Transaction

DATABASE OPERATIONS (VS) TRANSACTION

Database Operations → Insert, Update, Delete

Transaction → Business Operations → Funds Transfer, New Account Opening

A Transaction is a set of DML operations that performs a Business Unit of Work.

Example :- Transfer $50 from account A to B

Txn from THEORY

1. **Read**(A)
2. A := A – 50
3. **Write**(A)

4. **Read**(B)
5. B := B + 50
6. **Write**(B)

Txn from ORACLE

1. Update ACCOUNT
   set balance = balance-50
   where account_no=A;

2. Update ACCOUNT
   set balance = balance+50
   where account_no=B;

READ(X)

1. *Find the **address of the disk block** that contains **item X**.*
2. *Copy that disk block into a **buffer** in **main memory** (if block is **not** in main memory buffer).*
3. *Copy **item X** from the **buffer** to the **program variable** named X.*

WRITE(X)

1) *Copy **item X** from the program variable named X into its correct location in the **buffer**.*
2) *Store the updated block from the **buffer** back to **disk.***

HOW TRANSACTION BEGINS & HOW IT ENDS ?

Transaction Begins → When we execute the first DML statement after opening the session.

Transaction Ends → Txn ends with COMMIT (or) ROLLBACK (or) DDL (or)DCL.

2 Types of Transactions → READ ONLY [Only Select is allowed] and

READ WRITE [Insert, Update, Delete, and Select are allowed]

Default Option → READ WRITE

TRANSACTION PROPERTIES

DBMS must ensure that the following properties will be satisfied by **every Transaction** :-

A → **A**tomicity

C → **C**onsistency

I → **I**solation

D → **D**urability

TRANSACTION PROPERTIES

Atomicity

1. **read_from_acoount**(A)
2. A := A − 50
3. **write_to_account**(A)
4. **read_from_accont**(B)
5. B := B + 50
6. **write_to_account**(B)

Atomicity:-Either **all operations (or) none** with respect to the execution of the transaction.

1. If the **transaction fails** after step 3 and before step 6, money will be "lost" leading to an **inconsistent database state.**
2. Failure could be due to software (or) hardware.
3. The system should ensure that updates of a **partially executed transaction** are not reflected in the database.

TRANSACTION PROPERTIES

Consistency

1. **read_from_acoount**(A)
2. A := A – 50
3. **write_to_account**(A)
4. **read_from_accont**(B)
5. B := B + 50
6. **write_to_account**(B)

Consistency:- **Before** execution of a transaction, the database is in consistent state and even **after** execution, the database must be in consistent state.

1. The **sum of A and B** remains same before and after execution of the transaction.
2. A transaction must see a consistent database and must leave a consistent database.
3. During transaction execution the database may be temporarily inconsistent.

TRANSACTION PROPERTIES

Isolation

Txn: **T1** :- Transfer **$50** from A to B

1. **read_from_acoount**(A)
2. A := A – 50
3. **write_to_account**(A)
4. **read_from_accont**(B)
5. B := B + 50
6. **write_to_account**(B)

Txn: **T2** :- Transfer **$200** from A to B

1. **read_from_acoount**(A)
2. A := A – 200
3. **write_to_account**(A)
4. **read_from_accont**(B)
5. B := B + 200
6. **write_to_account**(B)

Total No of Transactions = 2 [T1, T2]

How many ways the 2 transactions can be executed ? **3** →

1. **T1** then **T2** serially
2. **T2** then **T1** serially
3. **T1 & T2** together

1. **T1** then **T2** serially → **10 seconds [5+5]**
2. **T2** then **T1** serially → **10 seconds [5+5]**
3. **T1 & T2** together → **5 seconds** [**12** operations executes concurrently]

TRANSACTION PROPERTIES

Isolation:- When transactions executes concurrently, the **net effect** must be **identical** to one of the possible ***serial execution***.

How many ways the 2 transactions can be executed ? **3** ➡

1. **T1** then **T2** serially
2. **T2** then **T1** serially
3. **T1 & T2** together

Now **Isolation** guarantee the following :-

3. **T1 & T2** together = 1. **T1** then **T2** serially

or

3. **T1 & T2** together = 2. **T2** then **T1** serially

Schedule

Txn : **T1** → Transfers $50 from A to B

Txn : **T2**→ Transfers 10% of the balance from A to B

1

Serial Schedule:- T1 then T2

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

2

Serial Schedule:- T2 then T1

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

3

Concurrent Schedule:- T2 & T1

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)	
	read(B)
	$B := B + temp$
	write(B)

Concurrent Schedule [T1& T2] = Serial Schedule 1 [T1 then T2]

TRANSACTION PROPERTIES

Advantages of executing multiple transactions Concurrently are as follows:-

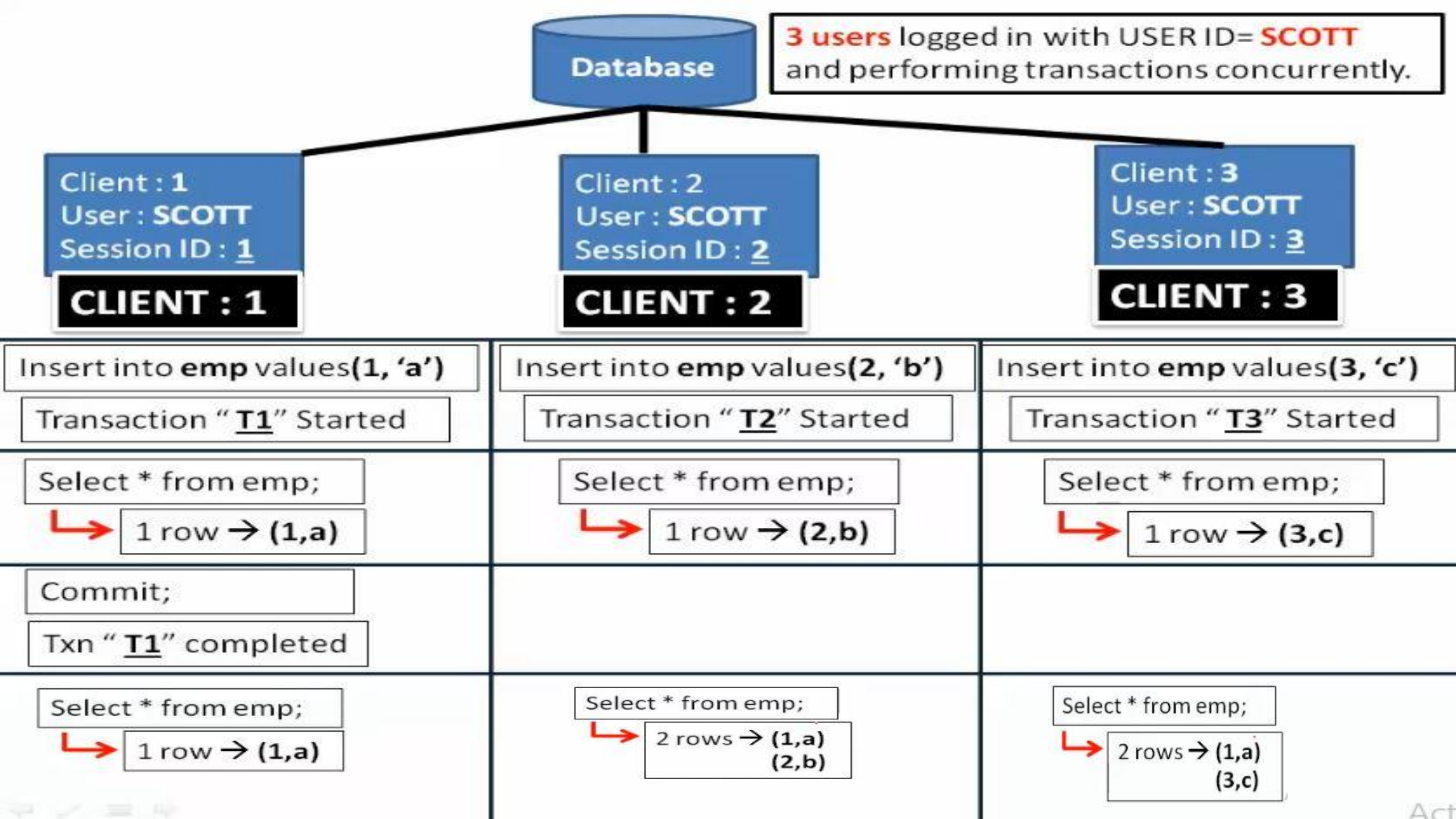
1. **Increased processor and disk utilization**, leading to better transaction ***throughput.***
2. **Reduced average response time** for transactions: short transactions need not wait behind long ones.

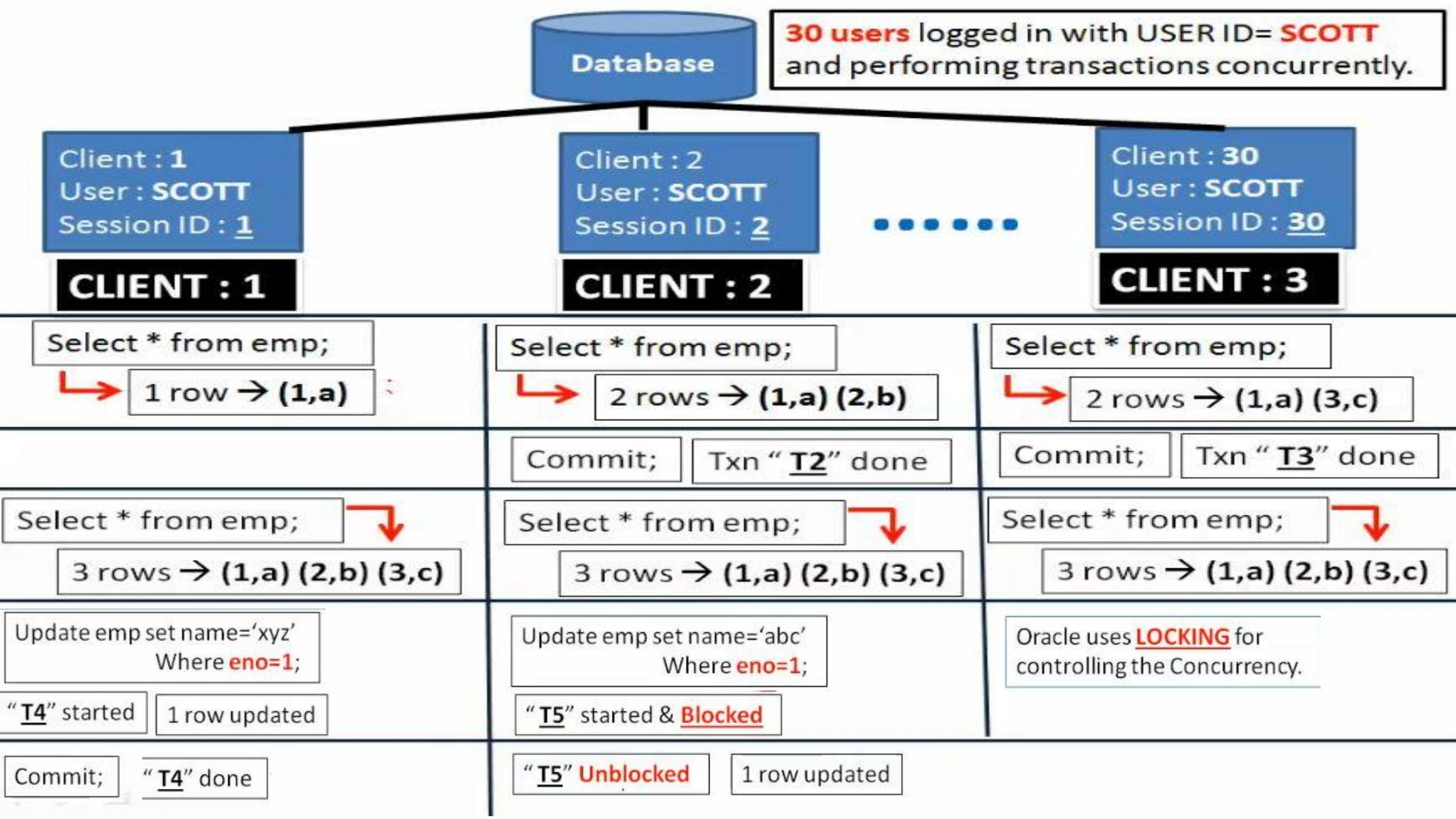
Transaction Properties

Durabilty

1. **read_from_acoount**(A)
2. A := A – 50
3. **write_to_account**(A)
4. **read_from_accont**(B)
5. B := B + 50
6. **write_to_account**(B)

Durability:- Once the transaction is completed, the updates must be permanent in the database even if there are Disk Failure (or) Operating System Failure (or) hardware failures.


