UNIT –IV

Transaction Processing
ACID Properties
States of Transaction
Types of Schedules
Implementation Of Atomicity And Durability
Recoverability
Implementation Of Isolation

1.Transaction Processing

•A transaction is a logical unit of work of database processing that includes one or more database access operations.

•A transaction can be defined as **an action or series of actions that is carried out by a single user or application program to perform operations for accessing the contents of the database.** The operations can include **retrieval, insertion, deletion and modification**

•A transaction must be either **completed or aborted**.

•A transaction is a program unit whose execution may change the contents of a database. It can either be embedded within an application program or can be specified interactively via a high level **query language such as SQL.**

•If the database is in a **consistent state before** a transaction executes, then the database should still be in **consistent state after** its execution.

•Therefore, to ensure these conditions and preserve the integrity of the database **transaction must be atomic** (also called serializability).

•**Atomic transaction** is a transaction in which either **all actions associated with the transaction** are executed to completion or none are performed.

Basic operations on database are read and write

1. read_item(X): Reads a database item named X into a program variable. To simplify our notation, we assume that the program variable is also named X.

2. write_item(X): Writes the value of program variable X into the database item named X

Example:

❖You are working on a system for a bank. A customer goes to the ATM and instructs it to transfer Rs. 1000 from savings to a checking account.

❖This simple transaction requires two steps:

- **Subtracting the money** from the savings account balance.

Savings -1000

- **Adding the money** to the checking account balance.

Checking + 1000

❖The code to create this transaction will require **two updates** to the database.
❖For example, there will be two SQL statements: one **UPDATE command to decrease the balance in savings** and a second **UPDATE command to increase the balance in the checking account**.
❖You have to consider what would happen **if a machine crashed between these two operations.** The money has already been subtracted from the savings account will not be added to the checking account. It is lost.
❖You might consider performing the addition to checking first, but then the customer ends up with extra money, and the bank loses.
❖The point is that both changes must be made successfully.
❖Thus, a transaction is defined as a set of changes that must be made together.

There are **several reasons for a transaction to fail** in the middle of execution.

1. Computer failure: a hardware, software or network error occurs in the computer system during transaction execution.

2. Transaction or system: some operations in the transaction may cause it to fail such as integer overflow or division by 0. The user may also interrupt the transaction during its execution

3. Local errors or exception conditions detected by the transaction :during transaction execution , certain condition may occur that necessitate the cancellation of transaction.

Example: insufficient account balance in a banking database may cause a transaction to be cancelled.

4. Concurrency control enforcement: this method may decide to abort the transaction because several transactions are in a state of deadlock.

5. Disk failure: some disk blocks may lose their data because of a disk read/write head crash. This may happen during a read/ write operation of a transaction.

6. Physical problem & catastrophes: this refers to an endless list of problems that include fire, theft etc.

2. ACID PROPERTIES

- ACID properties are used for maintaining the integrity of database during transaction processing.
- ACID stands for Atomicity, Consistency, Isolation, and Durability.

- Atomicity(All or nothing):-

- requires that all operations of a transaction be completed, if not, the transaction is aborted
- In other words, a transaction is treated as single, individual logical unit of work.

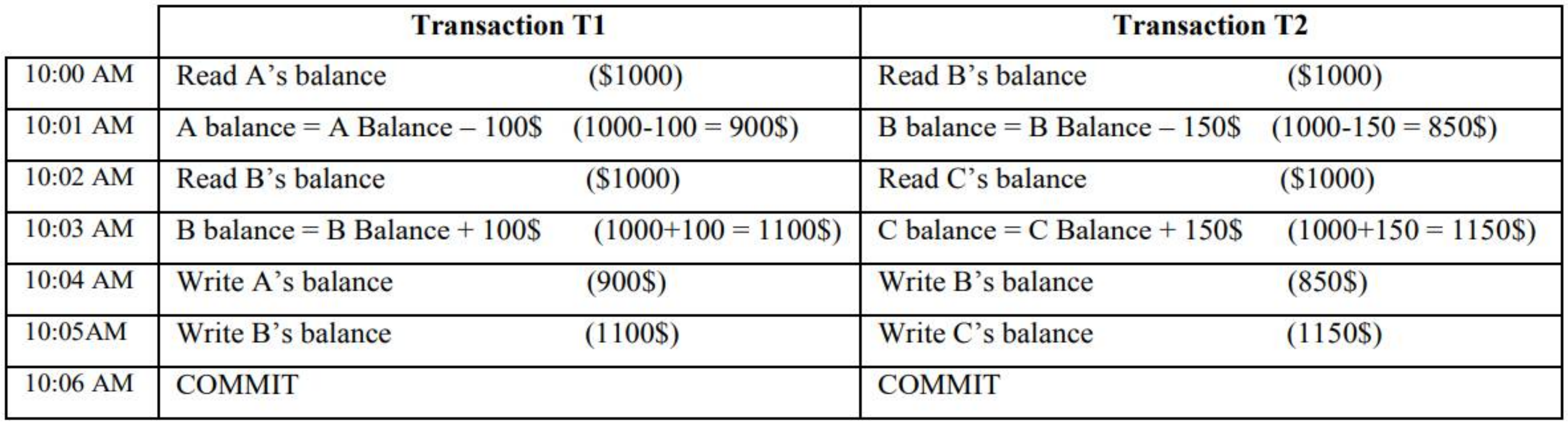
- Example: Transferring $100 from account A to account B. (Assume initially, account A balance = $400 and account B balance = 700$.).
- Transferring $100 from account A to account B has two operations
- a) Debiting 100$ from A's balance ($400 -$100 = $300)
- b) Crediting 100$ to B's balance ($700+$100 = $800)
- Let's say first operation (a) passed successfully while second (b) failed, in this case A's balance would be 300$ while B would be having 700$ instead of 800$. This is unacceptable in a banking system.
- Either the transaction should fail without executing any of the operation or it should process both the operations. The Atomicity property ensures that.

- Consistency(No violation of integrity constraints):-
- every transaction sees a consistent database instance.
- In other words, execution of a transaction must leave a database in either its prior stable state or a new stable state that reflects the new modifications (updates) made by the transaction.
- For example, transferring funds from one account to another, the consistency property ensures that the total values of funds in both the accounts is the same before and end of the transaction.
- ✓ i.e., Assume initially, A balance = $400 and B balance = 700$.
- ✓ The total balance of A + B = 1100$ (Before transferring 100$ from A to B)
- ✓ The total balance of A + B = 1100$ (After transferring 100$ from A to B)

- Isolation(concurrent changes invisibles):-
- the data used during the execution of a transaction cannot be used by a second transaction until the first one is completed.
- This property isolates transactions from one another. In other words, if a transaction T1 is being executed and is using the data item X, that data item cannot be accessed by any other transaction (T2…Tn) until T1 ends.

