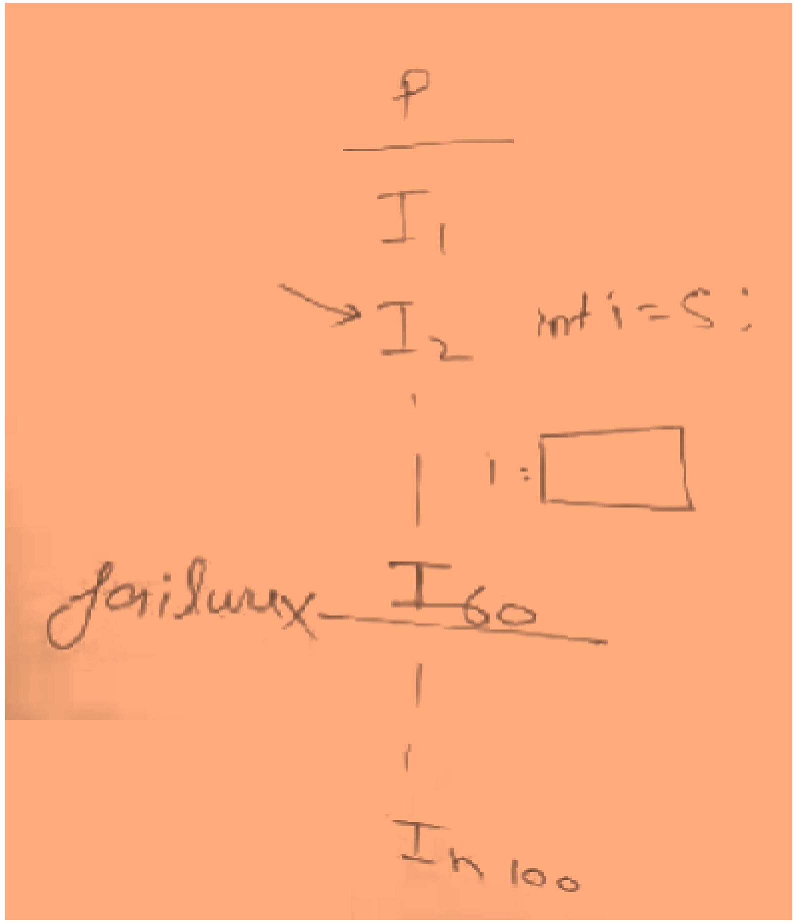
TRANSACTION



P

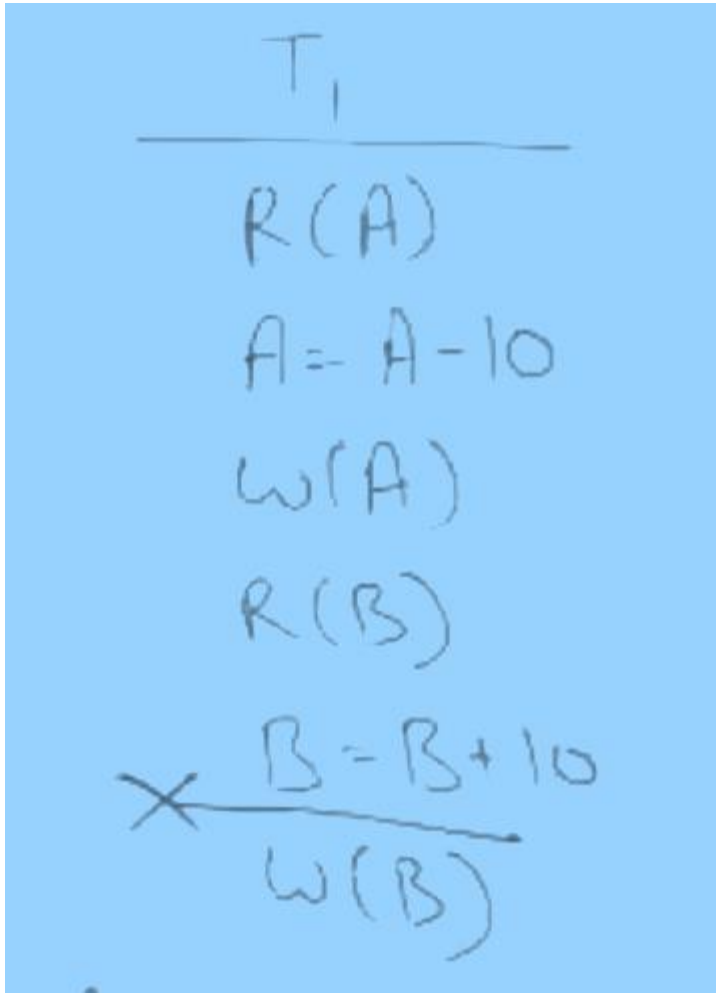
I_1

→ I_2 int i=5;

i = □

failure X I_{60}

$I_{n\,100}$



T_1
R(A)
A=A-10
W(A)
R(B)
B=B+10
X
W(B)

Transaction

"Transaction is a set of operations which are all logically related."

OR

"Transaction is a single logical unit of work formed by a set of operations."

Operations in Transaction-

The main operations in a transaction are-

- Read Operation
- Write Operation

1. Read Operation-

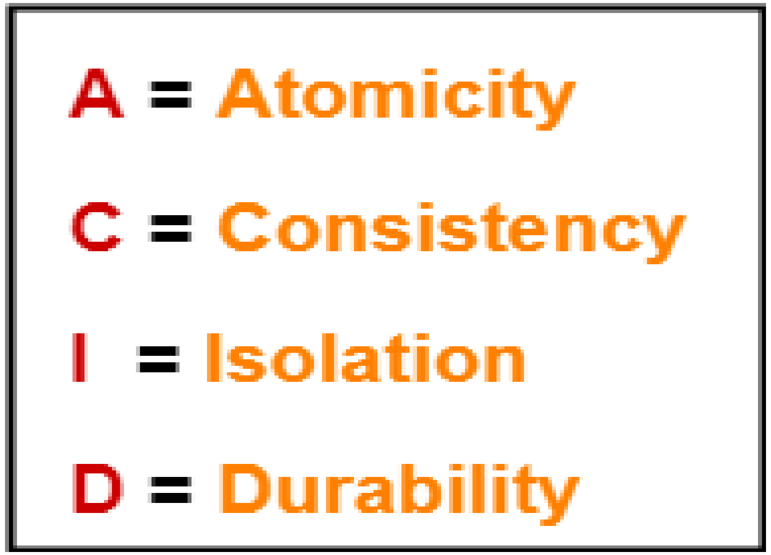
- Read operation reads the data from the database and then stores it in the buffer in main memory.
- For example- **Read(A)** instruction will read the value of A from the database and will store it in the buffer in main memory.

2. Write Operation-

- Write operation writes the updated data value back to the database from the buffer.
- For example- Write(A) will write the updated value of A from the buffer to the database.

ACID Properties-

- It is important to ensure that the database remains consistent before and after the transaction.
- To ensure the consistency of database, certain properties are followed by all the transactions occurring in the system.
- These properties are called as **ACID Properties** of a transaction.



A = Atomicity

C = Consistency

I = Isolation

D = Durability

1. Atomicity-

- This property ensures that either the transaction occurs completely or it does not occur at all.
- In other words, it ensures that no transaction occurs partially.
- That is why, it is also referred to as “**All or nothing rule**“.
- It is the responsibility of Transaction Control Manager to ensure atomicity of the transactions.

2. Consistency-

- This property ensures that integrity constraints are maintained.
- In other words, it ensures that the database remains consistent before and after the transaction.
- It is the responsibility of DBMS and application programmer to ensure consistency of the database.

3. Isolation-

- This property ensures that multiple transactions can occur simultaneously without causing any inconsistency.
- During execution, each transaction feels as if it is getting executed alone in the system.
- A transaction does not realize that there are other transactions as well getting executed parallely.
- Changes made by a transaction becomes visible to other transactions only after they are written in the memory.
- The resultant state of the system after executing all the transactions is same as the state that would be achieved if the transactions were executed serially one after the other.
- It is the responsibility of concurrency control manager to ensure isolation for all the transactions.

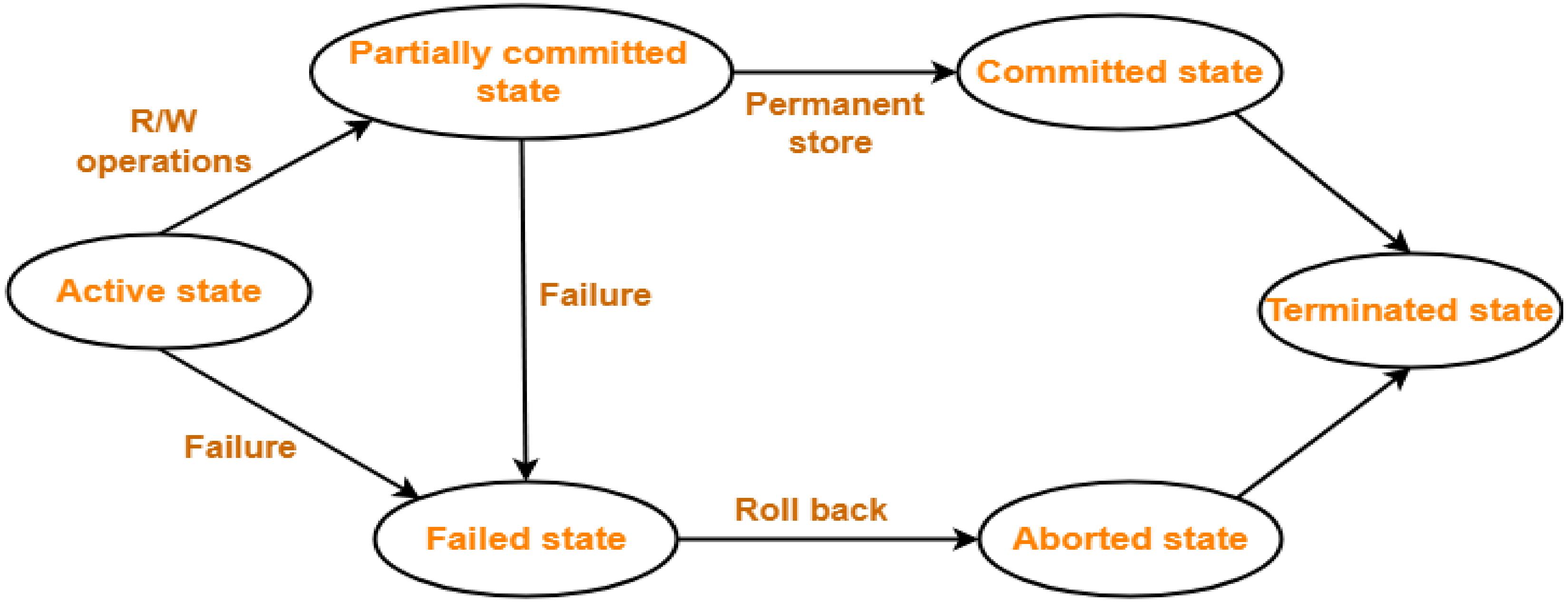
4. Durability-

- This property ensures that all the changes made by a transaction after its successful execution are written successfully to the disk.
- It also ensures that these changes exist permanently and are never lost even if there occurs a failure of any kind.
- It is the responsibility of recovery manager to ensure durability in the database.

Transaction States-

A transaction goes through many different states throughout its life cycle. These states are called as **transaction states**. Transaction states are as follows-

- **Active state**
- **Partially committed state**
- **Committed state**
- **Failed state**
- **Aborted state**
- **Terminated state**



Transaction States in DBMS

1. Active State-

- This is the first state in the life cycle of a transaction.
- A transaction is called in an **active state** as long as its instructions are getting executed.
- All the changes made by the transaction now are stored in the buffer in main memory.

2. Partially Committed State-

- After the last instruction of transaction has executed, it enters into a partially committed state.
- After entering this state, the transaction is considered to be partially committed.
- It is not considered fully committed because all the changes made by the transaction are still stored in the buffer in main memory.

3. Committed State-

- After all the changes made by the transaction have been successfully stored into the database, it enters into a **committed state**.
- Now, the transaction is considered to be fully committed.

NOTE-

- After a transaction has entered the committed state, it is not possible to roll back the transaction.
- In other words, it is not possible to undo the changes that has been made by the transaction.
- This is because the system is updated into a new consistent state.
- The only way to undo the changes is by carrying out another transaction called as **compensating transaction** that performs the reverse operations.

4. Failed State-

- When a transaction is getting executed in the active state or partially committed state and some failure occurs due to which it becomes impossible to continue the execution, it enters into a **failed state**.

5. Aborted State-

- After the transaction has failed and entered into a failed state, all the changes made by it have to be undone.
- To undo the changes made by the transaction, it becomes necessary to roll back the transaction.
- After the transaction has rolled back completely, it enters into an aborted state.

6. Terminated State-

- This is the last state in the life cycle of a transaction.
- After entering the committed state or aborted state, the transaction finally enters into a terminated state where its life cycle finally comes to an end.

Concurrency

Concurrency is the ability of a database to allow multiple users to perform multiple transactions.

Advantages of Concurrency

- **Reduce Waiting Time**
- **Reduce Respose Time**
- **Maximize Resourse utilization**
- **Increse Efficency**

Concurrency Problems in DBMS-

- When multiple transactions execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems.
- Such problems are called as **concurrency problems**.

The concurrency problems are-

- **Dirty Read Problem**
- **Unrepeatable Read Problem**
- **Lost Update Problem**
- **Phantom Read Problem**

1. Dirty Read Problem-

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

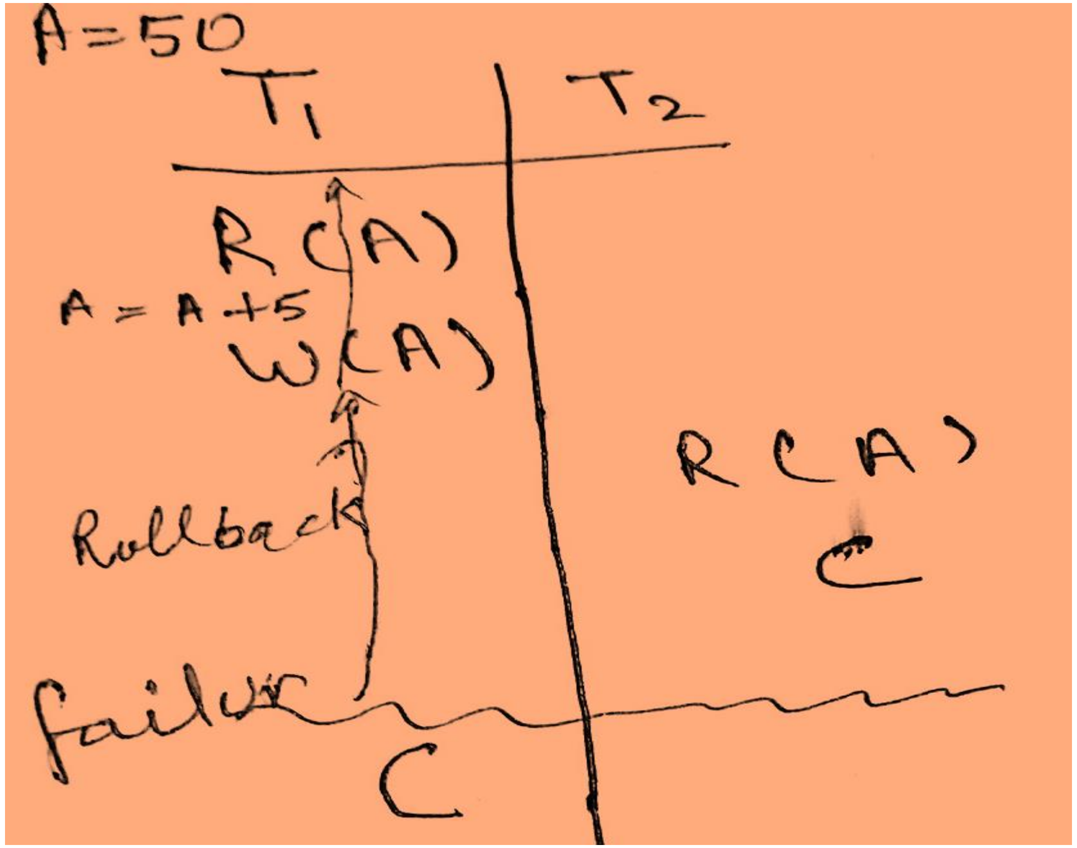
This read is called as dirty read because-

- There is always a chance that the uncommitted transaction might roll back later.
- Thus, uncommitted transaction might make other transactions read a value that does not even exist.
- This leads to inconsistency of the database.

NOTE-

- **Dirty read does not lead to inconsistency always.**
- **It becomes problematic only when the uncommitted transaction fails and roll backs later due to some reason.**

Transaction T1	Transaction T2
R (A)	
W (A)	
	R (A) // Dirty Read
	W (A)
	Commit
Failure	



Here,

T1 reads the value of A.

T1 updates the value of A in the buffer.

T2 reads the value of A from the buffer.

T2 writes the updated the value of A.

T2 commits.

T1 fails in later stages and rolls back.

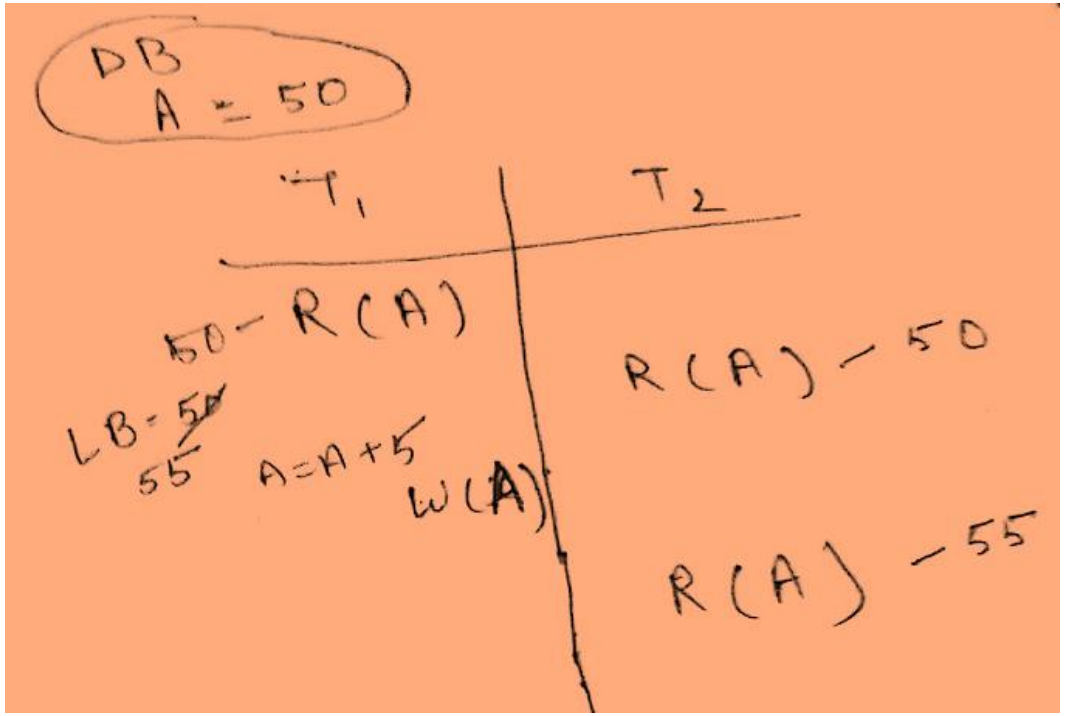
In this example,

- T2 reads the dirty value of A written by the uncommitted transaction T1.
- T1 fails in later stages and roll backs.
- Thus, the value that T2 read now stands to be incorrect.
- Therefore, database becomes inconsistent.