Database
3 users logged in with USER ID= SCOTT and performing transactions concurrently.
Client : 1
User : SCOTT
Session ID : 1
CLIENT : 1
Client : 2
User : SCOTT
Session ID : 2
CLIENT : 2
Client : 3
User : SCOTT
Session ID : 3
CLIENT : 3
Insert into emp values(1, 'a')
Transaction " T1" Started
Insert into emp values(2, 'b')
Transaction " T2" Started
Insert into emp values(3, 'c')
Transaction " T3" Started
Select * from emp;
1 row → (1,a)
Select * from emp;
1 row → (2,b)
Select * from emp;
1 row → (3,c)
Commit;
Txn " T1" completed
Select * from emp;
1 row → (1,a)
Select * from emp;
2 rows → (1,a)
(2,b)
Select * from emp;
2 rows → (1,a)
(3,c)


Database
30 users logged in with USER ID= SCOTT and performing transactions concurrently.
Client : 1
User : SCOTT
Session ID : 1
Client : 2
User : SCOTT
Session ID : 2
Client : 30
User : SCOTT
Session ID : 30
CLIENT : 1
CLIENT : 2
CLIENT : 3
Select * from emp;
1 row → (1,a)
Select * from emp;
2 rows → (1,a) (2,b)
Select * from emp;
2 rows → (1,a) (3,c)
Commit;
Txn " T2" done
Commit;
Txn " T3" done
Select * from emp;
3 rows → (1,a) (2,b) (3,c)
Select * from emp;
3 rows → (1,a) (2,b) (3,c)
Select * from emp;
3 rows → (1,a) (2,b) (3,c)
Update emp set name='xyz' Where eno=1;
" T4" started
1 row updated
Update emp set name='abc' Where eno=1;
" T5" started & Blocked
Oracle uses LOCKING for controlling the Concurrency.
Commit;
" T4" done
" T5" Unblocked
1 row updated

HOW CONCURRENCY IS CONTROLLED IN ORACLE?

Observations

Only **One transaction** will be active at any point of time for each **session**.

During the transaction execution, the **inconsistent data** will not be shown to other transactions.

All other transactions will be viewing the old consistent data.

Oracle uses **LOCKING** to control the concurrency [for Conflicting Operations]

*Two actions are said to **conflict** if*

1. *They belong to **different transactions**. [one belong to T1 & other belongs to T2]*
2. *Both operations refers to the **same** database object.*
3. *One of them is **Write**.*

T1	T2	CONFLICT (or) NOT
Read(A)	**Read(A)**	**No**
Read(A)	Write(A)	Conflict
Write(A)	Read(A)	Conflict
Write(A)	Write(A)	Conflict

Dirty Read [RW Conflict]

The following Transactions will be used to explain this concept:-

T1 → Transfers $500 from A to B

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

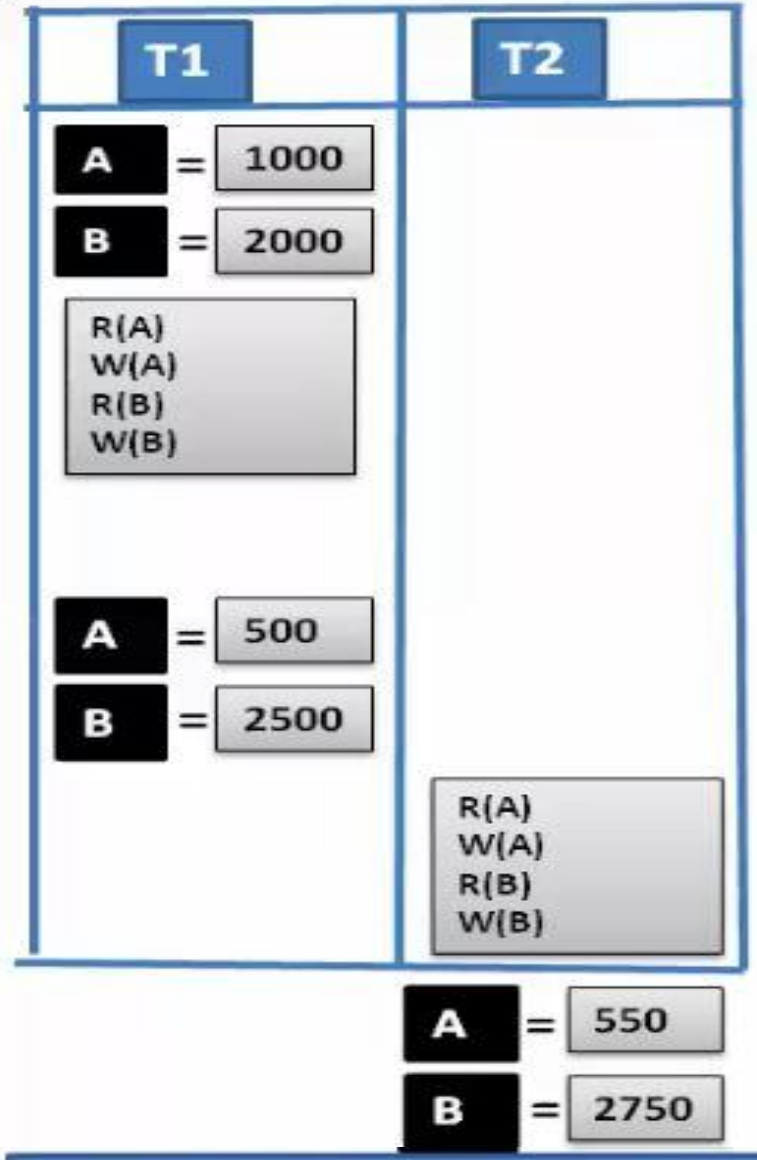
T2 → Adds 10% interest to A and B

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

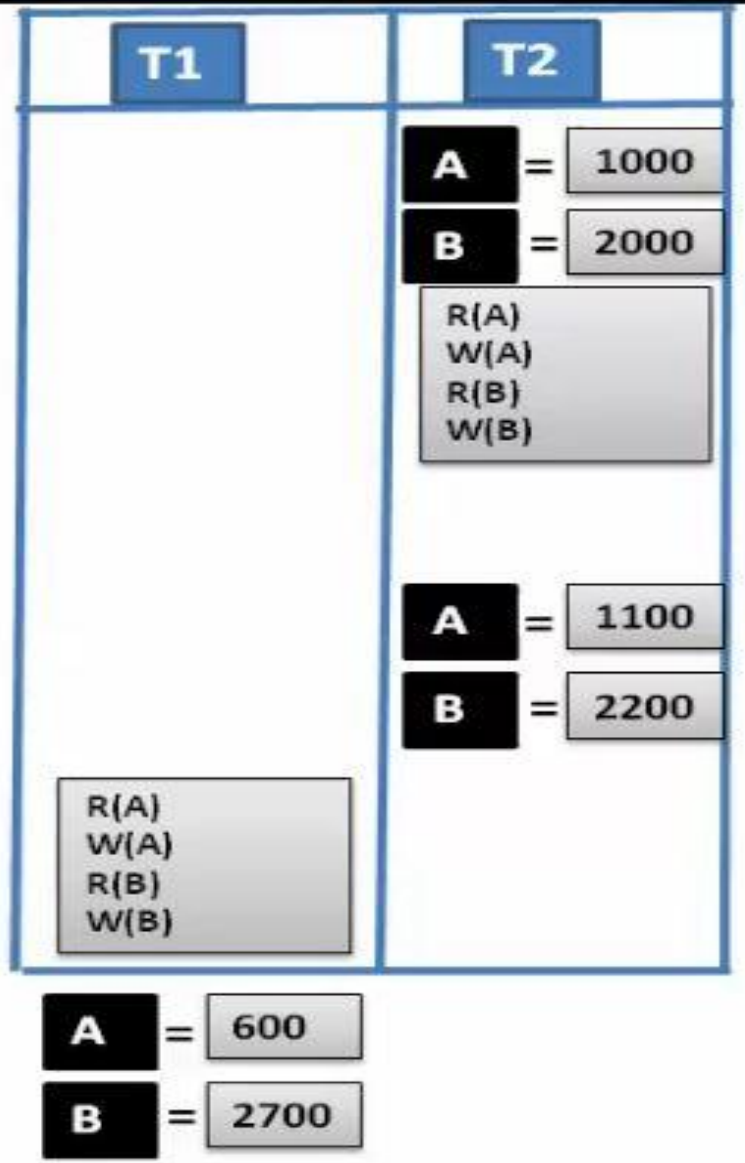
Consider the following initial Values:-

A	=	1000
B	=	2000

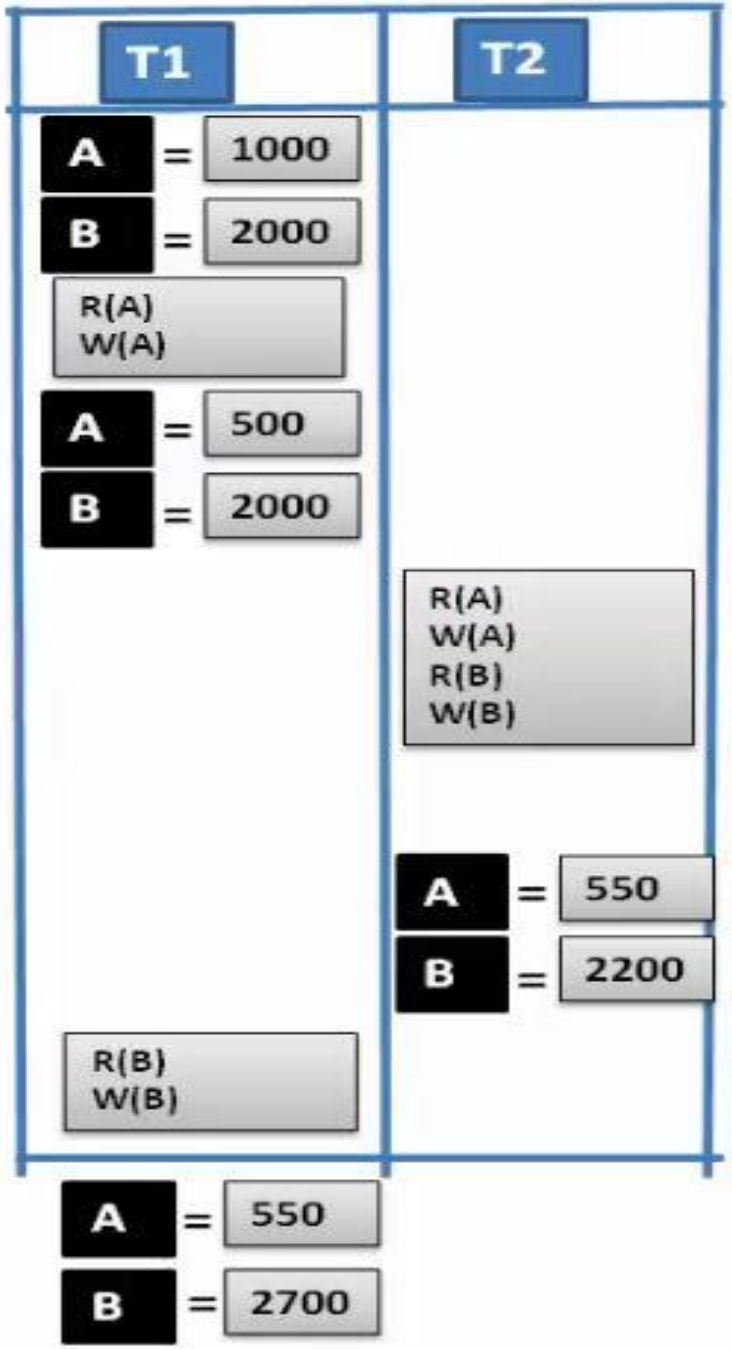
Serial Execution : T1 then T2



Serial Execution : T2 then T1



Concurrent Execution : T1 & T2



Concurrent Execution is Not a Serializable Schedule.