For example,

- Transaction T1: Transfer 100$ from account A to account B
- Transaction T2: Transfer 150$ from account B to account C
- Assume initially, A balance = B balance = C balance = $1000



	Transaction T1		Transaction T2	
10:00 AM	Read A's balance	($1000)	Read B's balance	($1000)
10:01 AM	A balance = A Balance – 100$	(1000-100 = 900$)	B balance = B Balance – 150$	(1000-150 = 850$)
10:02 AM	Read B's balance	($1000)	Read C's balance	($1000)
10:03 AM	B balance = B Balance + 100$	(1000+100 = 1100$)	C balance = C Balance + 150$	(1000+150 = 1150$)
10:04 AM	Write A's balance	(900$)	Write B's balance	(850$)
10:05AM	Write B's balance	(1100$)	Write C's balance	(1150$)
10:06 AM	COMMIT		COMMIT	

- After completion of Transaction T1 and T2, A balance = 900$, B balance = 1100$, C balance =1150$. But B balance should be 950$. The B balance is wrong due to execution of T1 and T2 parallel and in both the transactions, Account B is common. The last write in account B is at 10:05 AM, so that B balance is 1100$ (write in account B at 10:04 AM is overwritten).

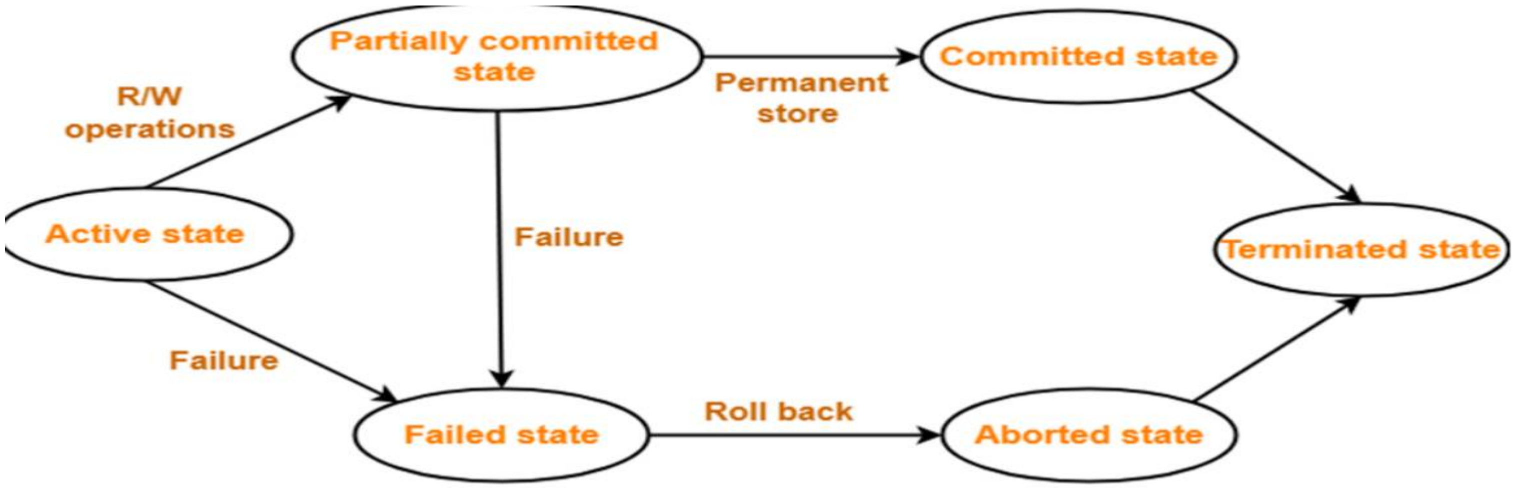
- Durability(committed update persist):-

- It states that the changes made by a transaction are permanent(the database reaches a consistent state).
- They cannot be lost by either a system failure or by the erroneous operation of a faulty transaction.

- For example, assume account A balance = 1000$. If A withdraw 100$ today, then the A balance = 900$. After two days or a month, A balance should be 900$, if no other transactions done on A

3. STATES OF TRANSACTION

A transaction goes through many different states throughout its life cycle. These states are called as transaction states. They are:



Active State:

- This is the first state in the life cycle of a transaction.
- Once the transaction starts executing, then it is said to be in active state.
- During this state it performs operations like READ and WRITE on some data items. All the changes made by the transaction are now stored in the buffer in main memory. They are not updated in database.
- From active state, a transaction can go into either a partially committed state or a failed state.

Partially Committed State:

- When the transaction executes its last statement, then the transaction is said to be in partially committed state.
- Still, all the changes made by the transaction are stored in the buffer in main memory, but they are not updated in the database.
- From partially committed state, a transaction can go into one of two states, a committed state or a failed state

Committed State:

- After all the changes made by the transaction have been successfully updated in the database, it enters into a committed state and the transaction is considered to be fully committed.
- After a transaction has entered the committed state, it is not possible to roll back (undo) the transaction. This is because the system is updated into a new consistent state and the changes are made permanent.
- The only way to undo the changes is by carrying out another transaction called as compensating transaction that performs the reverse operations.

Failed State:

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

Aborted State:

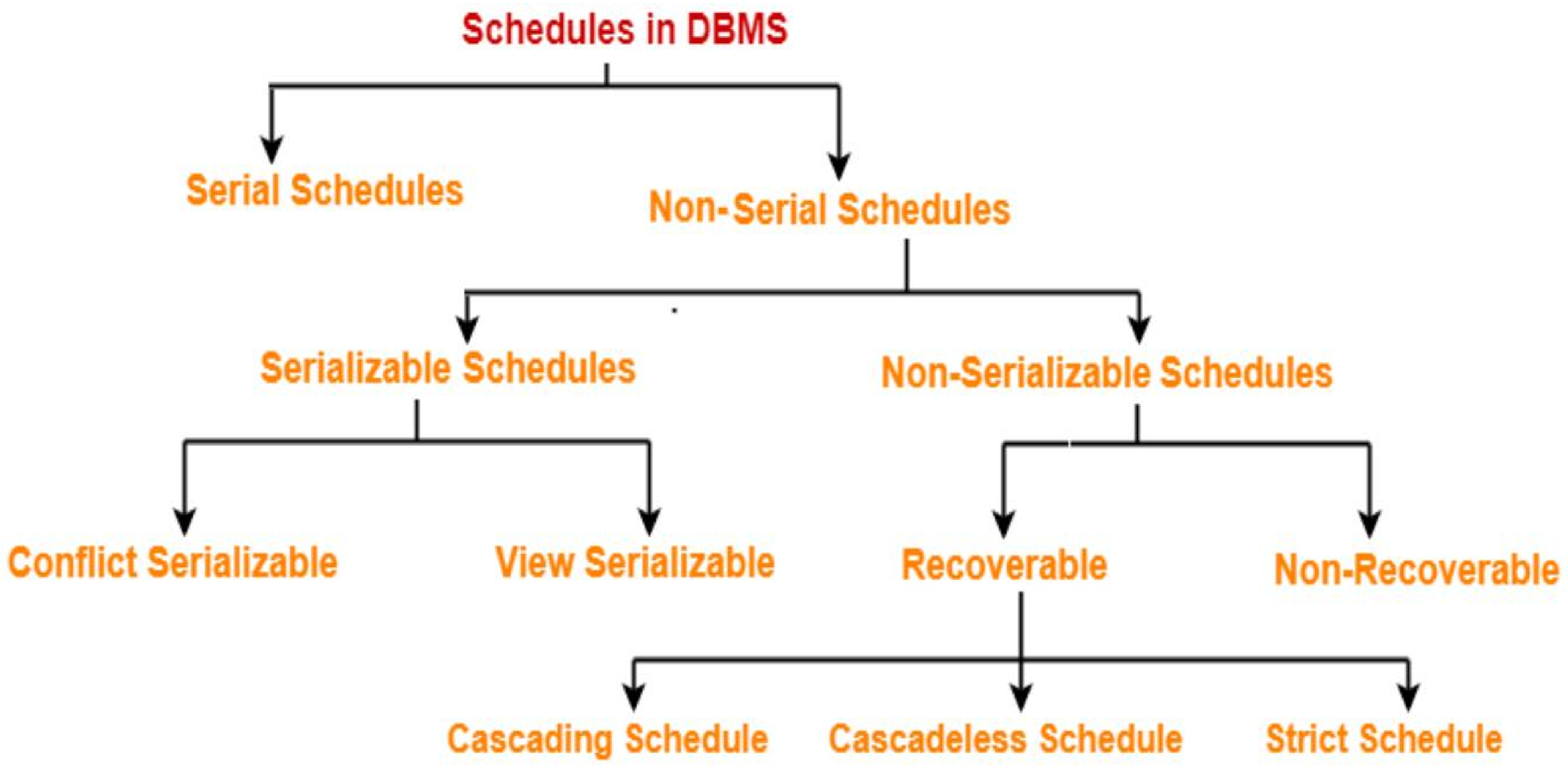
- After the transaction has failed and entered into a failed state, all the changes made by it have to be undone.
- To undo the changes made by the transaction, it becomes necessary to roll back the transaction.
- After the transaction has rolled back completely, it enters into an aborted state.