2. Unrepeatable Read Problem-

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

Transaction T1	Transaction T2
R (X)	
	R (X)
W (X)	
	R (X) // Unrepeated Read



Here,

- T1 reads the value of X (= 10 say).
- T2 reads the value of X (= 10).
- T1 updates the value of X (from 10 to 15 say) in the buffer.
- T2 again reads the value of X (but = 15).

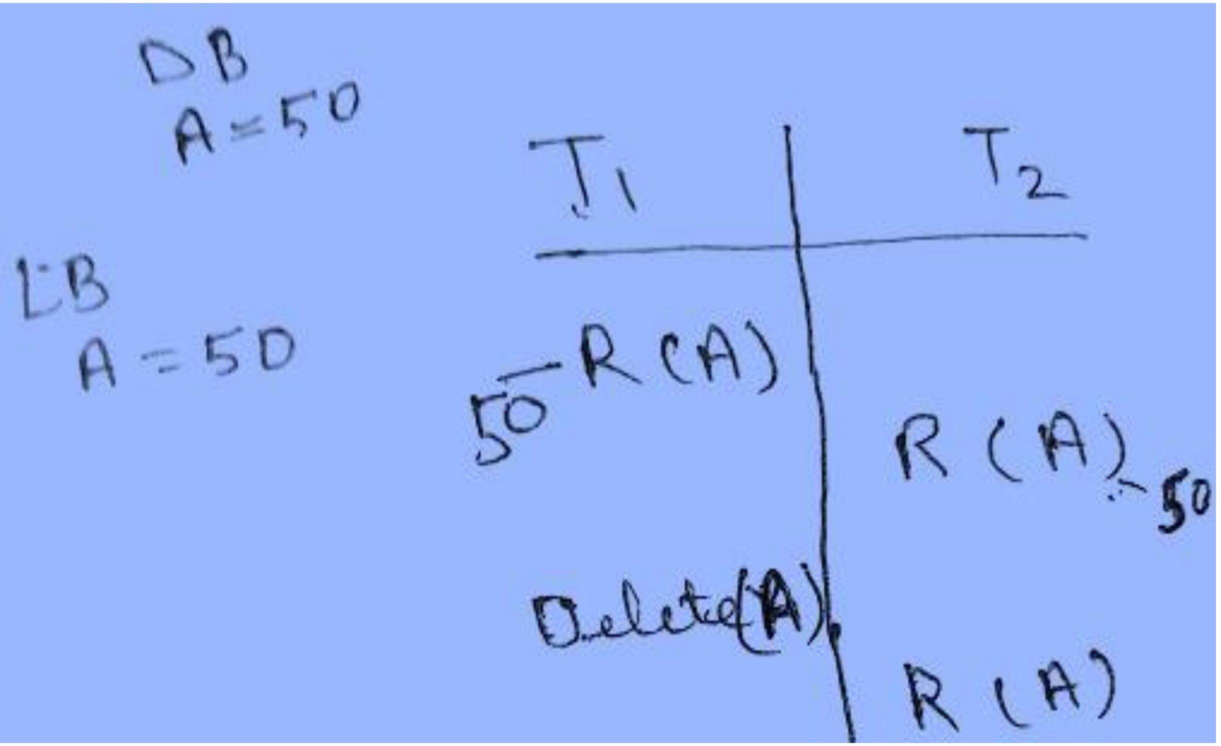
In this example,

- T2 gets to read a different value of X in its second reading.
- T2 wonders how the value of X got changed because according to it, it is running in isolation.

3. Phantom Read Problem-

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

Transaction T1	Transaction T2
R (X)	
	R (X)
Delete (X)	
	Read (X)



DB
A=50

LB
A=50

T1	T2
50 R(A)	
	R(A) 50
Delete(A)	
	R(A)

Here,

T1 reads X.

T2 reads X.

T1 deletes X.

T2 tries reading X but does not find it.

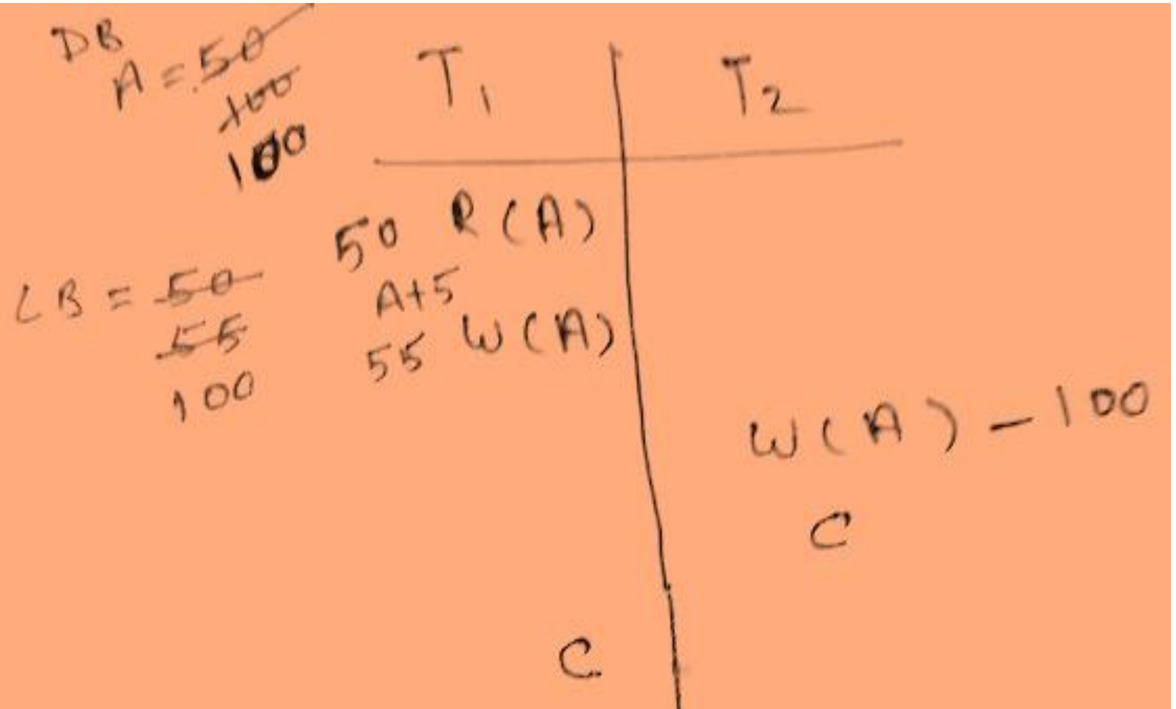
In this example,

- T2 finds that there does not exist any variable X when it tries reading X again.
- T2 wonders who deleted the variable X because according to it, it is running in isolation.

4. Lost Update Problem (write - write conflict)-

This problem occurs when multiple transactions execute concurrently and updates from one or more transactions get lost.

Transaction T1	Transaction T2
R (A)	
W (A)	
⋮	
	W (A)
	Commit
Commit	



DB
A = ~~50~~
~~100~~
100

LB = ~~50~~
~~55~~
100

	T1	T2
50	R(A)	
	A+5	
55	W(A)	
		W(A) – 100
		C
	C	

Here,

T1 reads the value of A (= 10 say).

T2 updates the value to A (= 15 say) in the buffer.

T2 does blind write A = 25 (write without read) in the buffer.

T2 commits.

When T1 commits, it writes A = 25 in the database.

In this example,

- T1 writes the over written value of X in the database.
- Thus, update from T1 gets lost.

NOTE-

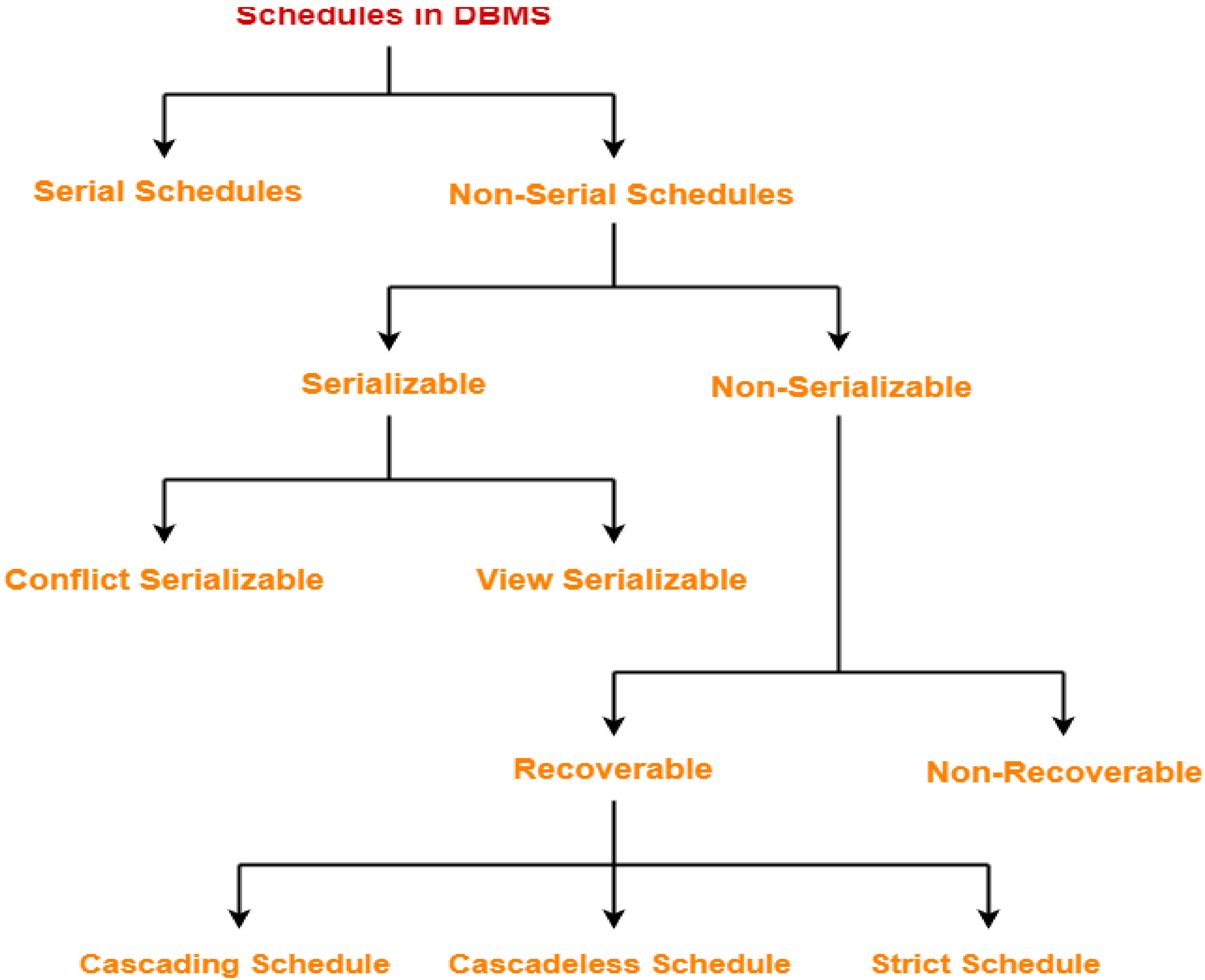
- **This problem occurs whenever there is a write-write conflict.**
- **In write-write conflict, there are two writes one by each transaction on the same data item without any read in the middle.**

Avoiding Concurrency Problems-

- To ensure consistency of the database, it is very important to prevent the occurrence of above problems.
- Concurrency Control Protocols help to prevent the occurrence of above problems and maintain the consistency of the database.

Schedule

The order in which the operations of multiple transactions appear for execution is called as a schedule.



Schedules in DBMS

Serial Schedules

Non-Serial Schedules

Serializable

Non-Serializable

Conflict Serializable

View Serializable

Recoverable

Non-Recoverable

Cascading Schedule

Cascadeless Schedule

Strict Schedule

Serial Schedules-

In serial schedules,

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

Transaction T1	Transaction T2
R (A)	
W (A)	
R (B)	
W (B)	
Commit	
	R (A)
	W (B)
	Commit

Characteristics-

Serial schedules are always-

- Consistent
- Recoverable
- Cascadeless
- Strict

Transaction T1	Transaction T2
	R (A)
	W (B)
	Commit
R (A)	
W (A)	
R (B)	
W (B)	
Commit	

Non-Serial Schedules-

In non-serial schedules,

- Multiple transactions execute concurrently.
- Operations of all the transactions are inter leaved or mixed with each other.

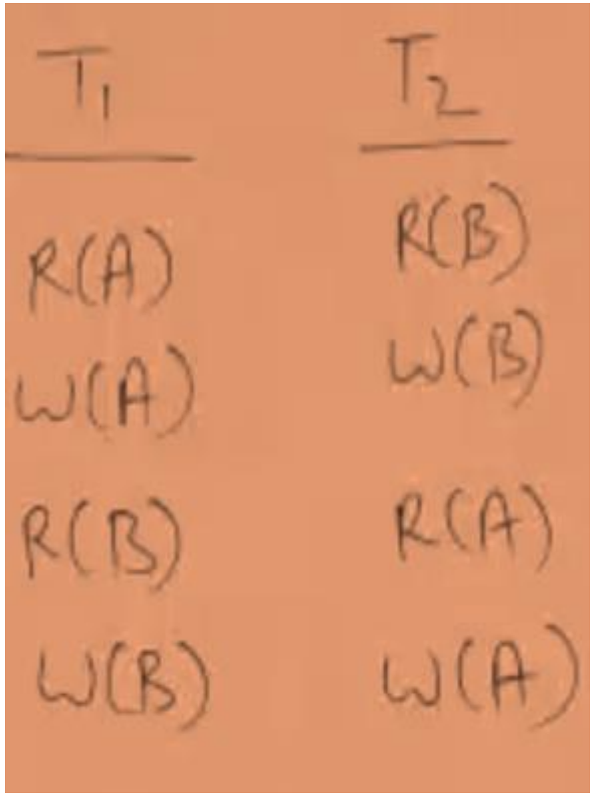
Transaction T1	Transaction T2
R (A)	
W (B)	
	R (A)
R (B)	
W (B)	
Commit	
	R (B)
	Commit

Characteristics-

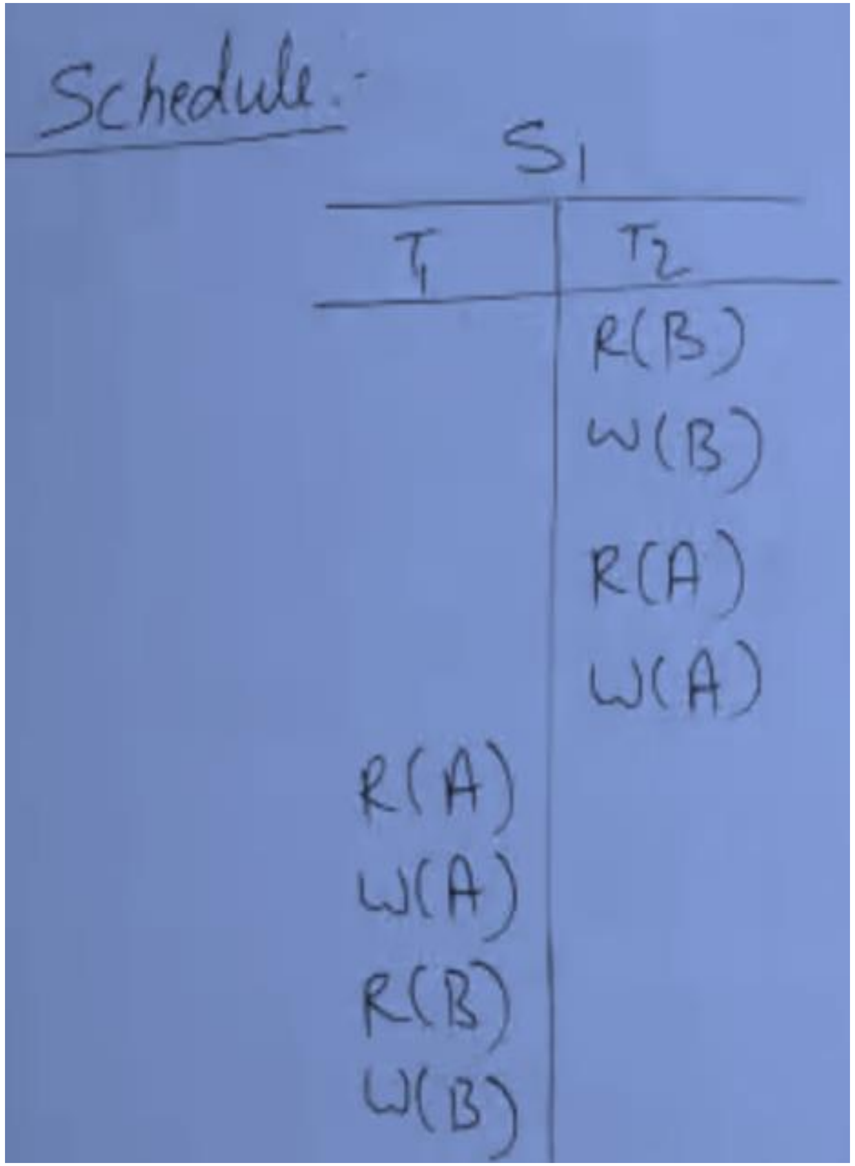
Non-serial schedules are **NOT** always-

- Consistent
- Recoverable
- Cascadeless
- Strict

Transaction T1	Transaction T2
	R (A)
R (A)	
W (B)	
	R (B)
	Commit
R (B)	
W (B)	
Commit	



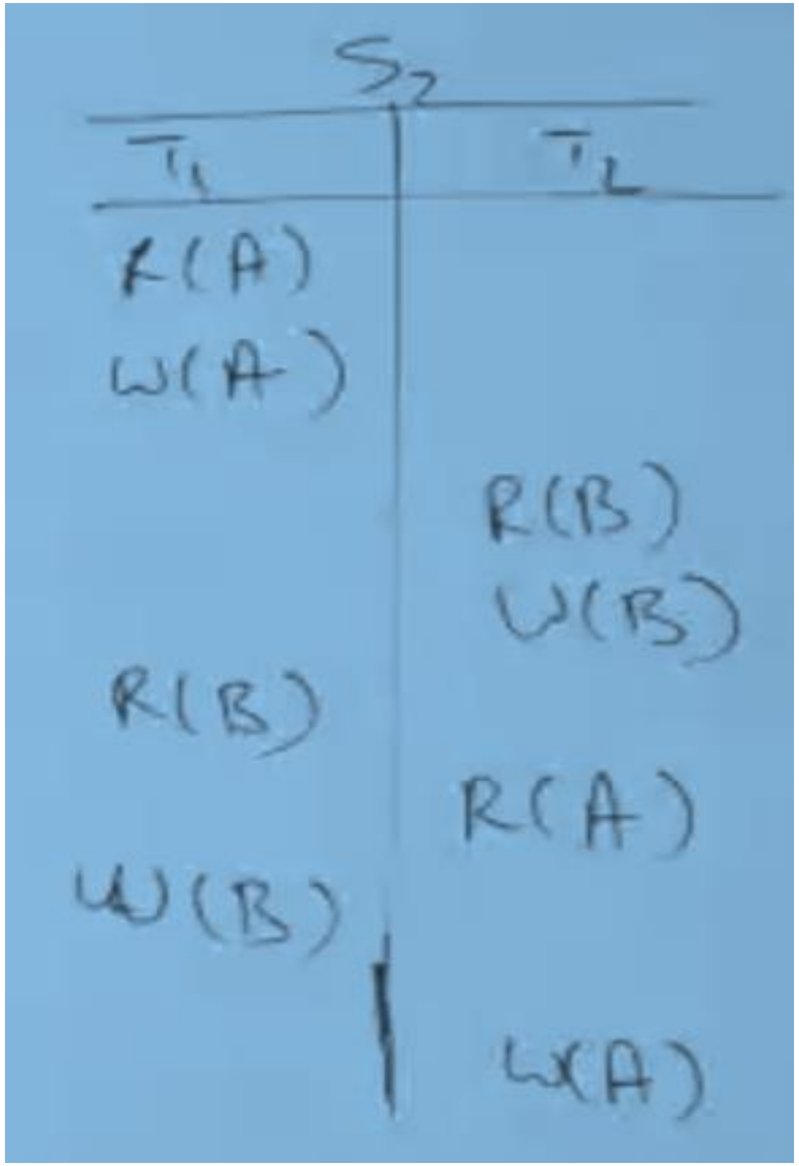
T1	T2
R(A)	R(B)
W(A)	W(B)
R(B)	R(A)
W(B)	W(A)



Schedule:-

S1

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



S2

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

Finding Number Of Schedules-

Consider there are n number of transactions T1, T2, T3 …. , Tn with N1, N2, N3 …. , Nn number of operations respectively.

