Transactions

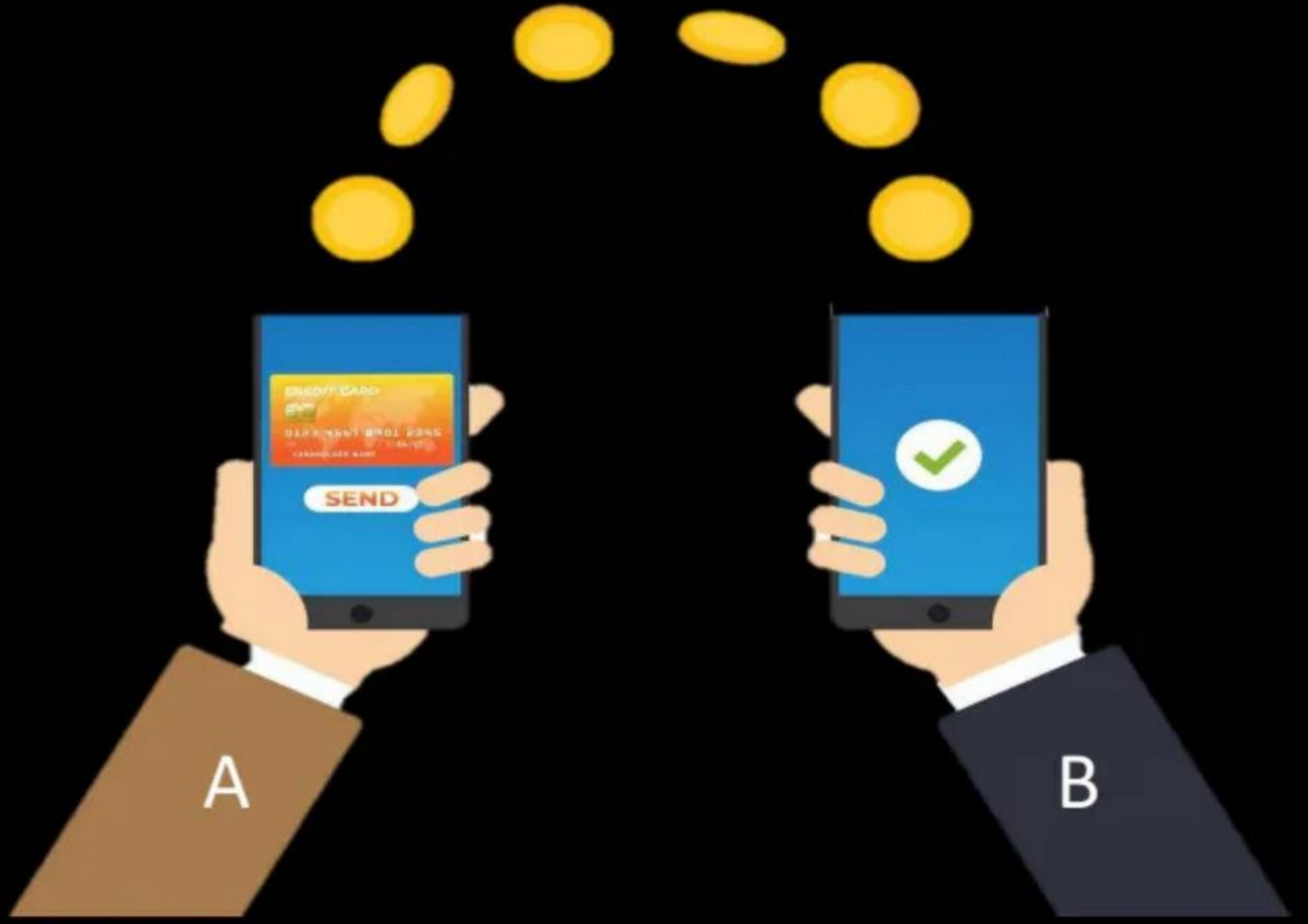
A transaction is a collection of operations
that forms a single logical unit of work

Understanding transactions from an example !