Reason :- T2 read the uncommitted data [A] of T1

Unrepeatable Read [RW Conflicts]

The following Transactions will be used to explain this concept:-

T1 → Reads 'A' two times & assigns "A" to "B & C"

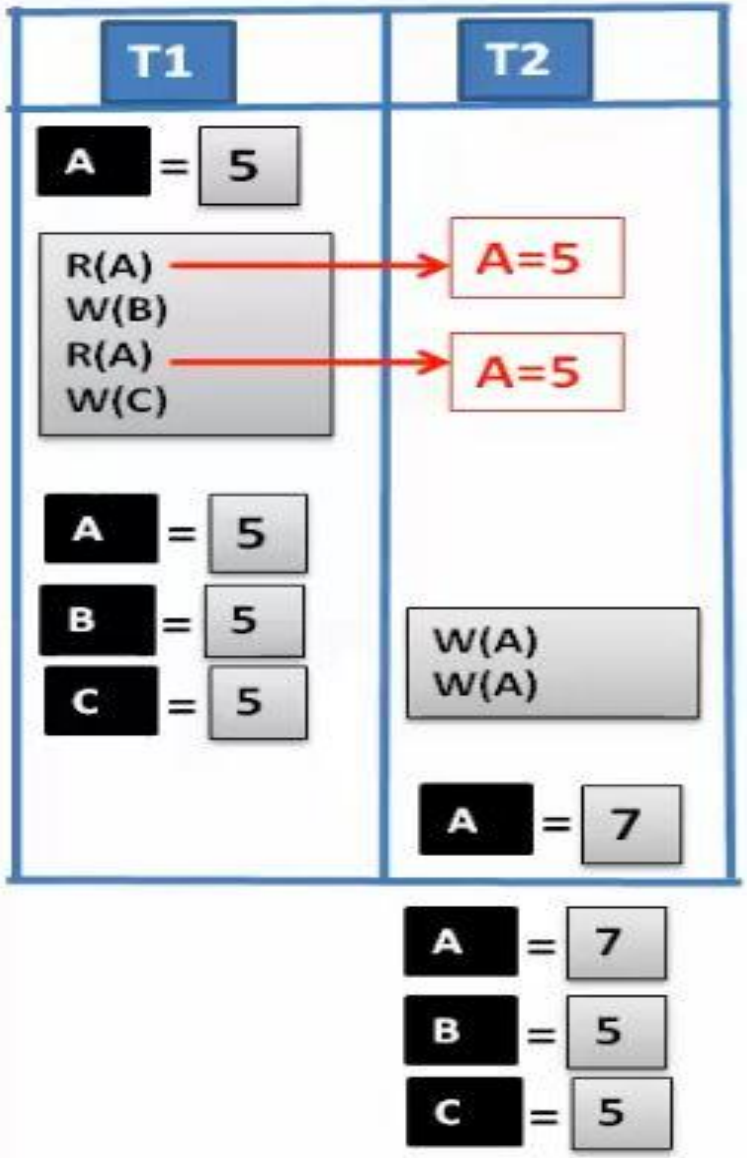
R(A)
W(B) → B = A
R(A)
W(C) → C = A

T2 → Increment's 'A' two times

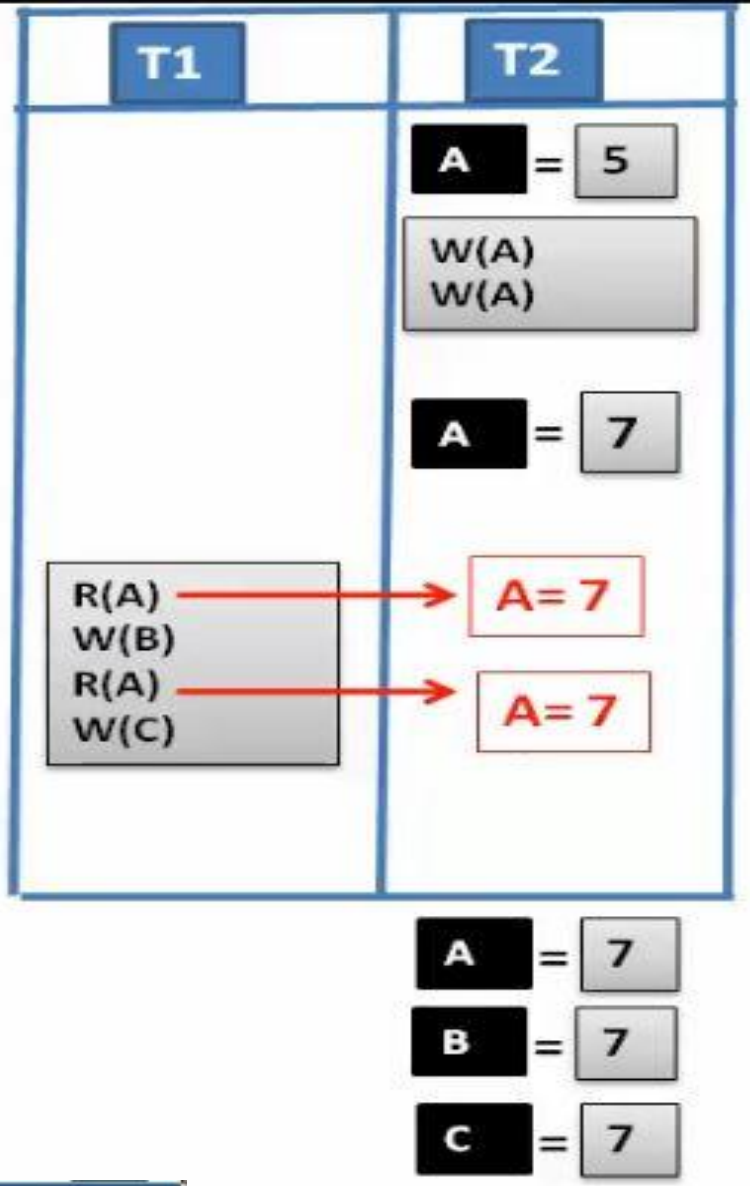
W(A)
W(A)

Consider the following initial Values:- **A** = 5

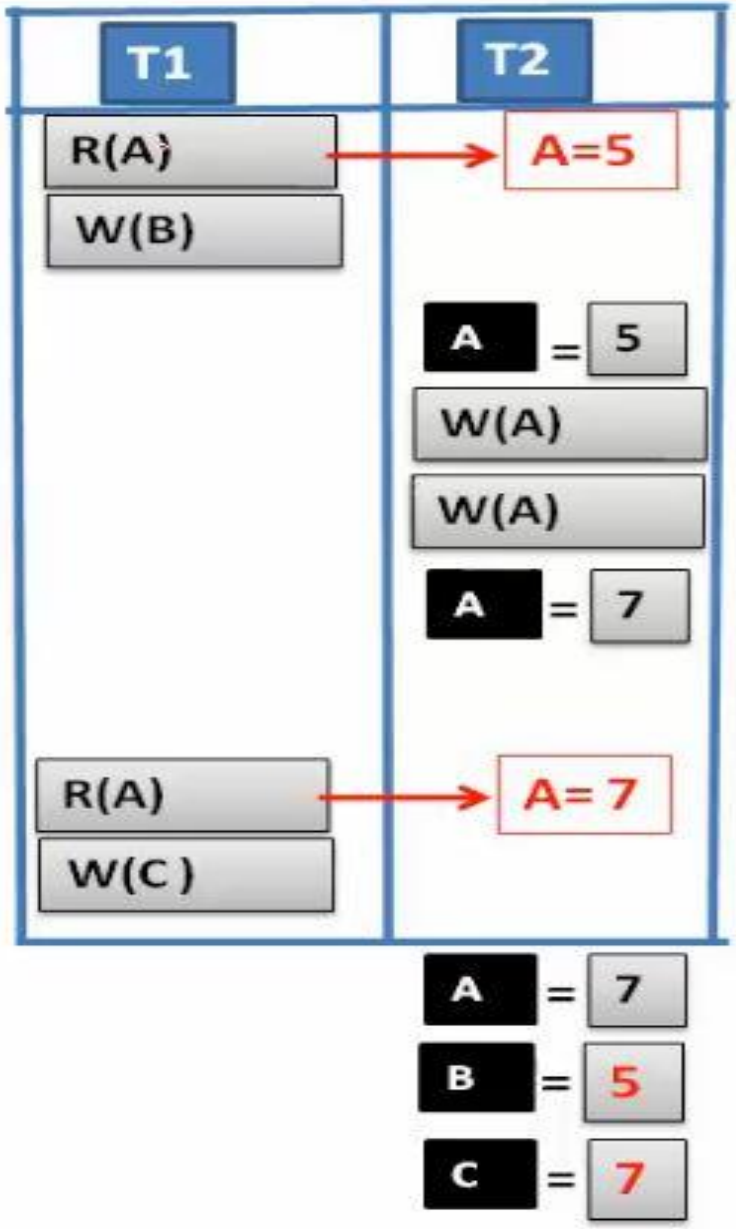
Serial Execution : T1 then T2



Serial Execution : T2 then T1



Concurrent Execution : T1 & T2



Concurrent Execution is Not a Serializable Schedule.

Reason :- Read(A) returned 2 distinct values during the life time of Transaction 'T1'

Blind Write [WW Conflicts]

- *Akbar* **[A]** *and Birbal* **[B]** *are 2 employees and Transactions T1 & T2 wants to equal their salaries.*

- *The following Transactions will be used to explain this concept:-*

T1 → Equals A & B' salary to $1111

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

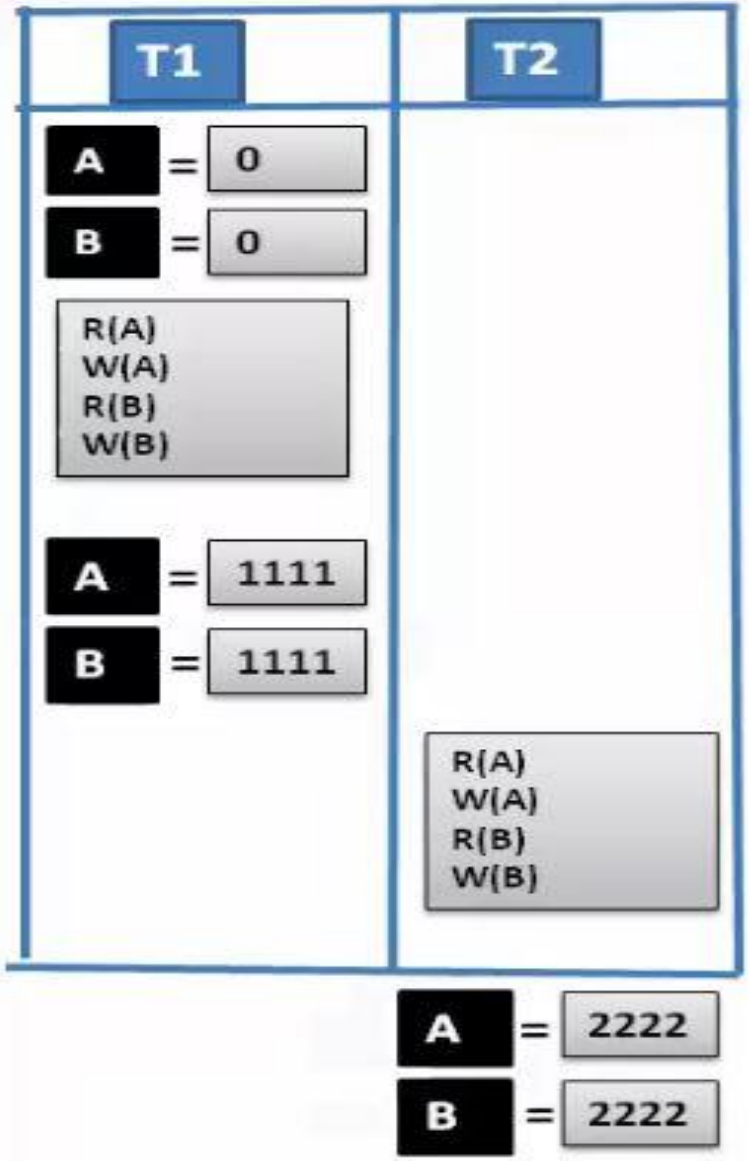
T2 → Equals A & B' salary to $2222

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

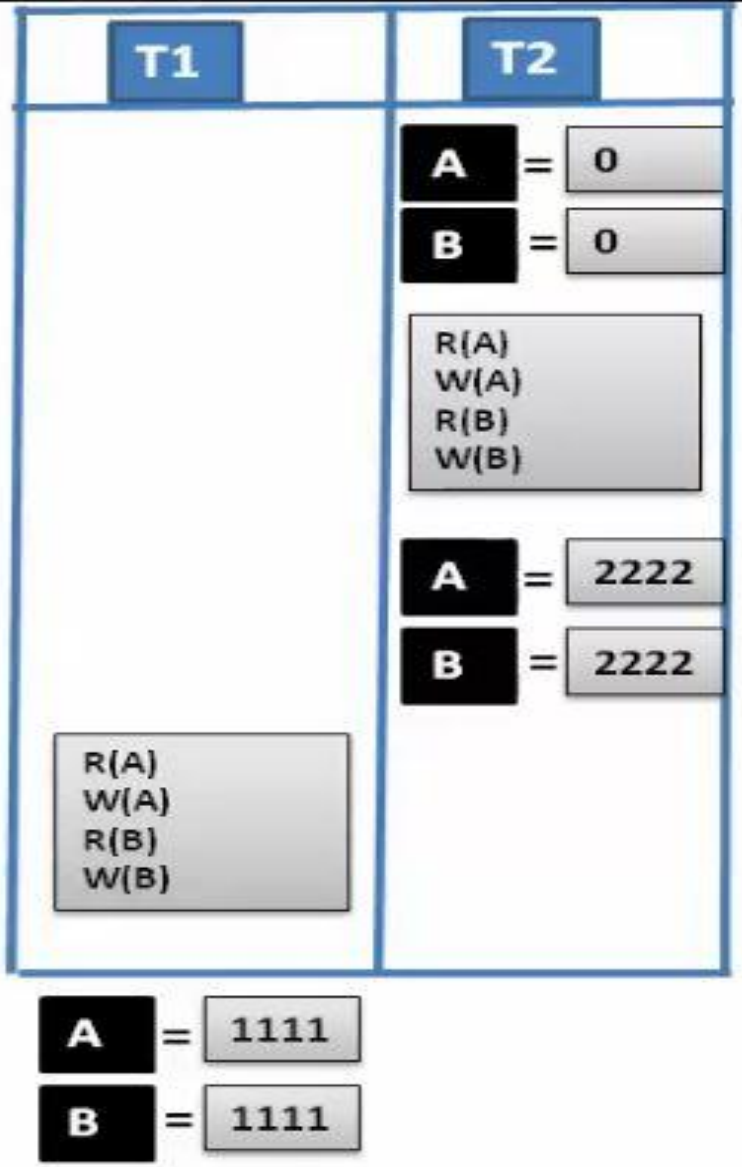
- *Consider the following initial Values:-* **A** = **0**