Total Number of Schedules-

Total number of possible schedules (serial + non-serial) is given by-

$$\boxed{\frac{N1 + N2 + N3 + \dots + Nn}{N1! \times N2! \times N3! \times \dots \times Nn!}}$$

Total Number of Serial Schedules-

Total number of serial schedules
= Number of different ways of arranging n transactions
= n!

Total Number of Non-Serial Schedules-

Total number of non-serial schedules
= Total number of schedules – Total number of serial schedules

Q. Consider there are three transactions with 2, 3, 4 operations respectively, find-

- How many total number of schedules are possible?
- How many total number of serial schedules are possible?
- How many total number of non-serial schedules are possible?

Total Number of Schedules-

Using the above formula, we have-

$$\text{Total number of schedules} = \frac{(2+3+4)!}{2! \times 3! \times 4!}$$

$$= 1260$$

Total Number of Serial Schedules-

Total number of serial schedules
= Number of different ways of arranging 3 transactions
= 3!
= 6

Total Number of Non-Serial Schedules-

Total number of non-serial schedules
= Total number of schedules – Total number of serial schedules
= 1260 – 6
= 1254

Serializability in DBMS-

- Some non-serial schedules may lead to inconsistency of the database.
- Serializability is a concept that helps to identify which non-serial schedules are correct and will maintain the consistency of the database.

Serializable Schedules-

If a given non-serial schedule of 'n' transactions is equivalent to some serial schedule of 'n' transactions, then it is called as a **serializable schedule**.

Characteristics-

Serializable schedules behave exactly same as serial schedules.

Thus, serializable schedules are always-

- Consistent
- Recoverable
- Casacadeless
- Strict

Types of Serializability-

Serializability is mainly of two types-

- Conflict Serializability
- View Serializability

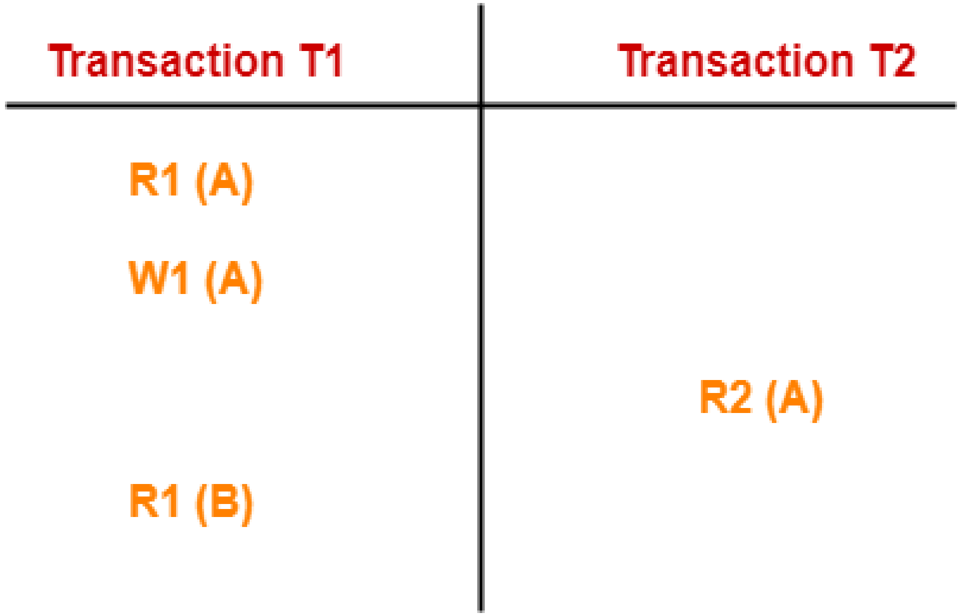
Conflict Serializability-

If a given non-serial schedule can be converted into a serial schedule by swapping its non-conflicting operations, then it is called as a **conflict serializable schedule**.

Conflicting Operations-

Two operations are called as **conflicting operations** if all the following conditions hold true for them-

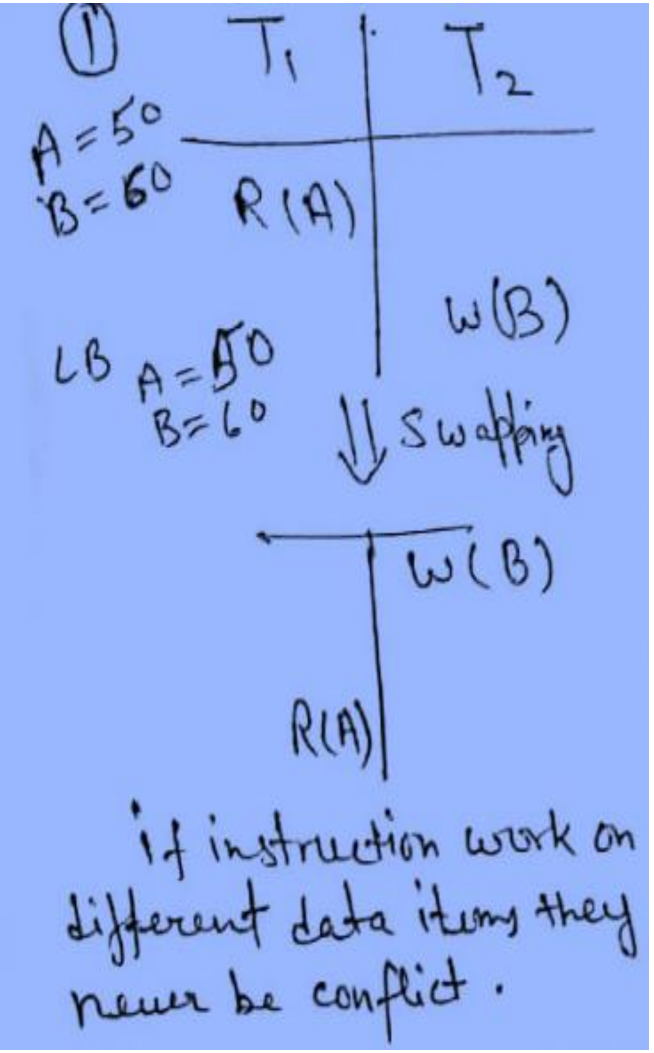
- Both the operations belong to different transactions
- Both the operations are on the same data item
- At least one of the two operations is a write operation



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

In this schedule,

- W1 (A) and R2 (A) are called as conflicting operations.
- This is because all the above conditions hold true for them.



①

A=50
B=60

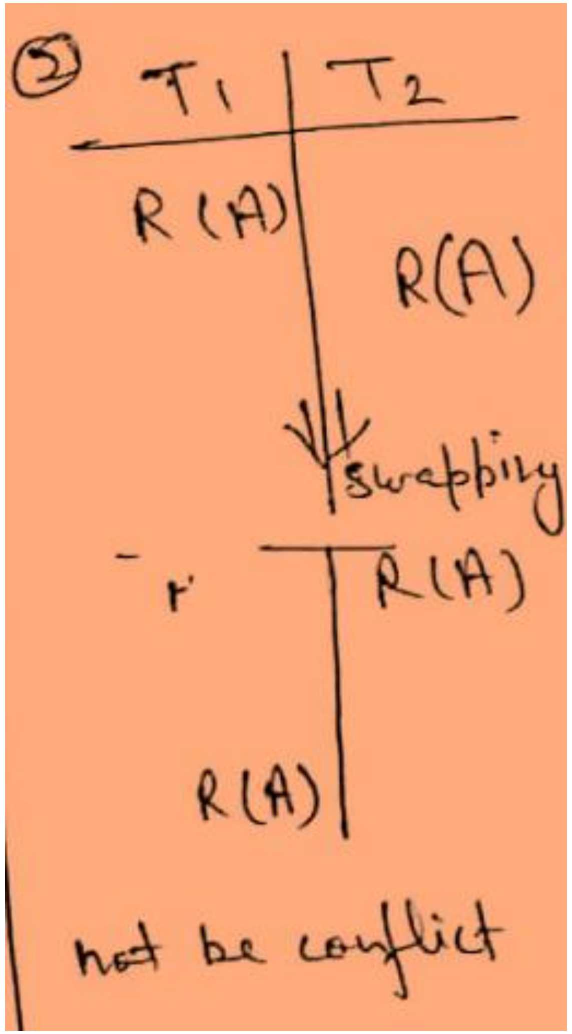
T1	T2
R(A)	
	W(B)

LB A=50
B=60

⇓ Swapping

	W(B)
R(A)	

if instruction work on different data items they never be conflict.



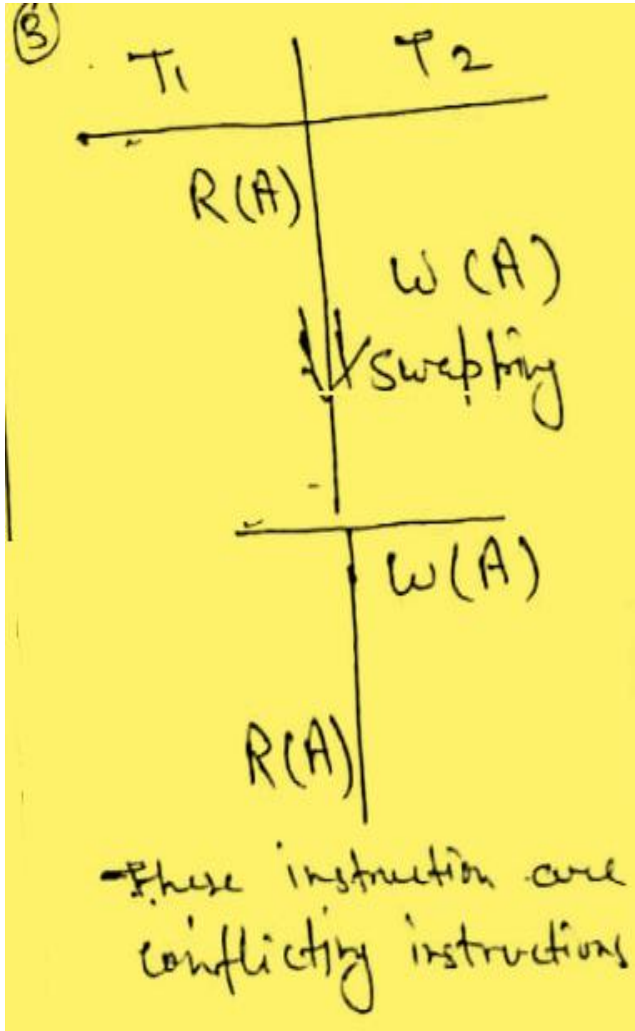
②

T1	T2
R(A)	
	R(A)

⇓ swapping

	R(A)
R(A)	

not be conflict



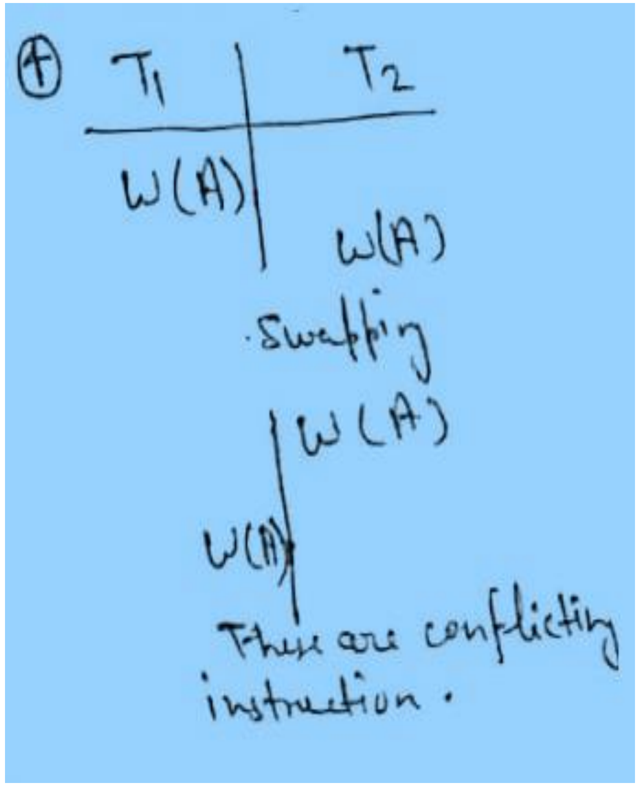
③

T1	T2
R(A)	
	W(A)

⇓ swapping

	W(A)
R(A)	

These instruction are conflicting instructions



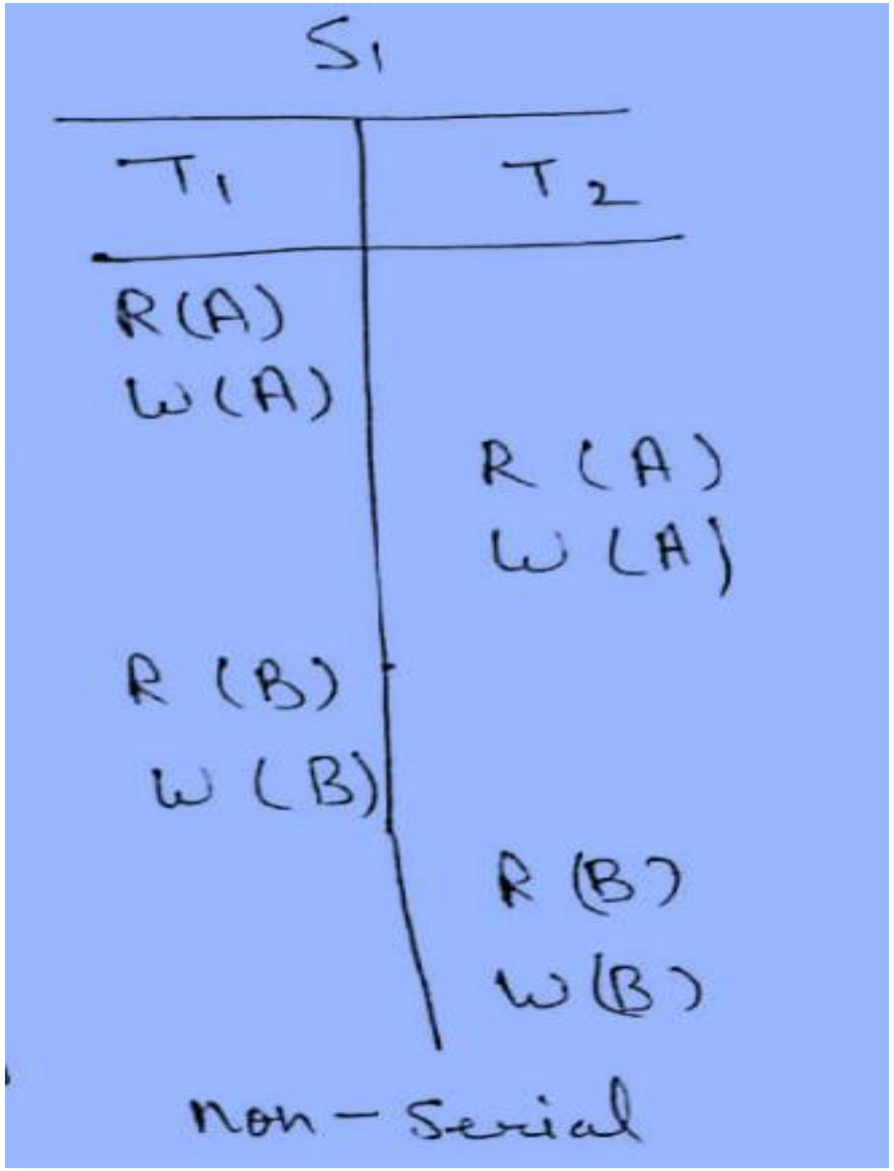
④

T1	T2
W(A)	
	W(A)

Swapping

	W(A)
W(A)	

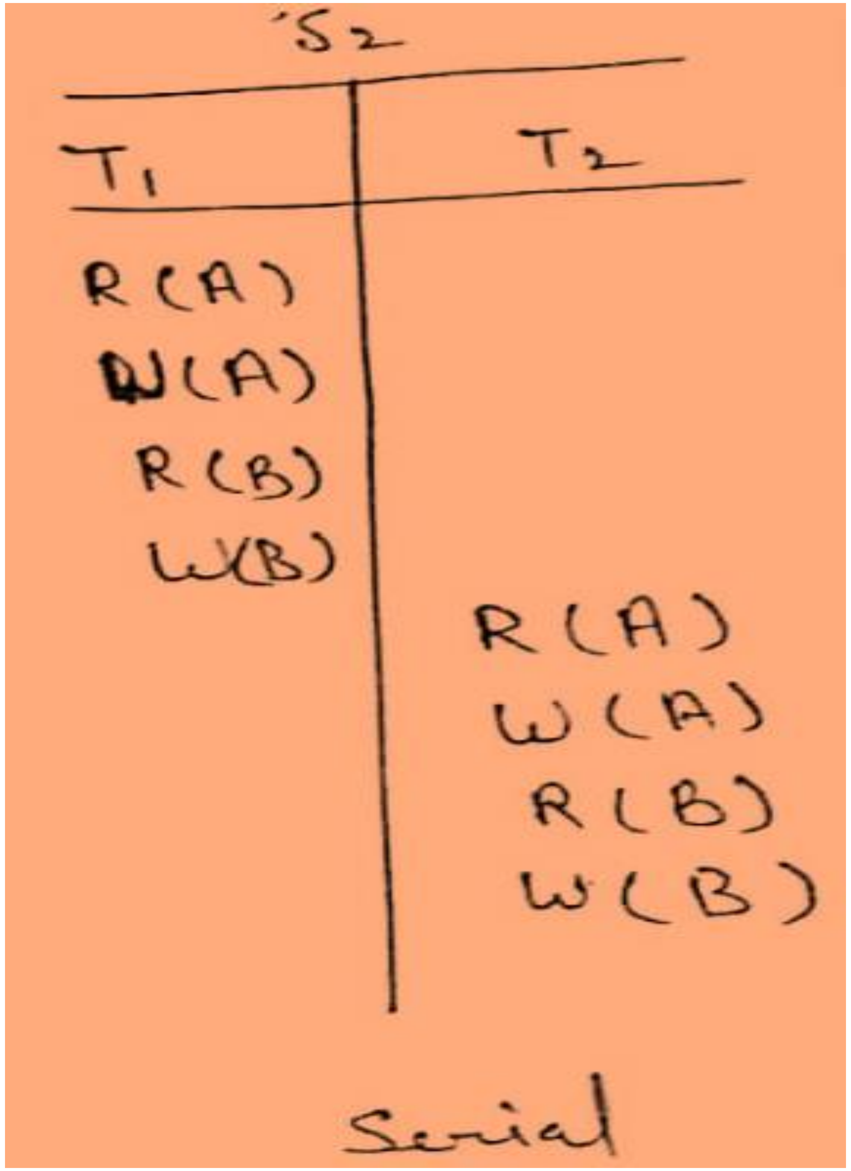
These are conflicting instruction.



