Transaction Management

By :
Dr. Rinkle Rani
Associate Professor, CSED
TIET, Patiala

Transaction :

A **transaction** is a set of logically related operations. For example, you are transferring money from your bank account to your friendôs account, the set of operations would be like this:

Simple Transaction Example

1. Read your account balance
2. Deduct the amount from your balance
3. Write the remaining balance to your account
4. Read your friendôs account balance
5. Add the amount to his account balance
6. Write the new updated balance to his account

Although only read, write and update operations are used in the above example but the transaction can have operations like read, write, insert, update, delete etc.

In DBMS, we write the above 6 steps transaction like this:
Lets say your account is A and your friendôs account is B, you are transferring 10000 from A to B, the steps of the transaction are:

1. R(A);
2. A = A - 10000;
3. W(A);
4. R(B);
5. B = B + 10000;
6. W(B);

In the above transaction **R** refers to the **Read operation** and **W** refers to the **write operation**.

Transaction failure in between the operations

The main problem that can happen during a transaction is that the transaction can fail before finishing the all the operations in the set. This can happen due to power failure, system crash etc. This is a serious problem that can leave database in an inconsistent state.

Assume that transaction fail after third operation then the amount would be deducted from your account but your friend will not receive it.

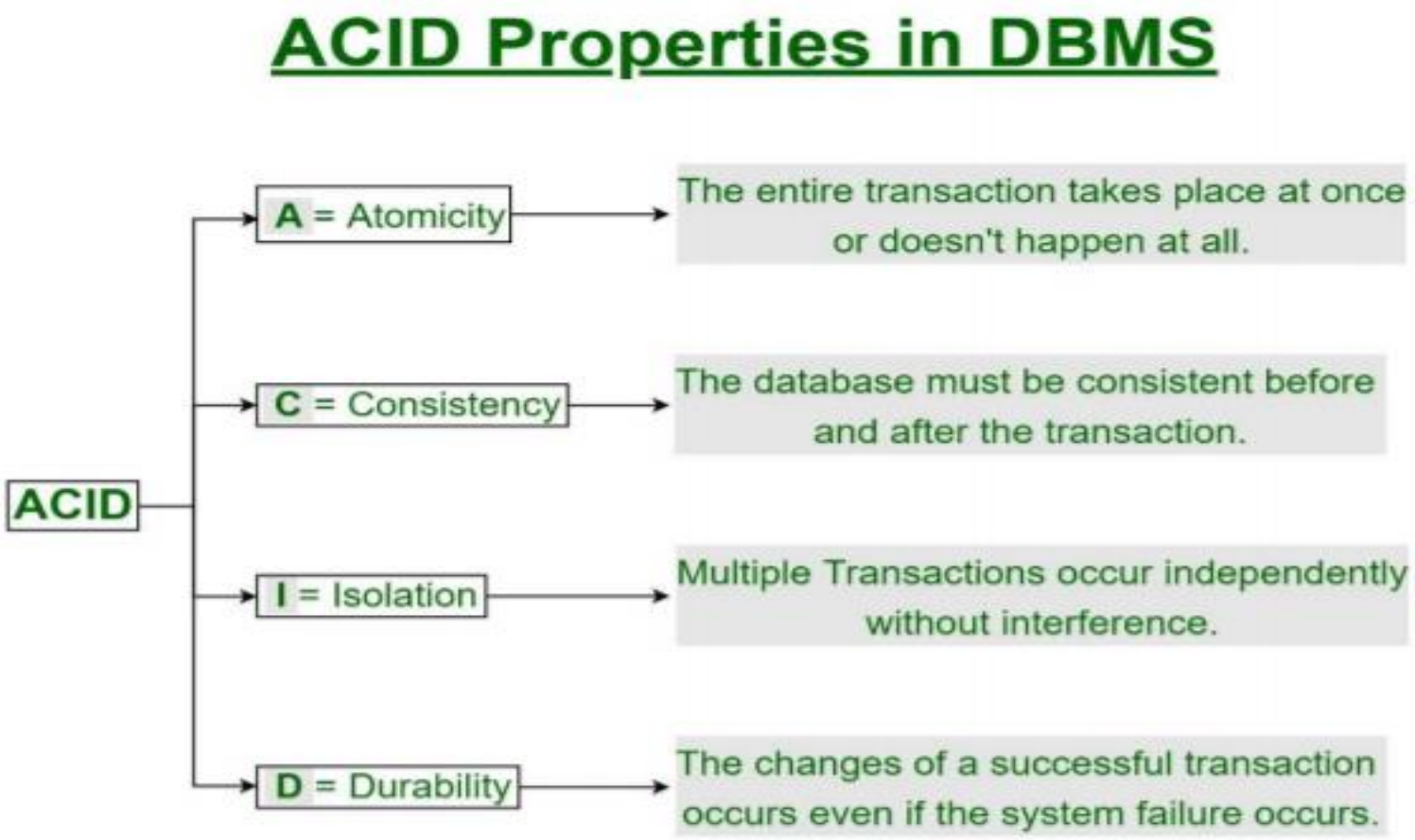
To handle this problem, there are following two operations.

Commit: If all the operations in a transaction are completed successfully then commit those changes to the database permanently.

Rollback: If any of the operation fails then rollback all the changes done by previous operations.

ACID properties in DBMS

To ensure the integrity of data during a transaction the database system maintains the following properties. These properties are widely known as ACID properties:



Atomicity

By this, we mean that either the entire transaction takes place at once or doesnôt happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.

ð **Abort**: If a transaction aborts, changes made to database are not visible.