Initial A/C Balance of A = 100

Initial A/C Balance of B = 200

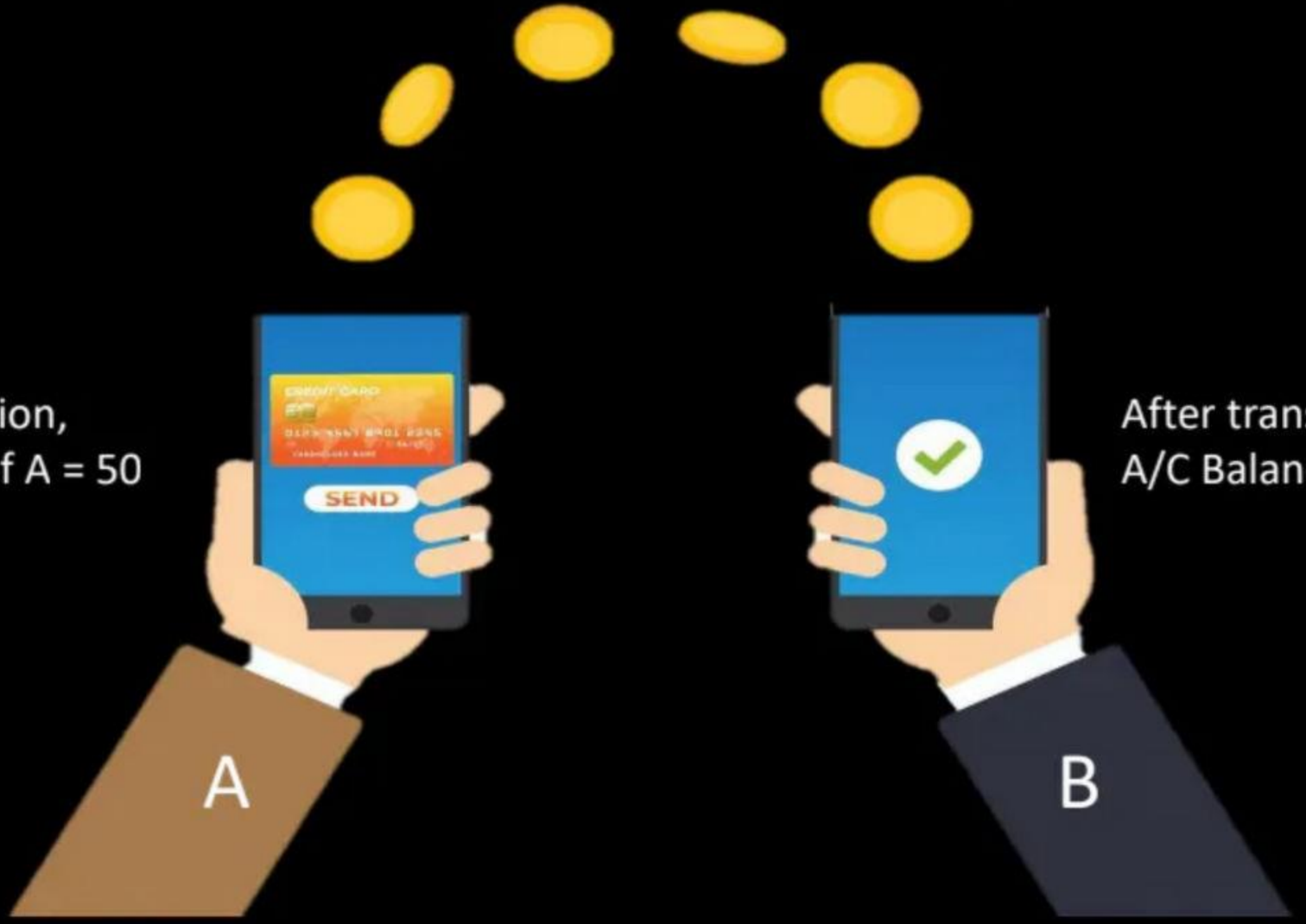
Suppose A wishes to transfer ₹50 to B



Initial A/C Balance of A = 100

Initial A/C Balance of B = 200

Suppose A wishes to transfer ₹50 to B

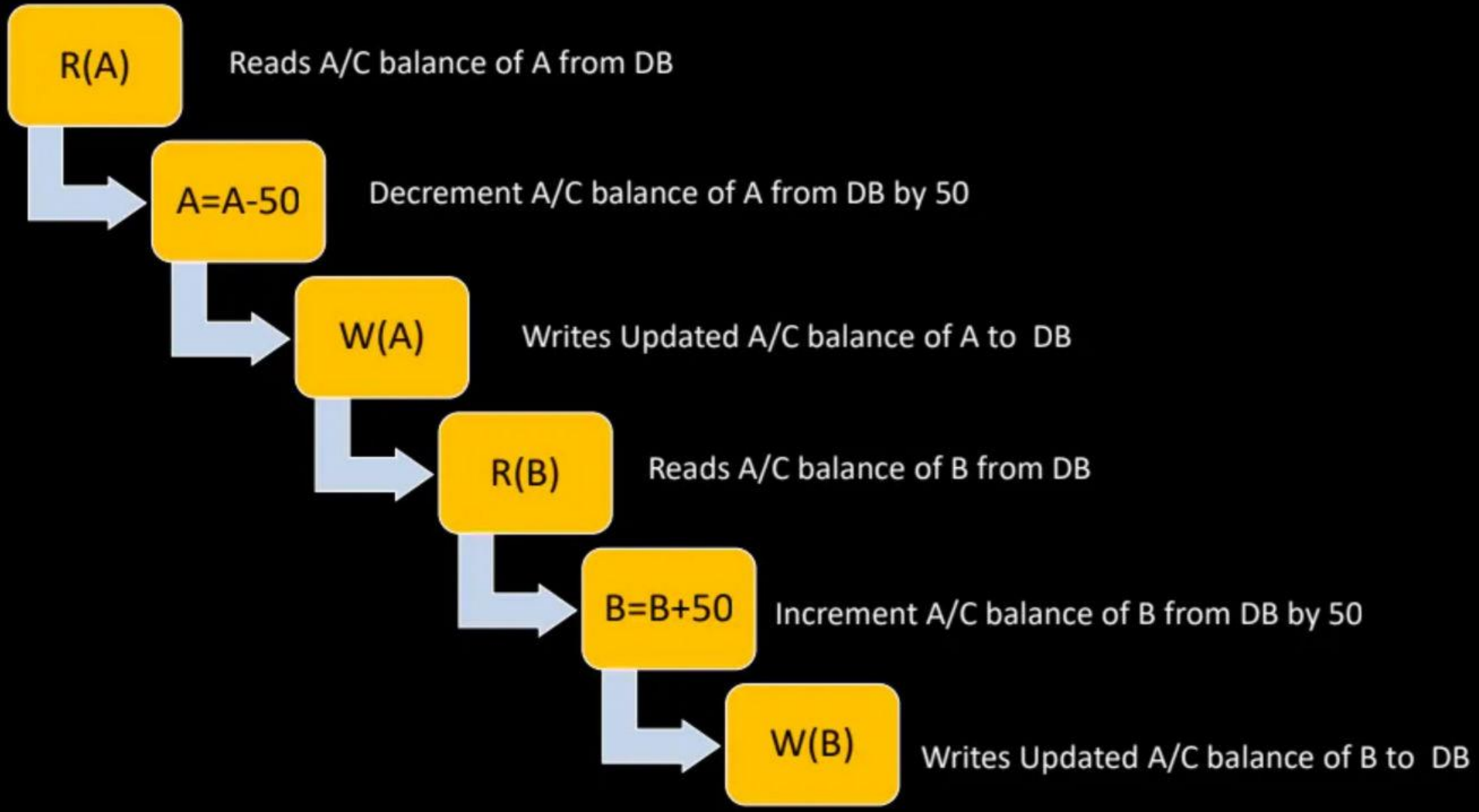


After transaction,
A/C Balance of A = 50

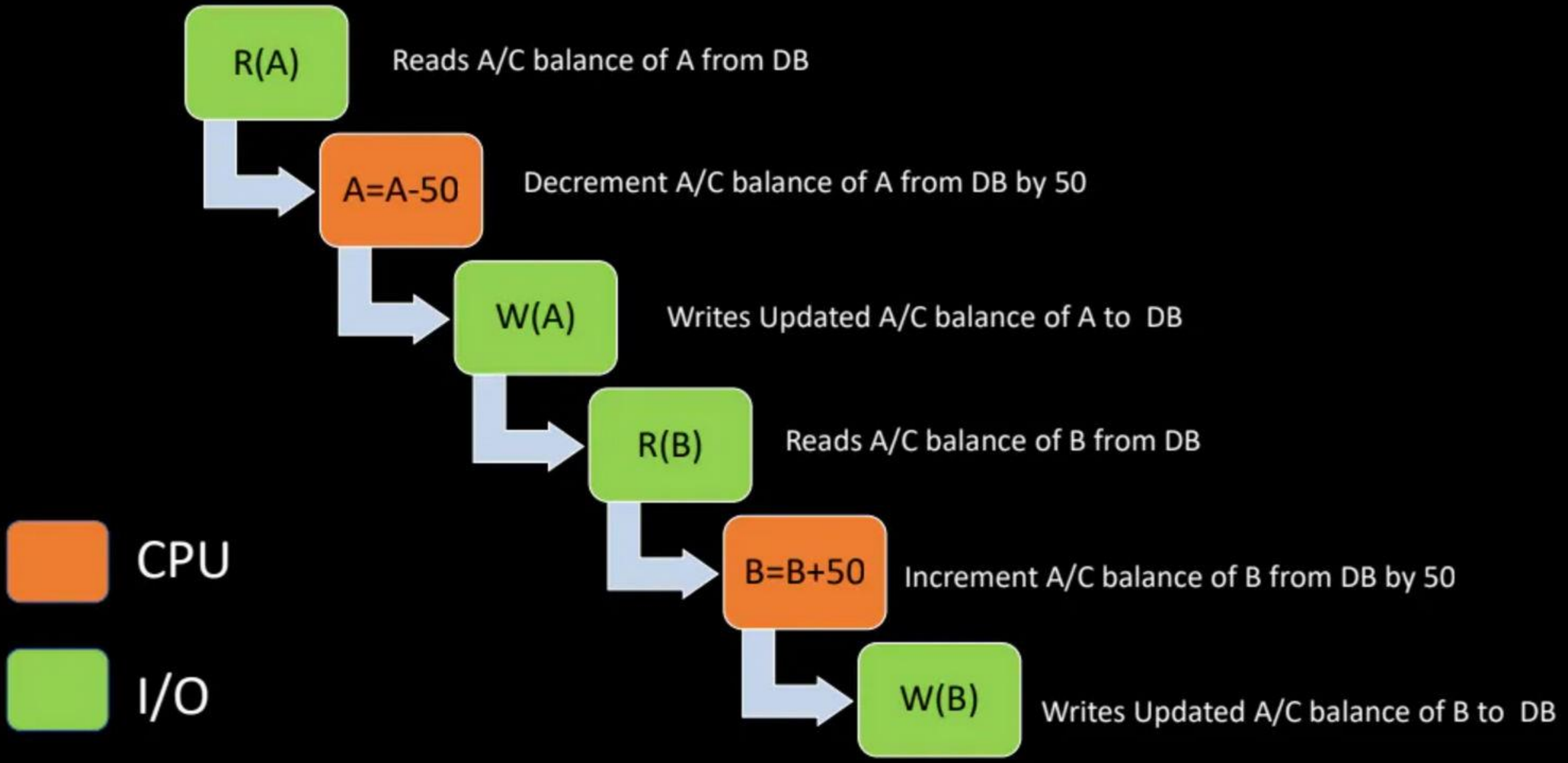
After transaction,
A/C Balance of B = 250

This looks like a simple
single logical unit of work right ?
But, there are many steps involved !

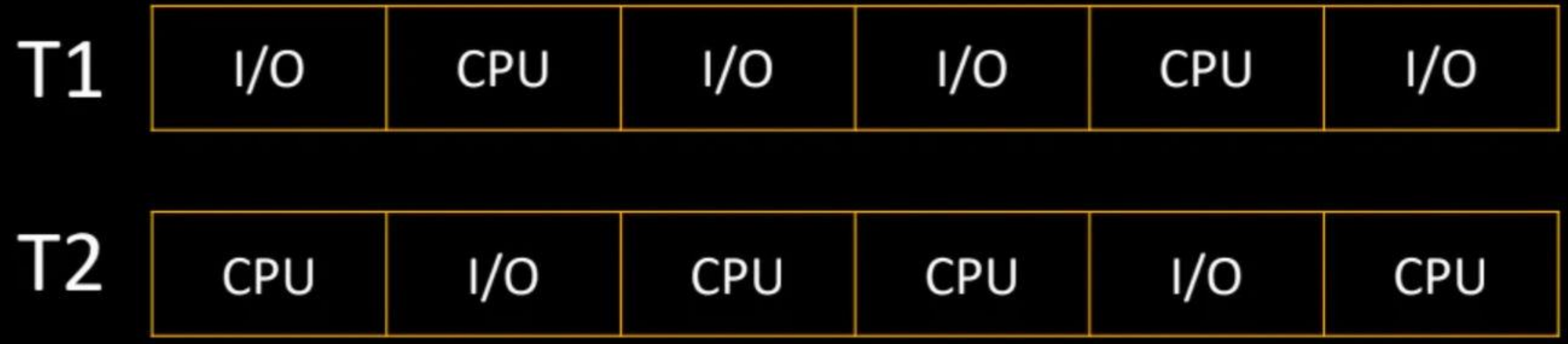
Now let us look at the various steps involved in this single transaction!



Certain steps require I/O Operations to be done
And Certain require CPU operations to be done



We know that if we have to increase efficiency
of a CPU it should be busy as much as possible by given
CPU to another process in idle time of executing process



There might be cases when
multiple transactions use same data,
This is where problem arises!
Let check an Example !

Suppose A= 100, T1 tries to decrement A by 50
and T2 decrements A by 25, the sequence of operations is shown below:

T1	T2
R(A)	
A=A-50	
	R(A)
	A=A-25
W(A)	
	W(A)

Lets see what happened here…….

Suppose A= 100, T1 tries to decrement A by 50
and T2 decrements A by 25, the sequence of operations is shown below:

T1	T2
R(A) //100	
A=A-50 //50	
	R(A) //100
	A=A-25 //75
W(A) //50	
	W(A) //75

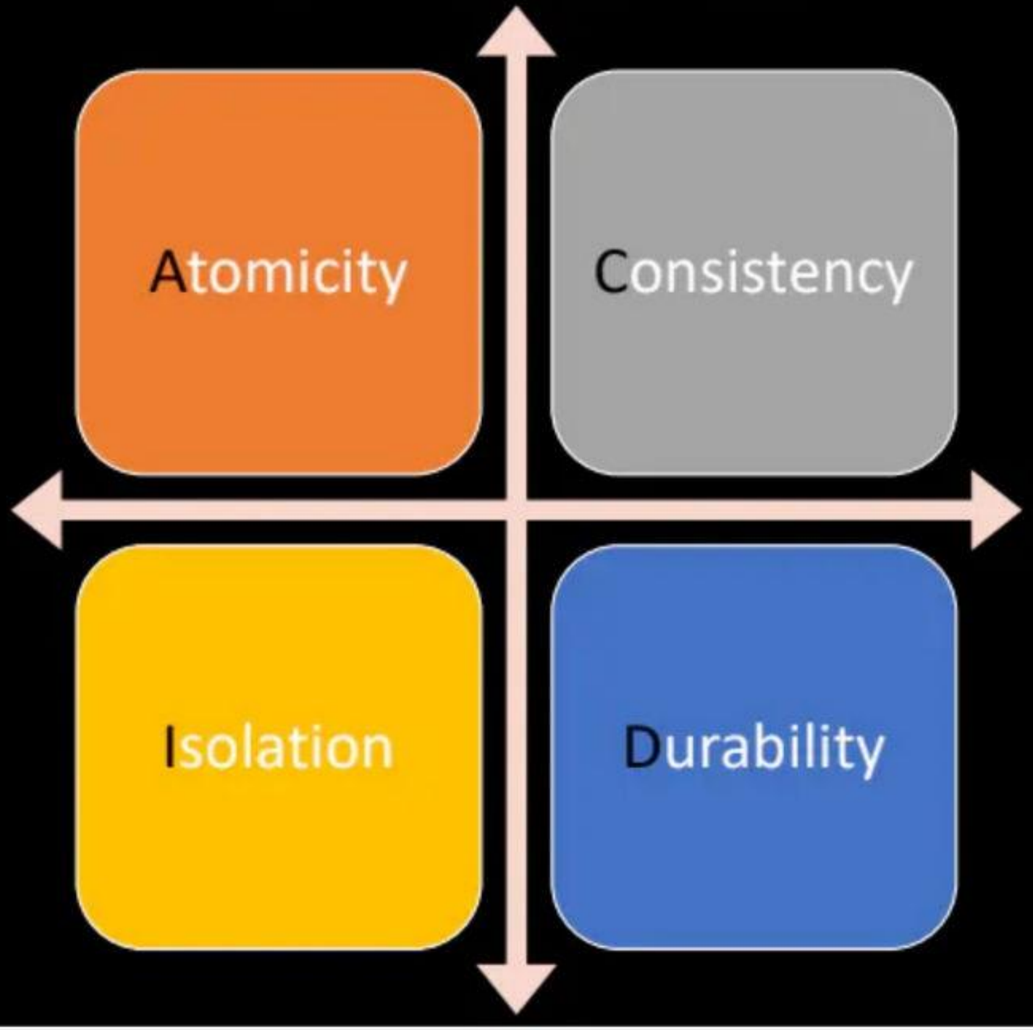
That's WRONG !

Now, you may think that first we can execute T1 and then T2 or T2 and then T1,
The answer would have been right !
But What about efficiency ???
We need a consistent system for which we need Transaction Management and Concurrency Control !

ACID Properties

To ensure the consistency of database, certain properties are followed by all the transactions occurring in the system.

ACID PROPERTIES



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.