S_1

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

non-Serial



S_2

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

Serial

Checking Whether a Schedule is Conflict Serializable Or Not-

Step-01:

Find and list all the conflicting operations.

Step-02:

Start creating a precedence graph by drawing one node for each transaction.

Step-03:

- Draw an edge for each conflict pair such that if X_i (V) and Y_j (V) forms a conflict pair then draw an edge from T_i to T_j.
- This ensures that T_i gets executed before T_j.

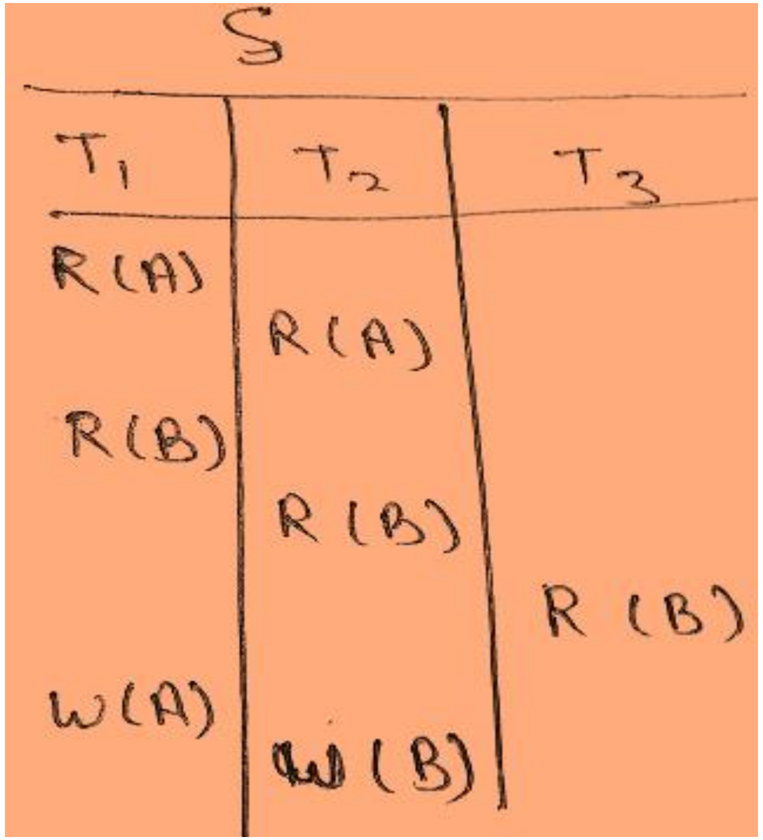
Step-04:

- Check if there is any cycle formed in the graph.
- If there is no cycle found, then the schedule is conflict serializable otherwise not.

NOTE-

- By performing the Topological Sort of the Directed Acyclic Graph so obtained, the corresponding serial schedule(s) can be found.
- Such schedules can be more than 1.

PRACTICE PROBLEMS BASED ON CONFLICT SERIALIZABILITY-



S

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

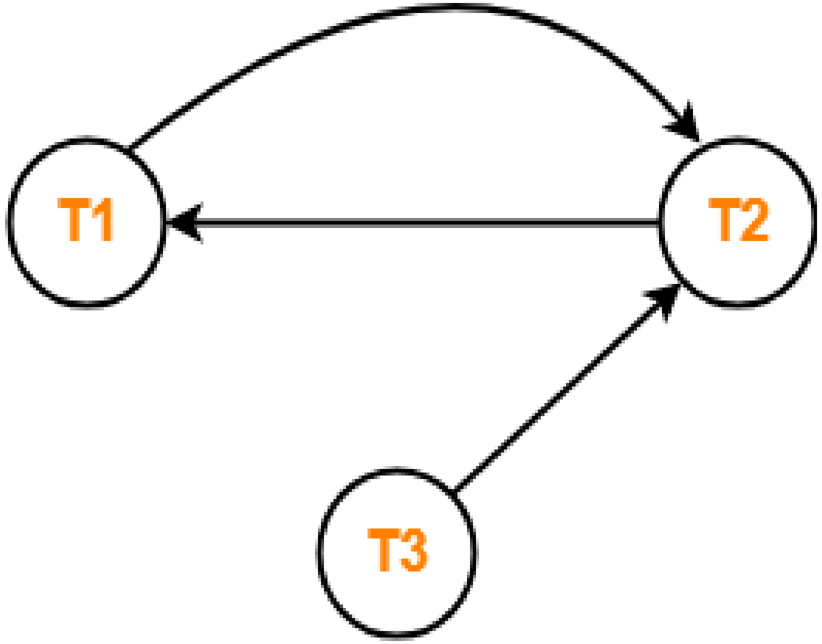
Step-01:

List all the conflicting operations and determine the dependency between the transactions-

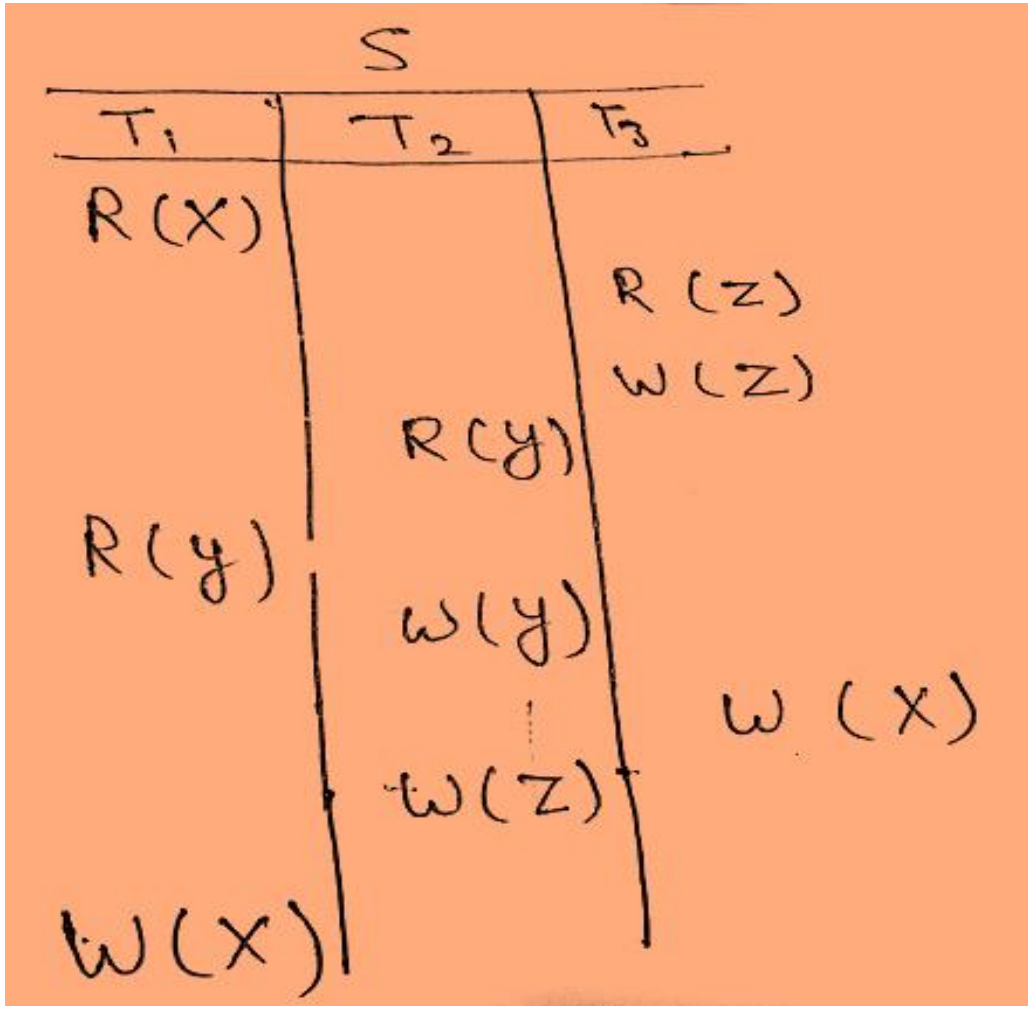
- $R_2(A)$, $W_1(A)$ ($T_2 \rightarrow T_1$)
- $R_1(B)$, $W_2(B)$ ($T_1 \rightarrow T_2$)
- $R_3(B)$, $W_2(B)$ ($T_3 \rightarrow T_2$)

Step-02:

Draw the precedence graph-

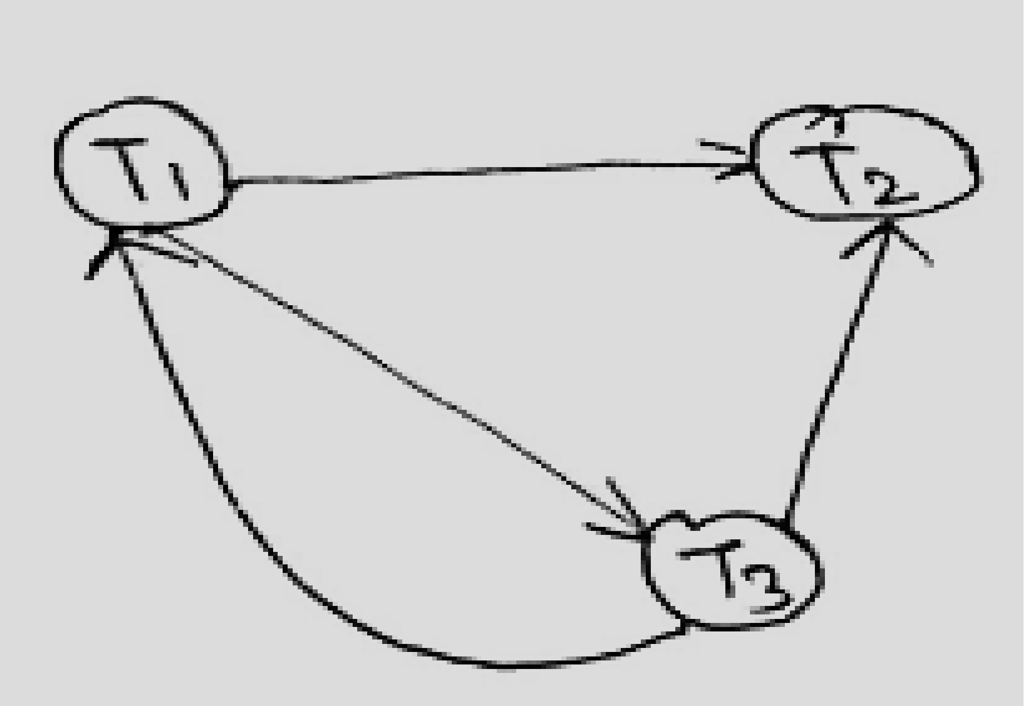


- Clearly, there exists a cycle in the precedence graph.
- Therefore, the given schedule S is not conflict serializable.

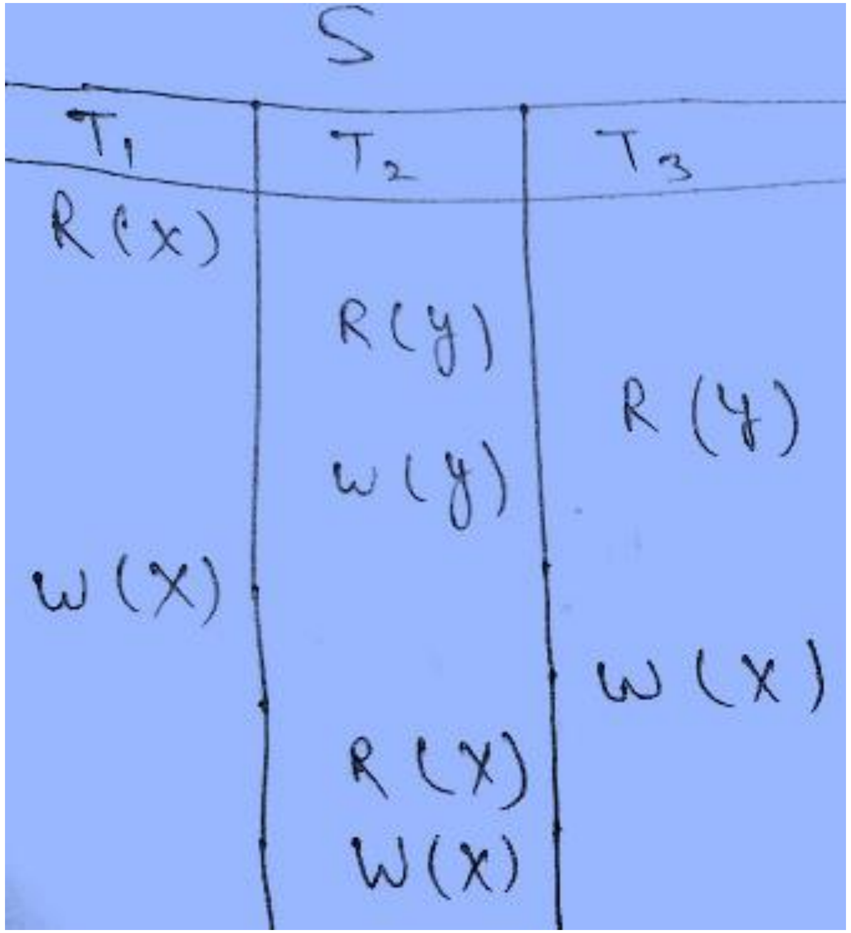


S

T1	T2	T3
R(X)		
		R(Z)
		W(Z)
	R(y)	
R(y)		
	w(y)	
		w(X)
	w(Z)	
W(X)		



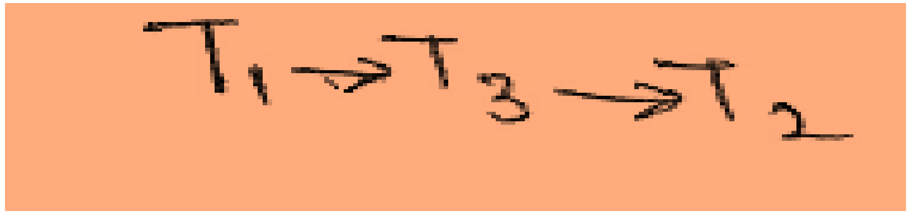
- Clearly, there exists a cycle in the precedence graph.
- Therefore, the given schedule S is not conflict serializable.
- Thus, Number of possible serialized schedules = 0.



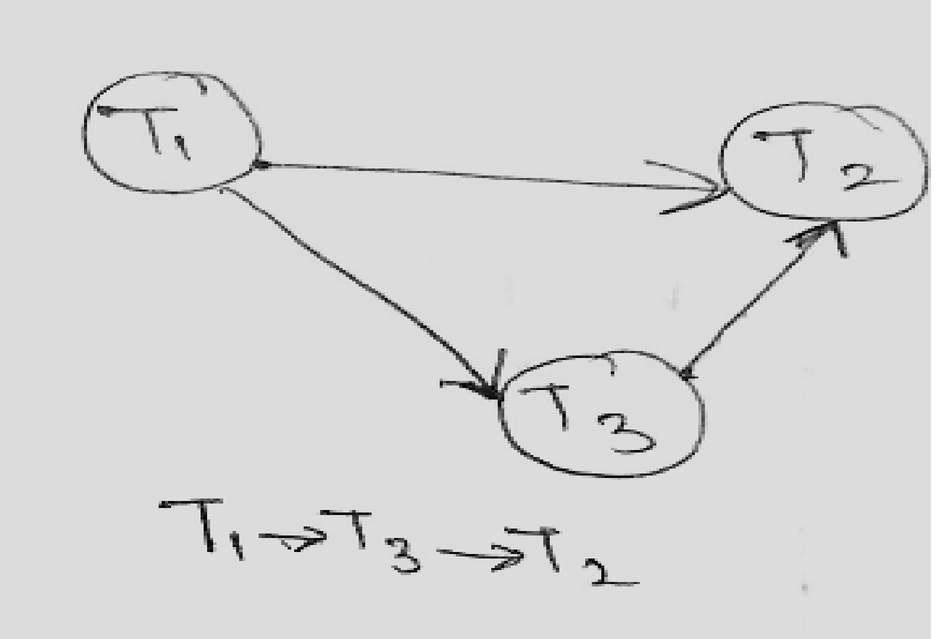
S

T_1	T_2	T_3
R(x)		
	R(y)	
		R(y)
	w(y)	
w(X)		
		w(x)
	R(X)	
	W(X)	

