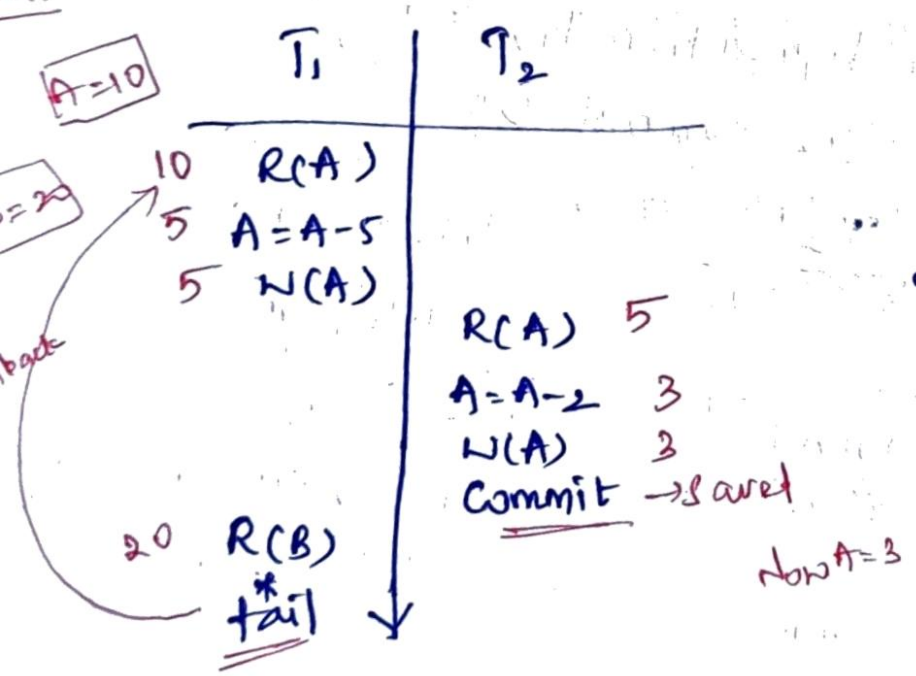
They doesn't seem to be equivalent but they are equivalent. That is nothing but view serializability. ~~They~~ do They are not conflict equivalent but they are view equivalent.

Recoverability:

Irrecoverable Vs Recoverable schedule

Irrecoverable schedule: The schedule which cannot be recovered are known as irrecoverable schedules.

Ex:



At some point of time due to some hardware failure, s/w failure or any reason, if the transaction T_1 gets failed, due to atomicity property it gets rollback so again the value of A=10 because T_1 is failed

so all the changes that it had done should be undo so again the latest value of A will be 10. But if we see carefully, T_2 also had done some operations, so all those operations of T_2 are lost which cannot be recovered

T_2 cannot be rollback bcz it has committed

∧ So these types of schedules are irrecoverable schedules

Recoverable Schedules:-

A schedule is said to be recoverable if a transaction which reads a data item written by another transaction commits only after the written transaction commits

~~Ex: Consider an example of a small organization, which pays Rs. 100/- for each overtime employee to the employees. If an employee earns more than~~

(OR)

Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules.

ex: If some transaction T_2 is reading value updated or written by some other transaction T_1, then the commit of T_2 must occur after the commit of T_1