ð **Commit**: If a transaction commits, changes made are visible.

Atomicity is also known as the óAll or nothing ruleô.

Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.

Before: X : 500	Y: 200
Transaction T	
T1	**T2**
Read (X) X: = X − 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

If the transaction fails after completion of **T1** but before completion of **T2**.(say, after **write(X)** but before **write(Y)**), then amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in entirety in order to ensure correctness of database state.

Consistency

This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database.

Referring to the example above,

The total amount before and after the transaction must be maintained.

Total **before T** occurs = **500 + 200 = 700**.

Total **after T occurs** = **400 + 300 = 700**.

Therefore, database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result T is incomplete.

Isolation ī In a database system where more than one transaction are being executed simultaneously and in parallel, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. No transaction will affect the existence of any other transaction.

Let **X**= 500, **Y** = 500.
Consider two transactions **T** and **T".**

T	T"
Read (X) X: = X*100 Write (X) Read (Y) Y: = Y – 50 Write (Y)	Read (X) Read (Y) Z: = X + Y Write (Z)

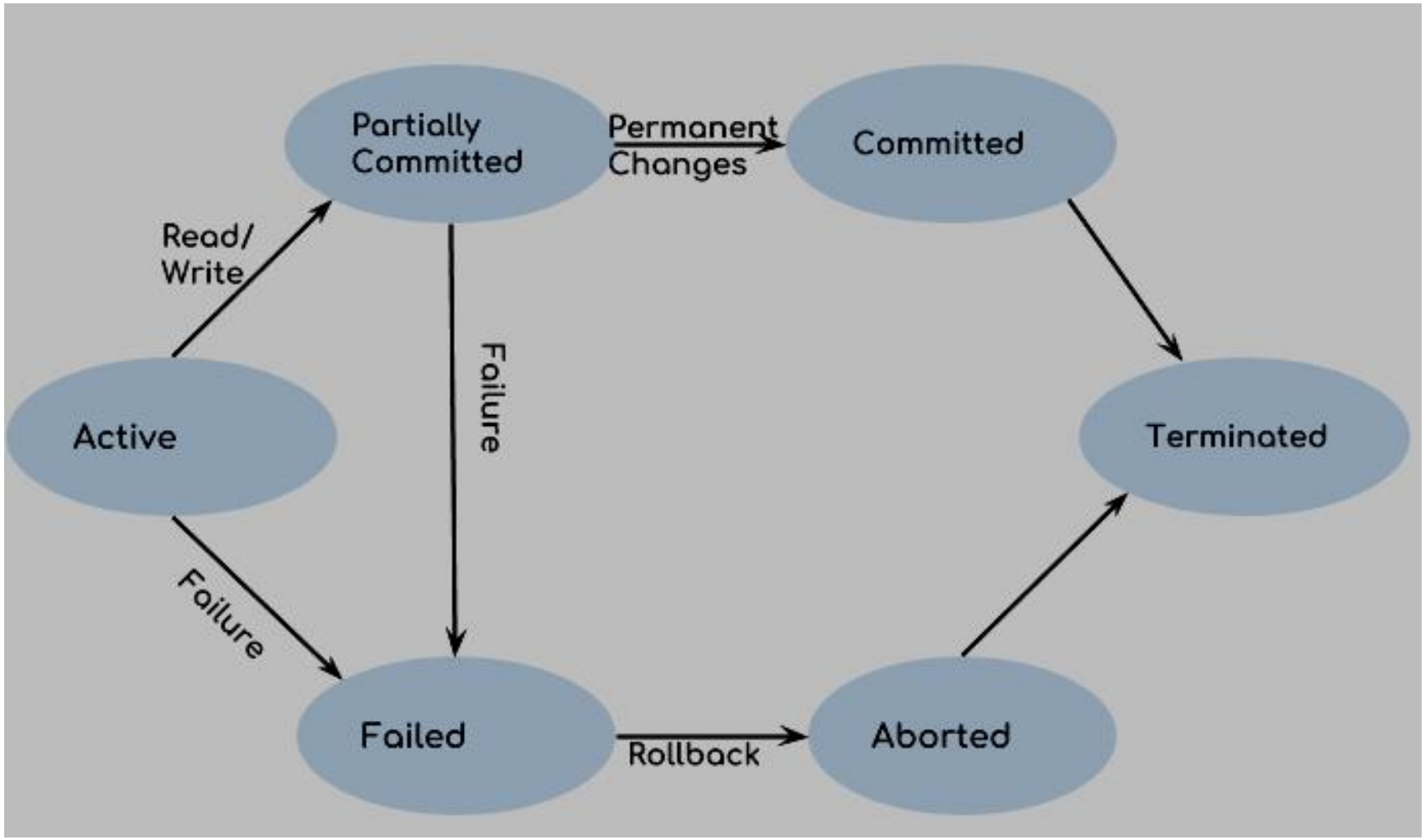
Suppose **T** has been executed till **Read (Y)** and then **T"** starts. As a result , interleaving of operations takes place due to which **T"** reads correct value of **X** but incorrect value of **Y** and sum computed by **T": (X+Y = 50, 000+500=50, 500)** is thus not consistent with the sum at end of transaction: **T: (X+Y = 50, 000 + 450 = 50, 450)**. Hence, transactions must take place in isolation .

Durability:

This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

DBMS Transaction States

A transaction in DBMS can be in one of the following states.



Active State

As we have discussed, that a transaction is a sequence of operations. If a transaction is in execution then it is said to be in active state. It doesnôt matter which step is in execution, until unless the transaction is executing, it remains in active state.