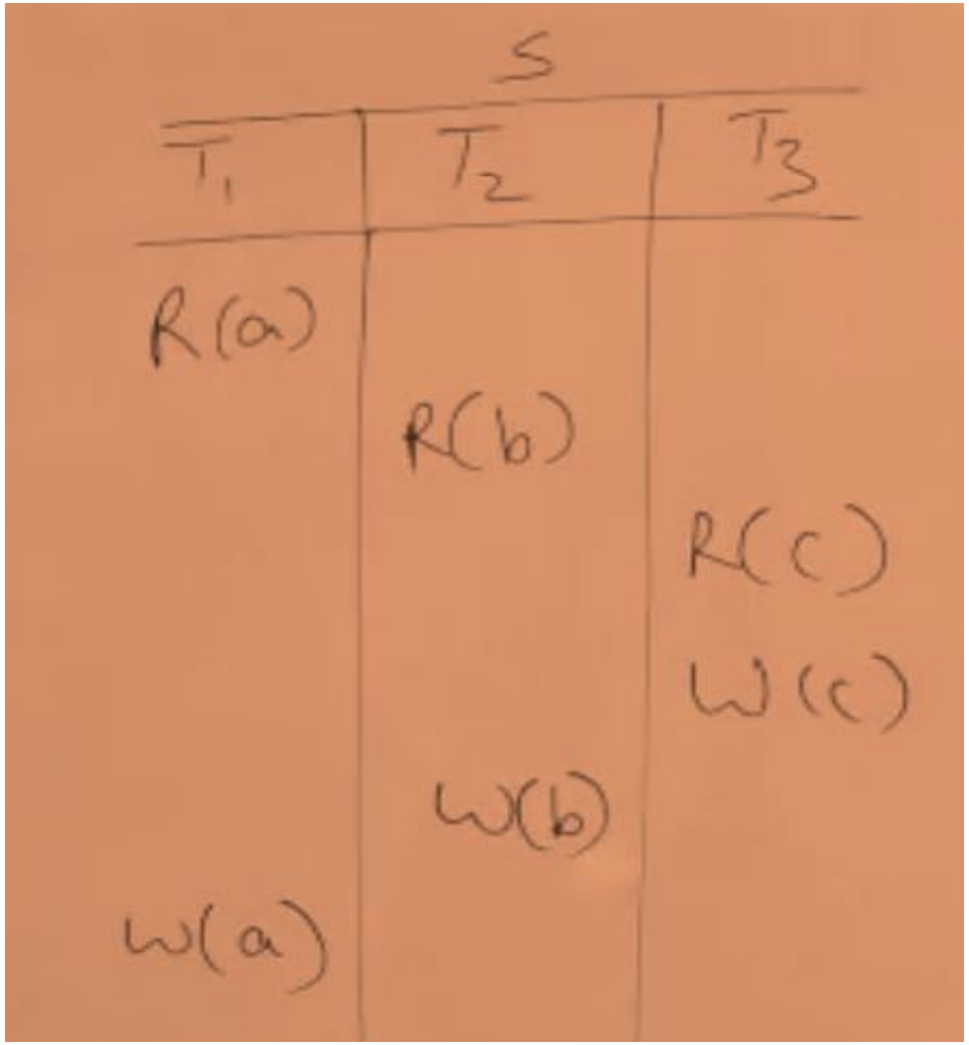
possible serialized schedules are-



$T_1 \rightarrow T_3 \rightarrow T_2$

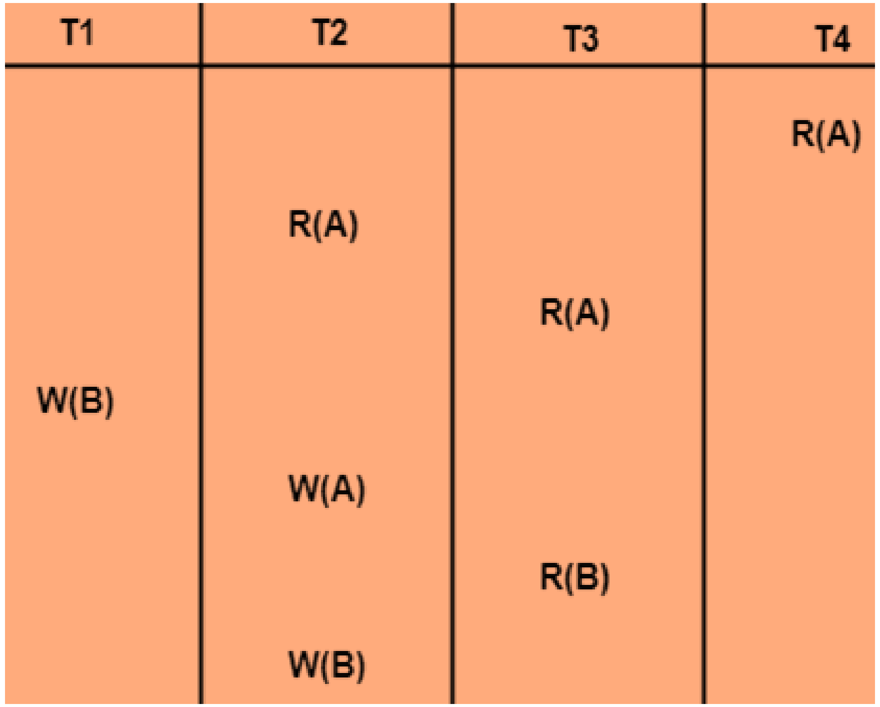


- Clearly, there exists no cycle in the precedence graph.
- Therefore, the given schedule S is conflict serializable.



S

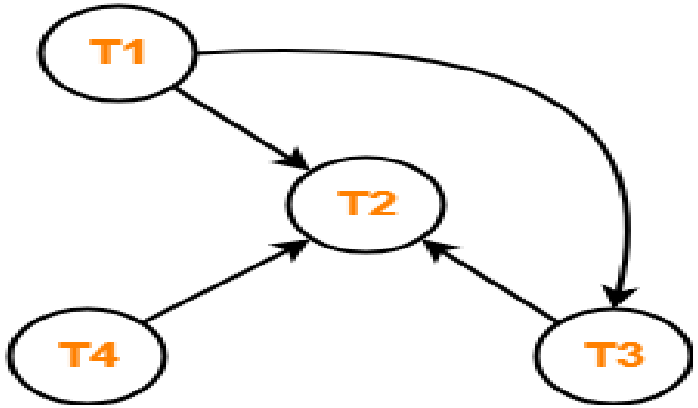
T_1	T_2	T_3
R(a)		
	R(b)	
		R(c)
		W(c)
	w(b)	
w(a)		



T1	T2	T3	T4
			R(A)
	R(A)		
		R(A)	
W(B)			
	W(A)		
		R(B)	
	W(B)		

conflicting operations and determine the dependency between the transactions-

- $R_4(A)$, $W_2(A)$ ($T_4 \rightarrow T_2$)
- $R_3(A)$, $W_2(A)$ ($T_3 \rightarrow T_2$)
- $W_1(B)$, $R_3(B)$ ($T_1 \rightarrow T_3$)
- $W_1(B)$, $W_2(B)$ ($T_1 \rightarrow T_2$)
- $R_3(B)$, $W_2(B)$ ($T_3 \rightarrow T_2$)



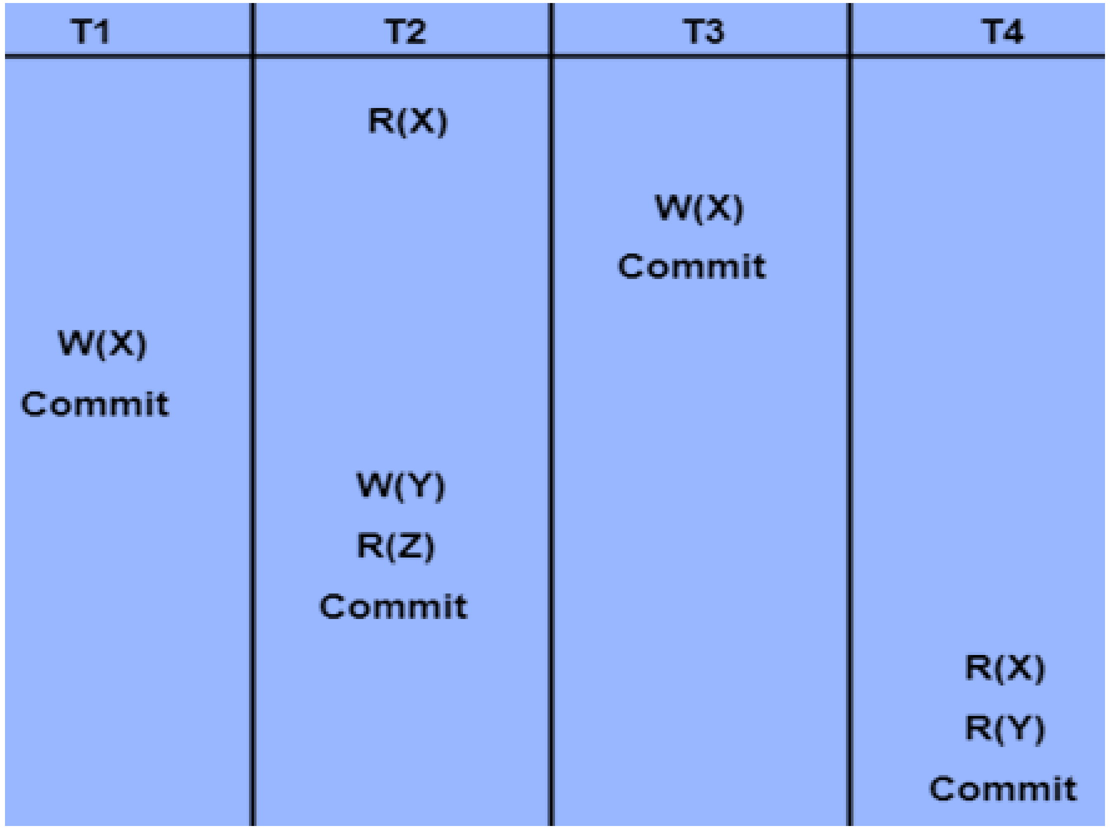
- Clearly, there exists no cycle in the precedence graph.
- Therefore, the given schedule S is conflict serializable

Finding the Serialized Schedules-

- All the possible topological orderings of the above precedence graph will be the possible serialized schedules.
- The topological orderings can be found by performing the Topological Sort of the above precedence graph.

After performing the topological sort, the possible serialized schedules are-

- $\mathbf{T_1 \rightarrow T_3 \rightarrow T_4 \rightarrow T_2}$
- $\mathbf{T_1 \rightarrow T_4 \rightarrow T_3 \rightarrow T_2}$
- $\mathbf{T_4 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2}$



T1	T2	T3	T4
	R(X)		
		W(X)	
		Commit	
W(X)			
Commit			
	W(Y)		
	R(Z)		
	Commit		
			R(X)
			R(Y)
			Commit

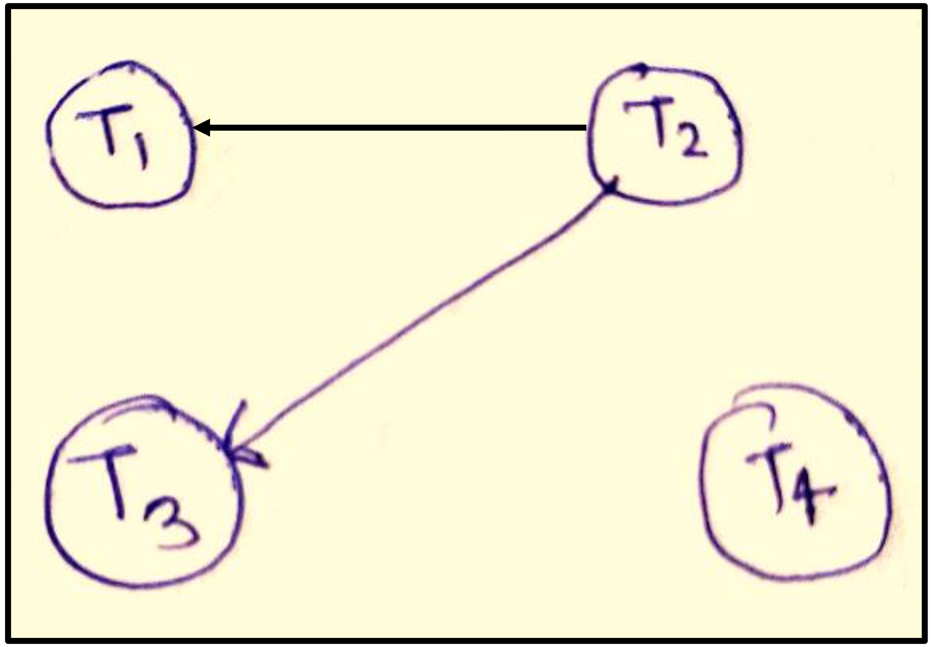
Checking Whether S is Recoverable Or Not-

. Conflict serializable schedules are always recoverable.

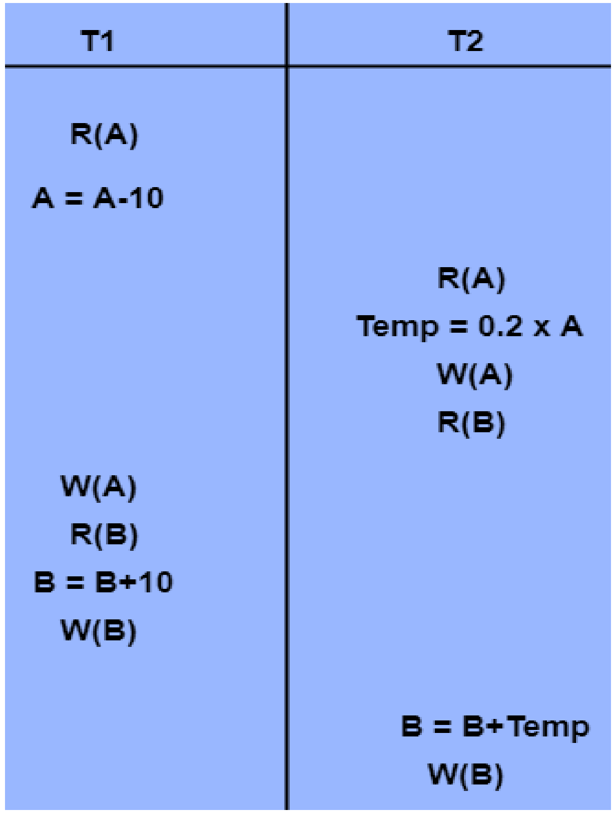
- Therefore, the given schedule S is recoverable.

Alternatively,

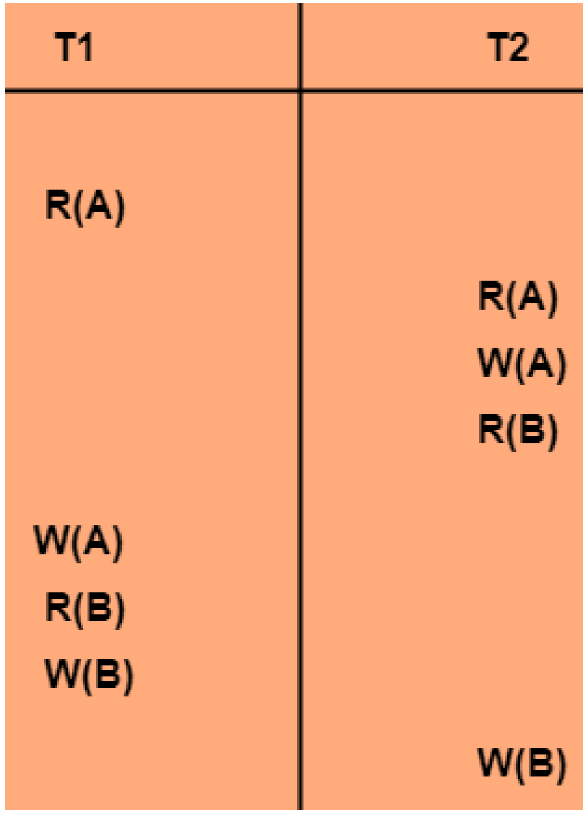
- There exists no dirty read operation.
- This is because all the transactions which update the values commits immediately.



- Clearly, there exists no cycle in the precedence graph.
- Therefore, the given schedule S is conflict serializable.



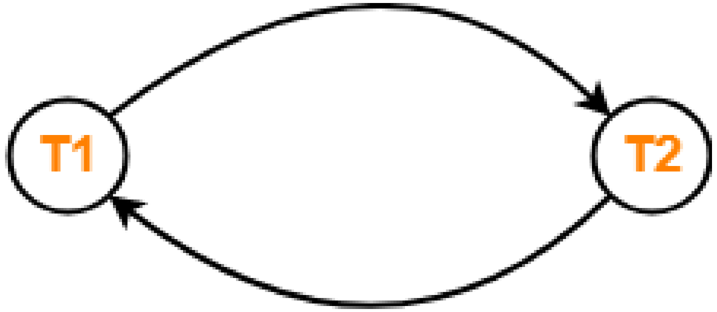
T1	T2
R(A)	
A = A-10	
	R(A)
	Temp = 0.2 x A
	W(A)
	R(B)
W(A)	
R(B)	
B = B+10	
W(B)	
	B = B+Temp
	W(B)



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

conflicting operations and determine the dependency between the transactions-

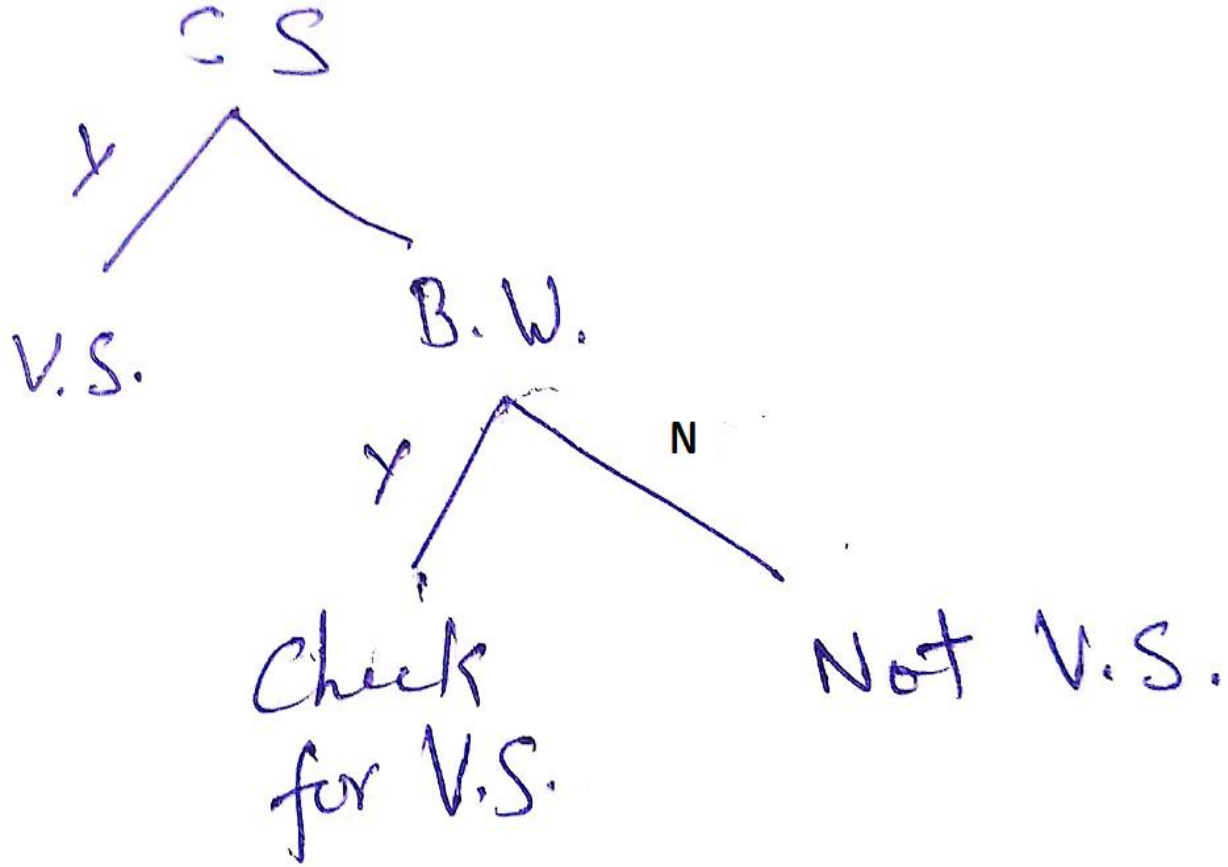
- $R_1(A)$, $W_2(A)$ $(T_1 \rightarrow T_2)$
- $R_2(A)$, $W_1(A)$ $(T_2 \rightarrow T_1)$
- $W_2(A)$, $W_1(A)$ $(T_2 \rightarrow T_1)$
- $R_2(B)$, $W_1(B)$ $(T_2 \rightarrow T_1)$
- $R_1(B)$, $W_2(B)$ $(T_1 \rightarrow T_2)$
- $W_1(B)$, $W_2(B)$ $(T_1 \rightarrow T_2)$