ex:-

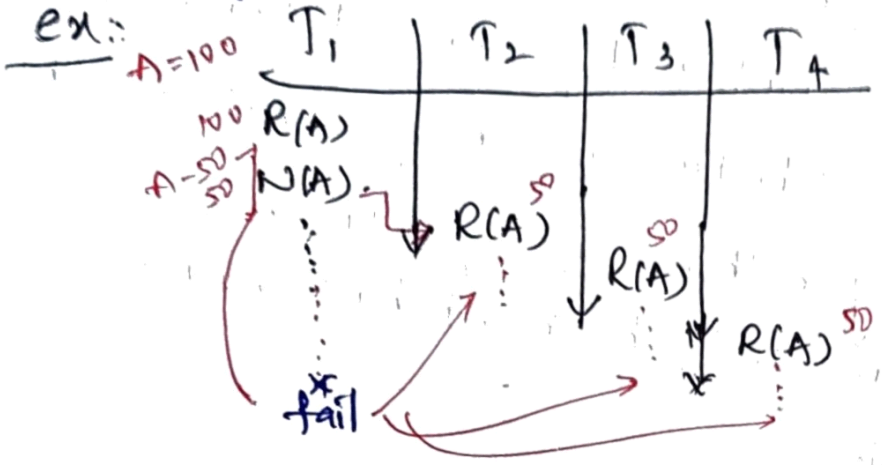
T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
Commit	
	commit

This is a recoverable schedule since T_1 commits before T_2, that makes the value read by T_2 correct.

Cascading Vs Cascadeless Schedule:-

Cascading: Basically cascading refers to that due to the occurrence of 1 event, multiple events are automatically occurring

ex:



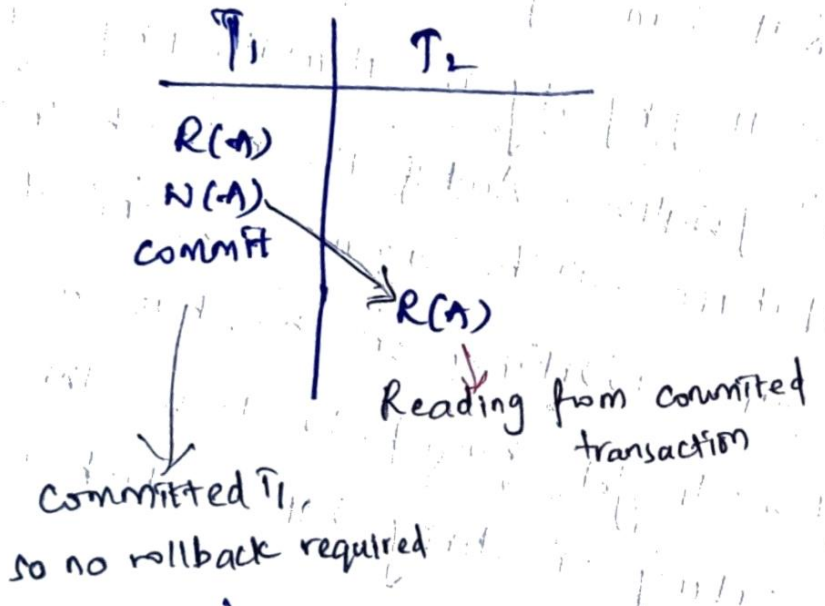
The changes done by T_1 ie, value 50 is read by T_2, T_3, T_4 Now all these transactions are going on and at some point of time T_1 gets failed so according to ACID property ie, atomicity property it should get rollback to 1st operation. And if we rollback that the value in the database will be again 100 (initial value). So T_2, T_3, T_4 are working on 50. but 50 value doesn't exist after T_1 gets failed, so these three transactions T_2, T_3, T_4 are working on dirty data which is called as Write-read problem.

So to remove this problem, automatically we have to abort all other transactions T_2, T_3 and T_4. and rollback which is called as cascading. T_1 got failed due to some outrage but all other transactions have to be forcefully aborted. So finally all four transactions gets failed.

Disadvantage: Performance and CPU utilization is degraded. CPU cycles gets wasted.

Cascadeless schedule:- In a cascadeless schedule, the written transaction T_1 commits before a read transaction T_2. This transac reads the values that are written by committed transaction T_1. Cascadeless schedules are schedules in which cascading rollback doesn't occur. Cascading rollback is a situation in which series of transaction needs to be rolled back even if a single transaction fails.

ex:



Note that in the above ~~transaction~~ example there is no dirty read problem, transaction T_2 is reading from another committed transaction T_1, so no rollback is required.