B = **0**

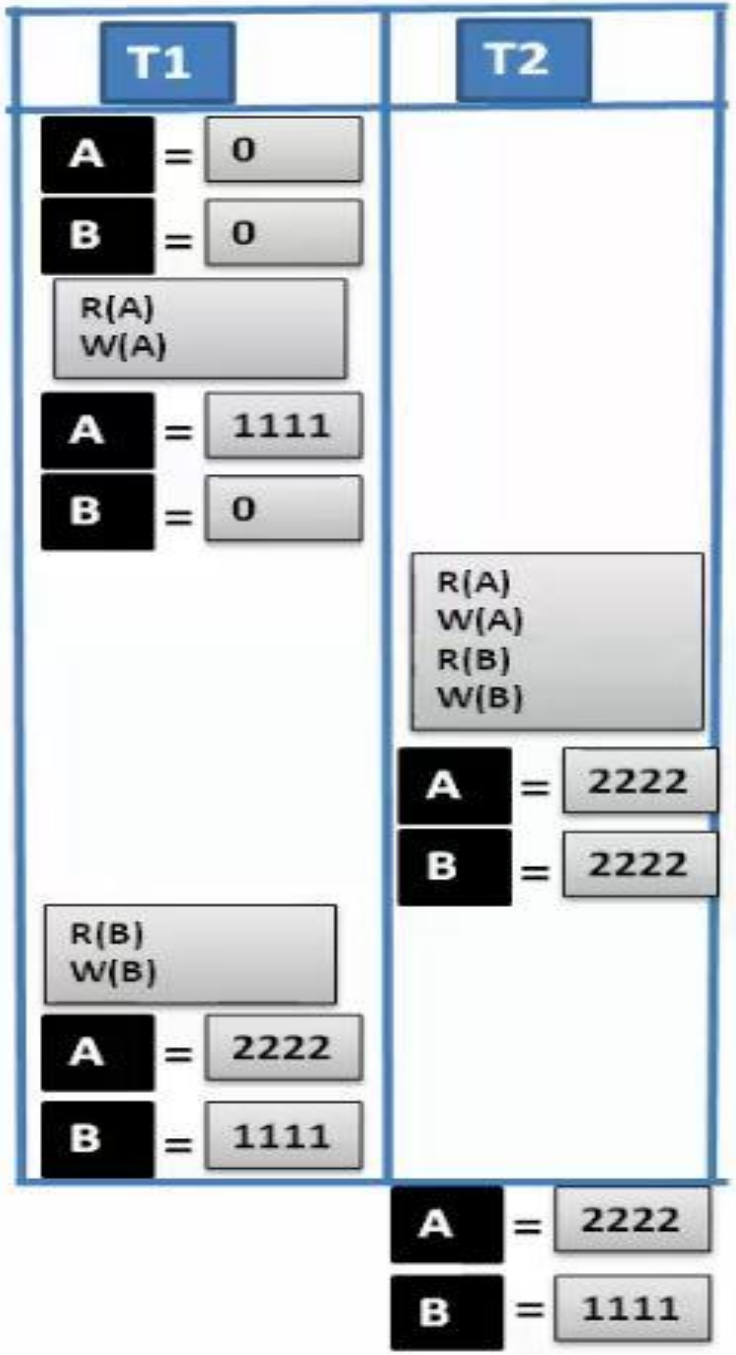
Serial Execution : T1 then T2



Serial Execution : T2 then T1



Concurrent Execution : T1 & T2



Concurrent Execution is Not a Serializable Schedule.

Reason :- T2 overwritten the modified data of uncommitted transaction T1

PROBLEMS IF NO CONTROL IN CONCURRENCY

Guidelines to overcome all the 3 anomalies:-

1. Every Transaction must **read** the **Committed Data** only. **[No Dirty Read Problem]**

2. If Transaction "T1" **read** the committed data, then no other Transaction should **modify** the data as long as "T1" is in progress. **[No Unrepeatable Read Problem]**

3. If Transaction "T1" has **modified** any data, then no other Transaction is allowed to **modify** the data again as long as "T1" is in progress. **[No Blind Write Problem]**

Using **LOCKING technique**, DBMS is overcoming the above mentioned problems

Testing for Serializability :-

T1 → Transfers 10 rupees from X to Y

T2 → Transfers 20 rupees from X to Y

X = 100
Y = 200

T1 & T2 Concurrent	
T1	T2
R(X)	
X = X -10	
W(X)	
	R(Y)
	Y = Y + 20
	W(Y)
R(Y)	
Y = Y + 10	
W(Y)	
COMMIT	
	R(X)
	X = X - 20
	W(X)
	COMMIT

X = 70
Y = 230

X = 100
Y = 200

T1 then T2 Serially	
T1	T2
R(X)	
X = X -10	
W(X)	
R(Y)	
Y = Y + 10	
W(Y)	
COMMIT	
	R(Y)
	Y = Y + 20
	W(Y)
	R(X)
	X = X - 20
	W(X)
	COMMIT

X = 70
Y = 230

X = 100
Y = 200

T2 then T1 Serially	
T1	T2
	R(Y)
	Y = Y + 20
	W(Y)
	R(X)
	X = X - 20
	W(X)
	COMMIT
R(X)	
X = X -10	
W(X)	
R(Y)	
Y = Y + 10	
W(Y)	
COMMIT	

X = 70
Y = 230

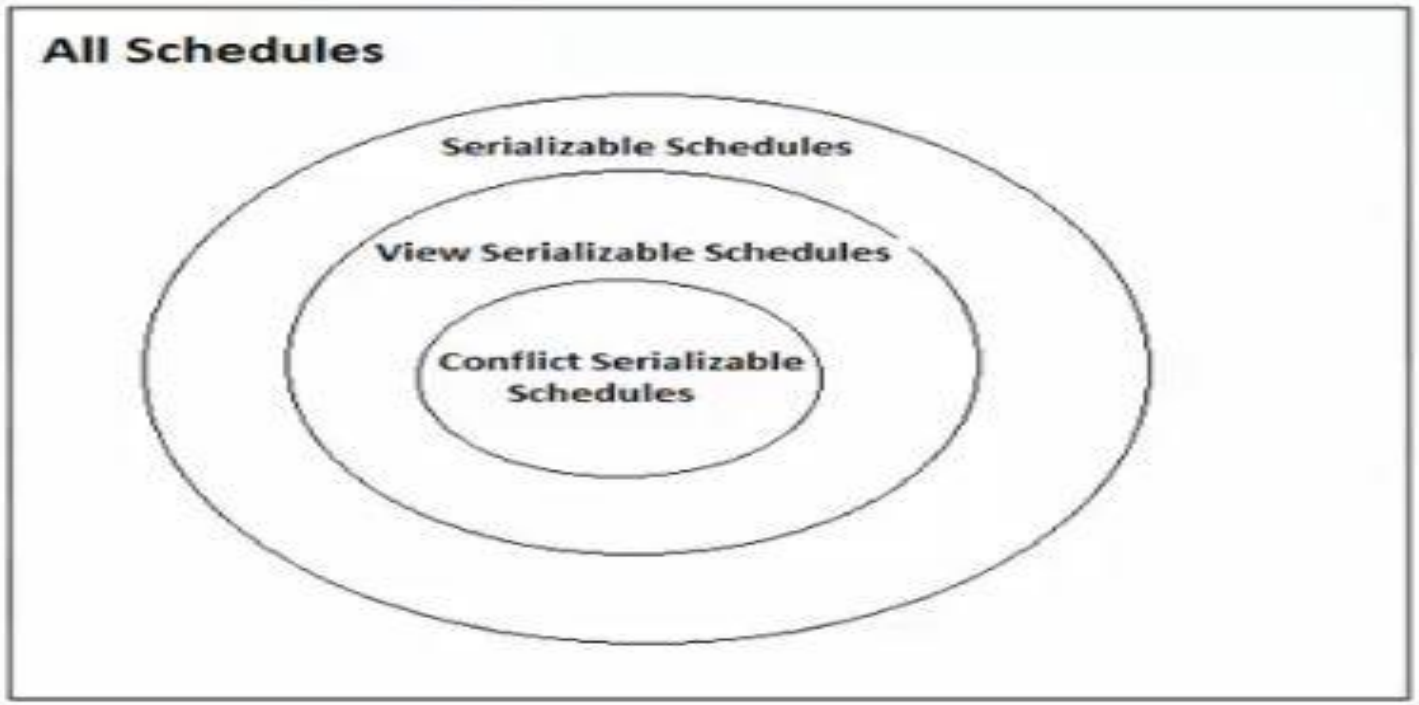
Schedule is Serializable to both
T1 → T2 and T2 → T1

Serializability

S → Set of Committed Transactions

- A schedule **'S'** is said to be **serializable** if and only if the net effect is identical to some complete serial schedule over 'S'.

- *As its difficult to verify whether the given schedule is serializable (or) not, we use the following types of Serializability to verify the Isolation property:-*



Serializability

Easier way to verify whether the schedule is serializable (or) not :-

1. **Conflict Serializability**
2. **View Serializability**

Conflict Serializability

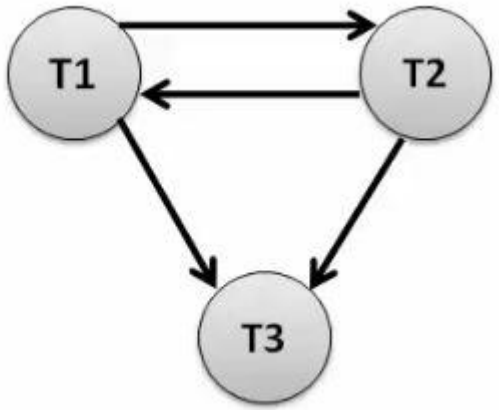
How to identify whether the given schedule is **Conflict Serializable** (or) Not ?

Precedence graph

Steps to draw a Precedence graph:-

1. A node for each committed transaction in S.
2. An arc from T_i to T_j if an action of T_i precedes and conflicts with one of T_j 's actions.
3. A schedule 'S' is Conflict Serializable, if and only if, the graph is **acyclic**.

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



Cycle is formed [T1 and T2]→ Not Conflict Serializable.

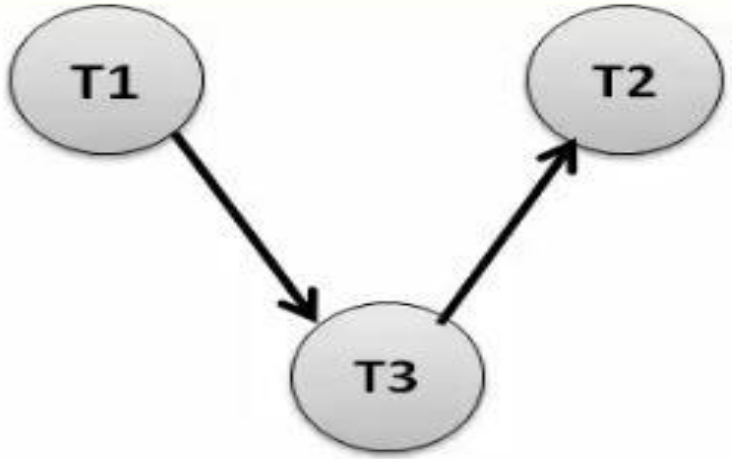
Conflict Serializability

Steps to draw a Precedence graph:-

1. *A node for each committed transaction in S.*
2. *An arc from T_i to T_j if an action of T_i precedes and conflicts with one of T_j 's actions.*
3. *A schedule 'S' is Conflict Serializable, if and only if, the graph is **acyclic**.*

SCHEDULE		
T1	T2	T3
R(X)		
	R(Y)	
		W(X)
		Commit;
	R(X)	
R(Y)		
Commit;		
	Commit	

No Cycle → Conflict Serializable → Serializable



Equivalent Serial Schedule = T1 → T3 → T2

Previous GATE Questions

Consider the following schedule.

T3	T4
Read(Q)	
	Write(Q)
Write(Q)	

Which out of the following is true?

a. Schedule is equivalent to

T3	T4
	Write(Q)
Read(Q)	
Write(Q)	

b. Schedule is equivalent to

T3	T4
Read(Q)	
Write(Q)	
	Write(Q)

c. Schedule is not conflict serializable
d. Schedule is conflict serializable

Previous GATE Questions

Consider the following schedule.

T3	T4
Read(Q)	
	Write(Q)
Write(Q)	

Which out of the following is true?

X

T_3 ⇄ T_4 → Cycle is formed

a. Schedule is equivalent to

T3	T4
	Write(Q)
Read(Q)	
Write(Q)	

b. Schedule is equivalent to

T3	T4
Read(Q)	
Write(Q)	
	Write(Q)

Answer = C

c. Schedule is not conflict serializable

d. Schedule is conflict serializable

Previous GATE Questions

Consider the following schedules involving the transactions. Which one of the following statements is TRUE ?