Failed State

If a transaction is executing and a failure occurs, either a hardware failure or a software failure then the transaction goes into failed state from the active state.

Partially Committed State

A transaction goes into ñpartially committedò state from the active state when there are read and write operations present in the transaction.

A transaction contains number of read and write operations. Once the whole transaction is successfully executed, the transaction goes into partially committed state where we have all the read and write operations performed on the main memory (local memory) instead of the actual database.

The reason why we have this state is because a transaction can fail during execution so if we are making the changes in the actual database instead of local memory, database may be left in an inconsistent state in case of any failure. **This state helps us to rollback the changes made to the database in case of a failure during execution.**

Committed State

If a transaction completes the execution successfully then all the changes made in the local memory during partially committed state are permanently stored in the database. You can also see in the above diagram that a transaction goes from partially committed state to committed state when everything is successful.

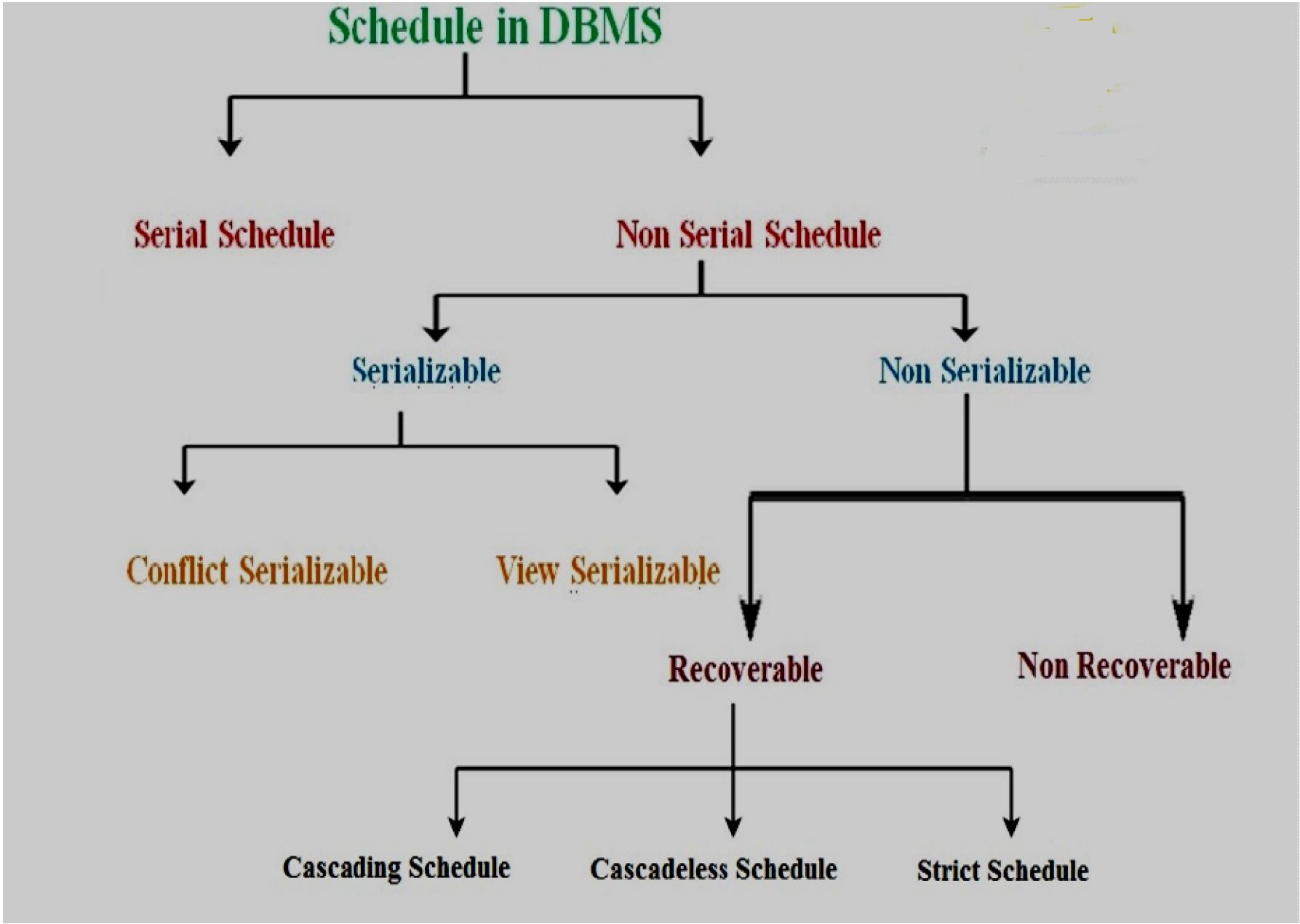
Aborted State

As we have seen above, if a transaction fails during execution then the transaction goes into a failed state. The changes made into the local memory (or buffer) are rolled back to the previous consistent state and the transaction goes into aborted state from the failed state. Refer the diagram to see the interaction between failed and aborted state.

What is a Schedule

We know that transactions are set of instructions and these instructions perform operations on database. When multiple transactions are running concurrently then there needs to be a sequence in which the operations are performed because at a time only one operation can be performed on the database. This sequence of operations is known as **Schedule**.

Schedule, as the name suggests, is a process of lining the transactions and executing them one by one.