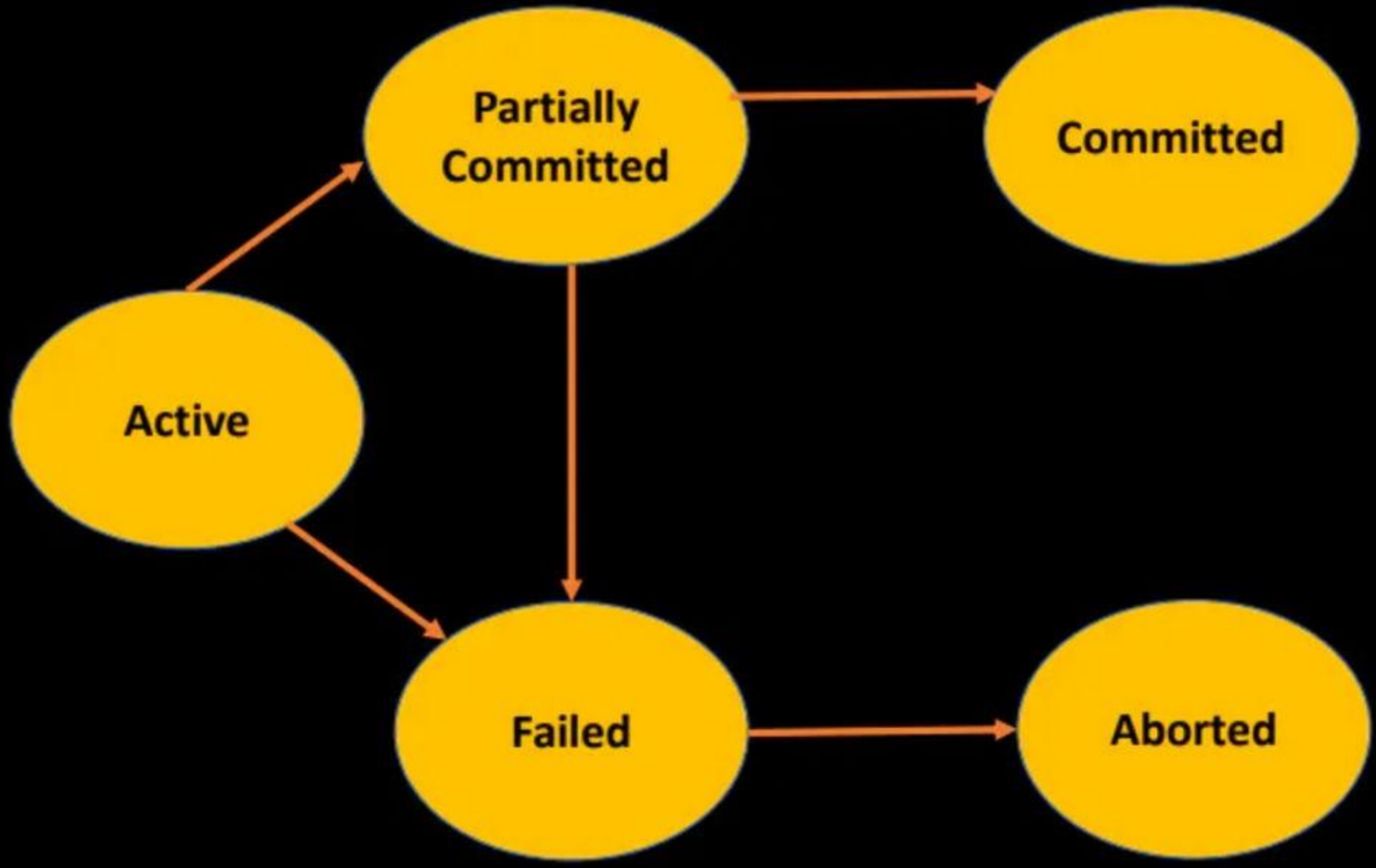
3. Isolation-

- 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 parallel.
- 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.

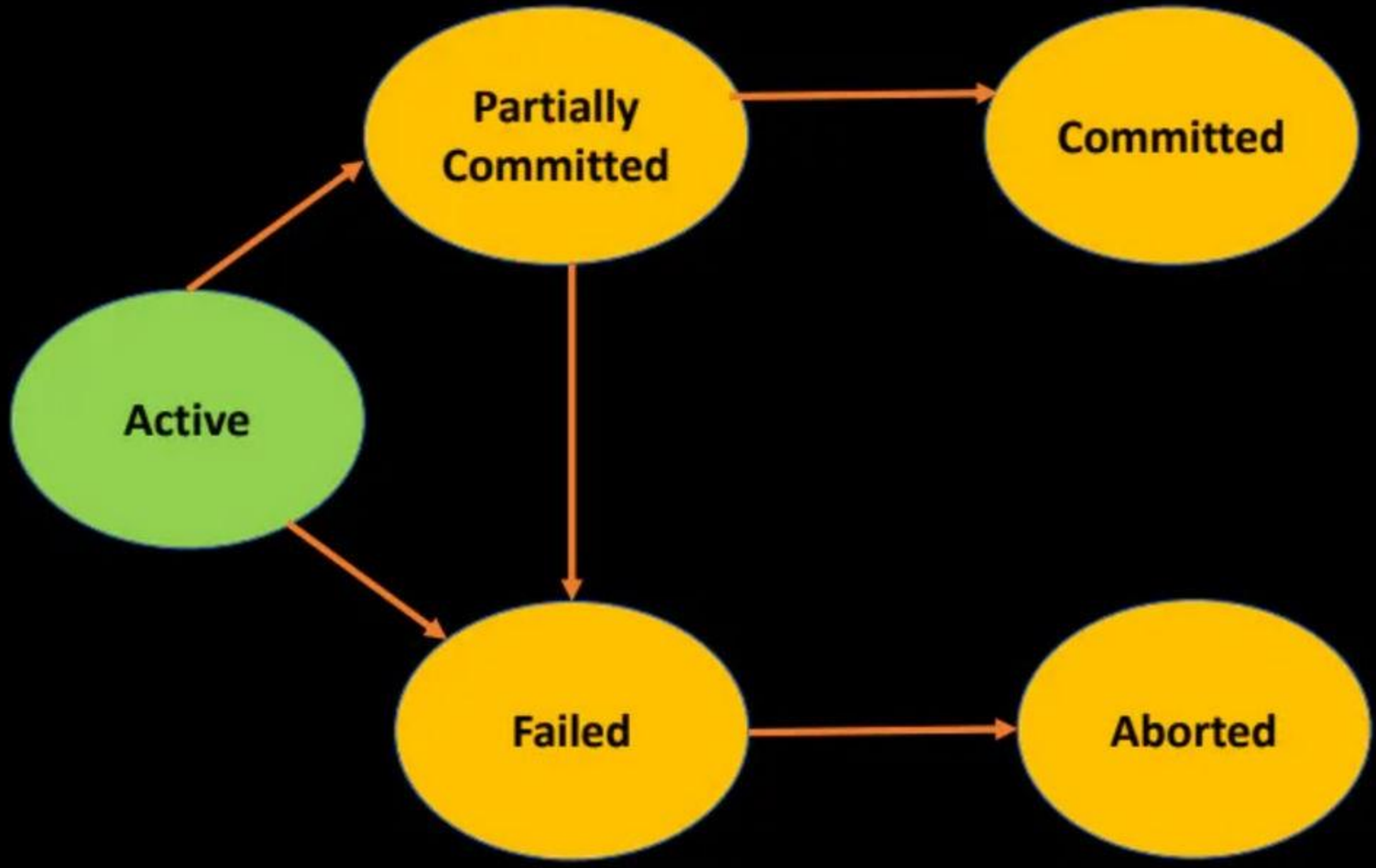
4. Durability-

- 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.