$S_1 : r_1(X); r_1(Y); r_2(X); r_2(Y); w_2(Y); w_1(X)$

$S_2: r_1(X); r_2(X); r_2(Y); w_2(Y); r_1(Y); w_1(X)$

a. Both S_1 and S_2 are conflict serializable
b. S_1 is conflict serializable and S_2 is not conflict serializable
c. S_1 is not conflict serializable and S_2 is conflict serializable
d. Both S_1 and S_2 are not conflict serializable

Previous GATE Questions

Consider the following schedules involving the transactions. Which one of the following statements is TRUE ?

S_1 : $r_1(X)$; $r_1(Y)$; $r_2(X)$; $r_2(Y)$; $w_2(Y)$; $w_1(X)$

S_2: $r_1(X)$; $r_2(X)$; $r_2(Y)$; $w_2(Y)$; $r_1(Y)$; $w_1(X)$

Schedule : S1

T1	T2
R1(X)	
R1(Y)	
	R2(X)
	R2(Y)
	W2(Y)
W1(X)	

Schedule : S2

T1	T2
R1(X)	
	R2(X)
	R2(Y)
	W2(Y)
R1(Y)	
W1(X)	

a. Both S_1 and S_2 are conflict serializable
b. S_1 is conflict serializable and S_2 is not conflict serializable
c. S_1 is not conflict serializable and S_2 is conflict serializable
d. Both S_1 and S_2 are not conflict serializable

Which of the following schedules are **conflict serializable** ?

T_1: $R_1[x]$ $W_1[x]$ $W_1[y]$

T_2: $R_2[x]$ $R_2[y]$ $W_2[y]$

a.S1 and S2
b.S2 and S3
c.S3 only
d.S4 only

S_1: $R_1[x]$ $R_2[x]$ $R_2[y]$ $W_1[x]$ $W_1[y]$ $W_2[y]$

S_2: $R_1[x]$ $R_2[x]$ $R_2[y]$ $W_1[x]$ $W_2[y]$ $W_1[y]$

S_3: $R_1[x]$ $W_1[x]$ $R_2[x]$ $W_1[y]$ $R_2[y]$ $W_2[y]$

S_4: $R_2[x]$ $R_2[y]$ $R_1[x]$ $W_1[x]$ $W_1[y]$ $W_2[y]$

Which of the following schedules are **conflict serializable** ?

T_1: $R_1[x]$ $W_1[x]$ $W_1[y]$

T_2: $R_2[x]$ $R_2[y]$ $W_2[y]$

S_1: $R_1[x]$ $R_2[x]$ $R_2[y]$ $W_1[x]$ $W_1[y]$ $W_2[y]$

S_2: $R_1[x]$ $R_2[x]$ $R_2[y]$ $W_1[x]$ $W_2[y]$ $W_1[y]$

S_3: $R_1[x]$ $W_1[x]$ $R_2[x]$ $W_1[y]$ $R_2[y]$ $W_2[y]$

S_4: $R_2[x]$ $R_2[y]$ $R_1[x]$ $W_1[x]$ $W_1[y]$ $W_2[y]$

a. S1 and S2
b. S2 and S3
c. S3 only
d. S4 only

Schedule : S1	
T1	T2
R1(X)	
	R2(X)
	R2(Y)
W1(X)	
W1(Y)	
	W2(Y)

Schedule : S2	
T1	T2
R1(X)	
	R2(X)
	R2(Y)
W1(X)	
	W2(Y)
W1(Y)	

$T_2 \rightarrow T_1$

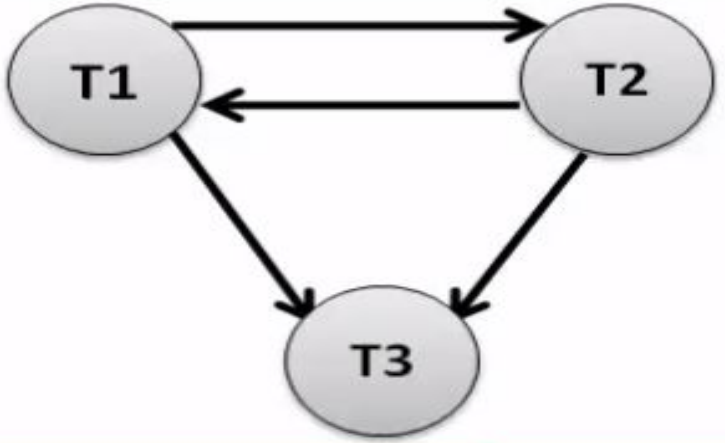
Schedule : S3	
T1	T2
R1(X)	
W1(X)	
	R2(X)
W1(Y)	
	R2(Y)
	W2(Y)

$T_1 \rightarrow T_2$

Schedule : S4	
T1	T2
	R2(X)
	R2(Y)
R1(X)	
W1(X)	
W1(Y)	
	W2(Y)

View Serializability

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

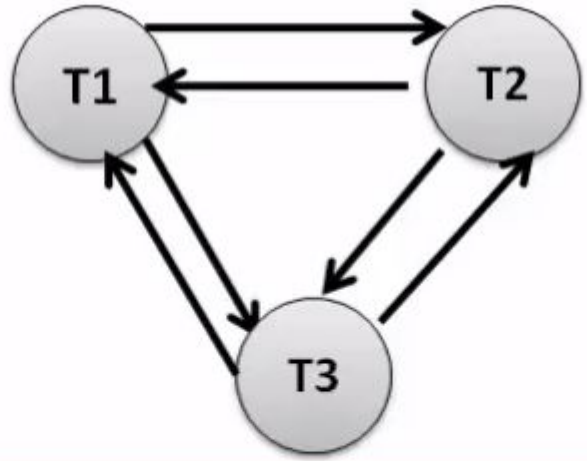


Cycle is formed [T1 and T2] → Not Conflict Serializable.

SNO	DATABASE OBJECT	INITIAL READ	WRITE-READ Sequencing	FINAL WRITE
1	A	T1	Nothing	T3

Equivalent Serial Schedule = T1 → T2 → T3

SCHEDULE		
T1	T2	T3
R(A)		
	R(B)	
	W(B)	
		R(A)
		W(A)
R(B)		
Commit		
		R(B)
		Commit
	R(A)	
	W(A)	
	Commit	



Cycle is formed [T1 ,T2,T3] → Not Conflict Serializable.

SNO	DATABASE OBJECT	INITIAL READ	WRITE-READ Sequencing	FINAL WRITE
1	A	T1	T3 → T2	T2
2	B	T2	T2 → T1 , T2→ T3	T2

Serial Schedule w.r.t object "A" = T1 → T3 → T2

Serial Schedule w.r.t object "B" = T2 → T1 → T3 (or) T2 → T3 → T1

Serializability

Verify whether the given Schedule is Serializable (or) not?

T1 → Transfers 10 rupees from X to Y

T2 → Transfers 20 rupees from X to Y

Is Schedule Conflict Serializable ? **NO**

Is Schedule View Serializable ? **NO**

T1	T2
R(X)	
X=X-10	
W(X)	
	R(Y)
	Y=Y+20
	W(Y)
R(Y)	
Y=Y+10	
W(Y)	
COMMIT;	
	R(X)
	X=X-20
	W(X)
	COMMIT;

TEST FOR CONFLICT SERIALIZABILITY :-

Cycle is formed → **Not Conflict Serializable**

TEST FOR VIEW SERIALIZABILITY :-

SNO	OBJECT	INITIAL READ	WRITE READ SEQUENCE	FINAL WRITE
1	X	T1	T1 → T2	T2
2	Y	T2	T2 → T1	T1

Serial Order from OBJECT X = T1 → T2

Serial Order from OBJECT Y = T2 → T1

Impossible

Schedule is **Not VIEW SERIALIZABLE**

Testing for Serializability :-

T1 → Transfers 10 rupees from X to Y

T2 → Transfers 20 rupees from X to Y

Not Conflict Serializable
&
Not View Serializable

X = 100
Y = 200

T1 & T2 Concurrent	
T1	T2
R(X)	
X = X -10	
W(X)	
	R(Y)
	Y = Y + 20
	W(Y)
R(Y)	
Y = Y + 10	
W(Y)	
COMMIT	
	R(X)
	X = X - 20
	W(X)
	COMMIT

X = 70
Y = 230

X = 100
Y = 200

T1 then T2 Serially	
T1	T2
R(X)	
X = X -10	
W(X)	
R(Y)	
Y = Y + 10	
W(Y)	
COMMIT	
	R(Y)
	Y = Y + 20
	W(Y)
	R(X)
	X = X - 20
	W(X)
	COMMIT

X = 70
Y = 230

X = 100
Y = 200

T2 then T1 Serially	
T1	T2
	R(Y)
	Y = Y + 20
	W(Y)
	R(X)
	X = X - 20
	W(X)
	COMMIT
R(X)	
X = X -10	
W(X)	
R(Y)	
Y = Y + 10	
W(Y)	
COMMIT	

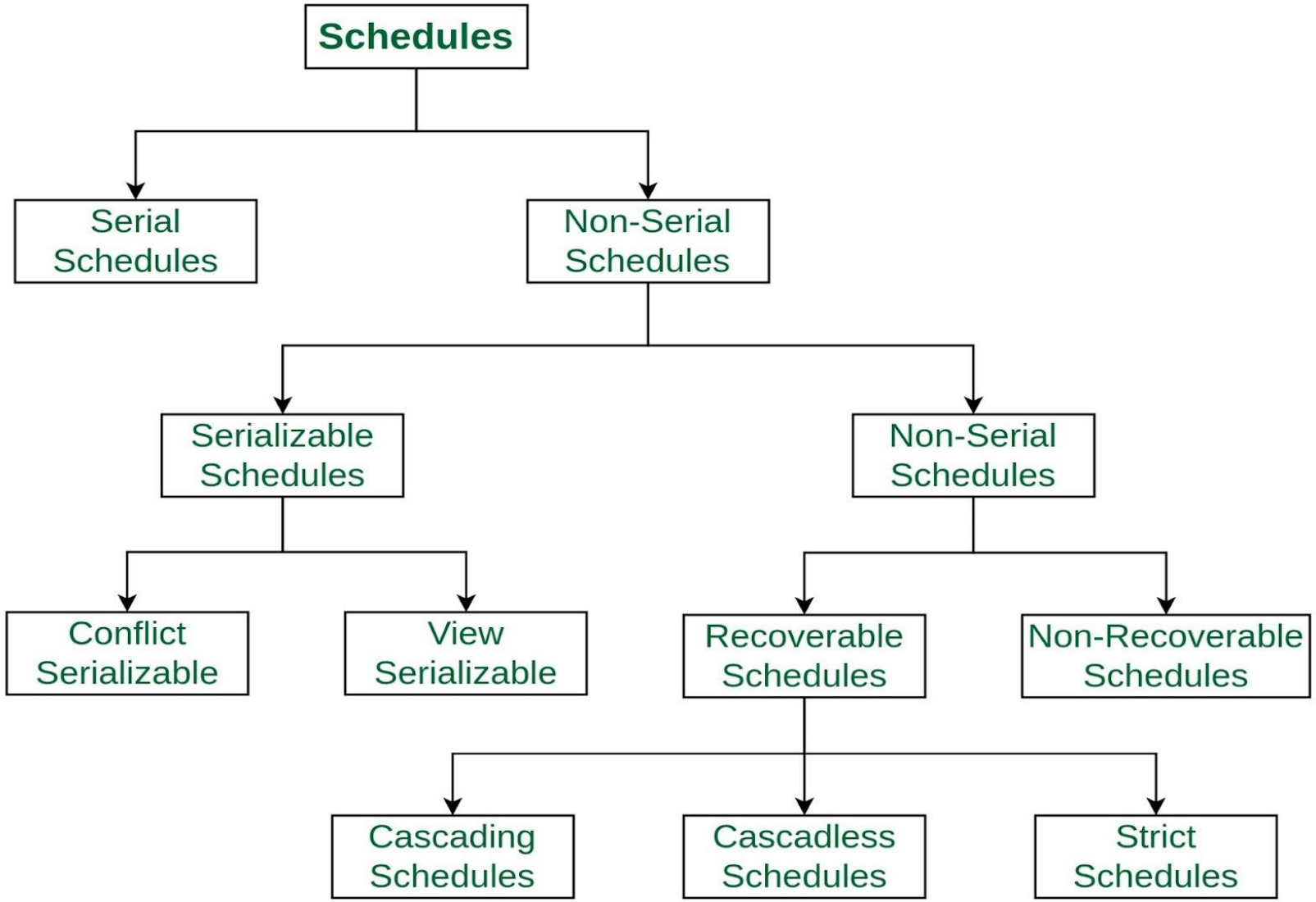
X = 70
Y = 230

Schedule is **Serializable** to both
T1 → T2 and **T2 → T1**

Types of Schedules in DBMS

Schedule, as the name suggests, is a process of lining the transactions and executing them one by one. When there are multiple transactions that are running in a concurrent manner and the order of operation is needed to be set so that the operations do not overlap each other, Scheduling is brought into play and the transactions are timed accordingly.

Types of schedules in DBMS



Serial Schedules: Schedules in which the transactions are executed non-interleaved, i.e., a serial schedule is one in which no transaction starts until a running transaction has ended are called serial schedules.