Schedule in DBMS
Serial Schedule
Non Serial Schedule
Serializable
Non Serializable
Conflict Serializable
View Serializable
Recoverable
Non Recoverable
Cascading Schedule
Cascadeless Schedule
Strict Schedule

Types of Schedules in DBMS

1) Serial Schedule

In **Serial schedule**, a transaction is executed completely before starting the execution of another transaction.
In other words, you can say that in serial schedule, a transaction does not start execution until the currently running transaction finished execution.
This type of execution of transaction is also known as **non-interleaved** execution.

Serial Schedule example

Here R refers to the read operation and W refers to the write operation. In this example, the transaction T2 does not start execution until the transaction T1 is finished.

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

Non-Serial Schedule:

- This is a type of Scheduling where the operations of multiple transactions are interleaved. This might lead to a rise in the concurrency problem.
- The transactions are executed in a non-serial manner, keeping the end result correct and same as the serial schedule.
- Unlike the serial schedule where one transaction must wait for another to complete all its operation, in the non-serial schedule, the other transaction proceeds without waiting for the previous transaction to complete.
- This sort of schedule does not provide any benefit of the concurrent transaction. The main disadvantage of a non-serial schedule is that there is a potential risk that the system may go into an inconsistent state.
- It can be of two types namely, Serializable and Non-Serializable Schedule.

Serializable Schedule

The objective of serializability is to find a non-serial schedule that allows transactions to execute concurrently without interfering with one another and producing a database state that a serial execution could produce.

A schedule is called serializable if both the serial and non-serial Schedule gives the same output.

These are of two types:

- Conflict Serializable Schedule
- View Serializable Schedule

Time	S_1	
	T_1	T_2
t_1	R(A)	
t_2	W(A)	
t_3		R(B)
t_4		W(B)
t_5	R(B)	
t_6	W(B)	
t_7		R(A)
t_8		W(A)

Conflict Serializable Schedule

If we try to convert a non-serial transaction to a serial transaction, we need to swap instructions between the non-serial Schedule.
So, after swapping non-conflicting instructions, if we can convert a non-serial schedule into a serial one, that Schedule is called conflict serializable. The non-serial Schedule is proved to be a consistent one.

Conflict Operations
Two operations are said to be conflict operations if they satisfy the following three conditions
(1) Both operations should belong to different transactions
(2) Both operations should use the Same data Item.
(3) There should be at least one Write operation between this two operation.

Example of Conflict Serializable Schedule

Consider a Schedule Given Below

S: R1(A) , W1(A) , R2(A) , W2(A),R1(B), W1(B),R2(B),W2(B)

Letôs see whether this Schedule is conflict serializable or not.

So W2(A) and R1(B) are non-conflicting operations to swap them.

After Swapping the order of Schedule will be

S: R1(A) , W1(A),R2(A),R1(B),W2(A),W1(B),R2(B),W2(B)

R2(A) and R1(B) are non-conflicting operations so that we can swap them. After swapping, the sequence will be like given below.

S: R1(A),W1(A),R1(B),R2(A),W2(A),W1(B),R2(B),W2(B)

R2(A) and W1(B) are non-conflicting operations because they are performed on a different data item to swap these operations.

After Swapping, the sequence will be like given below.

S: R1(A),W1(A),R1(B),R2(A), W1(B),W2(A),R2(B),W2(B)

R2(A) and W1(B) are non-conflicting operations, so we can swap them after swapping the sequence will be like given below.

S: R1(A),W1(A),R1(B),W1(B),R2(A),W2(A),R2(B),W2(B)

This resultant Schedule is a serial schedule, so here we can convert the given non-serial Schedule to a serial schedule by swapping the non-conflicting operations, so this verifies that the given Schedule is Conflict Serializable.

View Serializable Schedule

We have discussed that if a schedule is conflict serializable then the output must be consistent. But even if the schedule is not conflict serializable there may be a chance that the schedule is consistent. There comes the concept of view serializability.

A schedule is view serializable if it is view equivalent to a serial schedule.

Every conflict serializable schedule is view serializable, but not every view serializable schedule is conflict serializable.

An example of view serializable schedules is shown below-

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

S_2	
T_1	T_2
R(A) W(A) R(B) W(B)	
	R(A) W(A) R(B) W(B)

Non-Serializable Schedules-

A non-serial schedule which is not serializable is called as a non-serializable schedule.
A non-serializable schedule is not guaranteed to produce the the same effect as produced by some serial schedule on any consistent database.

Characteristics-

Non-serializable schedules-

may or may not be consistent
may or may not be recoverable

Recoverable Schedule

If in a schedule, a transaction performs a dirty read operation from an uncommitted transaction and its commit operation is delayed till the uncommitted transaction either commits or roll backs then such a schedule is known as a **Recoverable Schedule**.