States of Transaction

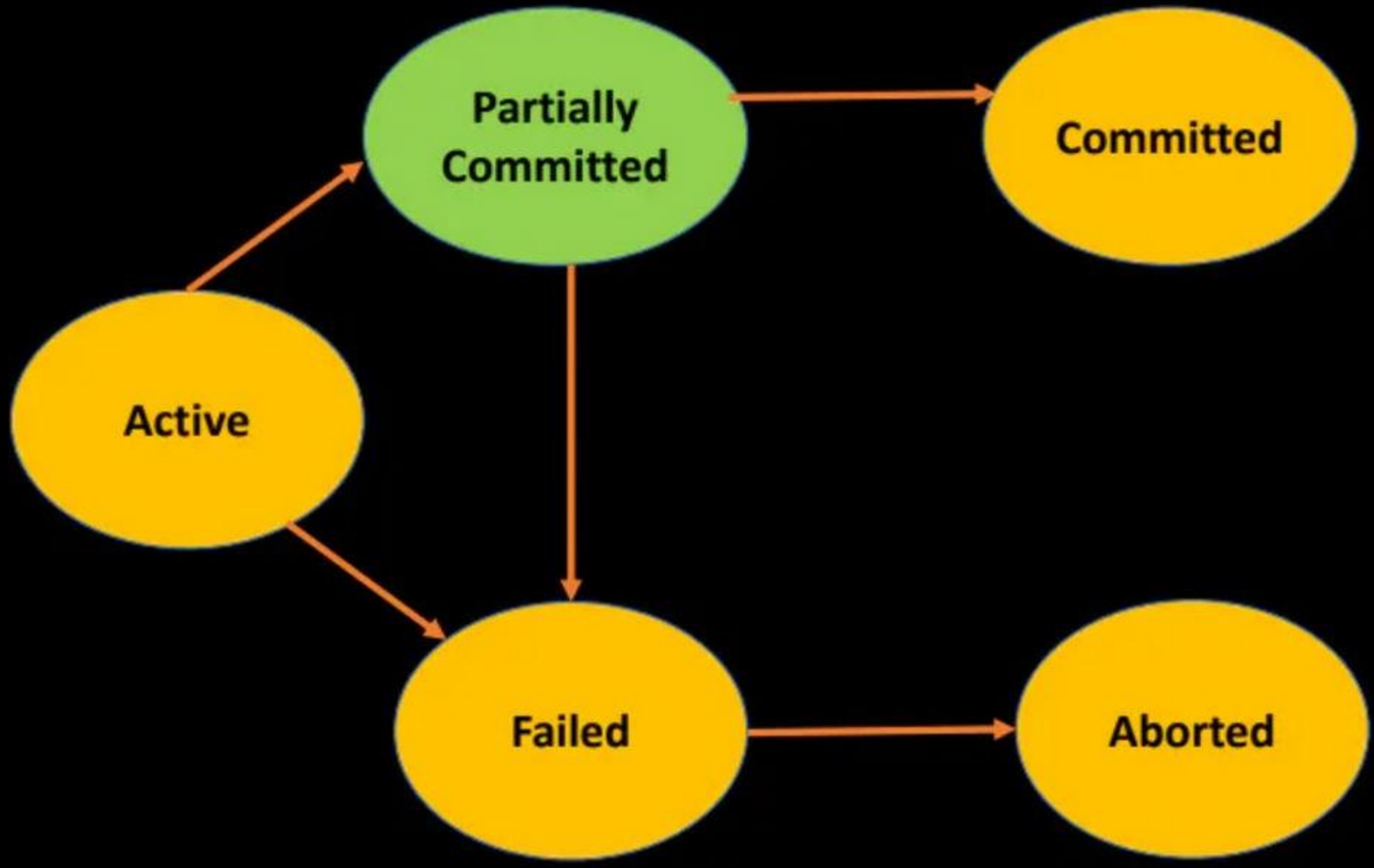


States of Transaction



Active state

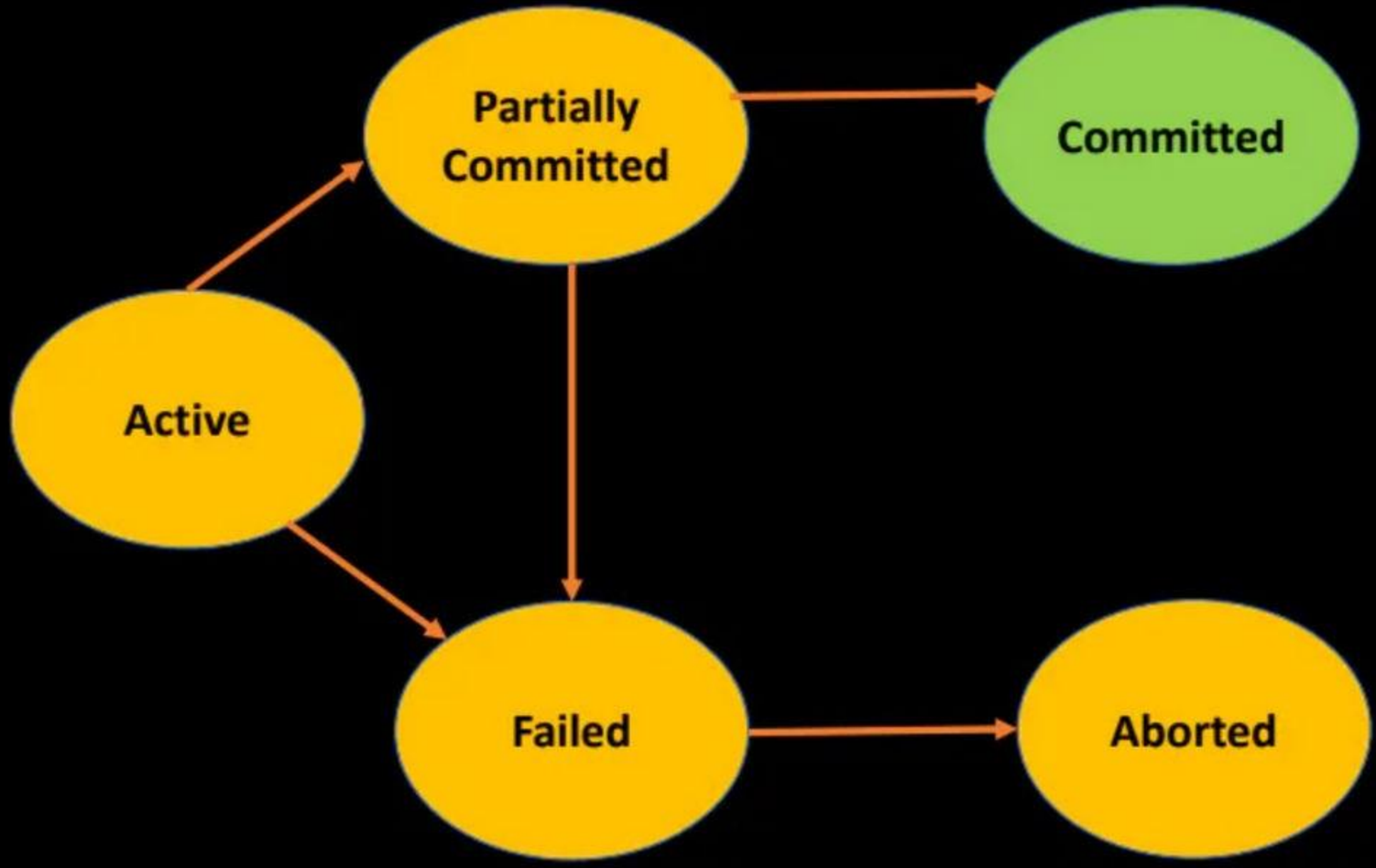
- The active state is the first state of every transaction. In this state, the transaction is being executed.
- For example: Insertion or deletion or updating a record is done here. But all the records are still not saved to the database.



Partially committed

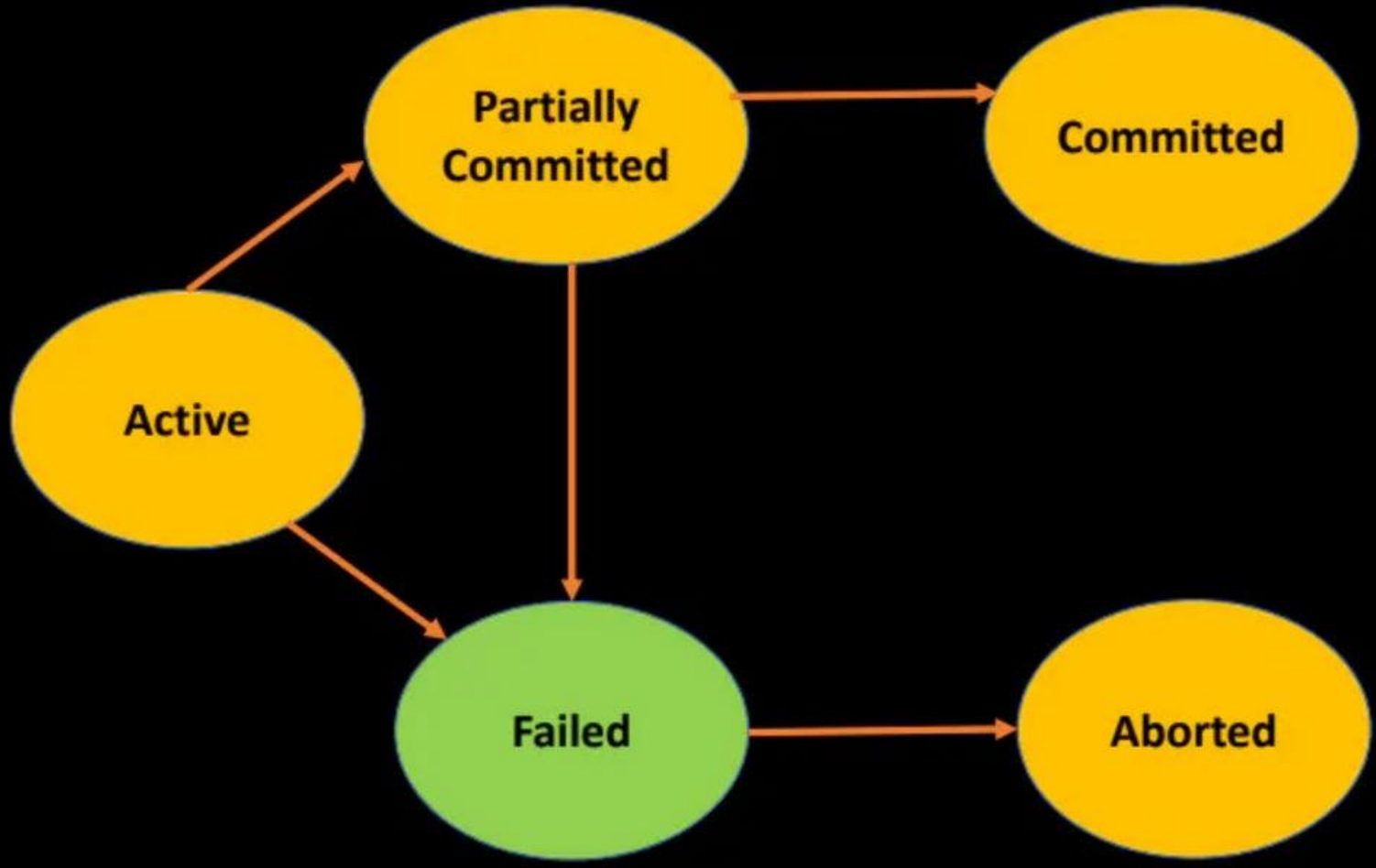
•In the partially committed state, a transaction executes its final operation, but the data is still not saved to the database.

EX. In the total mark calculation, a final display of the total marks step is executed in this state.



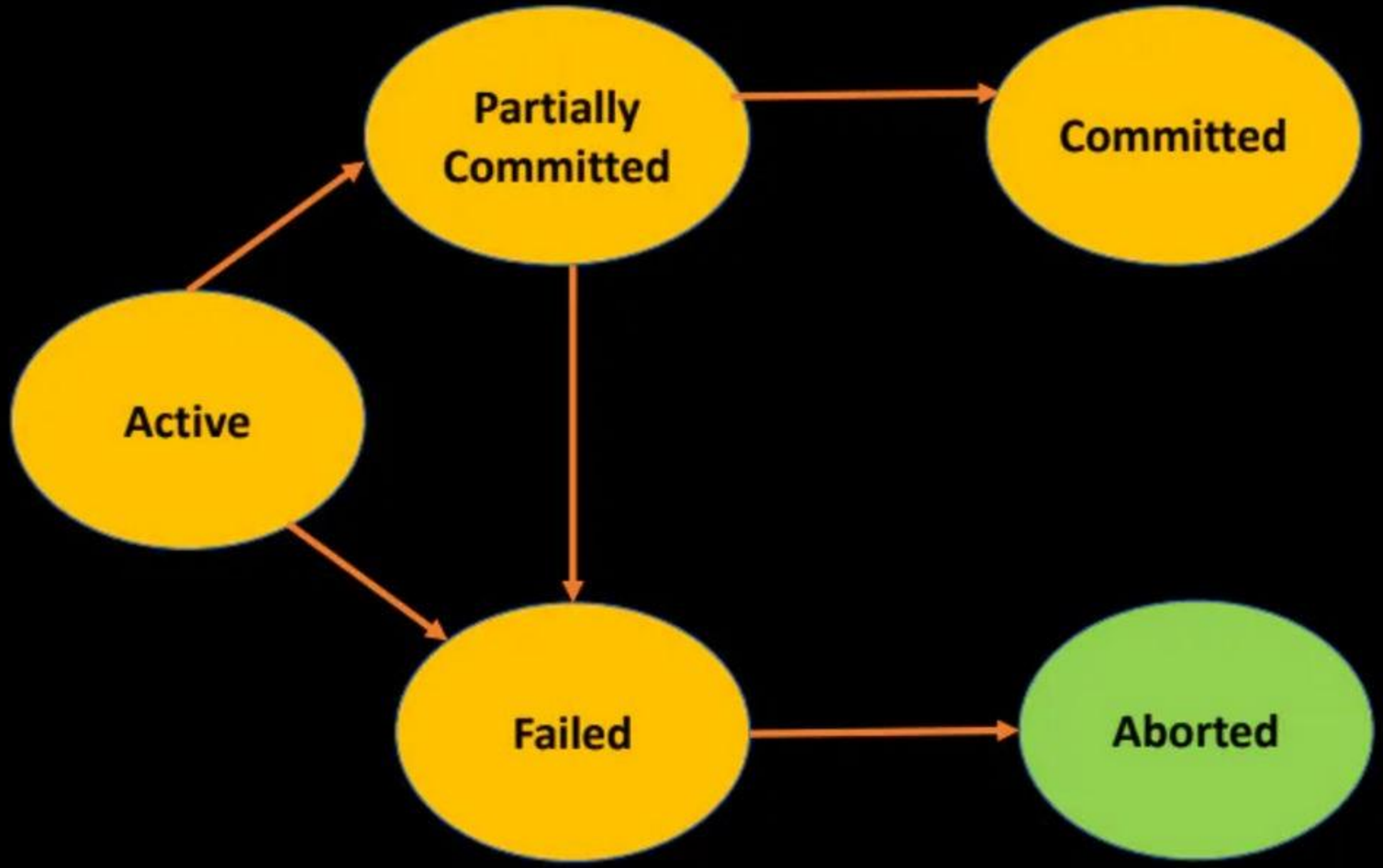
Committed

A transaction is said to be in a committed state if it executes all its operations successfully. In this state, all the effects are now permanently saved on the database system.