Example: Consider the following schedule involving two transactions T_1 and T_2.

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

where R(A) denotes that a read operation is performed on some data item 'A' This is a serial schedule since the transactions perform serially in the order T_1 —> T_2

Non-Serializable:

The non-serializable schedule is divided into two types, Recoverable and Non-recoverable Schedule.

Recoverable Schedule: Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction T_j is reading value updated or written by some other transaction T_i, then the commit of T_j must occur after the commit of T_i.

Example – Consider the following schedule involving two transactions T_1 and T_2.

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

This is a recoverable schedule since T_1 commits before T_2, that makes the value read by T_2 correct. There can be three types of recoverable schedule:

Cascading Schedule: Also called Avoids cascading aborts/rollbacks (ACA). When there is a failure in one transaction and this leads to the rolling back or aborting other dependent transactions, then such scheduling is referred to as Cascading rollback or cascading abort. Example:

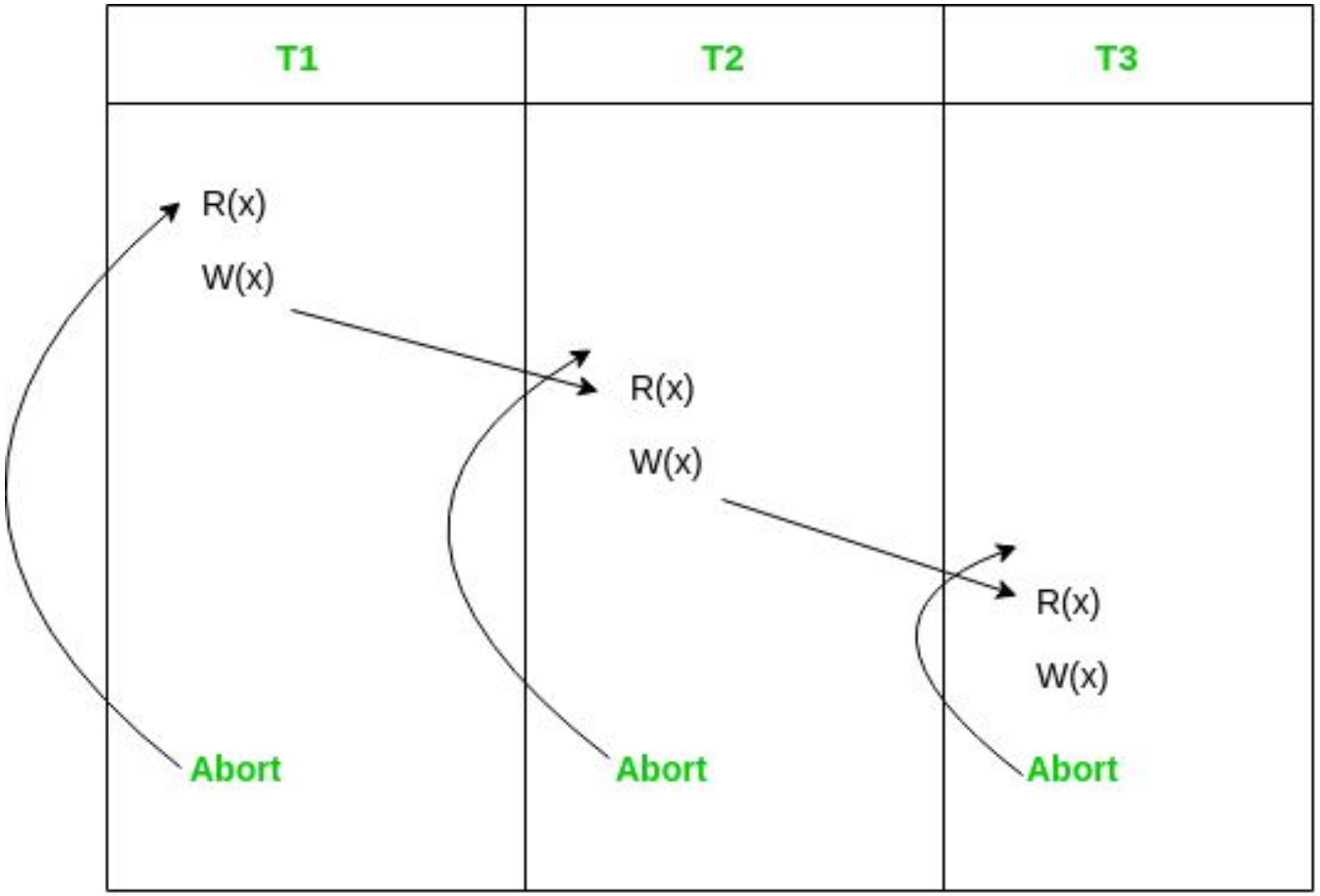


Figure - Cascading Abort

Cascadeless Schedule: Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules. Avoids that a single transaction abort leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

Example: Consider the following schedule involving two transactions T_1 and T_2.

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

This schedule is cascadeless. Since the updated value of **A** is read by T_2 only after the updating transaction i.e. T_1 commits.

Strict Schedule:

A schedule is strict if for any two transactions T_i, T_j, if a write operation of T_i precedes a conflicting operation of T_j (either read or write), then the commit or abort event of T_i also precedes that conflicting operation of T_j.

Example: Consider the following schedule involving two transactions T_1 and T_2.

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

This is a strict schedule since T_2 reads and writes A which is written by T_1 only after the commit of T_1.

Non-Recoverable Schedule:

Example: Consider the following schedule involving two transactions T_1 and T_2.

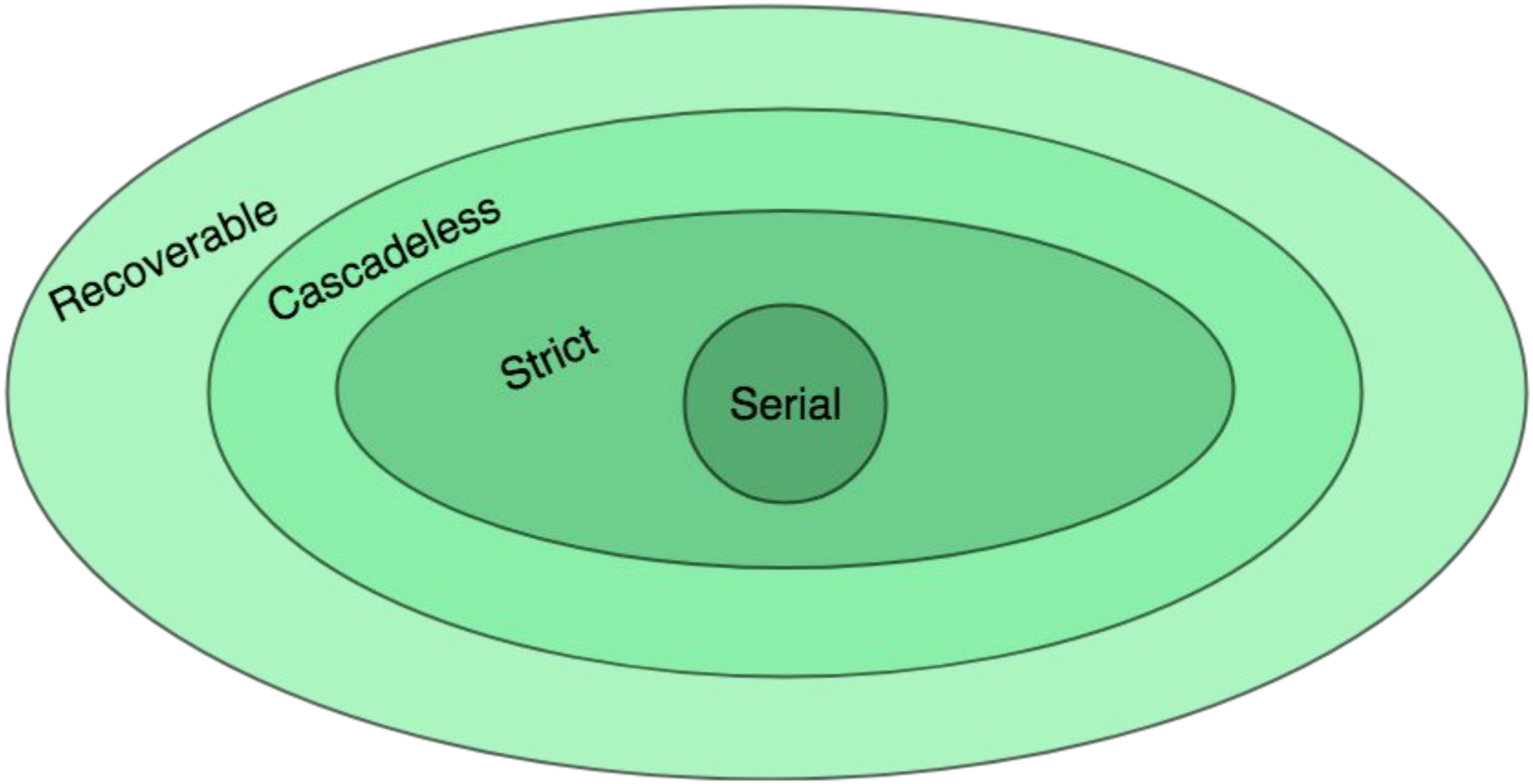
T_1	T_2
R(A)	
W(A)	
	W(A)
	R(A)
	commit
abort	

T_2 read the value of A written by T_1, and committed. T_1 later aborted, therefore the value read by T_2 is wrong, but since T_2 committed, this schedule is **non-recoverable**.

Note – It can be seen that:

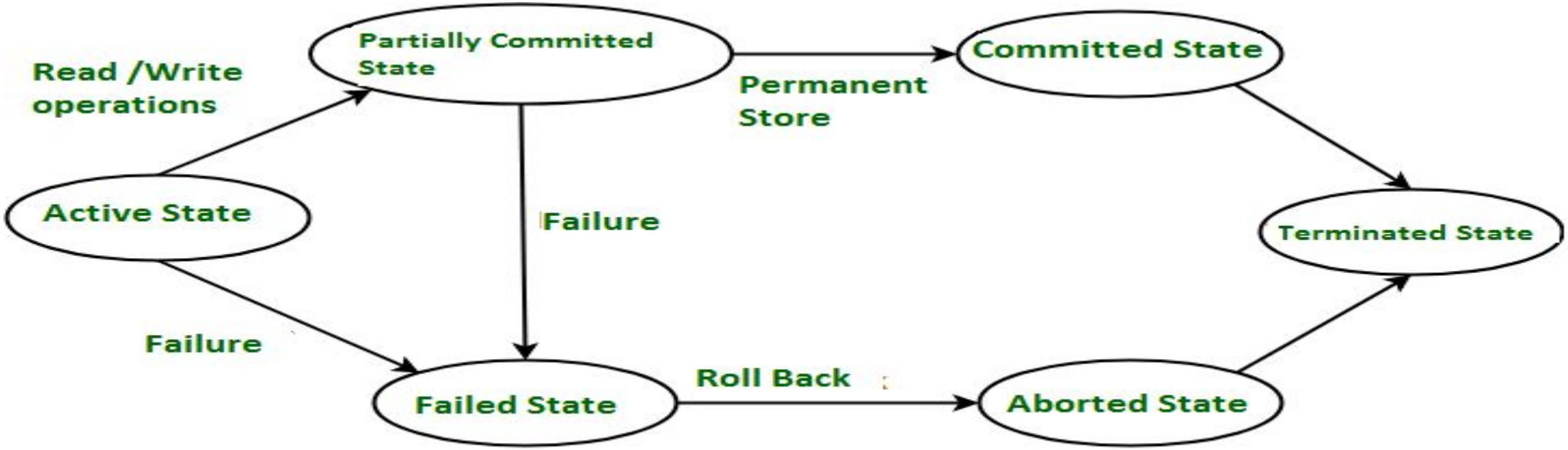
1. Cascadeless schedules are stricter than recoverable schedules or are a subset of recoverable schedules.
2. Strict schedules are stricter than cascadeless schedules or are a subset of cascadeless schedules.
3. Serial schedules satisfy constraints of all recoverable, cascadeless and strict schedules and hence is a subset of strict schedules.

The relation between various types of schedules can be depicted as:



Transaction States

States through which a transaction goes during its lifetime. These are the states which tell about the current state of the Transaction and also tell how we will further do processing we will do on the transactions. These states govern the rules which decide the fate of the transaction whether it will commit or abort.



Transaction States in DBMS

Active State – When the instructions of the transaction is running then the transaction is in active state. If all the read and write operations are performed without any error then it goes to “partially committed state”, if any instruction fails it goes to “failed state”.

Partially Committed – After completion of all the read and write operation the changes are made in main memory or local buffer. If the changes are made permanent on the Data Base then state will change to “committed state” and in case of failure it will go to “failed state”.

Failed State – When any instruction of the transaction fails it goes to “failed state” or if failure occurs in making permanent change of data on Data Base.

Aborted State – After having any type of failure the transaction goes from “failed state” to “aborted state” and in before states the changes are only made to local buffer or main memory and hence these changes are deleted or rollback.

Committed Stage – It is the stage when the changes are made permanent on the Data Base and transaction is complete and therefore terminated in “terminated state”.