Terminated State:

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

4. TYPES OF SCHEDULES –

SERIALIZABILITY In DBMS, schedules may be classified as

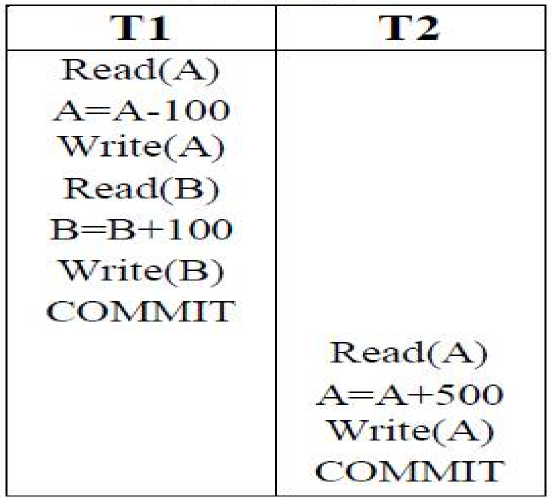


Serial Schedules:

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

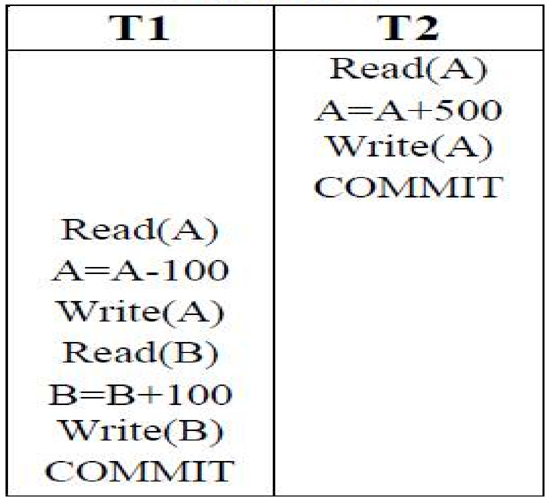
Examples:

Schedule-1



T1	T2
Read(A)	
A=A-100	
Write(A)	
Read(B)	
B=B+100	
Write(B)	
COMMIT	
	Read(A)
	A=A+500
	Write(A)
	COMMIT

Schedule-2



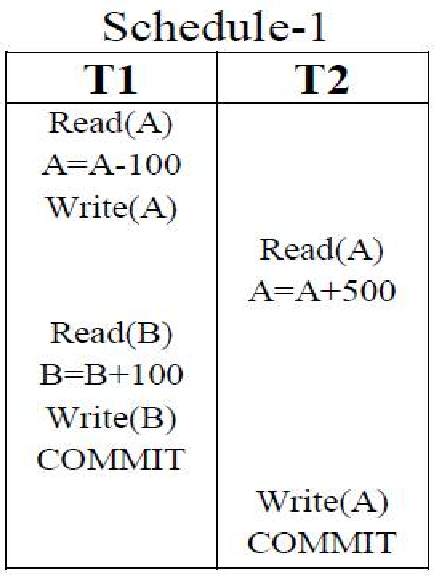
T1	T2
	Read(A)
	A=A+500
	Write(A)
	COMMIT
Read(A)	
A=A-100	
Write(A)	
Read(B)	
B=B+100	
Write(B)	
COMMIT	

- ❖In schedule 1, after T1 completes its execution, transaction T2 executes. So, schedule-1 is a *Serial Schedule*.
- ❖Similarly, in schedule-2, after T2 completes its execution, transaction T1 executes. So, schedule -2 is also an example of a *Serial Schedule*.

Non-Serial Schedules:

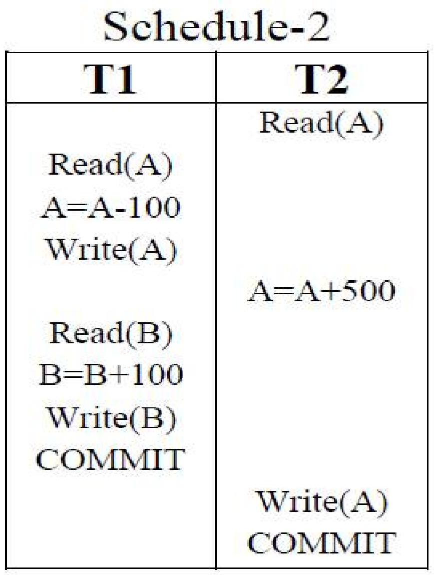
- ❖ In non-serial schedules, multiple transactions execute concurrently.
- ❖ Operations of all/some of the transactions are inter-leaved or mixed with each other.
- ❖ Some non-serial schedules may lead to inconsistency of the database and may produce wrong results

Examples:



Schedule-1

T1	T2
Read(A)	
A=A-100	
Write(A)	
	Read(A)
	A=A+500
Read(B)	
B=B+100	
Write(B)	
COMMIT	
	Write(A)
	COMMIT