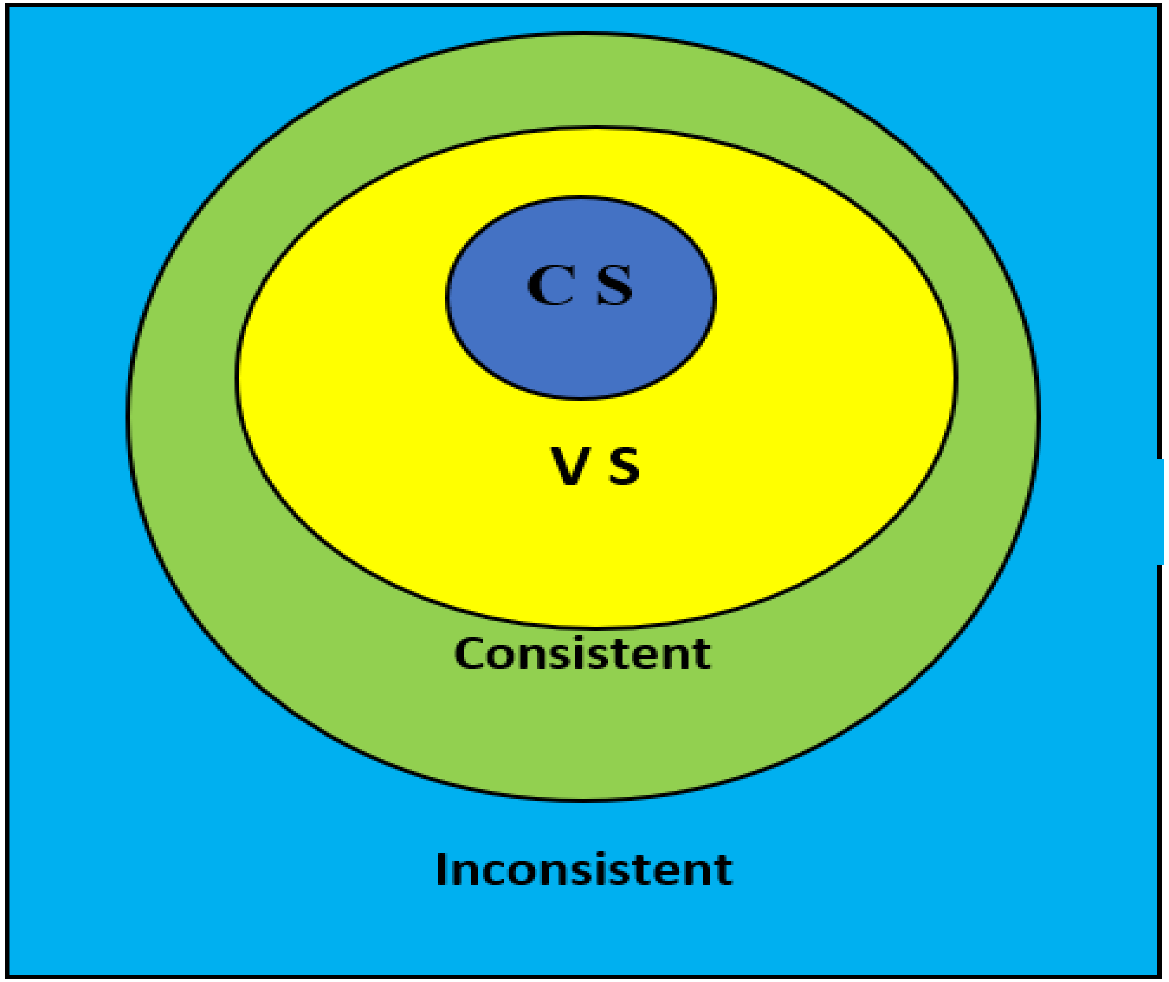


- Clearly, there exists a cycle in the precedence graph.
- Therefore, the given schedule S is not conflict serializable.
- Thus, Number of possible serialized schedules = 0.

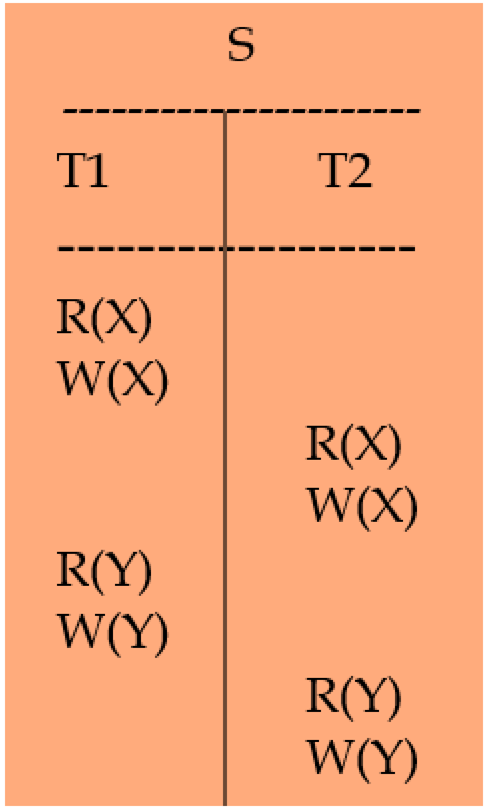
View Serializability in DBMS

- A schedule is view serializable when it is view equivalent to a serial schedule.
- All conflict serializable schedules are view serializable.
- The view serializable which is not a conflict serializable contains blind writes.



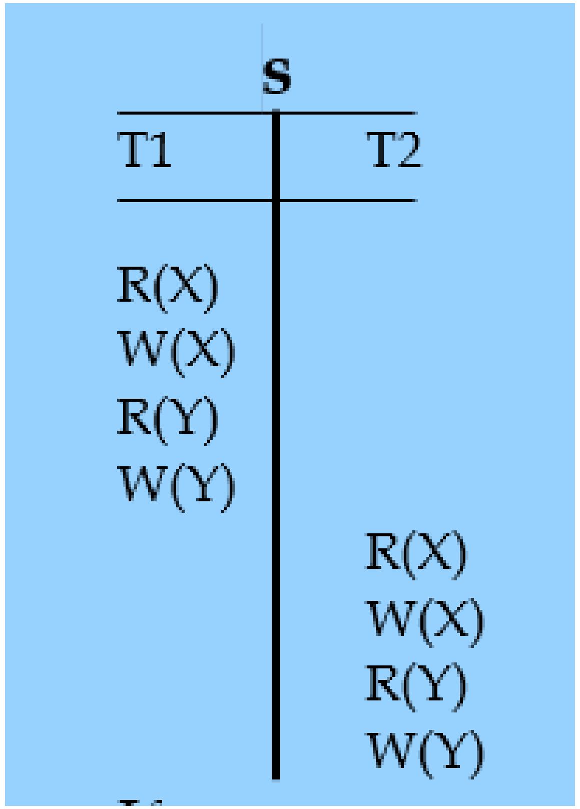


To check whether a given schedule is view serializable, we need to check whether the given schedule is **View Equivalent** to its serial schedule



S

T1	T2
R(X)	
W(X)	
	R(X)
	W(X)
R(Y)	
W(Y)	
	R(Y)
	W(Y)



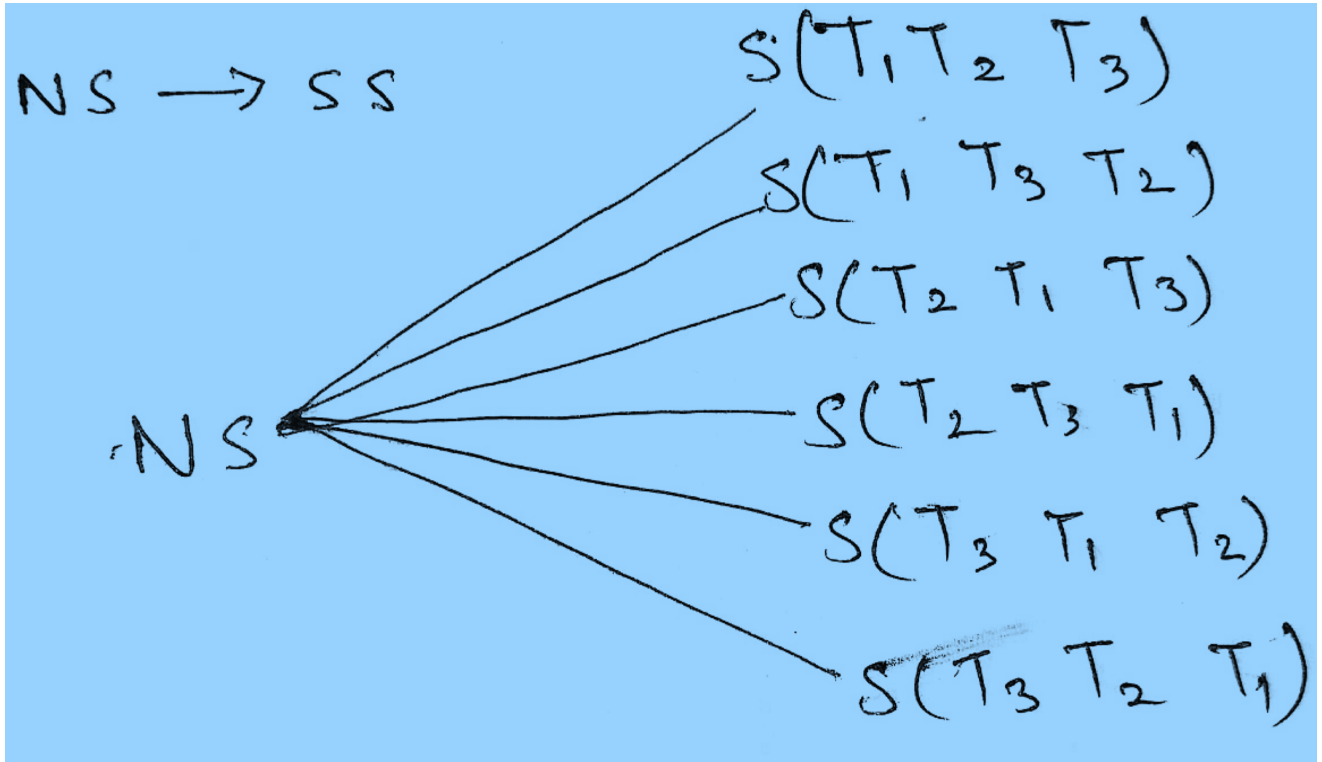
S

T1	T2
R(X)	
W(X)	
R(Y)	
W(Y)	
	R(X)
	W(X)
	R(Y)
	W(Y)

If we can prove that the given schedule is **View Equivalent** to its serial schedule then the given schedule is called **view Serializable**.

Why we need View Serializability?

We know that a serial schedule never leaves the database in inconsistent state because there are no concurrent transactions execution. However a non-serial schedule can leave the database in inconsistent state because there are multiple transactions running concurrently. By checking that a given non-serial schedule is view serializable, we make sure that it is a consistent schedule.



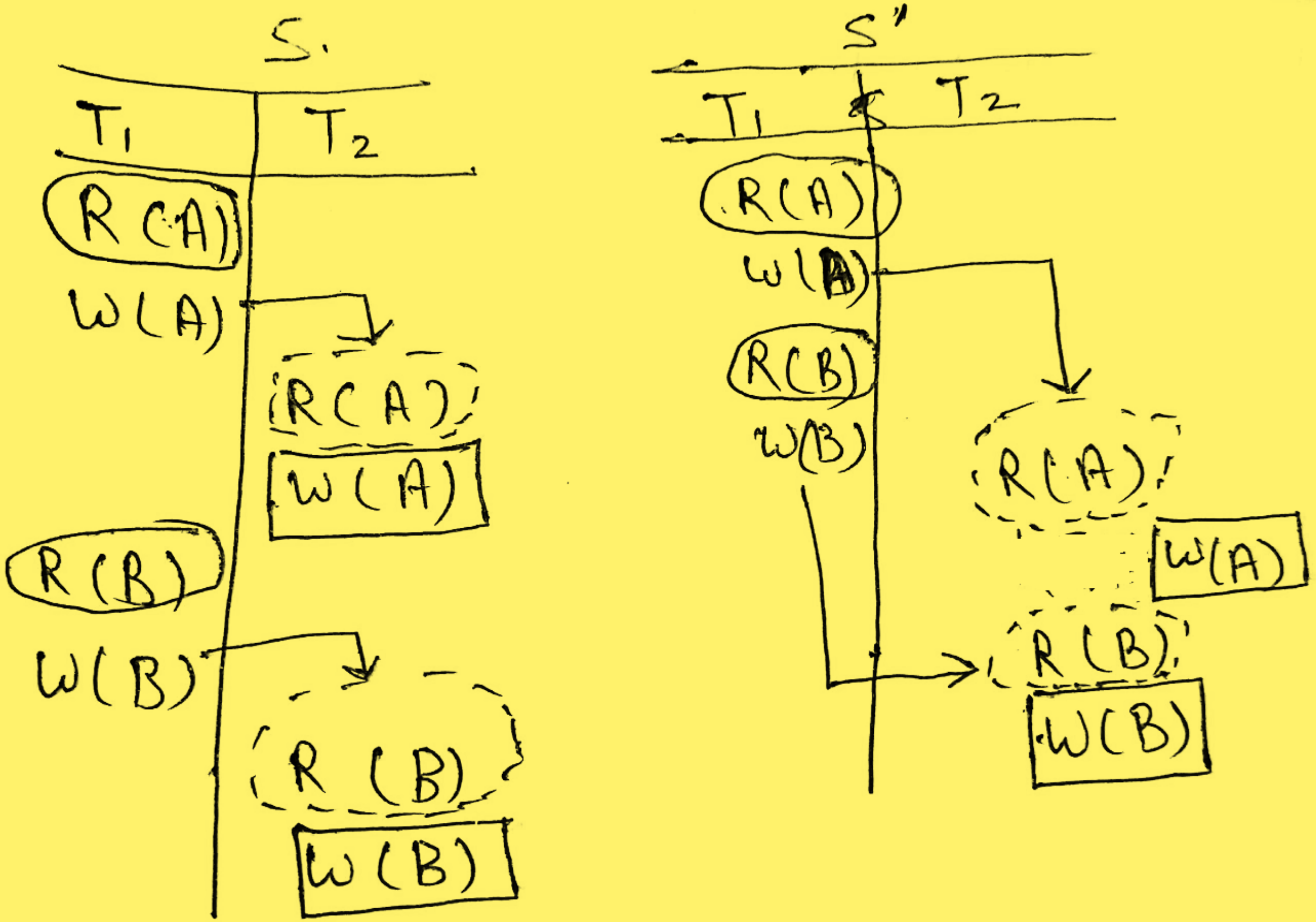
View Equivalent

1. **Initial Read:** Initial read of each data item in transactions must match in both schedules. For example, if transaction T1 reads a data item X before transaction T2 in schedule S1 then in schedule S2, T1 should read X before T2.

Read vs Initial Read: You may be confused by the term initial read. Here initial read means the first read operation on a data item, for example, a data item X can be read multiple times in a schedule but the first read operation on X is called the initial read. This will be more clear once we will get to the example in the next section of this same article.

2. **Final Write:** Final write operations on each data item must match in both the schedules. For example, a data item X is last written by Transaction T1 in schedule S1 then in S2, the last write operation on X should be performed by the transaction T1.

3. **Update Read:** If in schedule S1, the transaction T1 is reading a data item updated by T2 then in schedule S2, T1 should read the value after the write operation of T2 on same data item. For example, In schedule S1, T1 performs a read operation on X after the write operation on X by T2 then in S2, T1 should read the X after T2 performs write on X.



S.

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

S'

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

(I) **Initial Read** ○

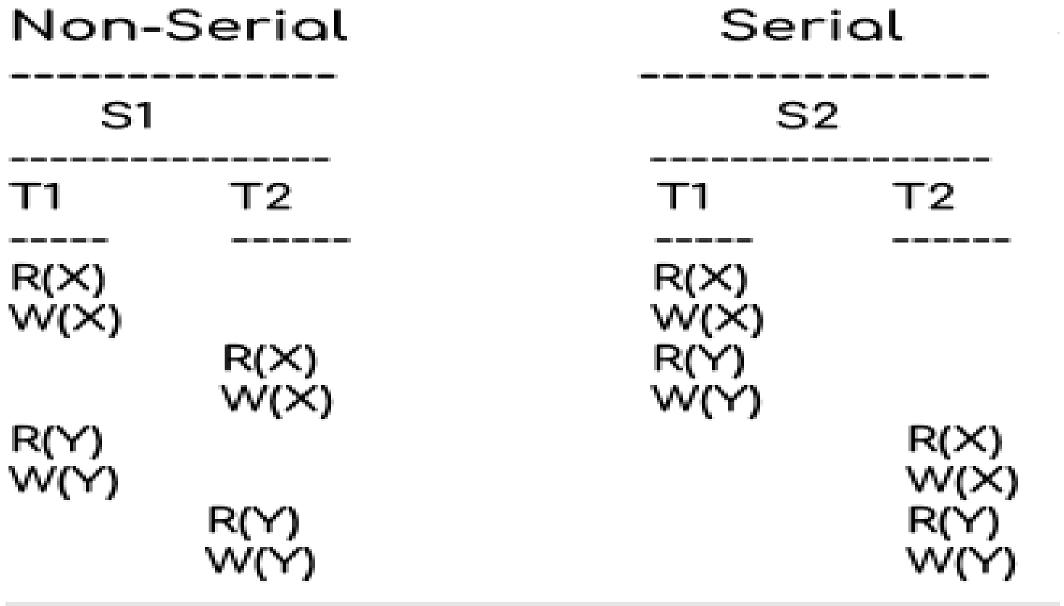
R	S	S'
A	T_1	T_1
B	T_1	T_1

(II) Final Write □

W	S	S'
A	T_2	T_2
B	T_2	T_2

(III) Update Read / Intermediate Read ◌

All the three conditions that checks whether the two schedules are view equivalent are satisfied which means S1 and S2 are view equivalent. So we can say that the schedule S1 is view serializable schedule.



```
Non-Serial                         Serial
--------------                 ----------------
      S1                              S2
----------------                ----------------
T1          T2                  T1          T2
-----       ------              -----       ------
R(X)                            R(X)
W(X)                            W(X)
            R(X)                R(Y)
            W(X)                W(Y)
R(Y)                                        R(X)
W(Y)                                        W(X)
           R(Y)                             R(Y)
           W(Y)                             W(Y)
```

Update Read

In S1, transaction T2 reads the value of X, written by T1. In S2, the same transaction T2 reads the X after it is written by T1.

In S1, transaction T2 reads the value of Y, written by T1. In S2, the same transaction T2 reads the value of Y after it is updated by T1.

Initial Read

In schedule S1, transaction T1 first reads the data item X. In S2 also transaction T1 first reads the data item X.

Lets check for Y. In schedule S1, transaction T1 first reads the data item Y. In S2 also the first read operation on Y is performed by T1.

