UNIT-V

TRANSACTION AND CONCURRENCY CONTROL

Transaction Management

A sequence of many actions that are considered to be one atomic unit of work. A transaction is a collection of operations involving data items in a database. There are four important properties of transactions that a DBMS must ensure to maintain data in database

Transaction

- The transaction is a set of logically related operation. It contains a group of tasks.
- A transaction is an action or series of actions. It is performed by a single user to perform operations for accessing the contents of the database.

Example: Suppose an employee of bank transfers Rs 800 from X's account to Y's account. This small transaction contains several low-level tasks:

X's Account

1. Open_Account(X)
2. Old_Balance = X.balance
3. New_Balance = Old_Balance - 800
4. X.balance = New_Balance
5. Close_Account(X)

Y's Account

1. Open_Account(Y)
2. Old_Balance = Y.balance
3. New_Balance = Old_Balance + 800
4. Y.balance = New_Balance
5. Close_Account(Y)

Operations of Transaction:

Following are the main operations of transaction:

Read(X): Read operation is used to read the value of X from the database and stores it in a buffer in main memory.

Write(X): Write operation is used to write the value back to the database from the buffer.

Let's take an example to debit transaction from an account which consists of following operations:

1. 1. R(X);
2. 2. X = X - 500;
3. 3. W(X);

Let's assume the value of X before starting of the transaction is 4000.

- The first operation reads X's value from database and stores it in a buffer.
- The second operation will decrease the value of X by 500. So buffer will contain 3500.
- The third operation will write the buffer's value to the database. So X's final value will be 3500.

But it may be possible that because of the failure of hardware, software or power, etc. that transaction may fail before finished all the operations in the set.

For example: If in the above transaction, the debit transaction fails after executing operation 2 then X's value will remain 4000 in the database which is not acceptable by the bank.

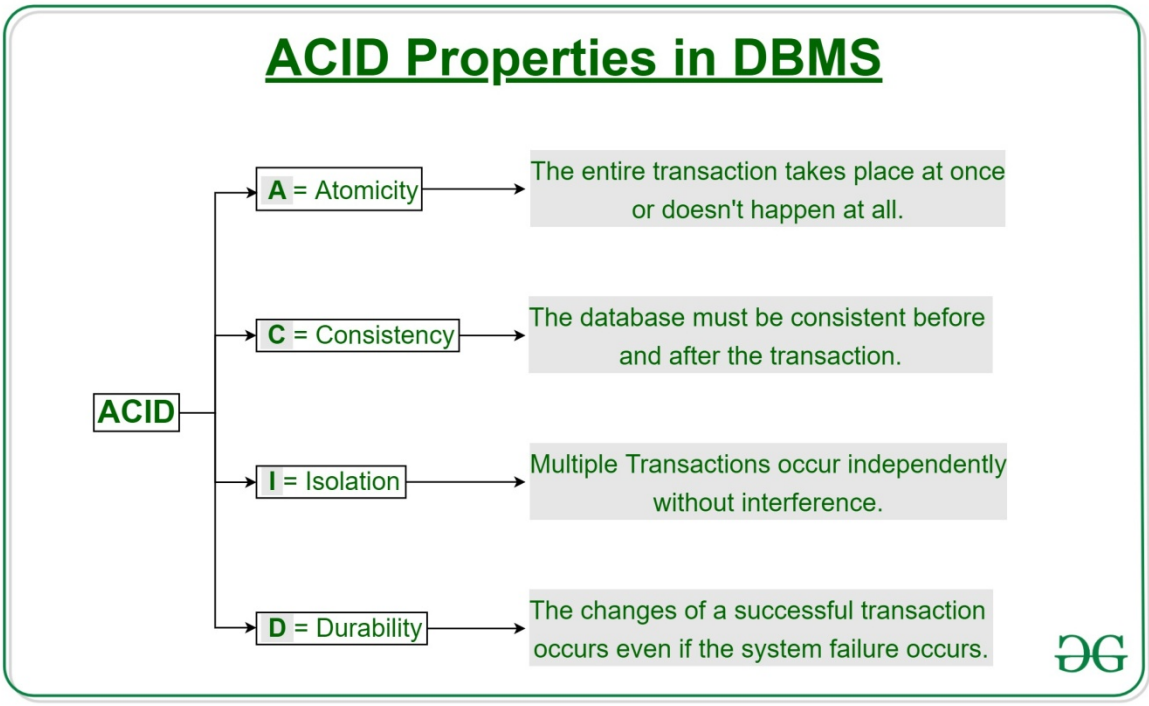
To solve this problem, we have two important operations:

Commit: It is used to save the work done permanently.