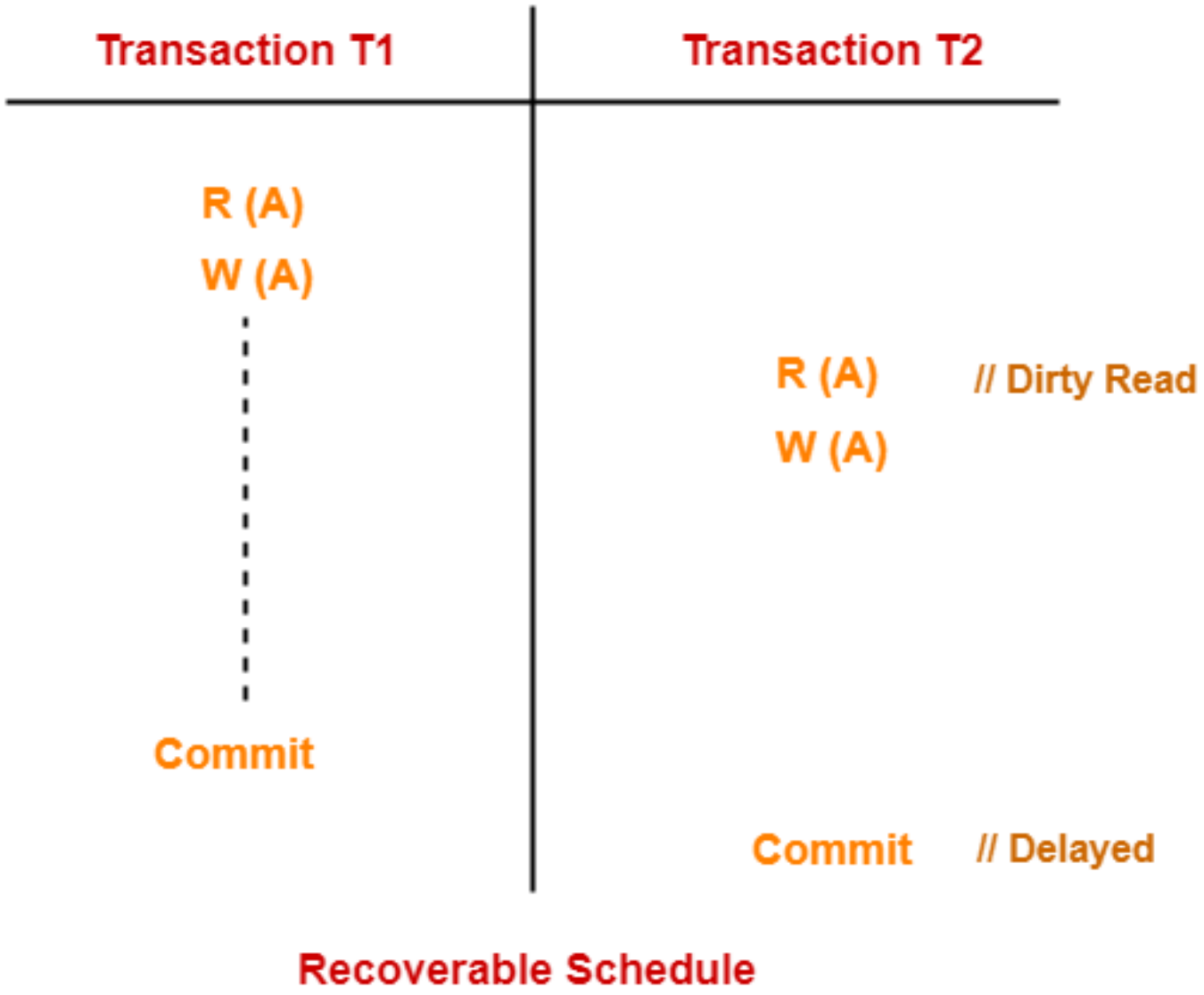
Here, The commit operation of the transaction that performs the dirty read is delayed. This ensures that it still has a chance to recover if the uncommitted transaction fails later.

Thumb Rules

- All conflict serializable schedules are recoverable.

- All recoverable schedules may or may not be conflict serializable.