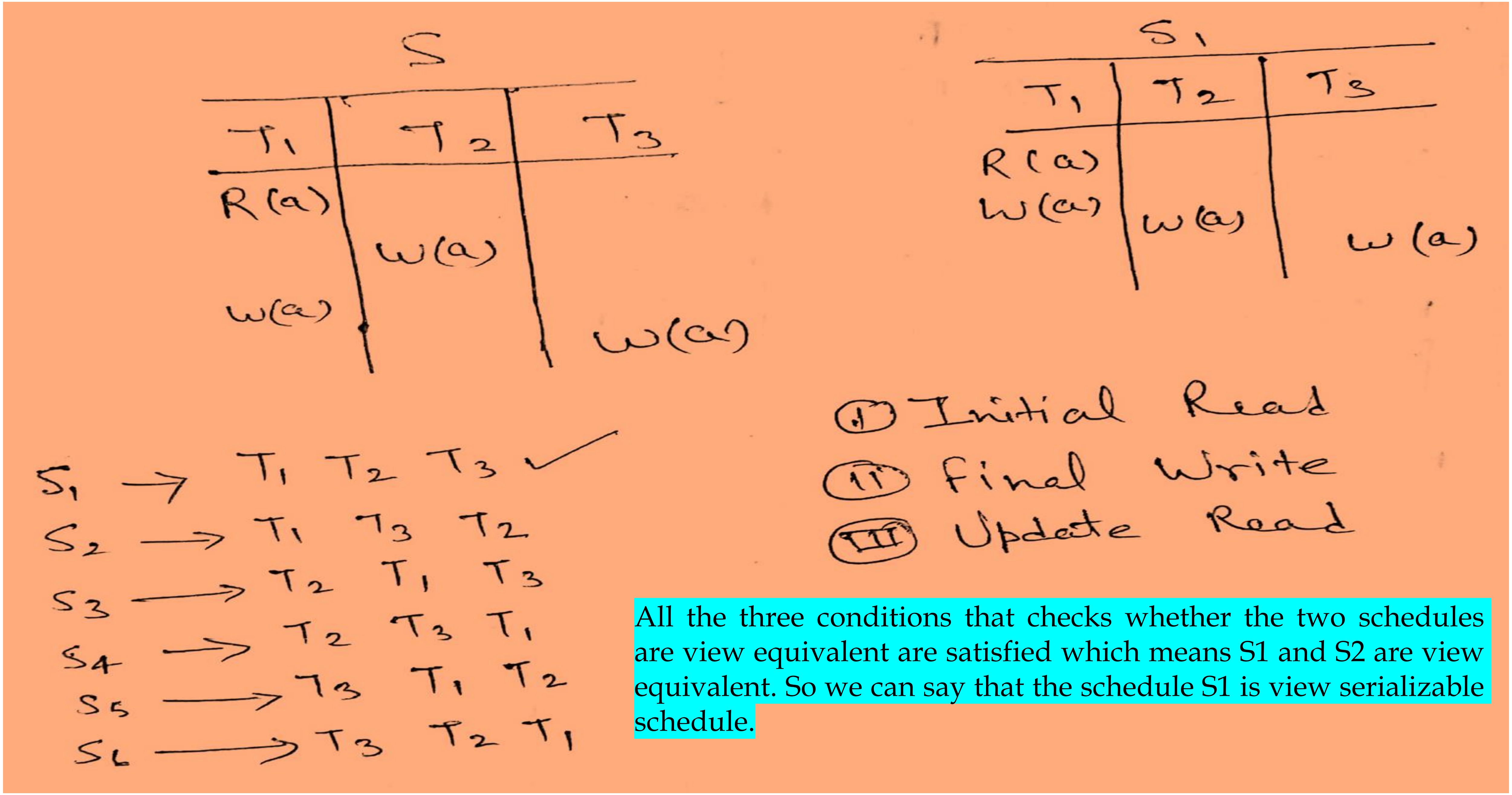
Final Write

In schedule S1, the final write operation on X is done by transaction T2. In S2 also transaction T2 performs the final write on X.

Lets check for Y. In schedule S1, the final write operation on Y is done by transaction T2. In schedule S2, final write on Y is done by T2.

Since all the three conditions that checks whether the two schedules are view equivalent are satisfied in this example, which means S1 and S2 are view equivalent. Also, as we know that the schedule S2 is the serial schedule of S1, thus we can say that the schedule S1 is view serializable schedule.

Q. Check given schedule is View Serializable or not



S

T1	T2	T3
R(a)		
	W(a)	
W(a)		
		W(a)

S1

T1	T2	T3

R(a)
W(a)		

	W(a)	
		W(a)

$S_1 \rightarrow T_1\ T_2\ T_3$ ✓

$S_2 \rightarrow T_1\ T_3\ T_2$

$S_3 \rightarrow T_2\ T_1\ T_3$

$S_4 \rightarrow T_2\ T_3\ T_1$

$S_5 \rightarrow T_3\ T_1\ T_2$

$S_6 \rightarrow T_3\ T_2\ T_1$

(I) Initial Read

(II) Final Write

(III) Update Read

All the three conditions that checks whether the two schedules are view equivalent are satisfied which means S1 and S2 are view equivalent. So we can say that the schedule S1 is view serializable schedule.

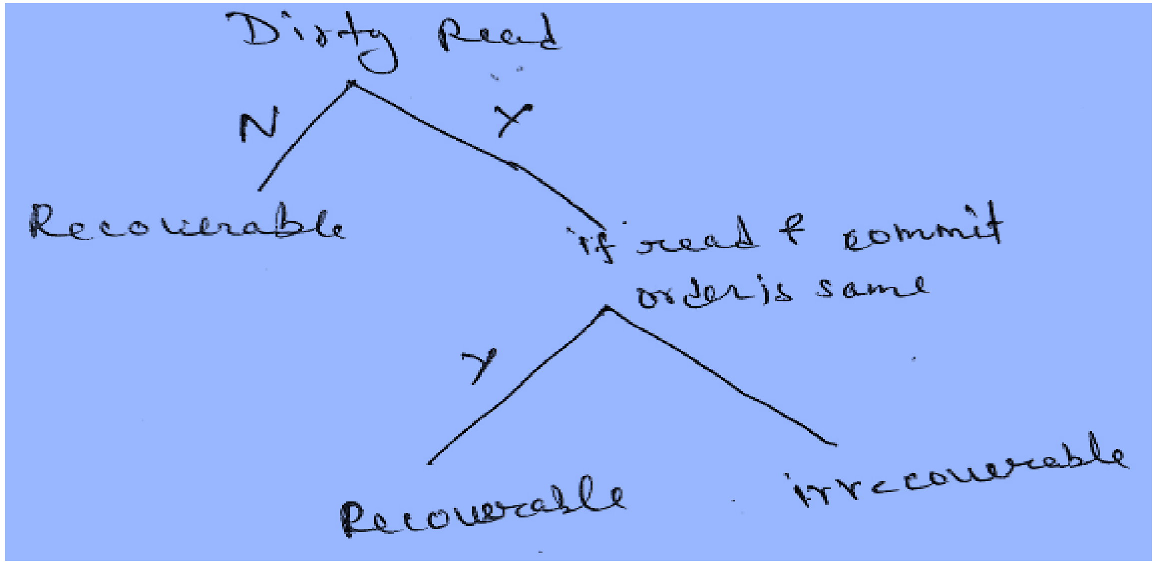
Recoverability

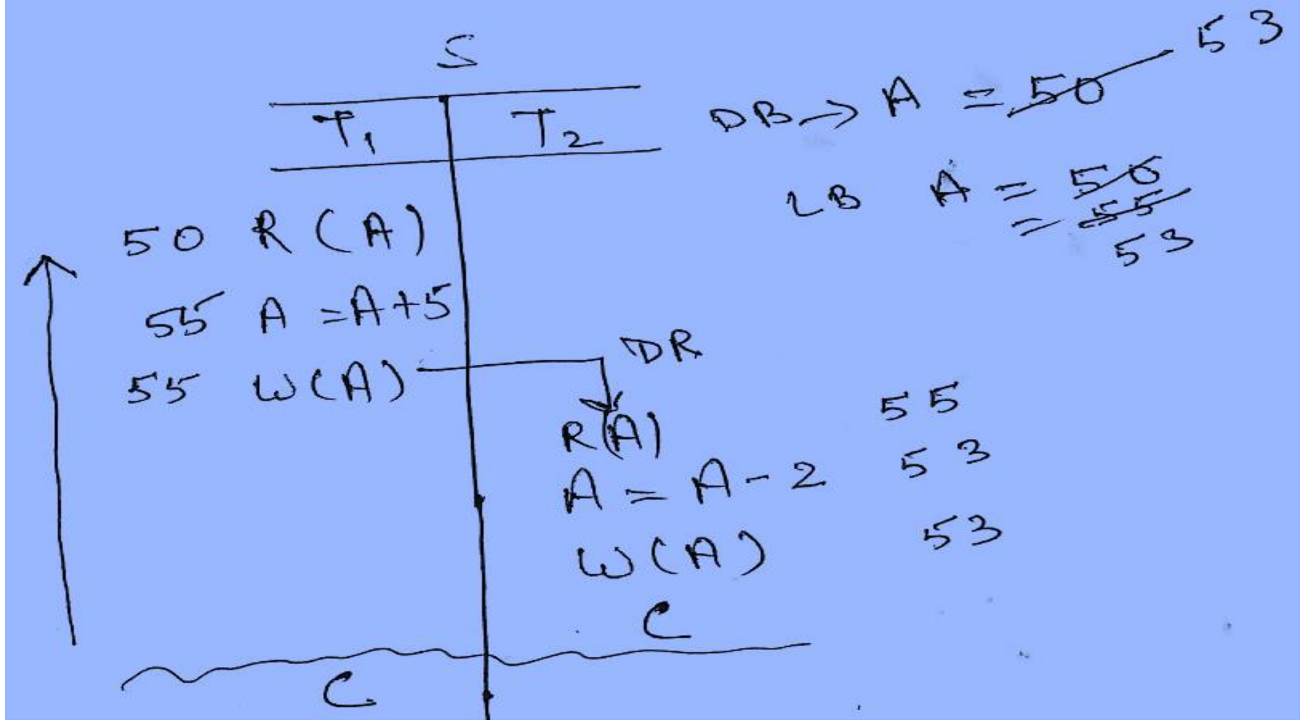
- ❑ Recoverable Schedule
- ❑ Irrecoverable Schedules-

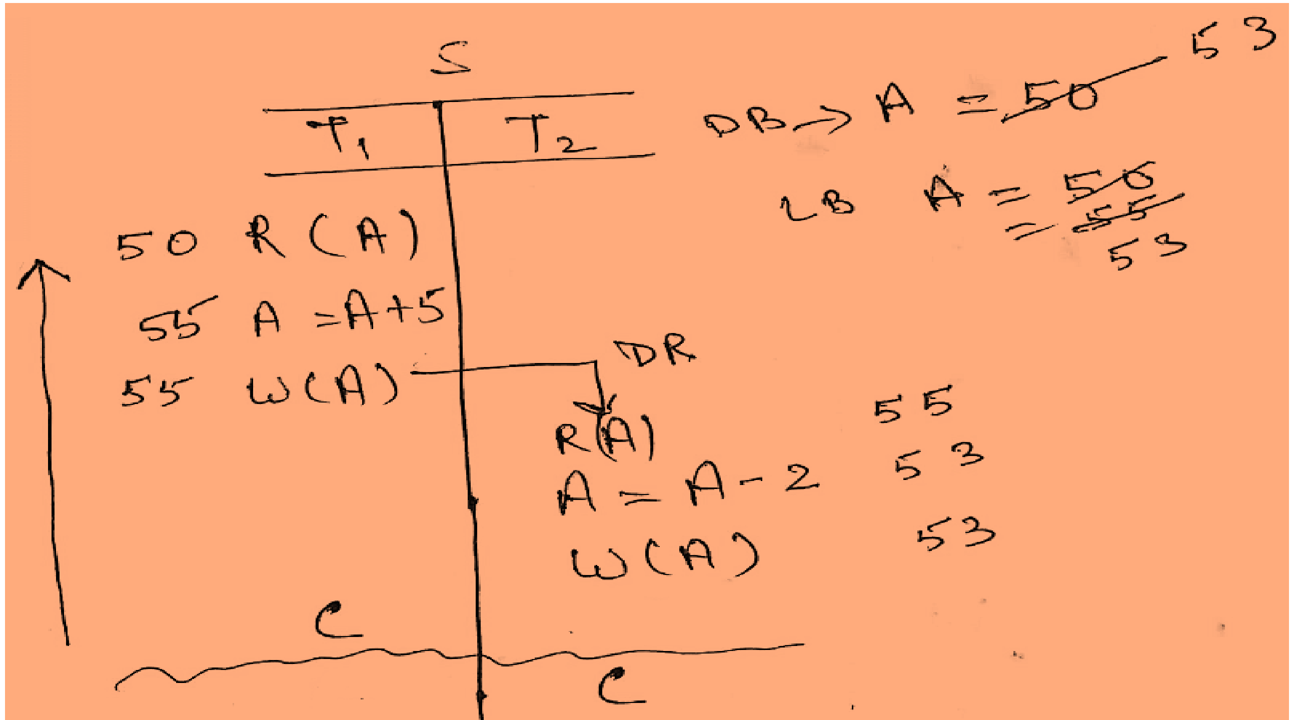
If in a schedule,

- A transaction performs a dirty read operation from an uncommitted transaction
- And commits before the transaction from which it has read the value

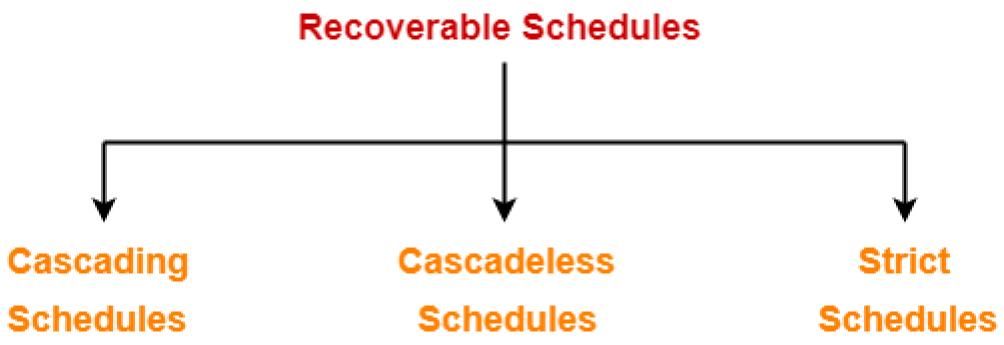
then such a schedule is known as an **Irrecoverable Schedule**.





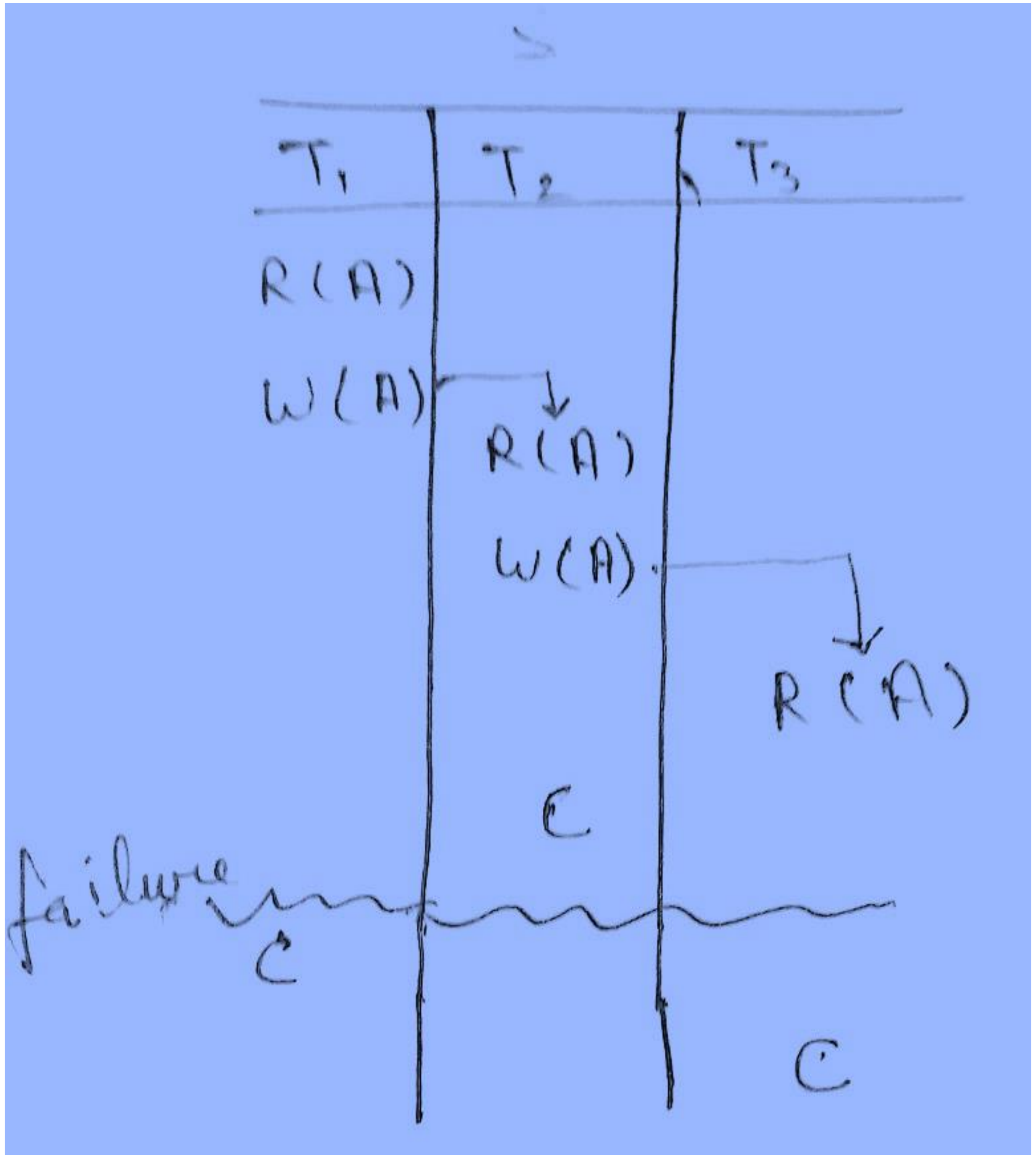


Types of Recoverable Schedules-



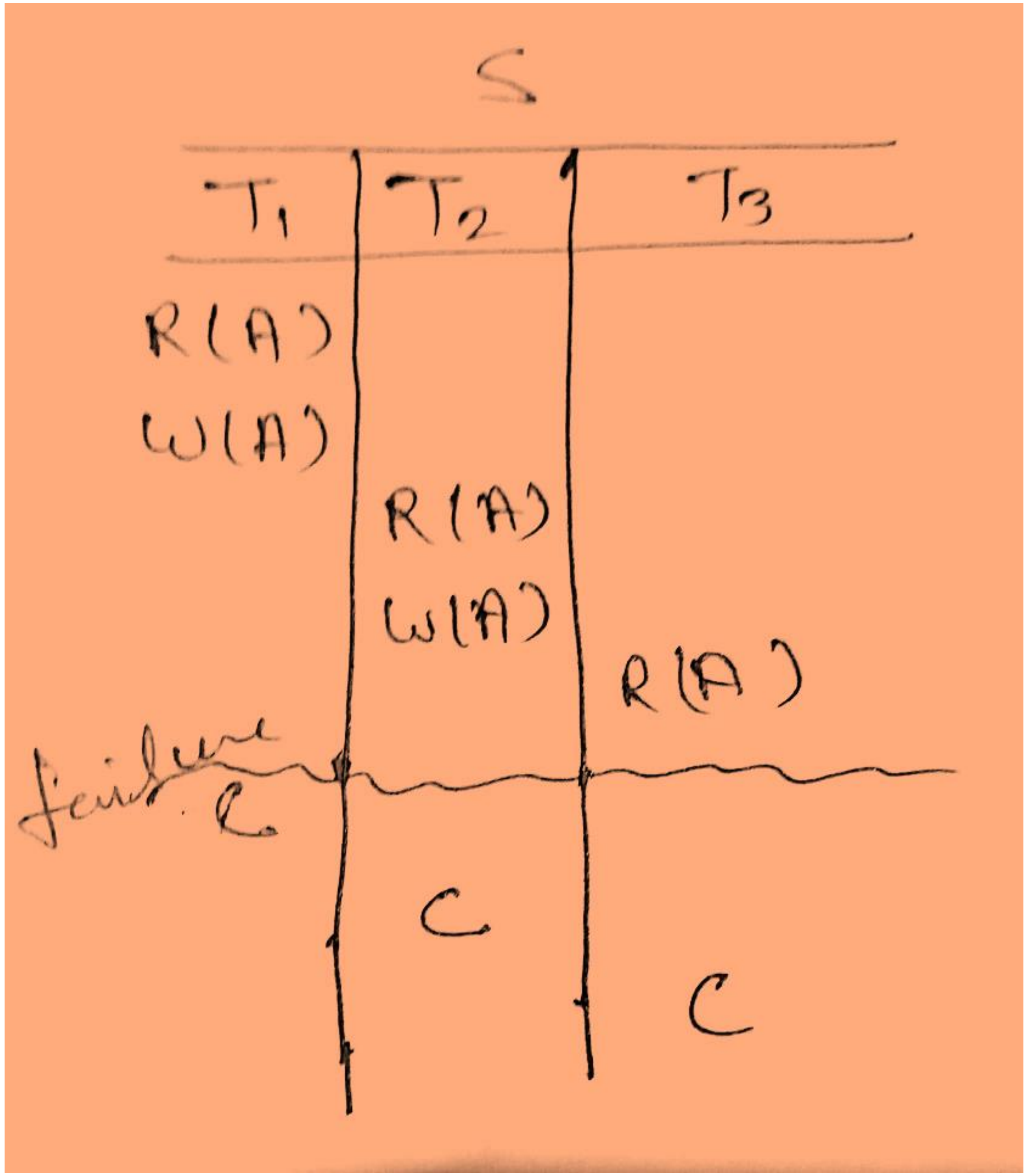
Cascading Schedule-

- If in a schedule, failure of one transaction causes several other dependent transactions to rollback or abort, then such a schedule is called as a **Cascading Schedule** or **Cascading Rollback** or **Cascading Abort**.
- It simply leads to the wastage of CPU time.