Rollback: It is used to undo the work done.

Transaction Properties:

A **transaction** is a single logical unit of work which accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.
In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties.



Atomicity
By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.
—**Abort**: If a transaction aborts, changes made to database are not visible.
—**Commit**: If a transaction commits, changes made are visible.
Atomicity is also known as the 'All or nothing rule'.

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

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

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

Consistency
This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database.
Referring to the example above,
The total amount before and after the transaction must be maintained.
Total **before T** occurs = **500 + 200 = 700**.
Total **after T occurs** = **400 + 300 = 700**.
Therefore, database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result T is incomplete.

Isolation
This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.
Let **X**= 500, **Y** = 500.
Consider two transactions **T** and **T".**

T	T''
Read (X) X: = X*100 Write (X) Read (Y) Y: = Y – 50 Write (Y)	Read (X) Read (Y) Z: = X + Y Write (Z)

Suppose **T** has been executed till **Read (Y)** and then **T''** starts. As a result , interleaving of operations takes place due to which **T''** reads correct value of **X** but incorrect value of **Y** and sum computed by
T'': (X+Y = 50, 000+500=50, 500)
is thus not consistent with the sum at end of transaction:
T: (X+Y = 50, 000 + 450 = 50, 450).
This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

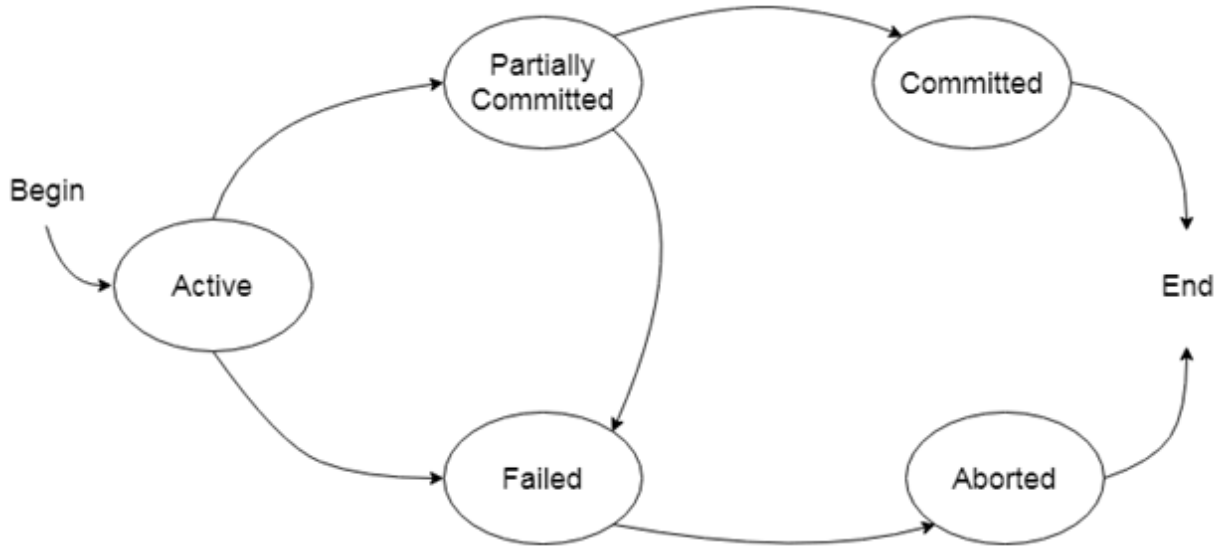
Durability:

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

The **ACID** properties, in totality, provide a mechanism to ensure correctness and consistency of a database in a way such that each transaction is a group of operations that acts a single unit, produces consistent results, acts in isolation from other operations and updates that it makes are durably stored.