Failed state

- If any of the checks made by the database recovery system fails, then the transaction is said to be in the failed state.
- In the example of total mark calculation, if the database is not able to fire a query to fetch the marks, then the transaction will fail to execute.



Aborted

•If any of the checks fail and the transaction has reached a failed state then the database recovery system will make sure that the database is in its previous consistent state. If not then it will abort or roll back the transaction to bring the database into a consistent state.

•If the transaction fails in the middle of the transaction then before executing the transaction, all the executed transactions are rolled back to its consistent state.

•After aborting the transaction, the database recovery module will select one of the two operations:

1. Re-start the transaction 2. Kill the transaction

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule in which T_1 is followed by T_2 :

T_1	T_2
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit

Schedule 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

- In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of ($A + B$).

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Problems with Concurrent Execution

1.) Lost Update Problem (W-W conflict)

In the lost update problem, update done to a data item by a transaction is lost as it is overwritten by the update done by another transaction.

Example:

A= 100

T1 (A sending Rs.50 to B)	T2 (A is incrementing its balance by 4%)
R(A)	
A=A-50	
	R(A)
	X=A*0.04
	A=A+X
W(A)	
	W(A)
R(B)	
B=B+50	
W(B)	

1.) Lost Update Problem (W-W conflict)

In the lost update problem, update done to a data item by a transaction is lost as it is overwritten by the update done by another transaction.

Example:

T1 (A sending Rs.50 to B)	T2 (A is incrementing its balance by 4%)
R(A) //100	
A=A-50 //50	
	R(A) //100
	X=A*0.04
	A=A+X //104
W(A) //50	
	W(A) //104
R(B)	
B=B+50	
W(B)	

A= ~~100~~
~~50~~
104

1.) Lost Update Problem (W-W conflict)

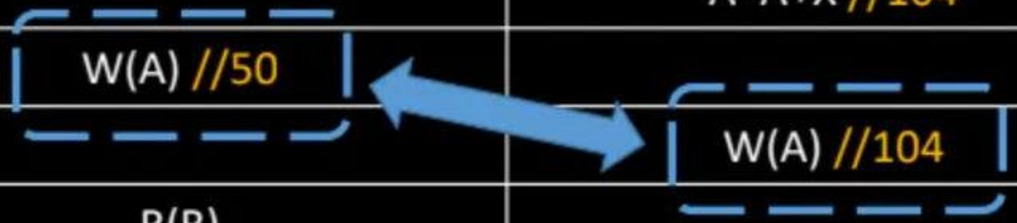
In the lost update problem, update done to a data item by a transaction is lost as it is overwritten by the update done by another transaction.

Example:

A= ~~100~~
~~50~~
104

T1 (A sending Rs.50 to B)	T2 (A is incrementing its balance by 4%)
R(A) //100	
A=A-50 //50	
	R(A) //100
	X=A*0.04
	A=A+X //104
W(A) //50	
	W(A) //104
R(B)	
B=B+50	
W(B)	

T1 (A sending Rs.50 to B)	T2 (A is incrementing its balance by 4%)
R(A) //100	
A=A-50 //50	
	R(A) //100
	X=A*0.04
	A=A+X //104
W(A) //50	
	W(A) //104
R(B)	
B=B+50	
W(B)	



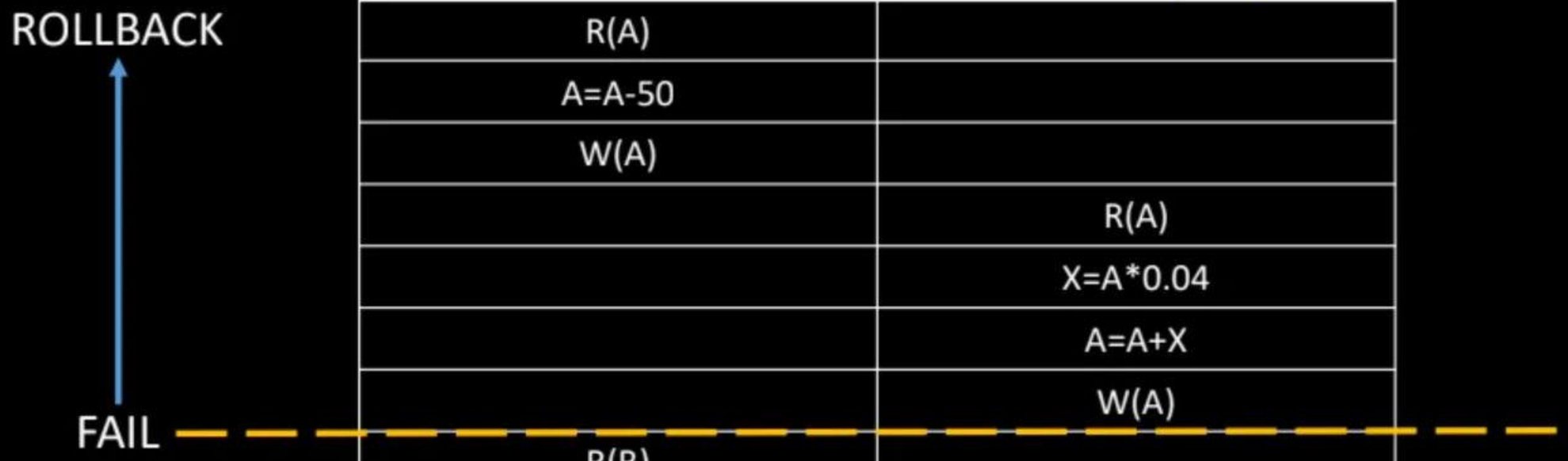
In write-write conflict, there are two writes one by each transaction on the same data item without any read in the middle.

COMMIT in DBMS is a transaction control language which is used to permanently save the changes done in the transaction in tables/databases.

ROLLBACK in DBMS is used to undo the transactions that have not been saved in database.

2.) Dirty Read Problem (W-R conflict)

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



T1 (A sending Rs.50 to B)	T2 (A is incrementing its balance by 4%)
R(A)	
A=A-50	
W(A)	
	R(A)
	X=A*0.04
	A=A+X
	W(A)
R(B)	
B+50	
W(B)	

2.) Dirty Read Problem (W-R conflict)

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

T1 (A sending Rs.50 to B)	T2 (A is incrementing its balance by 4%)
R(A) //100	
A=A-50 //50	
W(A) //50	
	R(A) //50
	X=A*0.04
	A=A+X // 52
	W(A) //52
R(B)	
B+50	
W(B)	

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.

3.) Unrepeatable Read Problem / R-W Conflict -

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.

T1	T2
R(A)	
	R(A)
	A=A-15000
	W(A)
R(A)	
A=A-10000	
W(A)	

3.) Unrepeatable Read Problem / R-W Conflict -

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.

A=20000

T1	T2
R(A) //20000	
	R(A) //20000
	A=A-15000 //5000
	W(A)
R(A) //5000	
A=A-10000	
W(A)	

In this example,

- T1 gets to read a different value of A in its second reading.
- T1 wonders how the value of A got changed because according to it, it is running in isolation.

4.) Phantom Tuple

T1	T2

E.NO	ENAME	SAL
1	A	5000
3	C	4000

SELECT *from Emp where SAL >=3000;

insert into Emp values(4,D,3500);

E.NO	ENAME	SAL
1	A	5000
3	C	4000
4	D	3500

SELECT *from Emp where SAL >=3000

SELECT is executed twice, but returns an additional row the second time that was not returned the first time, the row is a "**phantom**" tuple.

5.) Incorrect Summary Problem

Incorrect Summary issue occurs when one transaction takes summary over the value of all the instances of a repeated data-item, and second transaction update few instances of that specific data-item. In that situation, the resulting summary does not reflect a correct result.

5.) Incorrect Summary Problem

Lets say we wish to sum up 3 values – K =50, X=100, Y=200

T1	T2
	SUM=0
	R(K)
	SUM=SUM+K
R(X)	
X=X+500	
W(X)	
	R(X)
	SUM=SUM+X
	R(Y)
R(Y)	
Y=Y+200	
W(Y)	
	R(Y)
	SUM=SUM+Y

K = 50
X = 600
Y = 400
SUM =1050

Now, suppose
T1 tries to increase
Value of Y by 200
Then Summary will
Be different i.e. 1250

Schedule

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

Suppose there are two transactions T1 and T2 that have to perform operations as follows:

T1 = a1a2

T2 = b1b2

The various ways in which the operations can be executed is known as schedule, which are:

S1

T1	T2
a1	
a2	
	b1
	b2

S2

T1	T2
	b1
	b2
a1	
a2	

S3

T1	T2
a1	
	b1
	b2
a2	

S4

T1	T2
	b1
a1	
a2	
	b2

S5

T1	T2
a1	
	b1
a2	
	b2

S6

T1	T2
	b1
a1	
	b2
a2	

Suppose there are two transactions T1 and T2 that have to perform operations as follows:
T1 = a1a2
T2 = b1b2

The various ways in which the operations can be executed is known as schedule, which are:

S1

T1	T2
a1	
a2	
	b1
	b2

S2

T1	T2
	b1
	b2
a1	
a2	

S3

T1	T2
a1	
	b1
	b2
a2	

S4

T1	T2
	b1
a1	
a2	
	b2

S5

T1	T2
a1	
	b1
a2	
	b2

S6

T1	T2
	b1
a1	
	b2
a2	

Out of all possible schedules
S1 and S2 will execute without any problems(we discussed before), since they are 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.

In non-serial schedules,
- Multiple transactions execute concurrently.
- Operations of all the transactions are inter leaved or mixed with each other.

T1 = n operations
T2 = m operations
Then total schedules possible = (n+m)! /n!m!

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

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

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

1.How many total number of schedules are possible?

2.How many total number of serial schedules are possible?

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

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

1.How many total number of schedules are possible?

2.How many total number of serial schedules are possible?

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

Total Number of Schedules

= (2+3+4)! / 2! x 3! x 4! = 1260

Total number of serial schedules

= Number of different ways of arranging 3 transactions

= 3!

= 6

Total number of non-serial schedules

= Total number of schedules – Total number of serial schedules

= 1260 – 6

= 1254

Note:

Serial schedules are always consistent.

Non-serial schedules are not always consistent.

Types of Schedules

1.) Serial Schedule-

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

Characteristics-

Serial schedules are always-
- Consistent
- Recoverable
- Cascadeless
- Strict

We will see this later

2.) Complete Schedule-

A schedule is said to be complete if the last operation of each transaction is either abort or commit.