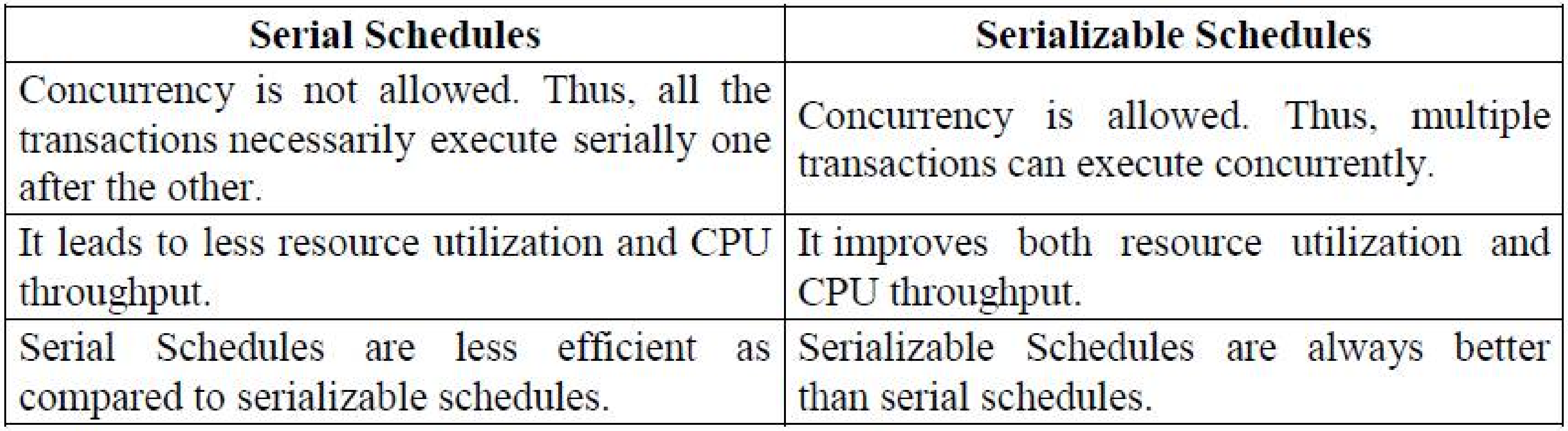
Schedule-2

T1	T2
	Read(A)
Read(A)	
A=A-100	
Write(A)	
	A=A+500
Read(B)	
B=B+100	
Write(B)	
COMMIT	
	Write(A)
	COMMIT

- ❖ In schedule-1 and schedule-2, the two transactions T1 and T2 executing concurrently. The operations of T1 and T2 are interleaved. So, these schedules are **Non-Serial Schedule**.

Serializable Schedules:

- ❖ A non-serial schedule of ‘n’ transactions is equivalent to some serial schedule of ‘n’ transactions, then it is called as a **serializable schedule**.
- ❖ In other words, the results produced by the transactions in a serial schedule are equal to the result produced by the same transactions in some non-serial schedule, then that non-serial schedule is called as serializability.
- ❖ Serializable schedules behave exactly same as serial schedules.
- ❖ Even though, Serial Schedule and Serializable Schedule produce same result, there are some differences they are



Serial Schedules	**Serializable Schedules**
Concurrency is not allowed. Thus, all the transactions necessarily execute serially one after the other.	Concurrency is allowed. Thus, multiple transactions can execute concurrently.
It leads to less resource utilization and CPU throughput.	It improves both resource utilization and CPU throughput.
Serial Schedules are less efficient as compared to serializable schedules.	Serializable Schedules are always better than serial schedules.

Serializability is mainly of two types. They are:

- ❖ Conflict Serializability
- ❖ View Serializability

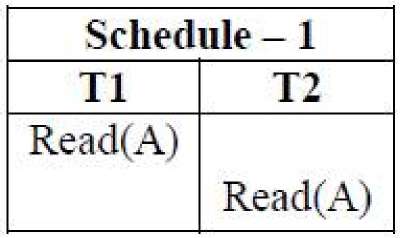
Conflict Serializability:

- ➢ If a given non-serial schedule can be converted into a serial schedule by swapping its non-conflicting operations, then it is called as a **conflict serializable schedule**.
- ➢ Two operations are called as **conflicting operations** if all the following conditions hold true

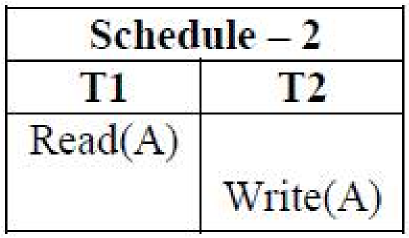
(1) Both the operations belong to different transactions

(2) Both the operations are on the **same data item**

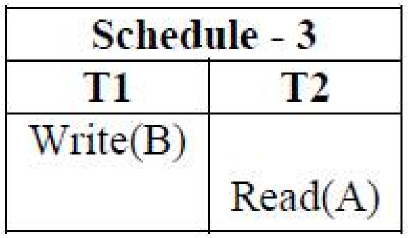
(3) At least one of the two operations is a write operation



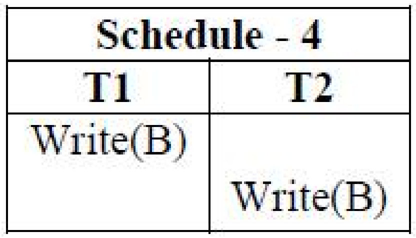
Schedule – 1	
T1	**T2**
Read(A)	
	Read(A)



Schedule – 2	
T1	**T2**
Read(A)	
	Write(A)



Schedule - 3	
T1	**T2**
Write(B)	
	Read(A)



Schedule - 4	
T1	**T2**
Write(B)	
	Write(B)

In Schedule -1, only rule (1) & (2) are true, but rule (3) is not holding. So, the operations are not conflict.

In Schedule -2, rule (1), (2) & (3) are true. So, the operations are conflict.

In Schedule -3, only rule (1) & (3) are true, but rule (2) is not holding. So, the operations are not conflict.

In Schedule -4, rule (1), (2) & (3) are true. So, the operations are conflict.

Testing of Conflict Serializability:

❖ Precedence Graph is used to test the Conflict Serializability of a schedule. The algorithm to draw precedence graph is

(1) Draw a node for each transaction in Schedule **S.**

(2) If T_a reads X value **written by T_b**, then draw arrow from $T_b \rightarrow T_a$.

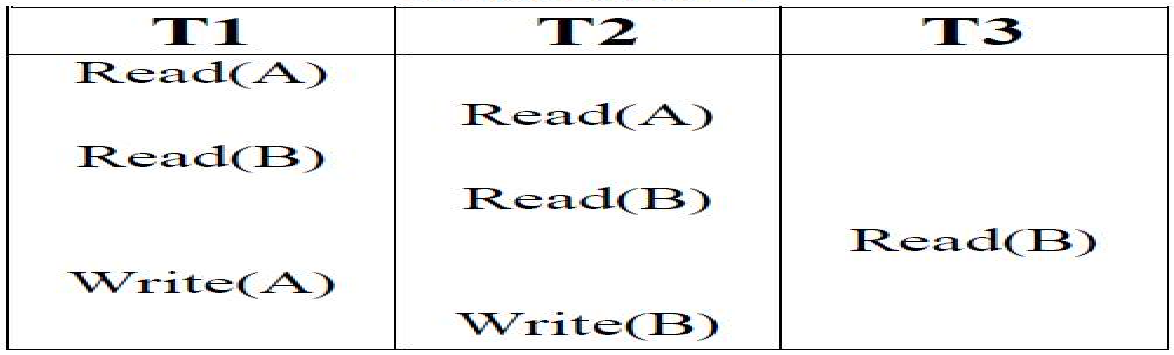
(3) If T_b writes X value **after** it has been **read by T_a**, then draw arrow from $T_a \rightarrow T_b$.

(4) If T_a writes X **after T_b writes X**, then draw arrow from $T_b \rightarrow T_a$.