States of Transaction

In a database, the transaction can be in one of the following states -



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.
- In the total mark calculation example, 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

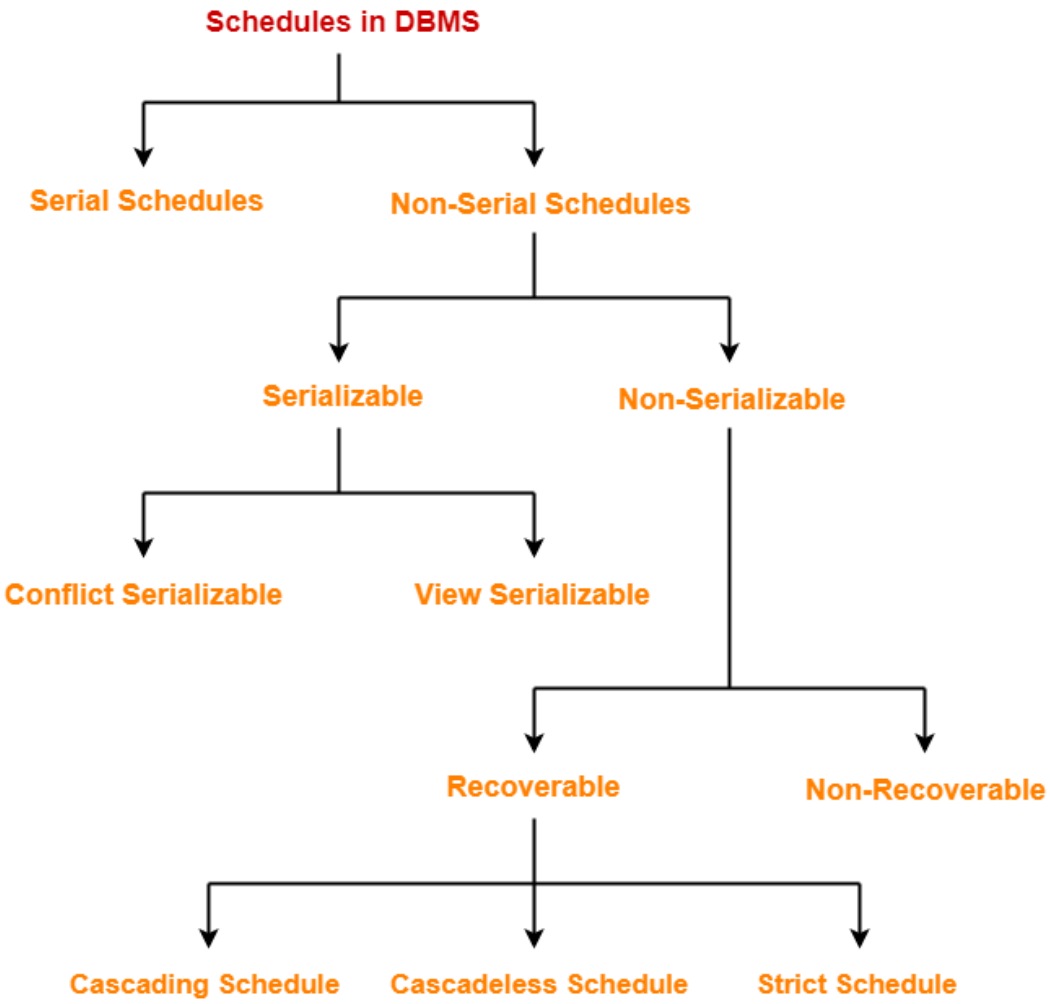
- If any of the checks fail and the transaction has reached a failed state then the database recovery system will make sure that the database is in its previous consistent state. If not then it will abort or roll back the transaction to bring the database into a consistent state.
- If the transaction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

Schedule

We have discussed-

- A schedule is the order in which the operations of multiple transactions appear for execution.
- Serial schedules are always consistent.
- Non-serial schedules are not always consistent.

In DBMS, schedules may be classified as-



Serializability in DBMS-

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

Serializable Schedules-

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

Characteristics-

Serializable schedules behave exactly same as serial schedules.

Thus, serializable schedules are always-

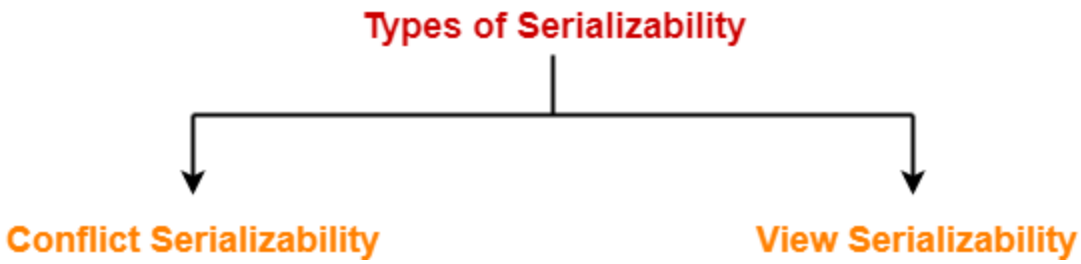
- Consistent
- **Recoverable**
- **Casacadeless**
- Strict