Terminated State – If there is any roll back or the transaction come from “committed state” then the system is consistent and ready for new transaction and the old transaction is terminated.

Introduction to Lock Management

*The part of the DBMS that keeps track of the locks issued to transactions is called the **lock manager**.*

- *The **lock manager** maintains an entry for each object identifier in a **lock table**.*
- ***DBMS** maintains a entry for each transaction in a **transaction table**.*

LOCK TABLE

Object-Id	No of Txn's holding lock	Nature of Lock	Queue of Lock Requests
123 [EMP]	6	Read (or) Shared	T4→T5→T6
234 [Dept]	1	Write (or) Exclusive	T5→T6
666 [JOB]	0		

TRANSACTION TABLE

TXN_ID	List of Locks held by Txn
T1	R(EMP),W(DEPT)
T2	R(EMP)
T3	

Deadlock & Specialize Locking

Dealing with Deadlocks

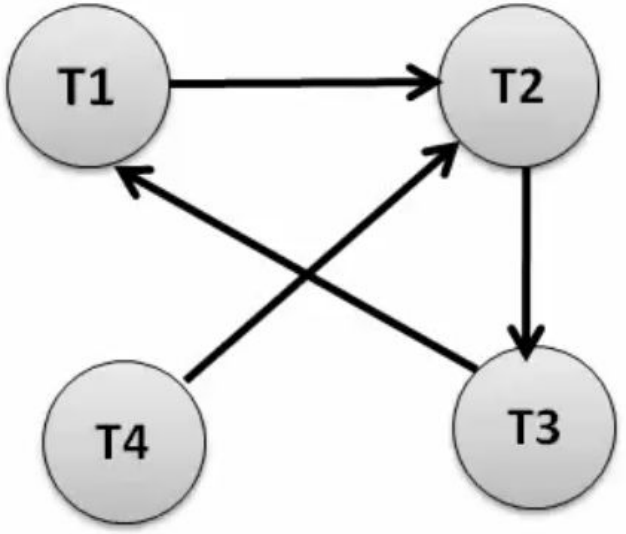
- The lock manager maintains a structure called a **waits-for graph** to detect deadlock cycles.
- Waits-for graph can be drawn using the **following steps**:-

1. There is a **node** for each **active transaction**.
2. There is an **arc** from T_i to T_j, if and only if, T_i is waiting for T_j to release the lock.

The lock manager **adds edges** to this graph when it queues lock requests and **removes** edges when it grants lock requests.

T1	T2	T3	T4
S(A)			
R(A)			
	X(B)		
	W(B)		
S(B)			
		S(C)	
		R(C)	
	X(C)		
			X(B)
		X(A)	

Schedule Illustrating Deadlock



Previous GATE Questions

Consider the following schedule:-

T_3	T_4
Lock→ X(B)	
Read(B)	
B:=B-50	
Write(B)	
	Lock→ S(A)
	Read(A)
	Lock → S(B)
Lock → X(A)	

a. No deadlock

b. Results in deadlock

c. T_4 is holding a shared-mode lock on A thus T_3 is granted exclusive mode lock on B

d. T_3 is holding an exclusive-mode lock on B thus T_4 is shared-mode lock on B

Previous GATE Questions

Consider the following schedule:-

T_3	T_4
Lock→ X(B)	
Read(B)	
B:=B-50	
Write(B)	
	Lock→ S(A)
	Read(A)
	Lock → S(B)
Lock → X(A)	

a. No deadlock

b. Results in deadlock

c. T_4 is holding a shared-mode lock on A thus T_3 is granted exclusive mode lock on B

d. T_3 is holding an exclusive-mode lock on B thus T_4 is shared-mode lock on B

Previous GATE Questions

Consider the following situation:-

- Transaction T_{25} is waiting for transaction T_{26} and T_{27}
- Transaction T_{27} is waiting for transaction T_{26}
- Transaction T_{26} is waiting for transaction T_{28}

Above situation may result into:-

a. Data not sufficient
b. No deadlock state
c. Deadlock State
d. Starvation of T25

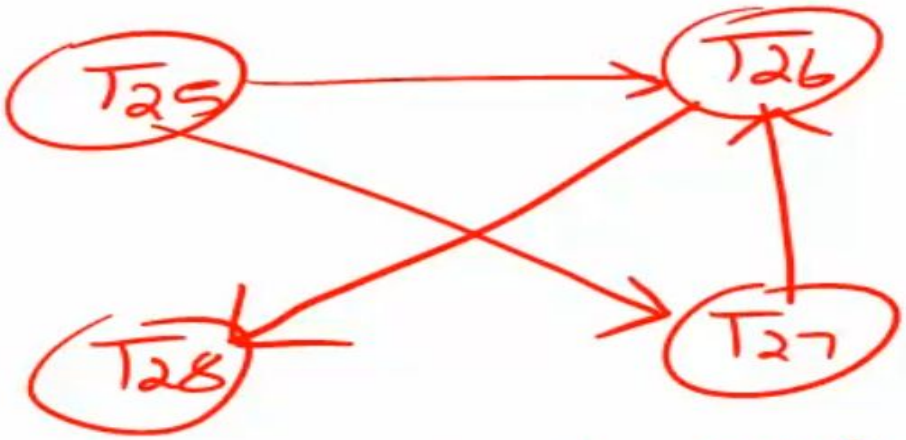
Previous GATE Questions

Consider the following situation:-

- Transaction T_{25} is waiting for transaction T_{26} and T_{27}
- Transaction T_{27} is waiting for transaction T_{26}
- Transaction T_{26} is waiting for transaction T_{28}

Above situation may result into:-

a. Data not sufficient
b. No deadlock state
c. Deadlock State
d. Starvation of T25



NO Cycle ⇒ NO Deadlock

Dealing with Deadlocks

Deadlock Prevention

- *We can prevent deadlocks by giving each transaction a **priority** to each transaction.*

- *If a transaction T_i requests a lock and transaction T_j holds a conflicting lock, the lock manager can use one of the following two policies:*

1. ***Wait-die**: If T_i has higher priority, it is allowed to wait; otherwise, it is aborted.*
2. ***Wound-wait**: If T_i has higher priority, abort T_j ;otherwise, T_i waits.*

Shared – Exclusive Locking

- Shared Lock(S):- if transaction locked data items in shared mode then allowed to read only
- Exclusive lock(X):- if transaction locked data items in shared mode then allowed to read and write both

T1	T2
S(A)	
R(A)	
U(A)	
	X(A)
	R(A)
	W(A)
	U(A)

Shared – Exclusive Locking

- Problem in shared/exclusive locking:

1. May not free from irrecoverability
2. May not be free from deadlock
3. May not be free from starvation

T1	T2
X(A)	
	X(B)
X(B)	
	X(A)

Deadlock

Shared – Exclusive Locking

- Problem in shared/exclusive locking:

T1	T2
X(A)	
R(A)	
W(A)	
U(A)	
	X(A)
	R(A)
	W(A)
	U(A)
	COMMIT
*	

IRRECOVERABLE

T1	T2	T3	T4
	S(A)		
X(A)	…		
	…	S(A)	
	U(A)	..	S(A)
		U(A)	…
			U(A)

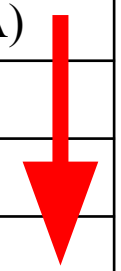
STARVATION

2 –Phase locking

- Growing phase:- locks are acquired and no locks are released
- Shrinking phase:- locks are released and no locks are acquired

T1	T2
X(A)	
R(A)	
W(A)	
	S(C)
	R(C)
	W(C)
S(B)	
R(B)	
U(A)	
U(B)	
	U(C)

T1	T2
S(A)	
	X(A)
X(B)	
*	
U(A)	
	X(D)
U(B)	
	U(A)
	U(B)



T1---->T2

2 –Phase locking

- Advantage :- always ensures serializability
- Drawback:-

1. May not free from irrecoverability
2. May not be free from deadlock
3. May not be free from starvation
4. May not be free from cascading rollback

2 –Phase locking

T1	T2	T3	T4
	S(A)		
X(A)	…		
	…	S(A)	
	U(A)	..	S(A)
		U(A)	…
			U(A)

STARVATION

T1	T2
X(A)	
	X(B)
X(B)	
	X(A)

DEADLOCK

2 –Phase locking

T1	T2	T3	T4
X(A)			
R(A)			
W(A)			
U(A)			
	S(A)		
	R(A)		
		S(A)	
		R(A)	
			S(A)
			R(A)
*			

IRRECOVERABLE & CASCADING ROLLBACK

2 –Phase locking

- Strict 2PL:

It should satisfy the basic 2PL and all exclusive locks should hold until commit/abort

- Rigorous 2PL:

It should satisfy the basic 2PL and all shared, exclusive locks should hold until commit/abort

2 –Phase locking

T1	T2	T3
X(A)		
R(A)		
W(A)		
U(A)		
	S(A)	
	R(A)	
		S(A)
		R(A)
*		

T1	T2
X(A)	
R(A)	
W(A)	
	S(A)
	R(A)
Commit	
U(A)	

2 –Phase locking

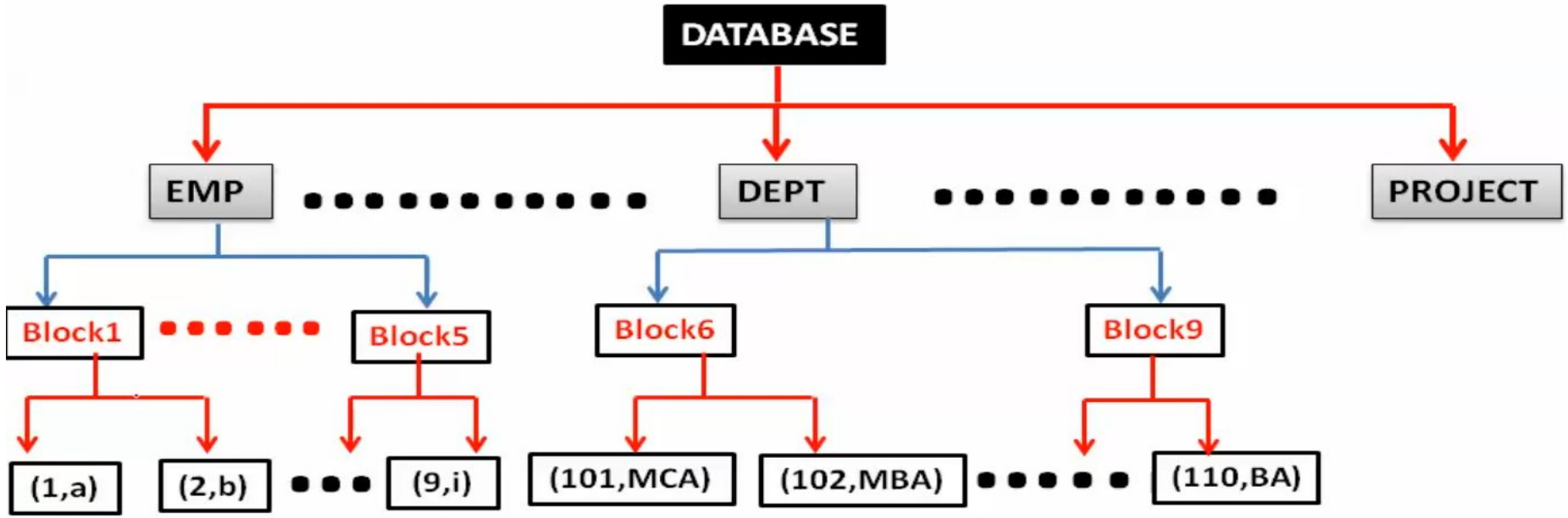
T1	T2
X(A)	
R(A)	
W(A)	
	S(A)
	R(A)
	…
	…
	…
Commit	
U(A)	

Multiple-Granularity Locking

- A database contains several ***files***, → Every table will be treated as a file
- a file is a collection of ***pages***, and → Every table has n-blocks (or) pages to store rows
- a page is a collection of ***records***. → 3 (or) m rows will be stored in a single block.