Example

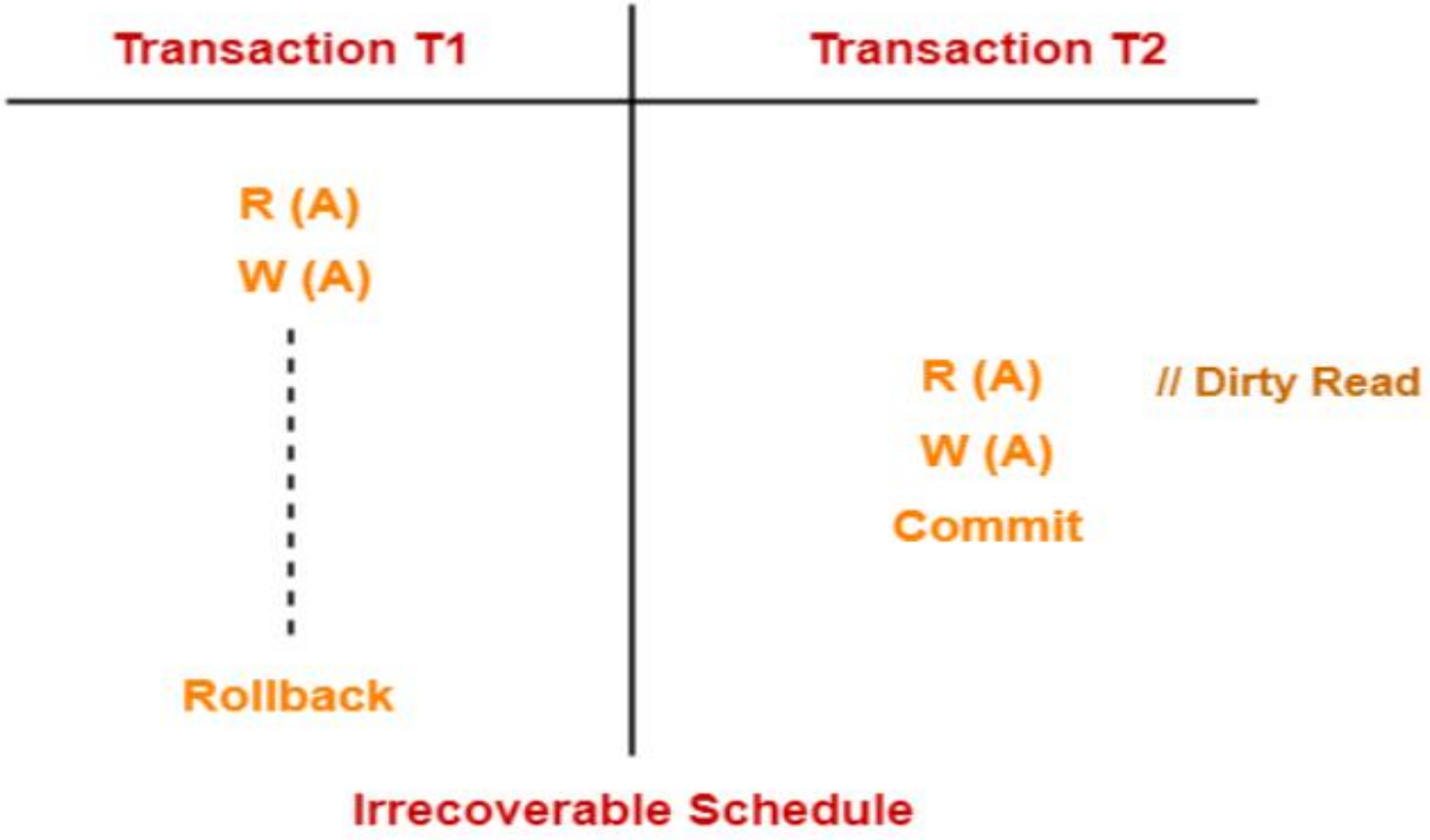


Recoverable Schedule

Irrecoverable Schedules-

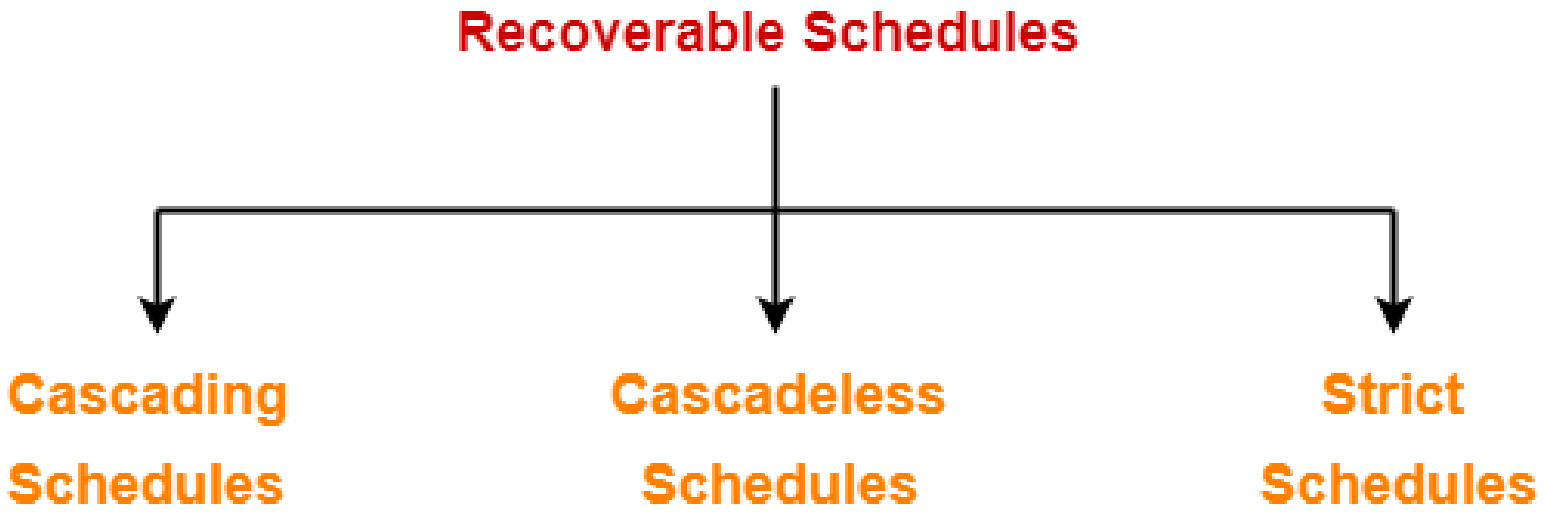
If in a schedule, A transaction performs a dirty read operation from an uncommitted transaction and commits before the transaction from which it has read the value then such a schedule is known as an **Irrecoverable Schedule**.