S

T1	T2	T3
R(A)		
W(A)		
	R(A)	
	W(A)	
		R(A)
	C	
C		
		C



S

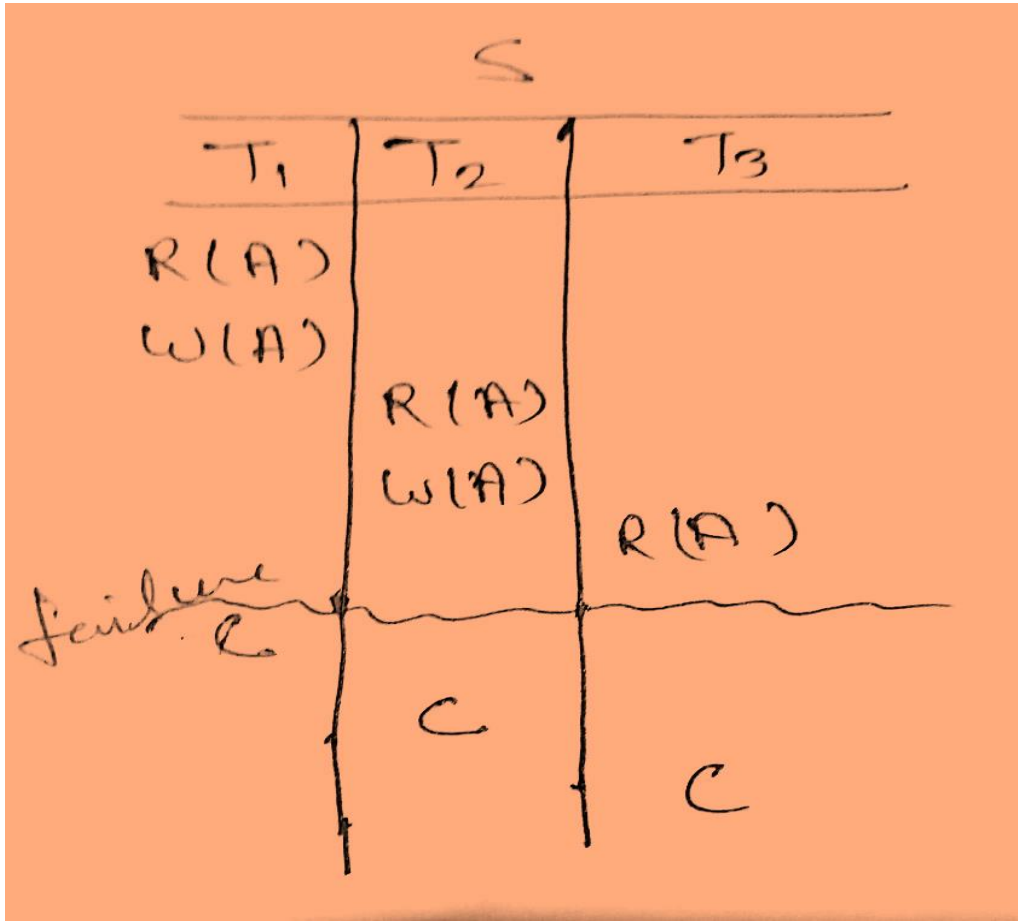
T1	T2	T3
R(A)		
W(A)		
	R(A)	
	W(A)	
		R(A)
C		
	C	
		C

Cascadeless Schedule-

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

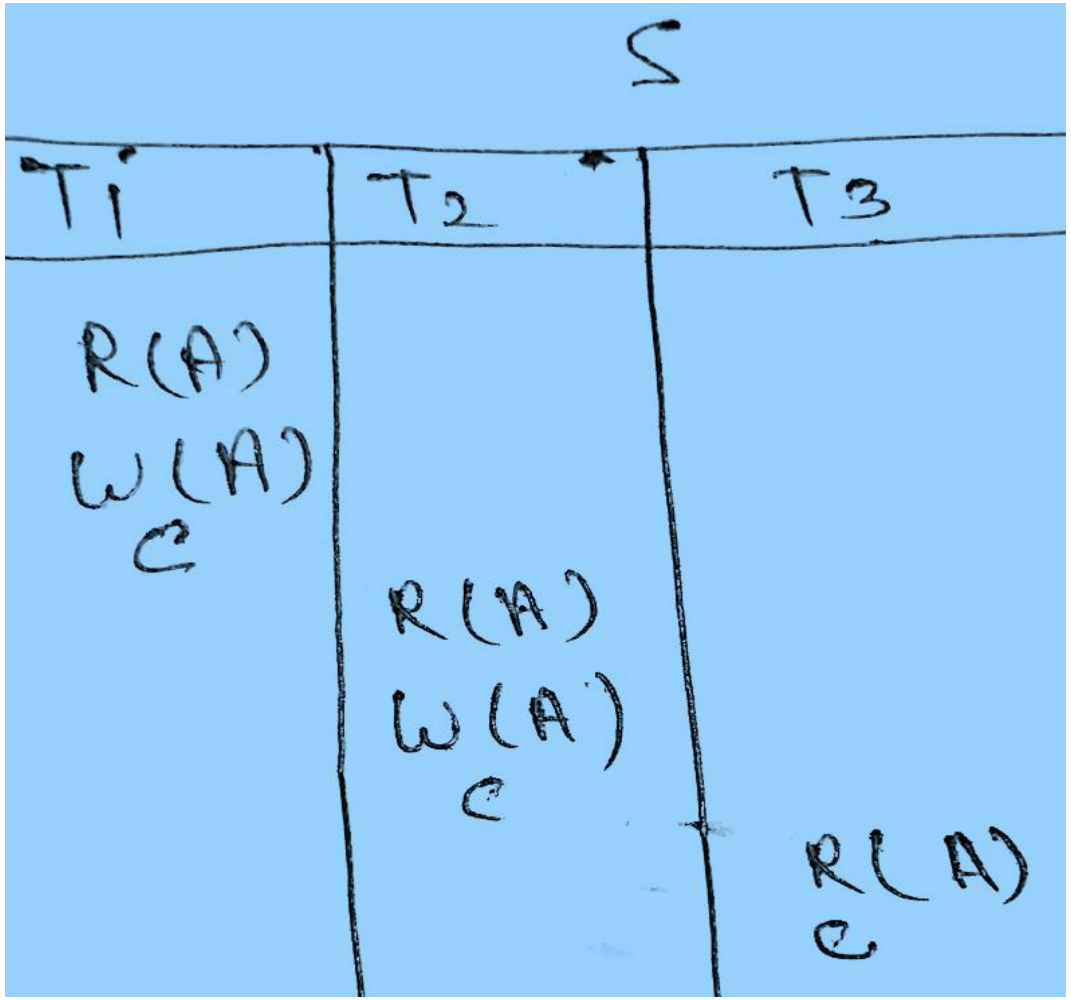
In other words,

- Cascadeless schedule allows only committed read operations.
- Therefore, it avoids cascading roll back and thus saves CPU time.



S

T1	T2	T3
R(A)		
W(A)		
	R(A)	
	W(A)	
		R(A)
failure R		
	C	
		C



S

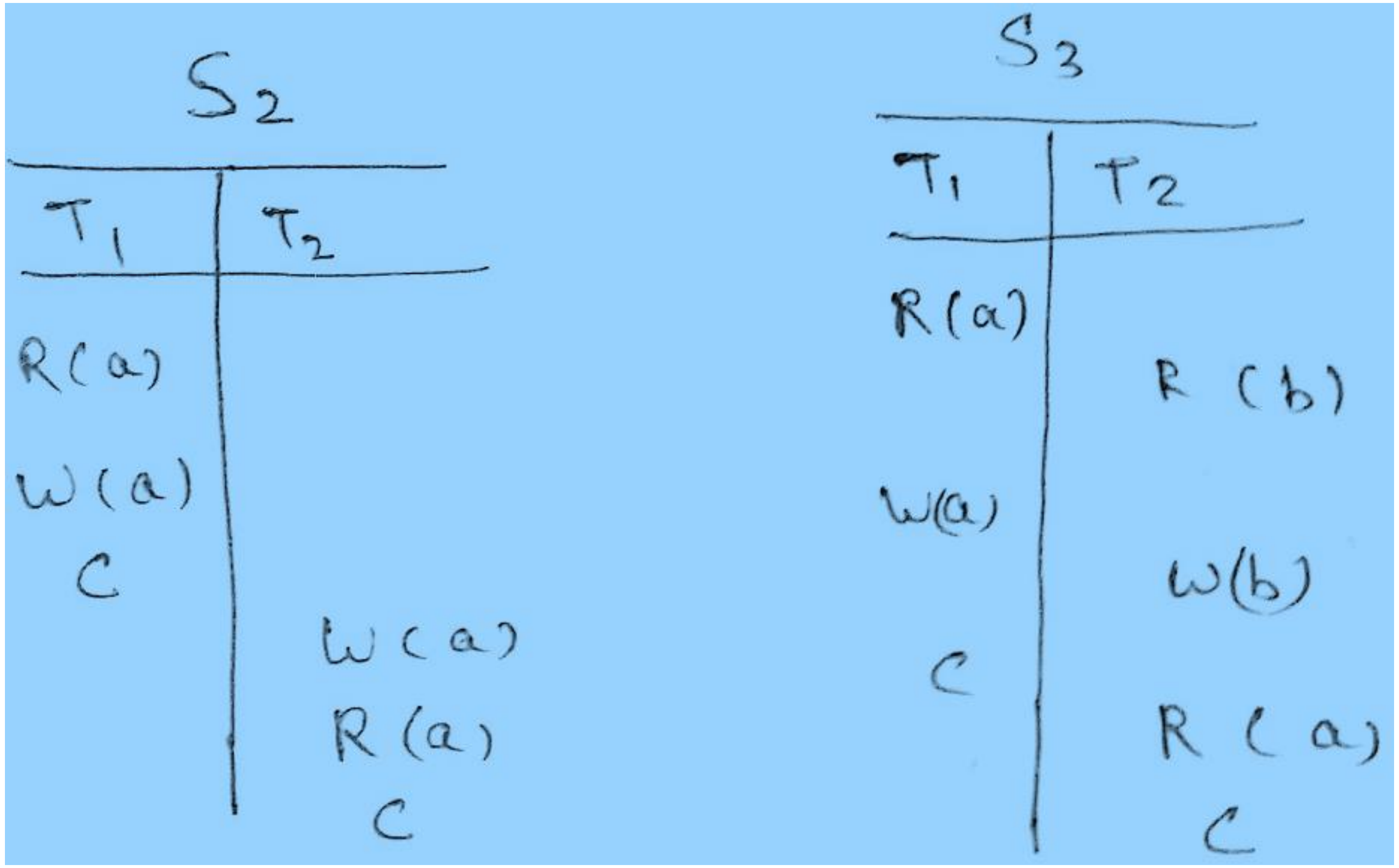
T1	T2	T3
R(A)		
W(A)		
C		
	R(A)	
	W(A)	
	C	
		R(A)
		C

Strict Schedule-

If in a schedule, a transaction is neither allowed to read nor write a data item until the last transaction that has written it is committed or aborted, then such a schedule is called as a **Strict Schedule**.

❑ Strict schedule allows only committed read and write operations.

❑ Clearly, strict schedule implements more restrictions than cascadeless schedule

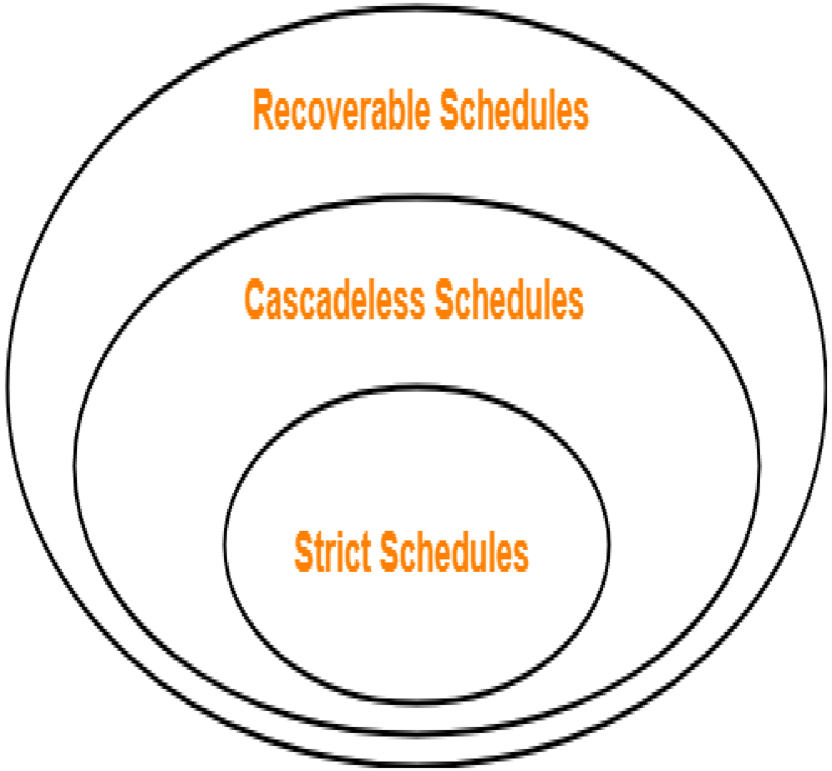


S_2

T_1	T_2
R(a)	
W(a)	
C	
	W(a)
	R(a)
	C

S_3

T_1	T_2
R(a)	
	R(b)
W(a)	
	W(b)
C	
	R(a)
	C



Types of Database Failures

There are many types of failures that can affect database processing. Some failures affect the main memory only, while others involve secondary storage. Following are the types of failure:

•Hardware failures: Hardware failures may include memory errors, disk crashes, bad disk sectors, disk full errors and so on. Hardware failures can also be attributed to design errors, inadequate (poor) quality control during fabrication, overloading (use of under-capacity components) and wearout of mechanical parts.

•Software failures: Software failures may include failures related to software such as, operating system, DBMS software, application programs and so on.

•System crashes: System crashes are due to hardware or software .

Database Recovery

is a process of recovering or restoring data in the database when a data loss occurs or data gets deleted by system crash, hacking, errors in the transaction, damage occurred coincidentally, by viruses, sudden terrible failure, commands incorrect implementation, etc. Data loss or failures happen in databases like other systems but the data stored in the database should be available whenever it's required. For fast restoration or recovery of data, the database must hold tools which recover the data efficiently. It should have atomicity means either the transactions showing the consequence of successful accomplishment perpetually in the database or the transaction must have no sign of accomplishment consequence in the database.

From any failure set of circumstances, there are both voluntary and involuntary ways for both, backing up of data and recovery. So, recovery techniques which are based on deferred update and immediate update or backing up data can be used to stop loss in the database.

Crash recovery

Crash recovery is the operation through which the database is transferred back to a compatible and operational condition. In DBMS, this is performed by rolling back insufficient transactions and finishing perpetrated transactions that even now existed in memory when the crash took place.

With many transactions being implemented with each second shows that, DBMS may be a tremendously complex system. The fundamental hardware of the system manages to sustain robustness and stiffness of software which depends upon its complex design. It’s anticipated that the system would go behind with some methodology or techniques to restore lost data when it fails or crashes in between the transactions.

Classification of failure

The following points are the generalization of failure into various classifications, to examine the source of a problem,

1. **Transaction failure**: a transaction must terminate when it arrives at a point from where it can't extend any further and when it fails to implement the operation. Transaction failure reasons could be,
   - **Logical errors:** The errors which take place in some code or any intrinsic error situation, where a transaction cannot properly fulfil.
   - **System errors:** The errors which take place when the database management system is not able to implement the active transaction, or it has to terminate it because of some conditions in a system.
2. **System Crash**: There are issues which may stop the system unexpectedly from outside and may create the system condition to crash. For example, disturbance or interference in the power supply may create the system condition of fundamental hardware or software to crash or failure.
3. **Disk Failure**: Disk failures comprise bad sectors evolution in the disk, disk inaccessibility, and head crash in the disk, other failures which damage disk storage completely or its particular parts.

Recovery and Atomicity

To recover and also to sustain the transaction atomicity, there are two types of methodology,

- Sustaining each transaction logs and before actually improving the database put them down onto some storage which is substantial.
- Sustaining shadow paging, in which on a volatile memory the improvements are completed and afterward, the real database is reformed.

Log-based Recovery

The log is an order of sequence of records, which sustains the operations record accomplished by a transaction in the database. Before the specific changes and improvements survive on a storage media which is stable and failing securely, it's essential that the logs area unit put down in storage.

Following are the workings of Log-based Recovery,

The log file is not damaged on a stable storage media.

Log-based recovery puts down a log regarding a transaction when a transaction begins to be involved in the system and starts implementation.

Recovery with Concurrent Transactions

The logs are interleaved when multiple transactions are being implemented in collateral. It would be difficult for the system of recovery to make an order of sequence of all logs again, and then start recovering at the time of recovery. Most recent times Database systems use the abstraction of 'checkpoints' to make this condition uncomplicated.

Checkpoint

The checkpoint is an established process where all the logs which are previously used are clear out from the system and stored perpetually in a storage disk. Checkpoint mention a point before which the DBMS was in a compatible state, and all the transactions were perpetrated.