❖ If the precedence graph has **no cycle**, then Schedule **S** is known as **conflict serializable**. If a precedence graph contains **a cycle**, then **S** is **not conflict serializable.**

Problem-01: Check whether the given schedule S is conflict serializable or not.
S : R1(A) , R2(A) , R1(B) , R2(B) , R3(B) , W1(A) , W2(B)
Solution:
Given that S : R1(A) , R2(A) , R1(B) , R2(B) , R3(B) , W1(A) , W2(B) .
The schedule for the above operations is

Schedule-1

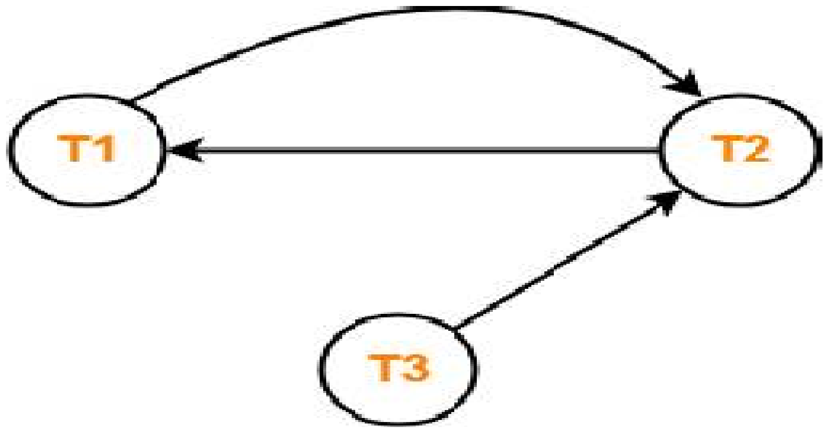


T1	T2	T3
Read(A)		
	Read(A)	
Read(B)		
	Read(B)	
		Read(B)
Write(A)		
	Write(B)	

List all the conflicting operations and determine the dependency between the transactions
(Thumb rule to find conflict operations: For each Write(X) in Ta, make a pair with each Read(X) and Write(X) in Tb. **The order is important in each pair** i.e., for example, Read after Write on X or write after read on X in the given schedule.**)**

❖ R2(A) , W1(A) (T2 → T1)
❖ R1(B) , W2(B) (T1 → T2)
❖ R3(B) , W2(B) (T3 → T2)

Draw the precedence graph:



There exists a cycle in the above graph. Therefore, the schedule S is not conflict serializable.

Problem-02: Check whether the given schedule S is conflict serializable schedule.

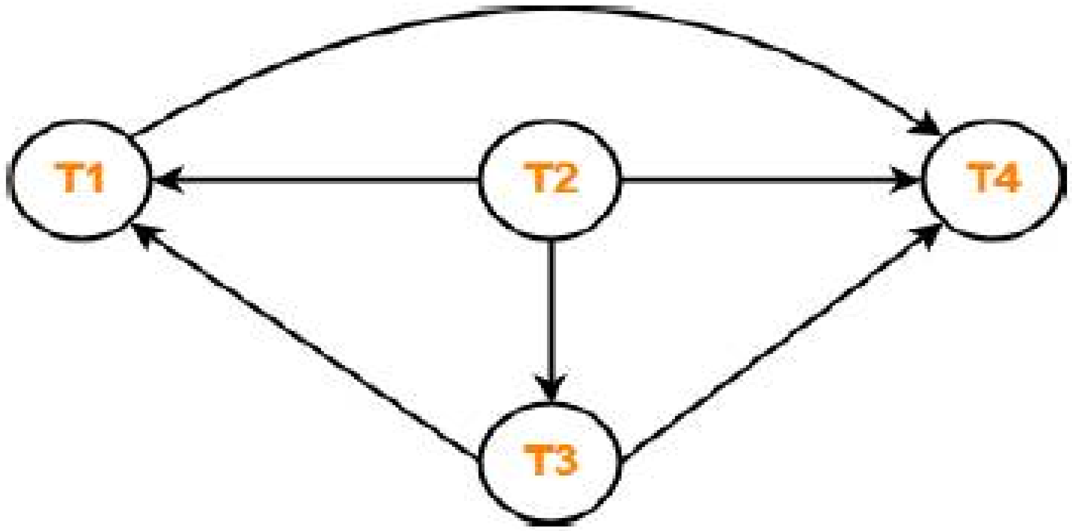
Schedule – S

T1	T2	T3	T4
	Read(X)		
		Write(X)	
		COMMIT	
Write(X)			
COMMIT			
	Write(Y)		
	Read(Z)		
	COMMIT		
			Read(X)
			Read(Y)
			COMMIT

Solution: List all the conflicting operations to determine the dependency between transactions.

$R_2(X)$, $W_3(X)$	$(T_2 \longrightarrow T_3)$
$W_3(X)$, $W_1(X)$	$(T_3 \longrightarrow T_1)$
$W_3(X)$, $R_4(X)$	$(T_3 \longrightarrow T_4)$
$R_2(X)$, $W_1(X)$	$(T_2 \longrightarrow T_1)$
$W_1(X)$, $R_4(X)$	$(T_1 \longrightarrow T_4)$
$W_2(Y)$, $R_4(Y)$	$(T_2 \longrightarrow T_4)$

Draw the precedence graph:



There exists no cycle in the precedence graph. Therefore, the schedule S is conflict serializable

View Serializability:

View Serializability Definition: If a given schedule is view equivalent to some serial schedule, then it is called as a view serializable schedule.

Two schedules S1 and S2 are said to be **view equivalent** if both of them satisfy the following three rules:

(1) **Initial Read:** The first read operation on each data item in both the schedule must be same.

- ❖ For each data item X, If first read on X is done by transaction Ta in schedule S1, then in schedule2 also the first read on X must be done by transaction Ta only.

(2) **Updated Read:** It should be same in both the schedules.

- ❖ If Read(X) of Ta followed by Write(X) of Tb in schedule S1, then in schedule S2 also, Read(X) of Ta must follow Write(X) of Tb ..

(3) **Final write:** The final write operation on each data item in both the schedule must be same.

- ❖ 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.

Problem 03: Check whether the given schedule S is view serializable or not

Schedule – 1

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

Solution:

For the given schedule-1, the serial schedule can be schedule -2

Schedule-1 (S1)

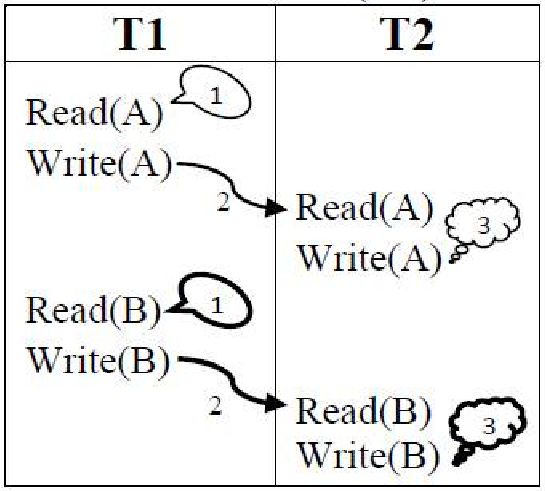
T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

Schedule-2 (S2)

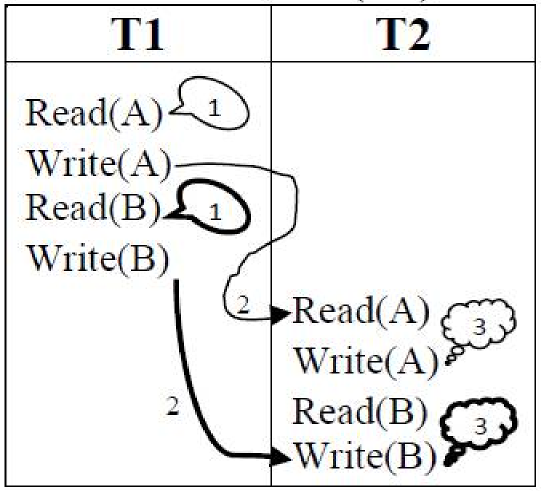
T1	T2
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)

Now let us check whether the three rules of view-equivalent satisfy or not.