Example-



Types of Recoverable Schedules-

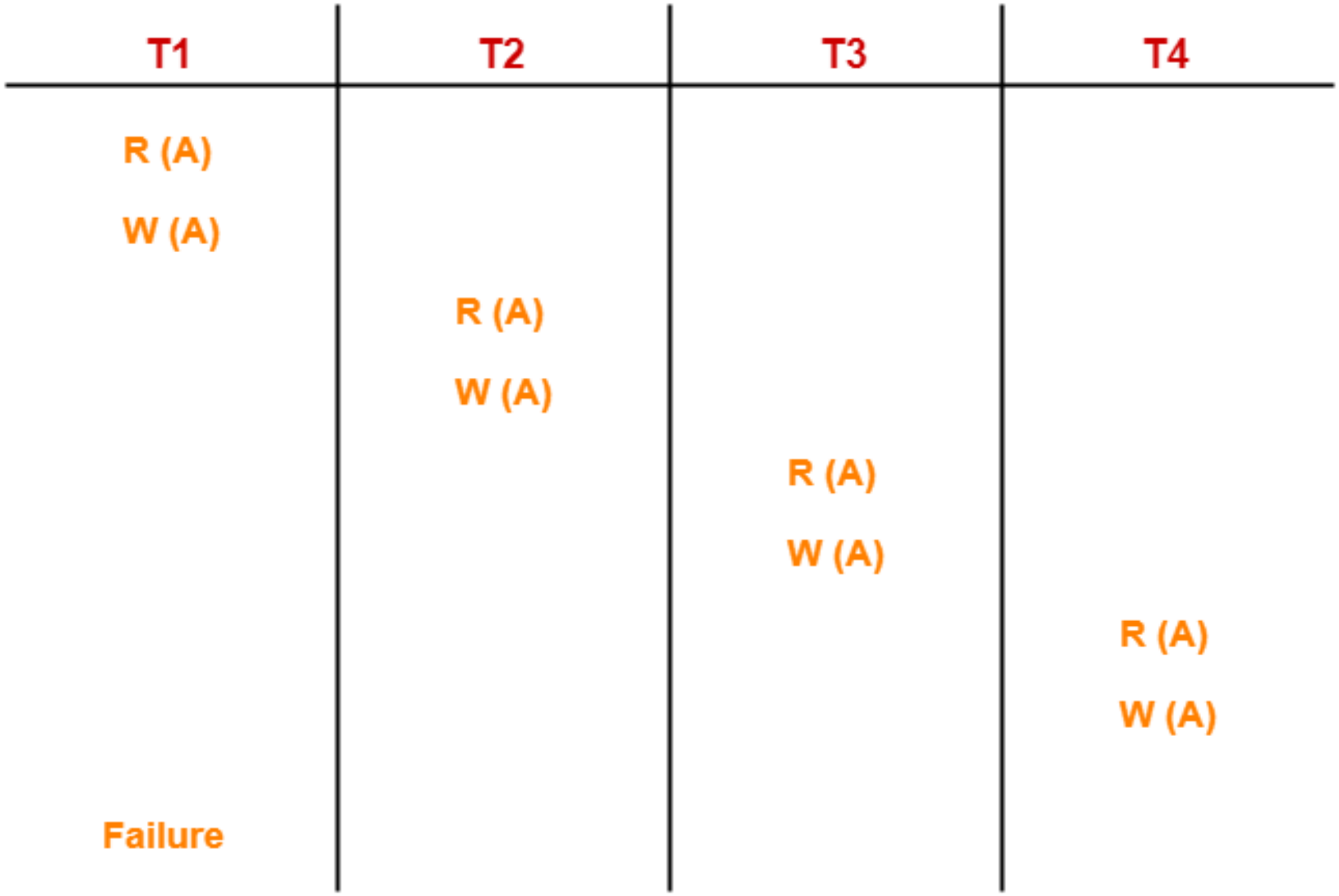
A recoverable schedule may be any one of these kinds-



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.



Cascading Recoverable Schedule

Here,
Transaction T2 depends on transaction T1.
Transaction T3 depends on transaction T2.
Transaction T4 depends on transaction T3.

In this schedule,
The failure of transaction T1 causes the transaction T2 to rollback.
The rollback of transaction T2 causes the transaction T3 to rollback.
The rollback of transaction T3 causes the transaction T4 to rollback.
Such a rollback is called as a **Cascading Rollback**.