Example:

T1	T2
R(A)	
	R(A)
A=A-50	
W(A)	
Commit	
	A=A+50
	W(A)
	Abort

3.) Irrecoverable and Recoverable Schedule-

Roll back

Fail

T1	T2
R(A)	
A=A-50	
W(A)	
	R(A)
	X=A*0.04
	A=A+X
	W(A)
	COMMIT
R(B)	
B=B+50	
W(B)	
COMMIT	

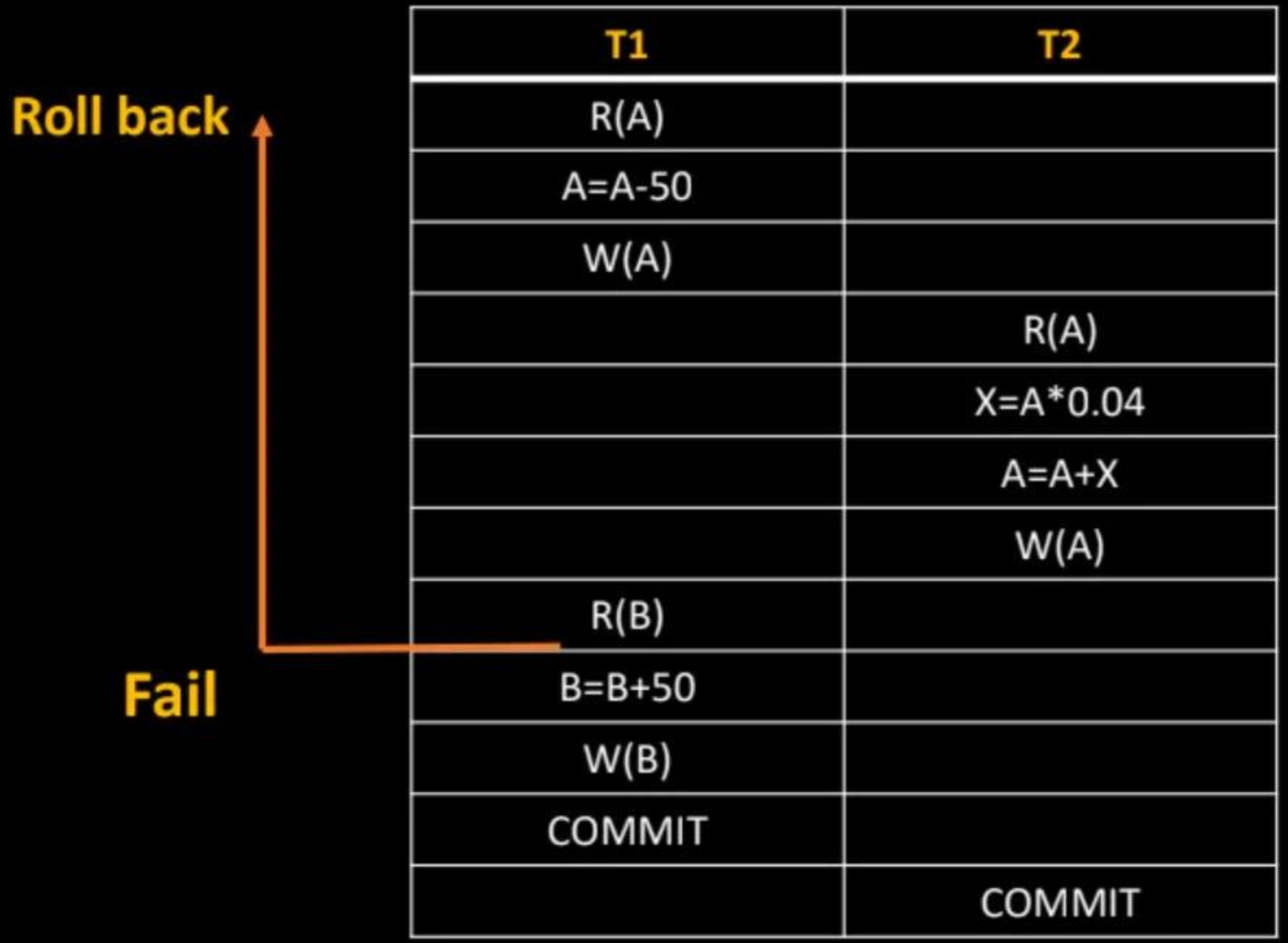
Irrecoverable

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**.

3.) Non Recoverable and Recoverable Schedule-



T1	T2
R(A)	
A=A-50	
W(A)	
	R(A)
	X=A*0.04
	A=A+X
	W(A)
R(B)	
B=B+50	
W(B)	
COMMIT	
	COMMIT

Recoverable

Recoverable Schedules

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.

Cascading Schedule / Cascading Rollback / Cascading Abort

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**.

Example:

ROLLBACK

FAIL

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

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**.

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

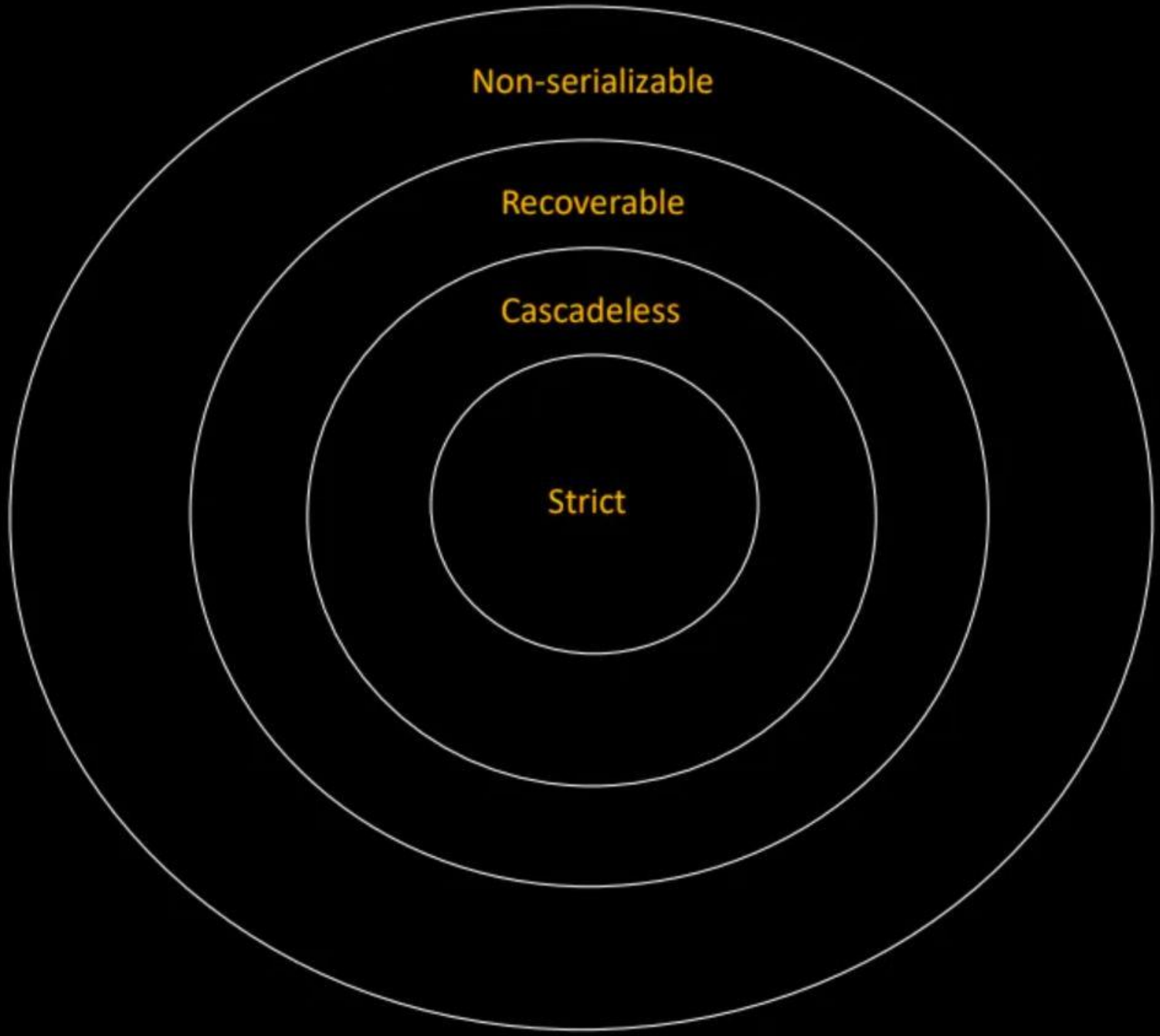
NOTE

- Cascadeless schedule allows only committed read operations.
- However, it allows uncommitted write operations.

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**.

T1	T2
R(A)	
W(A)	
COMMIT	
	W(A)


Non-serializable
Recoverable
Cascadeless
Strict

Serializable Schedule

A transaction schedule is serializable if its outcome is equal to the outcome of its transactions executed serially
i.e. Sequentially without overlapping in time

1.) Result equivalent Schedule
2.) Conflict Equivalent Schedule or Conflict Serializability
3.) View Equivalent Schedule or View Serializability

NOTE

Serial Schedules	Serializable Schedules
No concurrency is allowed. Thus, all the transactions necessarily execute serially one after the other.	Concurrency is allowed. Thus, multiple transactions can execute concurrently.

1.) Result equivalent Schedule

The two schedules S1 and S2 are said to be equivalent schedules if they produce the same final database state.

Q.) Which of the following schedules are result equivalent for the initial values X and Y (2,5) respectively.

T1	T2
R(X)	
X=X+5	
W(X)	
R(Y)	
Y=Y+5	
W(Y)	
	R(X)
	X=X*3
	W(X)

S1

T1	T2
R(X)	
X=X+5	
W(X)	
	R(X)
	X=X*3
	W(X)
R(Y)	
Y=Y+5	
W(Y)	

S2

T1	T2
	R(X)
	X=X*3
	W(X)
R(X)	
X=X+5	
W(X)	
R(Y)	
Y=Y+5	
W(Y)	

S3

Q.) Which of the following schedules are result equivalent for the initial values X and Y (2,5) respectively.

T1	T2
R(X) //2	
X=X+5 //7	
W(X)	
R(Y) //5	
Y=Y+5 //10	
W(Y) //10	
	R(X) //7
	X=X*3 //21
	W(X) //21

T1	T2
R(X) //2	
X=X+5 //7	
W(X) //7	
	R(X) //7
	X=X*3 //21
	W(X) //21
R(Y) //5	
Y=Y+5 //10	
W(Y) //10	

T1	T2
	R(X) //2
	X=X*3 //6
	W(X) //6
R(X) //6	
X=X+5 //11	
W(X) //11	
R(Y) //5	
Y=Y+5 //10	
W(Y) //10	

S1 ⟺ S2

X=21,Y=10 X=21,Y=10

S3

X=11,Y=10

2.) Conflict Equivalent Schedule or 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**.