Schedule-1 (S1)



Schedule-2 (S2)



Rule 1: Initial Read

First Read(A) is by T1 in S1 and in S2 also the first Read(A) is by T1 only.

First Read(B) is by T1 in S1 and in S2 also the first Read(B) is by T1 only.

Rule 2: Updated Read

Write(A) of T1 is read by T2 in S1 and in S2 also Write(A) of T1 is read by T2

Write(A) of T1 is read by T2 in S1 and in S2 also Write(A) of T1 is read by T2

Rule 3: Final Write

The final Write(A) is by T2 in S1 and in S2 also the final Write(A) is by T2 only

The final Write(B) is by T2 in S1 and in S2 also the final Write(B) is by T2 only

Conclusion: Hence, all the three rules are satisfied in this example, which means Schedule S1 and S2 are view equivalent. Also, it is proved that schedule S2 is the serial schedule of S1. Thus we can say that the S1 schedule is a view serializable schedule.

5. IMPLEMENTATION OF ATOMICITY AND DURABILITY

- ➢ The recovery-management component of a DBMS supports atomicity and durability by a variety of schemes. The simplest scheme to implement it is Shadow copy.

Shadow copy: In shadow-copy scheme,

- ❖ A transaction that wants to update the database first creates a complete copy of the database.
- ❖ All updates are done on the new database copy, leaving the original copy, untouched.
- ❖ If at any point the transaction has to be aborted, the system simply deletes the new copy. The old copy of the database has not been affected

- ➢ If the transaction complete successfully, then the database system updates the pointer db-pointer to point to the new copy of the database; the new copy then becomes the original copy of the database.
- ➢ The old copy of the database is then deleted. Figure below depicts the scheme, showing the database state before and after the update.