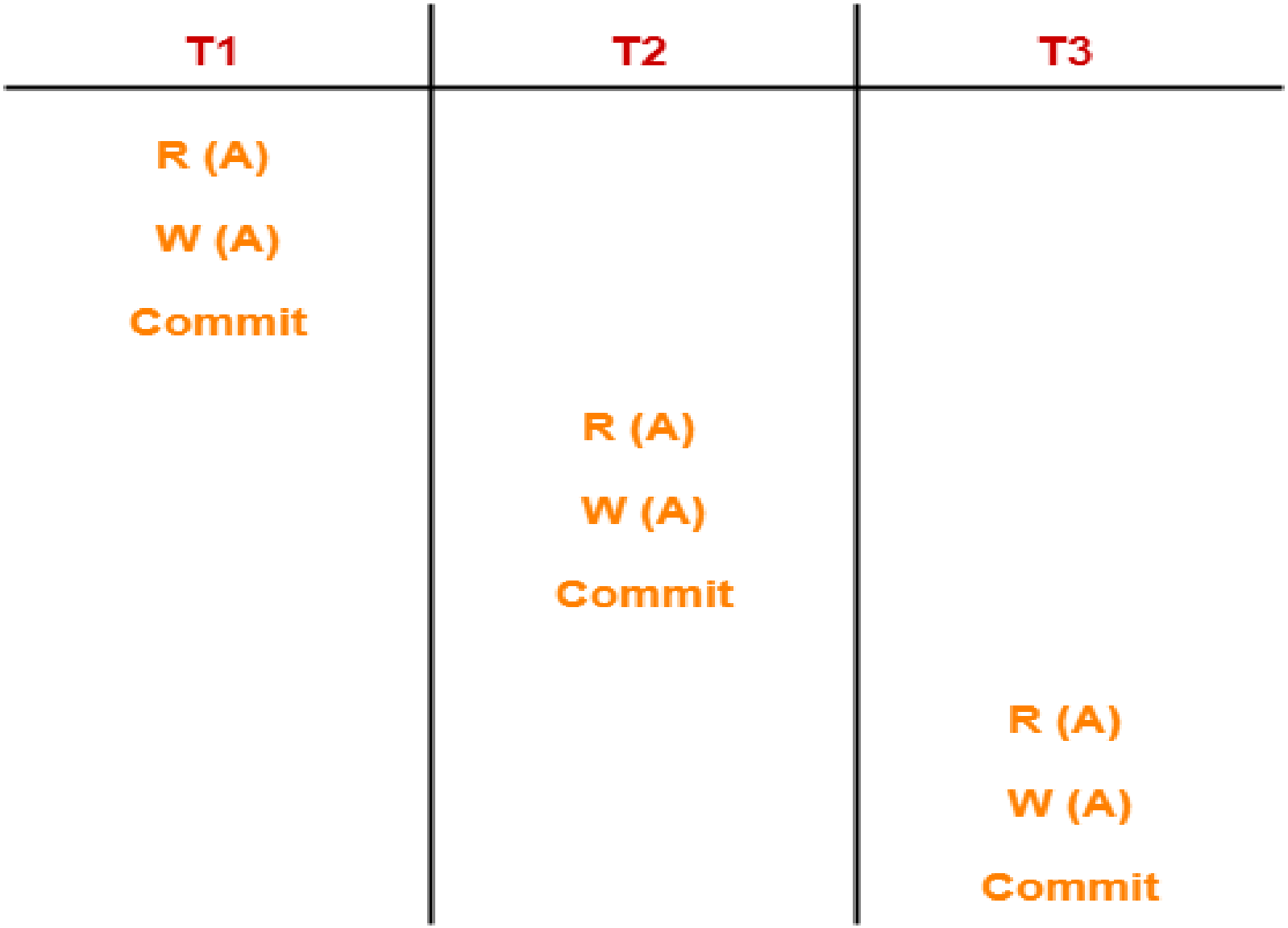
Cascadeless Schedule-

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.



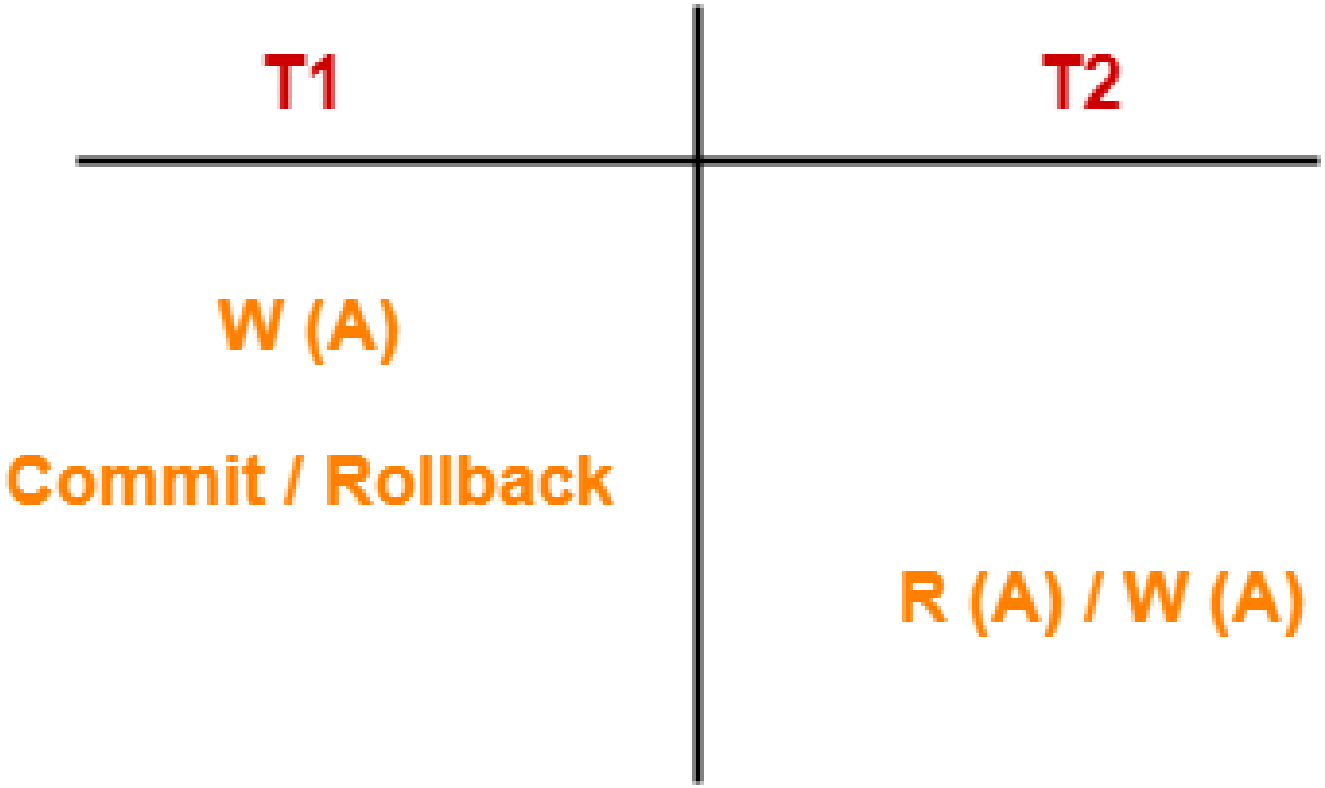
Cascadeless Schedule

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**.

In other words, Strict schedule allows only committed read and write operations.

Clearly, strict schedule implements more restrictions than cascadeless schedule.



Strict Schedule

Remember-

- Strict schedules are more strict than cascadeless schedules.
- All strict schedules are cascadeless schedules.
- All cascadeless schedules are not strict schedules.