Ti	Tj	
R(A)	W(A)	Conflicting Operations
W(A)	R(A)	
W(A)	W(A)	

R(A)	R(A)	Non- Conflicting Operations
Operation on different data		

S1

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

The order in which the conflicts are occurring,
$R_1(A) \rightarrow W_2(A)$
$W_1(A) \rightarrow W_2(A)$
$W_1(a) \rightarrow R_2(A)$

S2

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

The order in which the conflicts are occurring,
$R_1(A) \rightarrow W_2(A)$
$W_1(A) \rightarrow W_2(A)$
$W_1(a) \rightarrow R_2(A)$

Conflict Equivalent

Q1.) Test whether the following schedules are conflict equivalent or not.

S1 : R1(A) R2(B) W1(A) W2(B)

S2 : R2(B) R1(A) W2(B) W1(A)

Q1.) Test whether the following schedules are conflict equivalent or not.

S1 : R1(A) R2(B) W1(A) W2(B)

S2 : R2(B) R1(A) W2(B) W1(A)

S1

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

S2

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

Conflict equivalent

Q2.) Test whether the following schedules are conflict equivalent or not.

S1 : R1(A) W1(A) R2(A) W2(A) R1(B) W1(B) R2(B) W2(B)

S2 : R1(A) W1(A) R1(B) W1(B) R2(A) W2(A) R2(B) W2(B)

Q2.) Test whether the following schedules are conflict equivalent or not.

S1 : R1(A) W1(A) R2(A) W2(A) R1(B) W1(B) R2(B) W2(B)

S2 : R1(A) W1(A) R1(B) W1(B) R2(A) W2(A) R2(B) W2(B)

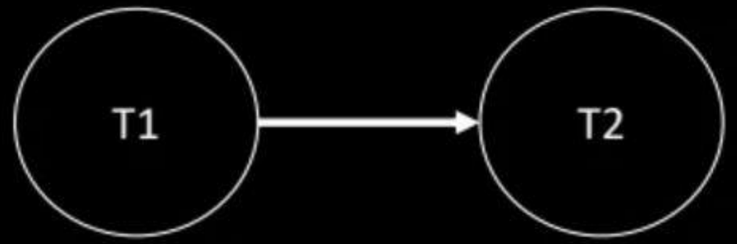
Answer : Conflict equivalent

Q3.) Test whether the following schedule is conflict serializable or not.

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

Q3.) Test whether the following schedule is conflict serializable or not.

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



Since no cycle is formed,
It is Conflict serializable.

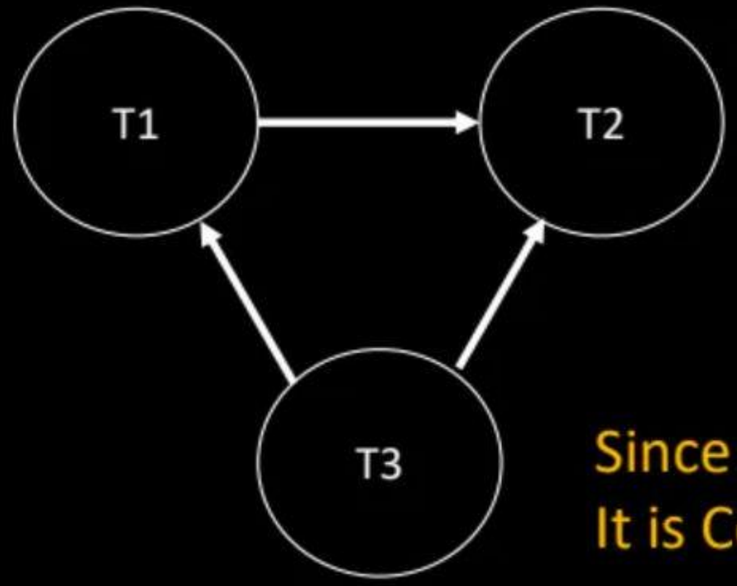
After performing topological ordering of the above precedence graph,
the possible serialized schedule = T1 T2

Q4.) Test whether the following schedule is conflict serializable or not.

T1	T2	T3
R(X)		
		R(X)
W(X)		
	R(X)	
	W(X)	

Q4.) Test whether the following schedule is conflict serializable or not.

T1	T2	T3
R(X)		
		R(X)
W(X)		
	R(X)	
	W(X)	



Since no cycle is formed,
It is Conflict serializable.

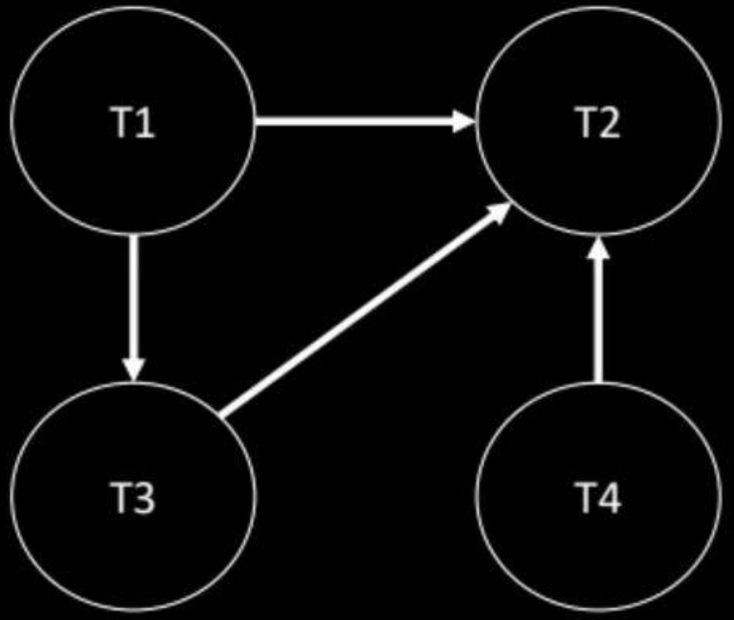
After performing topological ordering of the above precedence graph,
the possible serialized schedule = T3 T1 T2

Q5.) Test whether the following schedule is conflict serializable or not.

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

Q5.) Test whether the following schedule is conflict serializable or not.

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



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

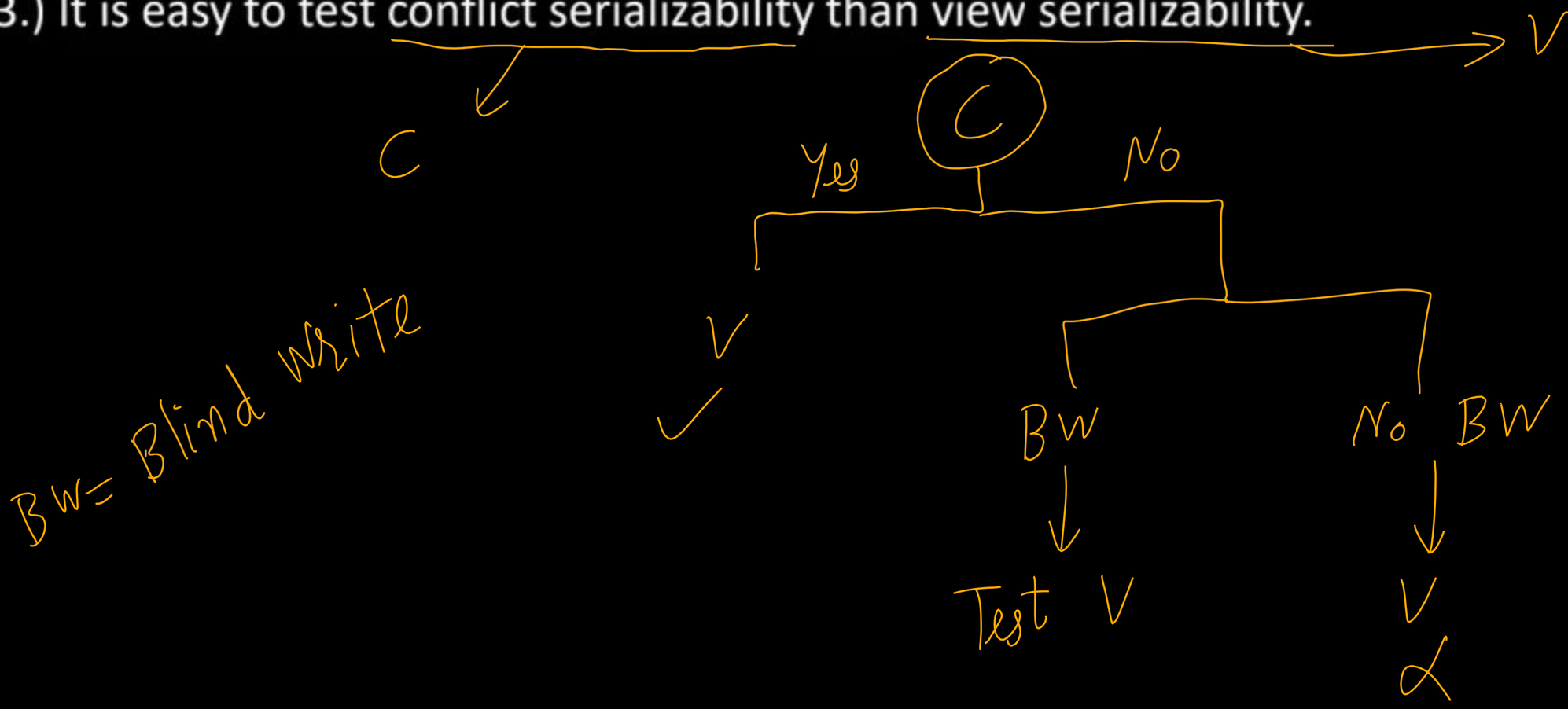
Since no cycle is formed, It is Conflict serializable.

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

1. $T_1 \rightarrow T_3 \rightarrow T_4 \rightarrow T_2$
2. $T_1 \rightarrow T_4 \rightarrow T_3 \rightarrow T_2$
3. $T_4 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2$

Comparison between conflict and view serializability:

1.) All conflict serializable schedules are also view serializable but reverse is not true
2.) It is easy to achieve conflict serializability than view serializability.
3.) It is easy to test conflict serializability than view serializability.



3.) View Serializability

A serial schedule and non serial schedule are said to be view equivalent if the satisfy all the following conditions:

Condition-1:
For each data item X, if transaction Ti reads X from the database initially in schedule S1, then in schedule S2 also, Ti must perform the initial read of X from the database.

Condition-2:
If transaction Ti reads a data item that has been updated by the transaction Tj in schedule S1, then in schedule S2 also, transaction Ti must read the same data item that has been updated by the transaction Tj.

Condition-3:
For each data item X, if X has been updated at last by transaction Ti in schedule S1, then in schedule S2 also, X must be updated at last by transaction Ti.

EXAMPLE: Check whether the two schedules are view equivalent or not.

T1	T2
R(A)	
A=A+10	
W(A)	
	R(A)
	A=A+10
	W(A)
R(B)	
B=B+20	
W(B)	
	R(B)
	B=B*1.1
	W(B)

T1	T2
R(A)	
A=A+10	
W(A)	
R(B)	
B=B+20	
W(B)	
	R(A)
	A=A+10
	W(A)
	R(B)
	B=B*1.1
	W(B)

EXAMPLE: Check whether the two schedules are view equivalent or not.