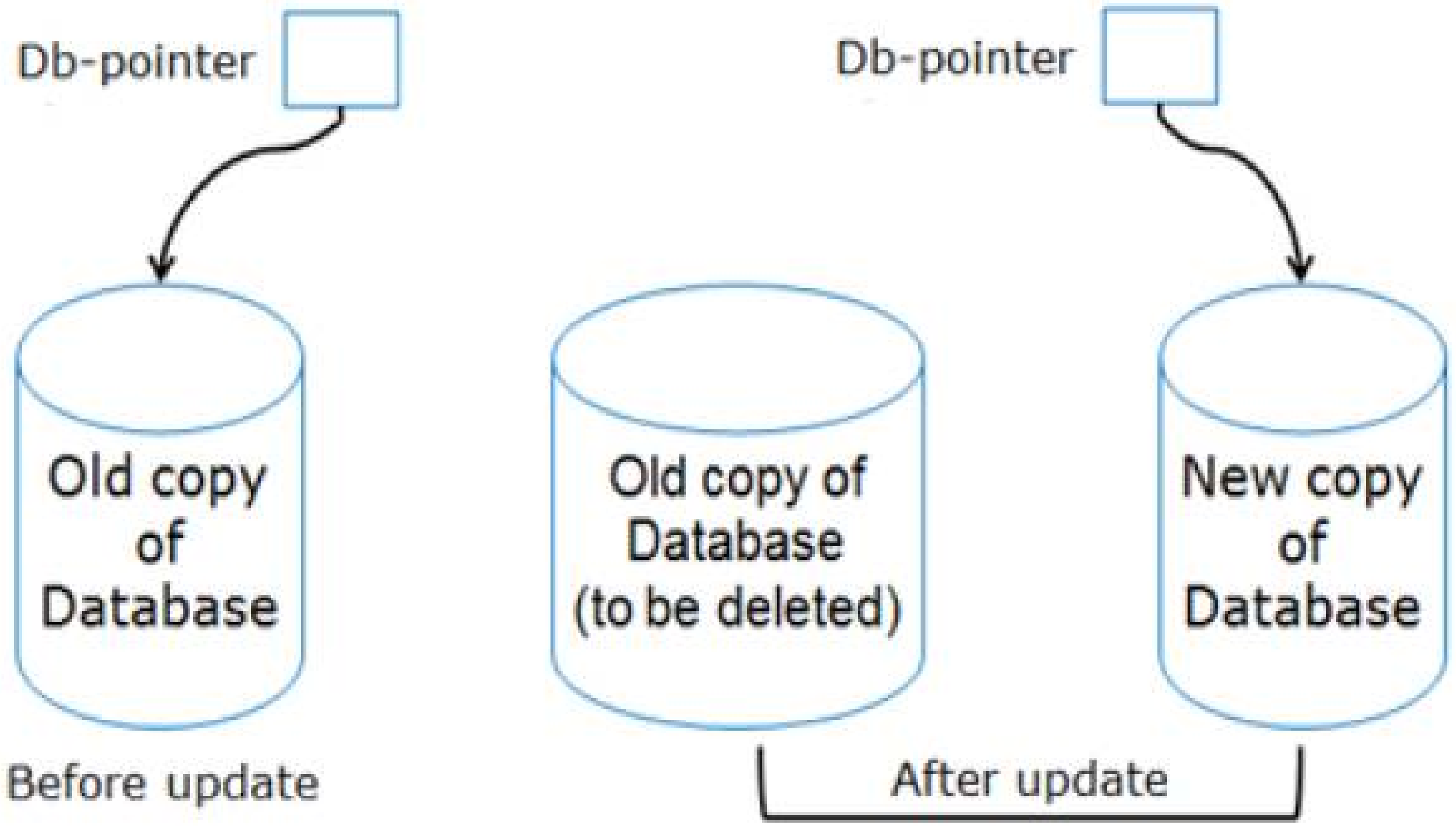


Figure: Shadow copy technique for atomicity and durability

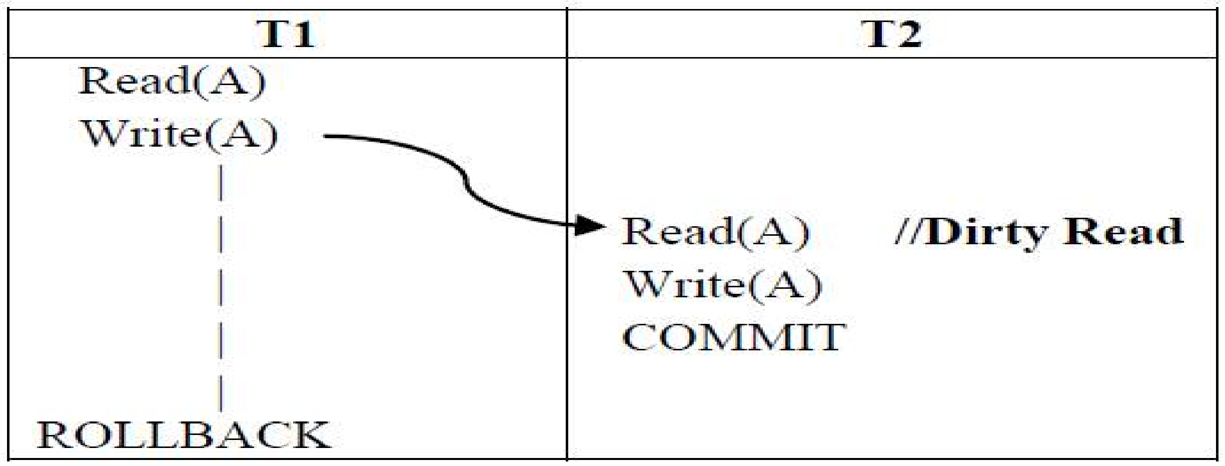
6. RECOVERABILITY

- ❖ During execution, if any of the transaction in a schedule is aborted, then this may leads the database into inconsistence state.
- ❖ If anything goes wrong, then the completed operations in the schedule needs to be undone.
- ❖ Sometimes, these undone operations may not possible. The recoverability of schedule depends on undone operations.
- ❖ If a transaction reads a data value that is updated by an uncommitted transaction, then this type of read is called as a **dirty read.**

i. Irrecoverable Schedule:

- ❖ In a schedule, if a transaction Ta performs a dirty read operation from other transaction Tb and Ta commits before Tb then such a schedule is known as an **Irrecoverable Schedule**

Example: Consider the following schedule



T1	T2
Read(A)	
Write(A)	
|	Read(A) **//Dirty Read**
|	Write(A)
|	COMMIT
|	
ROLLBACK	

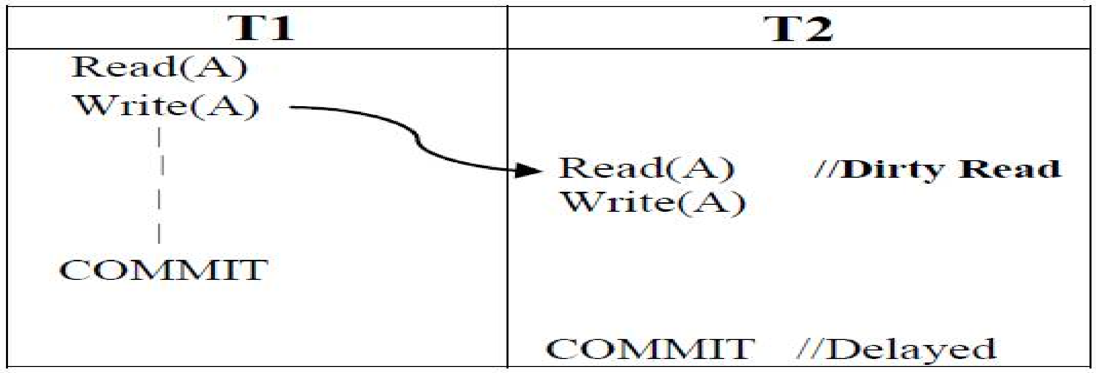
Here,

- ➢ T2 performs a dirty read operation
- ➢ T2 commits before T1.
- ➢ T1 fails later and roll backs.
- ➢ The value that T2 read now stands to be incorrect.
- ➢ T2 cannot recover since it has already committed.

ii. R**ecoverable Schedules:**

- ❖ In a schedule, if a transaction **Ta** performs a dirty read operation from other transaction **Tb** and **Ta** commit operation delayed till **Tb** commit, then such a schedule is known as an **recoverable Schedule**.

Example: Consider the following schedule-



T1	T2
Read(A)	
Write(A)	
|	Read(A) //**Dirty Read**
|	Write(A)
COMMIT	
	COMMIT //Delayed

Here,

- ❖ T2 performs a dirty read operation.
- ❖ The commit operation of T2 is delayed till T1 commits or roll backs.
- ❖ T1 commits later.
- ❖ T2 is now allowed to commit.
- ❖ In case, T1 would have failed, T2 has a chance to recover by rolling back.

Checking Whether a Schedule is Recoverable or Irrecoverable:

Check if there exists any dirty read operation.

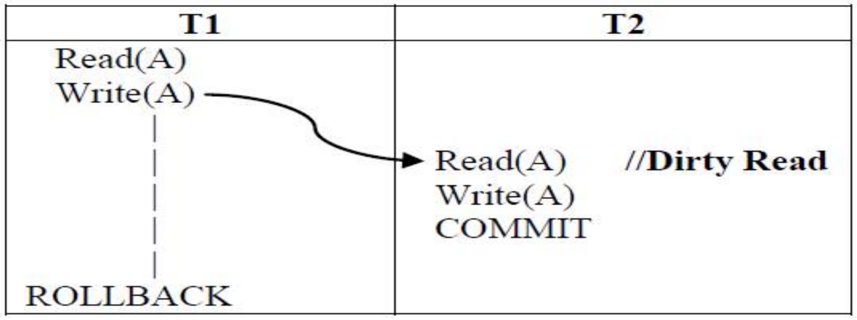
- If **there does not exist any dirty read operation**, then the schedule is **surely recoverable**.
- If **there exists any dirty read operation**, then
- If the commit operation of the transaction performing the dirty read occurs before the commit or abort operation of the transaction which updated the value, then the schedule is irrecoverable.
- If the commit operation of the transaction performing the dirty read is delayed till the commit or abort operation of the transaction which updated the value, then the schedule is recoverable.

7. IMPLEMENTATION OF ISOLATION

- Isolation determines how transactions integrity is visible to other users and systems. It means that **a transaction should take place in a system in such a way that it is the only one transaction that is accessing the resources in a database system.**
- Isolation level defines the degree to which a transaction must be isolated from the data modifications made by any other transactions in the database system.
- The phenomena's used to define levels of isolation are:

a) Dirty Read b) Non-repeatable Read c) Phantom Read

Dirty Read: If a transaction reads a data value updated by an uncommitted transaction, then this type of read is called as dirty read.