Locking → 3 options

1. File Level Lock (or) Table Level Lock
2. Page Level Lock
3. Record (or) Row Level Lock



Multiple-Granularity Locking

1. *Shared Lock* **[S]**
2. *Exclusive Lock* **[X]**

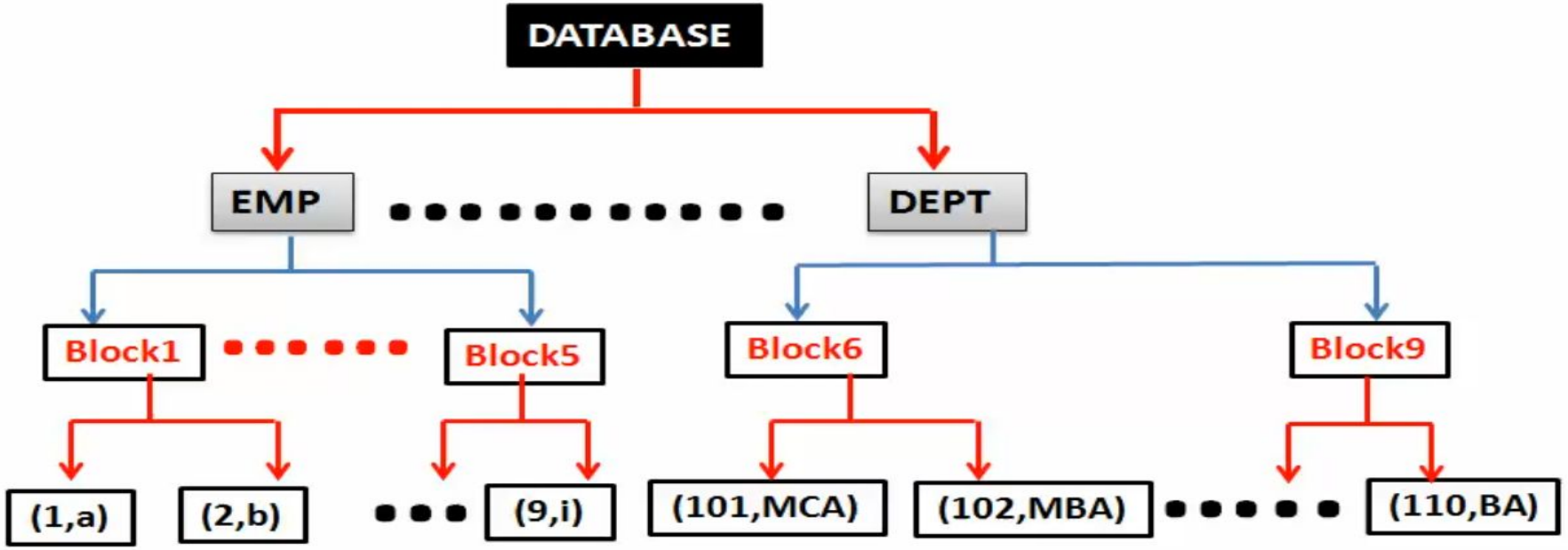
*Locks used to lock the **objects***

3. *Intention Shared* **[IS]**
4. *Intention Exclusive* **[IX]**

*Locks used to lock the **ancestors** of the locked object.*

- ❖ **IS** locks conflict only with **X** locks.
- ❖ **IX** locks conflict with **S** and **X** locks.

- ❖ *Locks must be **acquired** from **root-to-leaf order.***
- ❖ *Locks must be **released** in **leaf-to-root order.***



Two-phase Commit protocol

- The steps performed in the two phases are as follows −
- **Phase 1: Prepare Phase**
- After each slave has locally completed its transaction, it sends a “DONE” message to the controlling site. When the controlling site has received “DONE” message from all slaves, it sends a “Prepare” message to the slaves.
- The slaves vote on whether they still want to commit or not. If a slave wants to commit, it sends a “Ready” message.
- A slave that does not want to commit sends a “Not Ready” message. This may happen when the slave has conflicting concurrent transactions or there is a timeout.

Two-phase Commit protocol

- **Phase 2: Commit/Abort Phase**
- After the controlling site has received “Ready” message from all the slaves −
  - The controlling site sends a “Global Commit” message to the slaves.
  - The slaves apply the transaction and send a “Commit ACK” message to the controlling site.
  - When the controlling site receives “Commit ACK” message from all the slaves, it considers the transaction as committed.

Two-phase Commit protocol

- After the controlling site has received the first “Not Ready” message from any slave –

 The controlling site sends a “Global Abort” message to the slaves.

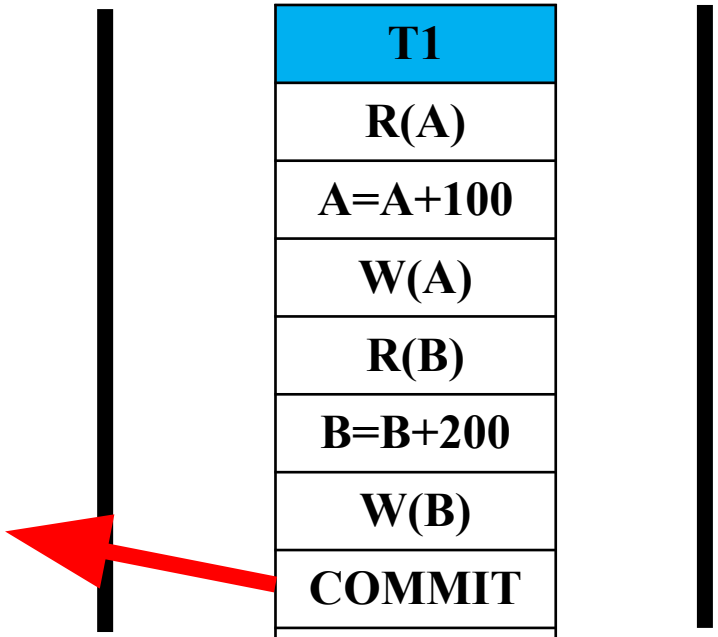
 The slaves abort the transaction and send a “Abort ACK” message to the controlling site.

 When the controlling site receives “Abort ACK” message from all the slaves, it considers the transaction as aborted.

Log based recovery (deferred database modification)

A = 100
B = 200

A = 200
B = 400



T1
R(A)
A=A+100
W(A)
R(B)
B=B+200
W(B)
COMMIT
*

Log
<T1,START>
<T1,A,200>
<T1,B,400>
<T1,COMMIT>

<TRANSACTION NUMBER, OBJECT, NEW VALUE>

REDO – NEW VALUE WILL BE UPDATED IN PERMANENT STORAGE

Log based recovery (deferred database modification)

A = 100
B = 200

T1
R(A)
A=A+100
W(A)
R(B)
B=B+200
W(B)
*

Log
<T1,START>
<T1,A,200>
<T1,B,400>

<TRANSACTION NUMBER, OBJECT, NEW VALUE>

NO OPERATION WILL BE DONE

Log based recovery (deferred database modification)

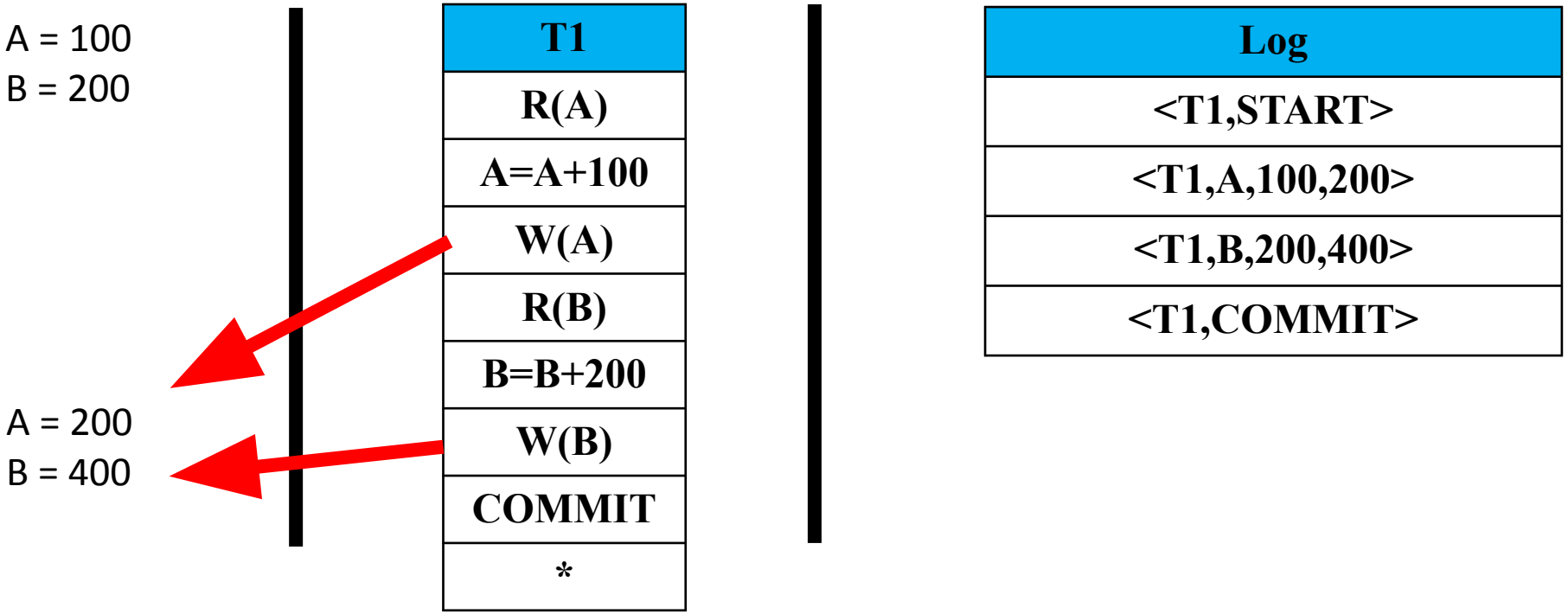
A = 100
B = 200
C = 500

Log
<T1,START>
<T1,A,200>
<T1,B,400>
<T1,COMMIT>
<T2,START>
<T2,C,500>
*

A = 200
B = 400
C = 500

T1 WILL BE REDO, FOR T2 NO OPERATION WILL BE DONE AS IT IS NOT COMMITED

Log based recovery (Immediate database modification)

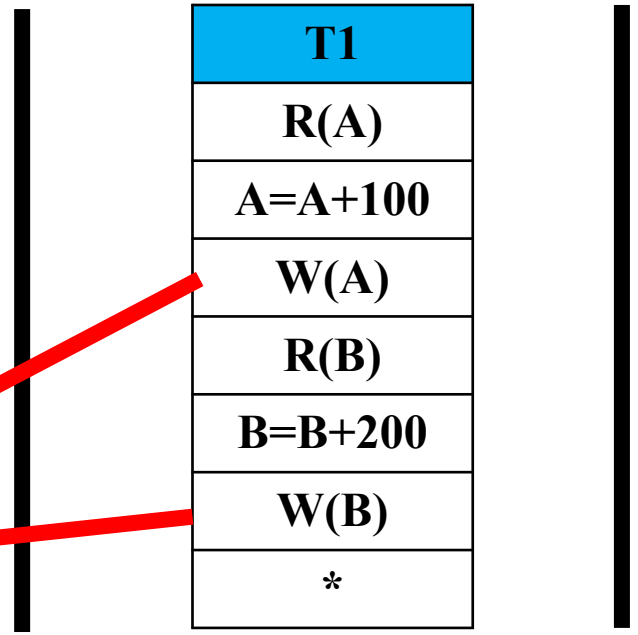


<TRANSACTION NUMBER, OBJECT,OLD VALUE, NEW VALUE>

REDO – NEW VALUE WILL BE UPDATED IN PERMANENT STORAGE

Log based recovery (Immediate database modification)

A = 100
B = 200



T1
R(A)
A=A+100
W(A)
R(B)
B=B+200
W(B)
*

A = 200
B = 400

Log
<T1,START>
<T1,A,100,200>
<T1,B,200,400>

<TRANSACTION NUMBER, OBJECT,OLD VALUE, NEW VALUE>

UNDO – OLD VALUE WILL BE UPDATED IN PERMANENT STORAGE

A = 100
B = 200

Log based recovery (Immediate database modification)

A = 100
B = 200
C = 500

Log
<T1,START>
<T1,A,100,200>
<T1,B,200,400>
<T1,COMMIT>
<T2,START>
<T2,C,500,600>
*

A = 200
B = 400
C = 500

T1 WILL BE REDO, T2 WILL BE UNDO

Timestamp ordering protocol

Rules for timestamp ordering protocol

1) Transaction Ti issue a Read(A) operation
   A) if WTS(A) > TS(Ti), then Rollback Ti
   B) Otherwise execute R(A) operation,
      set RTS(A)=Max{RTS(A),TS(Ti)}
2) Transaction Ti issue a Write(A) operation
   A) if RTS(A) > TS(Ti), then Rollback Ti
   B) if WTS(A) > TS(Ti), then Rollback Ti
   C) Otherwise execute write(A) operation,
      set WTS(A)=TS(Ti)

Timestamp-Based Concurrency Control

Which of the following Schedule is **Valid** as per **Time Stamp Protocol** ?

a.

S1	
T1	T2
R(A)	
	W(A)
	R(B)
W(B)	
	R(C)
W(C)	

b.

S2	
T1	T2
R(A)	
	W(A)
W(B)	
	R(B)
	R(C)
W(C)	

c.

S3	
T1	T2
	W(A)
R(A)	
W(B)	
	R(B)
	R(C)
W(C)	

d.

S4	
T1	T2
	W(A)
R(A)	
	R(B)
W(B)	
	R(C)
W(C)	

Timestamp-Based Concurrency Control

Which of the following Schedule is **Valid** as per **Time Stamp Protocol** ?

a.

S1	
T1	T2
R(A)	
	W(A)
	R(B)
W(B)	
	R(C)
W(C)	

b.

S2	
T1	T2
R(A)	
	W(A)
W(B)	
	R(B)
	R(C)
W(C)	

c.

S3	
T1	T2
	W(A)
R(A)	
W(B)	
	R(B)
	R(C)
W(C)	

d.

S4	
T1	T2
	W(A)
R(A)	
	R(B)
W(B)	
	R(C)
W(C)	

Answer = d, Since "T2" started first and accessed A, B, and C items before "T1".