Serial Schedules Vs Serializable Schedules-

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.
Serial schedules lead to less resource utilization and CPU throughput.	Serializable schedules improve both resource utilization and CPU throughput.
Serial Schedules are less efficient as compared to serializable schedules. (due to above reason)	Serializable Schedules are always better than serial schedules. (due to above reason)

Types of Serializability-

Serializability is mainly of two types-



1. Conflict Serializability
2. View Serializability

Conflict Serializability-

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

Conflicting Operations-

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

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

Example-

Consider the following schedule-

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

In this schedule,

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

Checking Whether a Schedule is Conflict Serializable Or Not-

Follow the following steps to check whether a given non-serial schedule is conflict serializable or not-

Step-01:

Find and list all the conflicting operations.

Step-02:

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

Step-03:

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

Step-04:

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

PRACTICE PROBLEMS BASED ON CONFLICT SERIALIZABILITY-

Problem-01:

Check whether the given schedule S is conflict serializable or not-

S : $R_1(A)$, $R_2(A)$, $R_1(B)$, $R_2(B)$, $R_3(B)$, $W_1(A)$, $W_2(B)$

Solution-

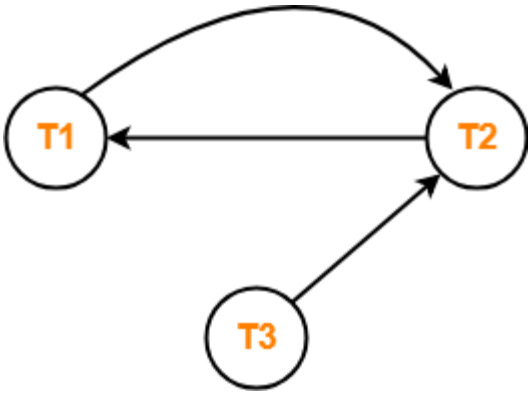
Step-01:

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

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

Step-02:

Draw the precedence graph-



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

Problem-02:

Check whether the given schedule S is conflict serializable and recoverable or not-

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

Solution-

Checking Whether S is Conflict Serializable Or Not-

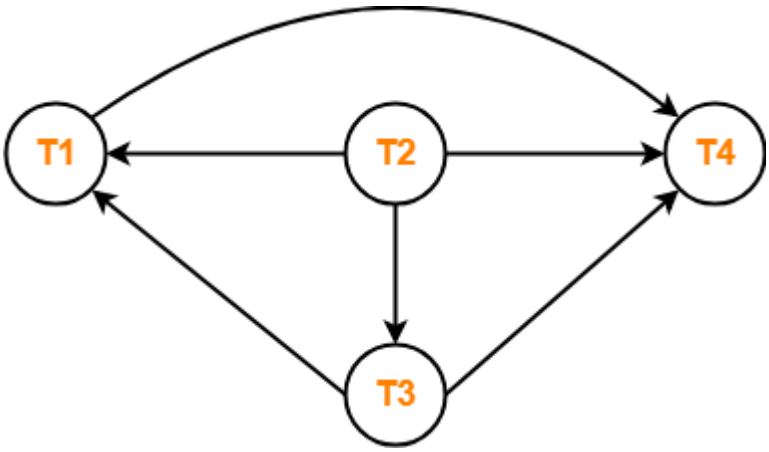
Step-01:

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

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

Step-02:

Draw the precedence graph-



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

Checking Whether S is Recoverable Or Not-

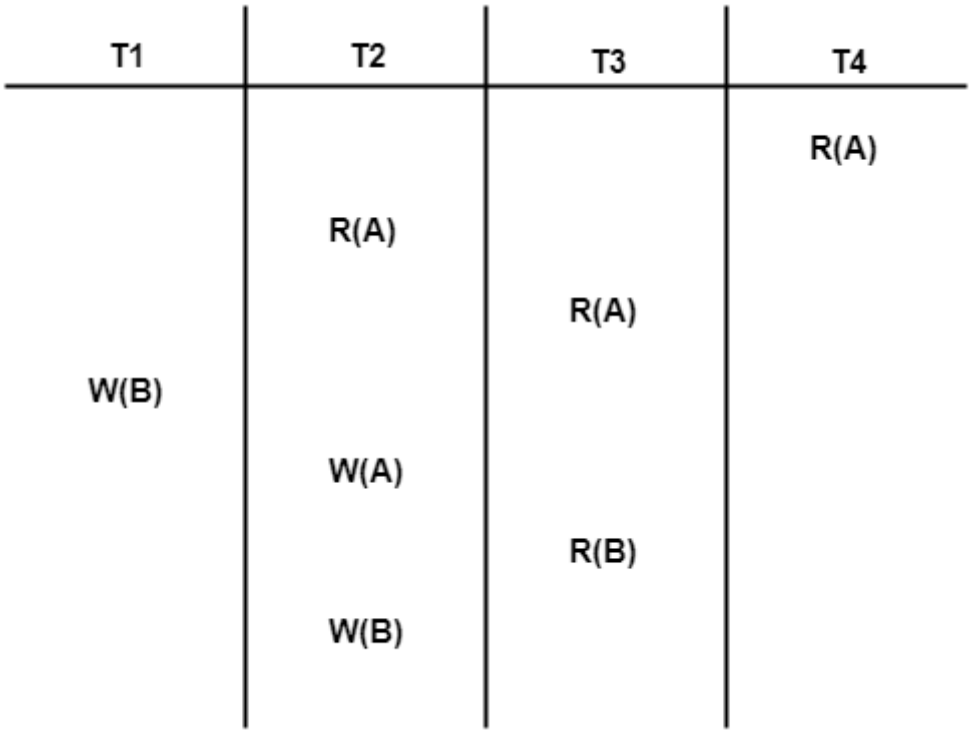
- Conflict serializable schedules are always recoverable.
- Therefore, the given schedule S is recoverable.

Alternatively,

- There exists no dirty read operation.
- This is because all the transactions which update the values commits immediately.
- Therefore, the given schedule S is recoverable.
- Also, S is a **Cascadeless Schedule**.

Problem-03:

Check whether the given schedule S is conflict serializable or not. If yes, then determine all the possible serialized schedules-



Solution-

Checking Whether S is Conflict Serializable Or Not-

Step-01:

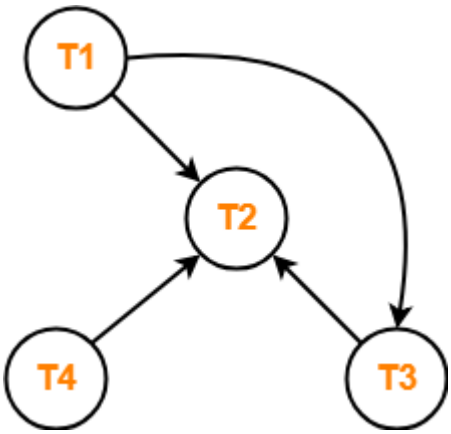
List all the conflicting operations and determine the dependency between the transactions-

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

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

Step-02:

Draw the precedence graph-



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

Finding the Serialized Schedules-

- Identify Indegrre of Vertices
  In-degree of T1-0
  In-degree of T2-3
  In-degree of T3-1
  Indegrre of T4-0
  Both T1 and T4 have In-degree 0 so possible serial schedules are
  T1-T4-T3-T2
  T4-T1-T3-T2

View Serializability-

If a given schedule is found to be view equivalent to some serial schedule, then it is called as a view serializable schedule.

View Equivalent Schedules-

Consider two schedules S1 and S2 each consisting of two transactions T1 and T2.

Schedules S1 and S2 are called view equivalent if the following three conditions hold true for them-

Condition-01:

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

Thumb Rule

"Initial readers must be same for all the data items".

Condition-02:

If transaction T_i reads a data item that has been updated by the transaction T_j in schedule S1, then in schedule S2 also, transaction T_i must read the same data item that has been updated by the transaction T_j.

Thumb Rule

"Write-read sequence must be same.".

Condition-03:

For each data item X, if X has been updated at last by transaction T_i in schedule S1, then in schedule S2 also, X must be updated at last by transaction T_i.

Thumb Rule

"Final writers must be same for all the data items".

Checking Whether a Schedule is View Serializable Or Not-

Method-01:

Check whether the given schedule is conflict serializable or not.

- If the given schedule is conflict serializable, then it is surely view serializable. Stop and report your answer.
- If the given schedule is not conflict serializable, then it may or may not be view serializable. Go and check using other methods.

Thumb Rules

- All conflict serializable schedules are view serializable.
- All view serializable schedules may or may not be conflict serializable.

Method-02:

Check if there exists any blind write operation.

(Writing without reading is called as a blind write).

- If there does not exist any blind write, then the schedule is surely not view serializable. Stop and report your answer.
- If there exists any blind write, then the schedule may or may not be view serializable. Go and check using other methods.