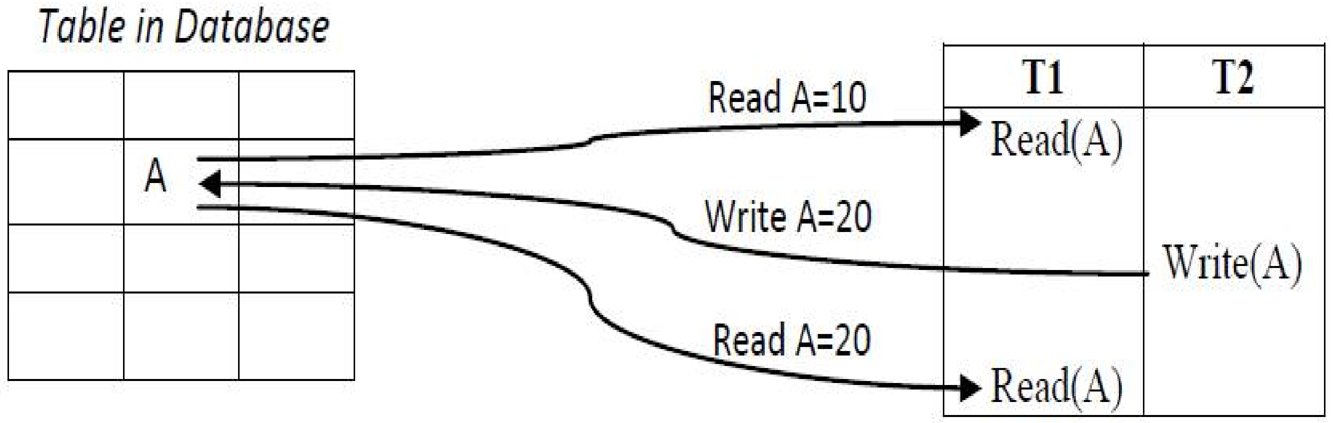


T1	T2
Read(A)	
Write(A)	
	Read(A) **//Dirty Read**
	Write(A)
	COMMIT
ROLLBACK	

As T1 aborted, the results produced by T2 become wrong. This is because T2 read A (Dirty Read) which is updated by T1.

Non-Repeatable Read:

❖ Non repeatable read occurs when a transaction read same data value twice and get a different value each time. It happens when a transaction reads once before and once after committed **UPDATES** from another transaction



First, T1 reads data item A and get A=10
Next, T2 writes data item A as A = 20
Last, T1 reads data item A and get A=20

Other example for Non-repeatable read:

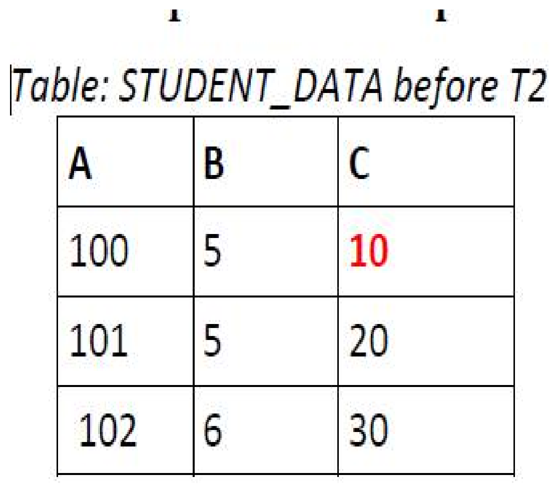


Table: STUDENT_DATA before T2

A	B	C
100	5	10
101	5	20
102	6	30

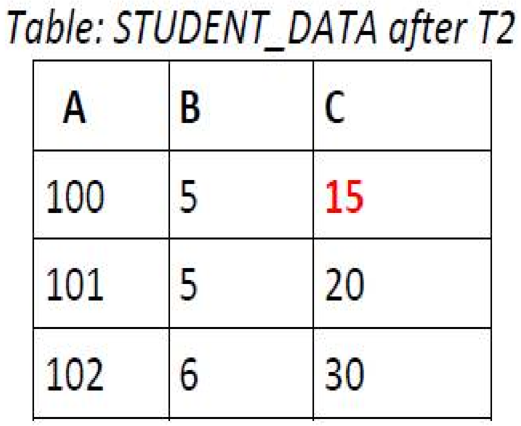


Table: STUDENT_DATA after T2

A	B	C
100	5	15
101	5	20
102	6	30

T1: SELECT SUM(C) FROM STUDENT_DATA WHERE B=5; Answer is (10+20) = **30**
T2: UPDATE STUDENT_DATA SET C = 15 WHERE A=100; Answer, in First row C changes to 15
T1: SELECT SUM(C) FROM STUDENT_DATA WHERE B=5; Answer is (15+20) = **35**

Phantom reads: Phantom reads occurs when a transaction read same data value twice and get a different value each time. It happens when a transaction reads once before and once after committed **INSERTS** and/or **DELETES** from another transaction.

Non-repeatable read	**Phantom read**
When T1 perform second read, there is no change in no of rows in the given table	When T1 perform second read, the no of rows either increase or decrease.
T2 perform UPDATE operation on the given table	T2 perform INSERT and/or DELETE operation on the given table

Example for Phantom read:

Table: STUDENT_DATA before T2

A	B	C
100	5	10
101	5	20
102	6	30

Table: STUDENT_DATA after T2

A	B	C
100	5	10
101	5	20
102	6	30
103	5	25

T1: SELECT SUM(C) FROM STUDENT_DATA WHERE B=5; Answer is (10+20) = 30
T2: INSERT INTO STUDENT_DATA VALUES(103, 5, 25);
Answer, in First row C changes to 15
T1: SELECT SUM(C) FROM STUDENT_DATA WHERE B=5; Answer is (10+20+25) = 55

Based on these three phenomena, SQL define four isolation levels. They are:

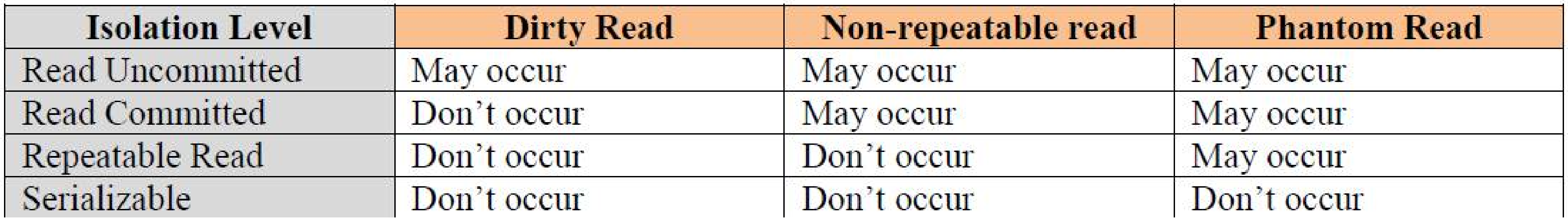
(1) **Read uncommitted:** This is the lowest level of isolation. In this level, one transaction may read the data item modified by other transaction which is not committed. It mean dirty read is allowed. In this level, transactions are not isolated from each other.

(2) **Read Committed:** This isolation level guarantees that any data read is committed at the moment it is read. Thus, it does not allow dirty read. The transaction holds a read/write lock on the data object, and thus prevents other transactions from reading, updating or deleting it.

(3) **Repeatable Read:** This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on all rows it inserts, updates, or deletes. Since other transaction cannot read, update or delete these rows, consequently it avoids non-repeatable read. So other transactions cannot read, update or delete these data items.

(4) **Serializable:** This is the highest isolation level. A *serializable* execution is guaranteed to be a serial schedule. Serializable execution is defined to be an execution of operations in which concurrently executing transactions appears to be serially executing

The table given below clearly depicts the relationship between isolation levels and the read phenomena and locks.



Isolation Level	Dirty Read	Non-repeatable read	Phantom Read
Read Uncommitted	May occur	May occur	May occur
Read Committed	Don’t occur	May occur	May occur
Repeatable Read	Don’t occur	Don’t occur	May occur
Serializable	Don’t occur	Don’t occur	Don’t occur

From the above table, it is clear that serializable isolation level is better than others