T1	T2
R(A)	
A=A+10	
W(A)	
	R(A)
	A=A+10
	W(A)
R(B)	
B=B+20	
W(B)	
	R(B)
	B=B*1.1
	W(B)

T1	T2
R(A)	
A=A+10	
W(A)	
R(B)	
B=B+20	
W(B)	
	R(A)
	A=A+10
	W(A)
	R(B)
	B=B*1.1
	W(B)

	S1	S2
A	T1, T2	T1, T2
B	T1, T2	T1, T2

Schedules S1 and S2 are view equivalent as all three conditions hold true.

S3

T1	T2
R(A)	
A=A+10	
	R(A)
	A=A+10
W(A)	
	W(A)
R(B)	
B=B+20	
	R(B)
	B=B*1.1
W(B)	
	W(B)

S1

T1	T2
R(A)	
A=A+10	
W(A)	
R(B)	
B=B+20	
W(B)	
	R(A)
	A=A+10
	W(A)
	R(B)
	B=B*1.1
	W(B)

S2

T1	T2
	R(A)
	A=A+10
	W(A)
	R(B)
	B=B*1.1
	W(B)
R(A)	
A=A+10	
W(A)	
R(B)	
B=B+20	
W(B)	

S3 is not view equivalent to S1 and S2

Comparison between conflict and view serializability:

1.) All conflict serializable schedules are also view serializable but reverse is not true
2.) It is easy to achieve conflict serializability than view serializability.
3.) It is easy to test conflict serializability than view serializability.

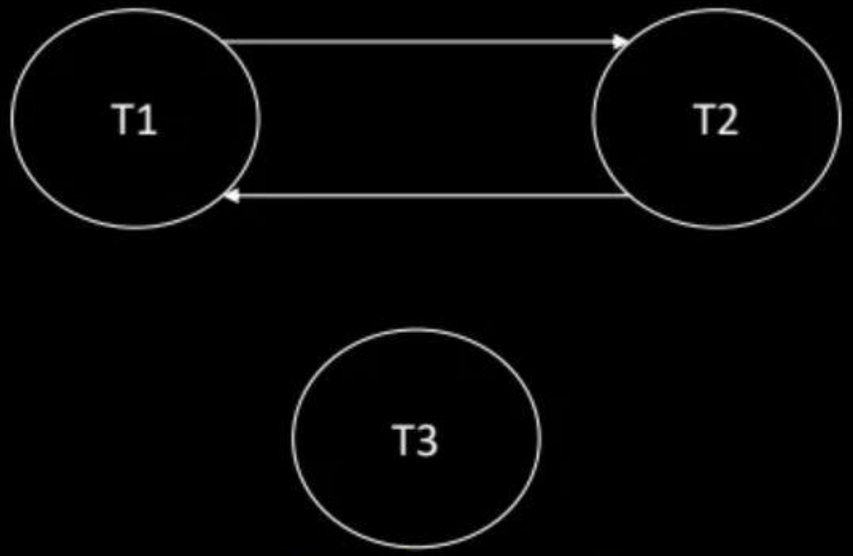
1.) Test the following schedule is view serializable

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

1.) Test the following schedule is view serializable

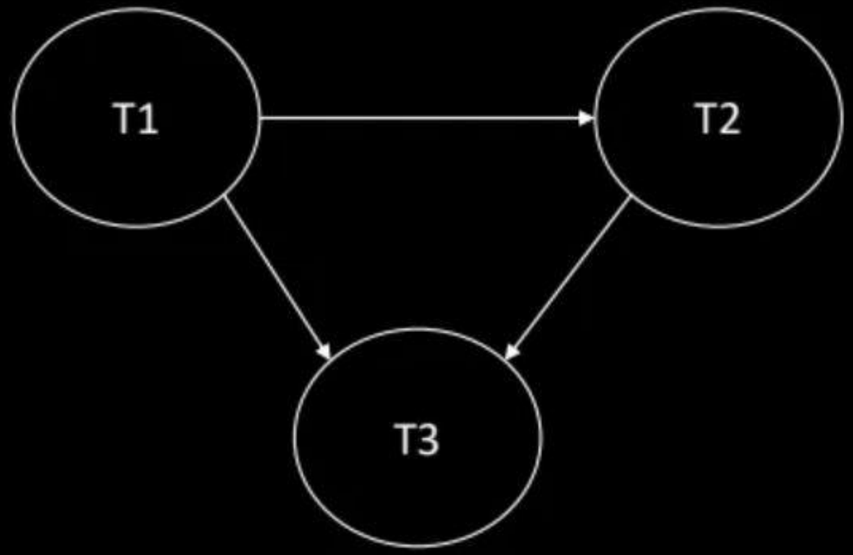
T1	T2	T3
R(A)		
	W(A)	
W(A)		
		W(A)

Cycle is formed,
Not conflict serializable
Also, Blind Writes present



Precedence graph

Testing view serializability by polygraph,



SERIAL SCHEDULE POSSIBLE T1 T2 T3

No Cycle formed, No W-R Dependencies therefore it is view serializable

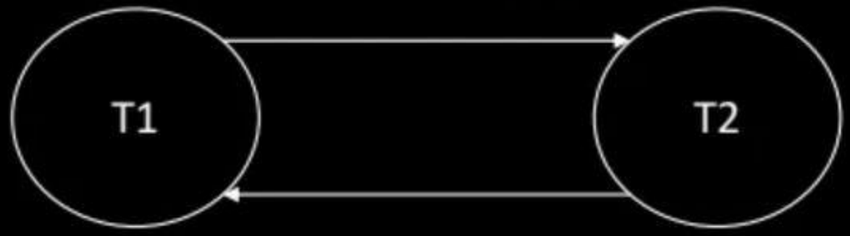
2.) Test the following schedule is view serializable

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

2.) Test the following schedule is view serializable

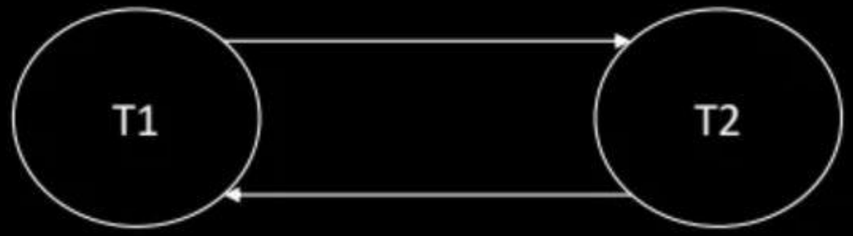
T1	T2
R(A)	
	W(A)
W(A)	

Precedence graph



Cycle is formed,
Not conflict serializable
Also, Blind Writes present

Testing view serializability by polygraph,



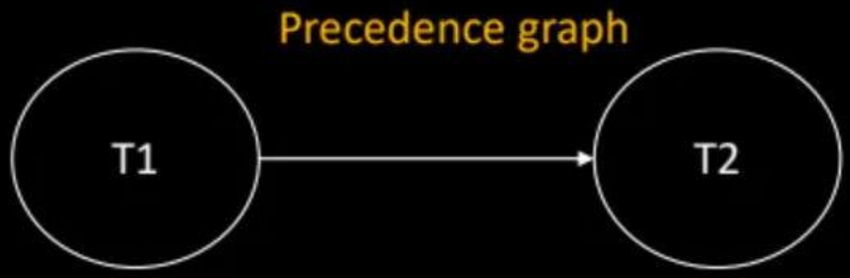
It is not view serializable

3.) Test the following schedule is view serializable

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

3.) Test the following schedule is view serializable

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



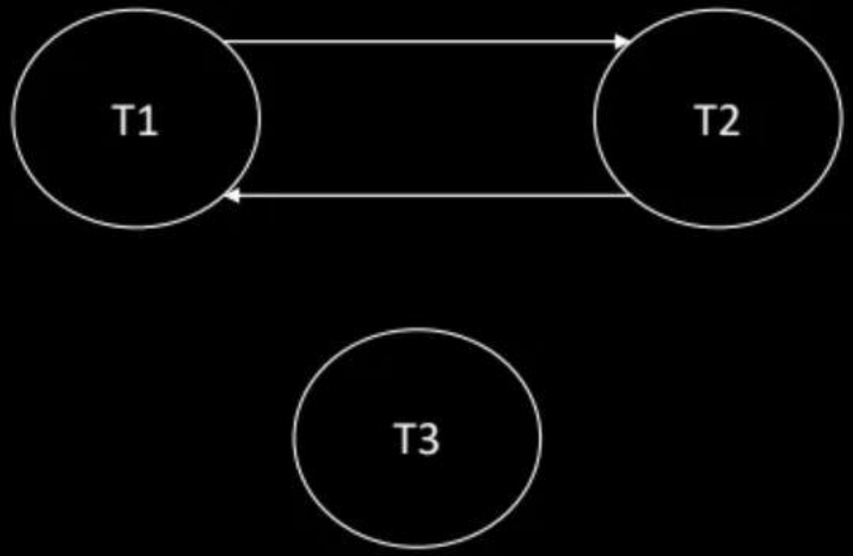
No cycle formed,
Therefore it is conflict serializable
And so it is view serializable too.

4.) Test the following schedule is view serializable

S = r1(A), w2(A), r3(A), w1(A) w3(A)

S = r1(A), w2(A), r3(A), w1(A) w3(A)

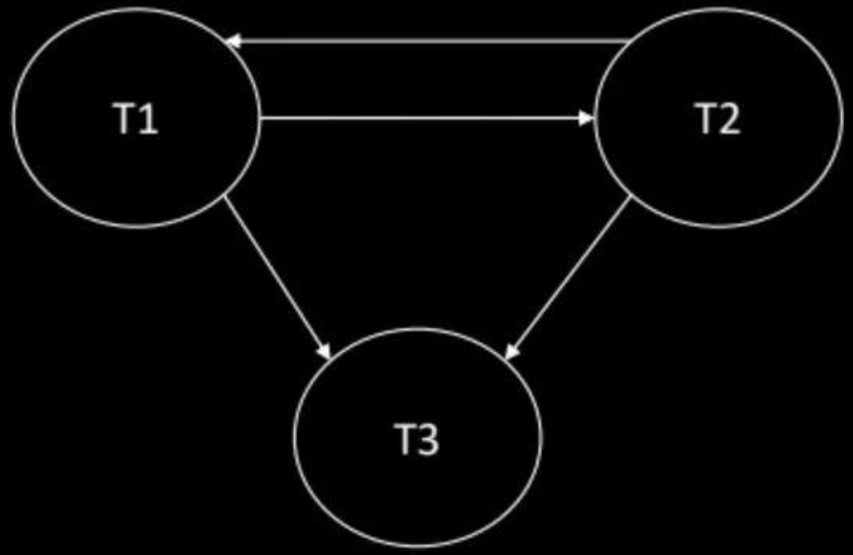
T1	T2	T3
r(A)		
	w(A)	
		r(A)
w(A)		
		w(A)



Precedence graph

Cycle is formed,
Not conflict serializable
Also, Blind Writes present

Testing view serializability by polygraph,



SERIAL SCHEDULE POSSIBLE T1 T2 T3

No Cycle formed, No W-R Dependencies therefore it is view serializable

5.) Test the following schedule is view serializable

S = T1: R(x), T2:W(x), T1: W(x), T2: abort, T1: Commit

5.) Test the following schedule is view serializable

S = T1: R(x), T2:W(x), T1: W(x), T2: abort, T1: Commit

Conflict serializable, Therefore View serializable