Thumb Rule

No blind write means not a view serializable schedule.

Method-03:

In this method, try finding a view equivalent serial schedule.

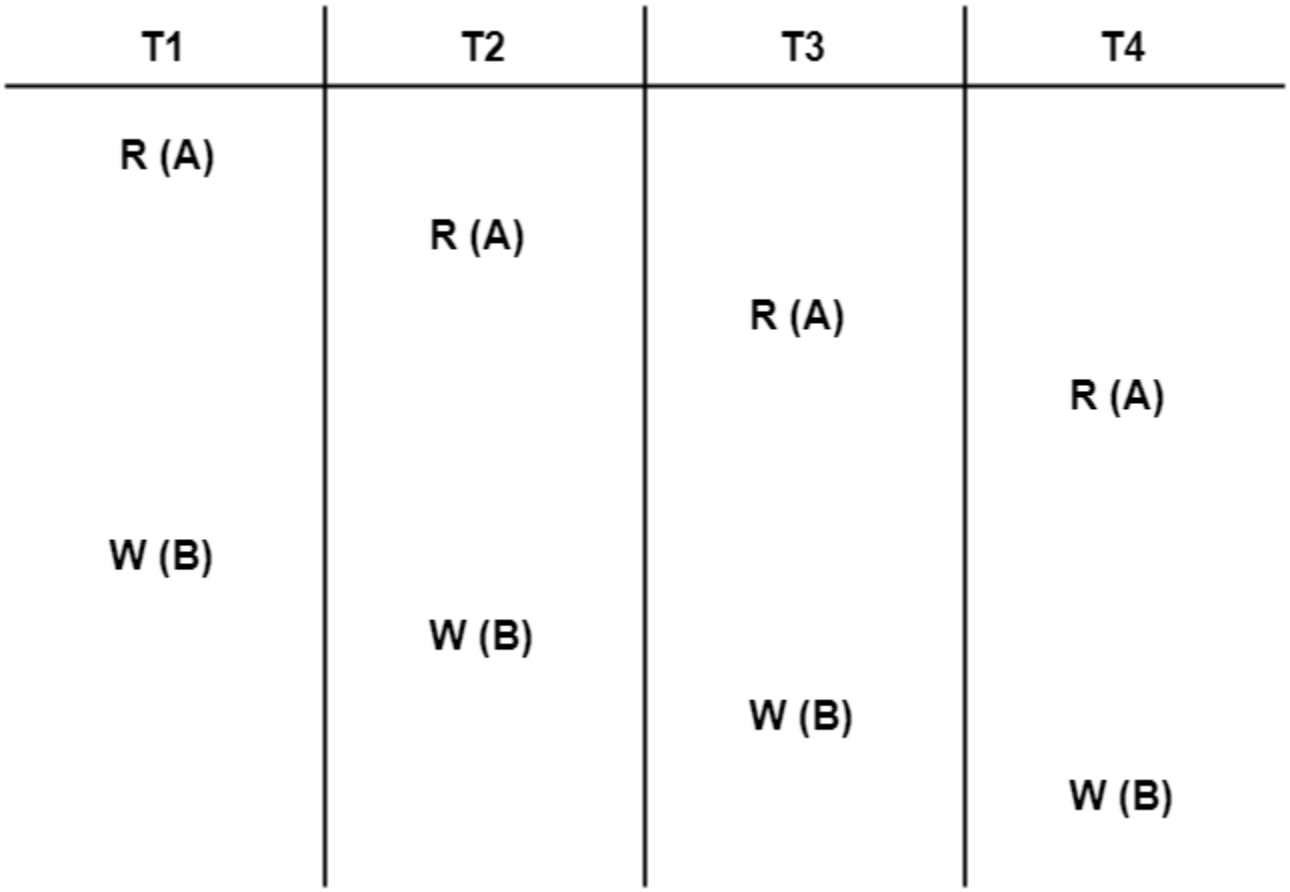
- By using the above three conditions, write all the dependencies.

- Then, draw a graph using those dependencies.
- If there exists no cycle in the graph, then the schedule is view serializable otherwise not.

PRACTICE PROBLEMS BASED ON VIEW SERIALIZABILITY-

Problem-01:

Check whether the given schedule S is view serializable or not-



Solution-

- We know, if a schedule is conflict serializable, then it is surely view serializable.
- So, let us check whether the given schedule is conflict serializable or not.

Checking Whether S is Conflict Serializable Or Not-

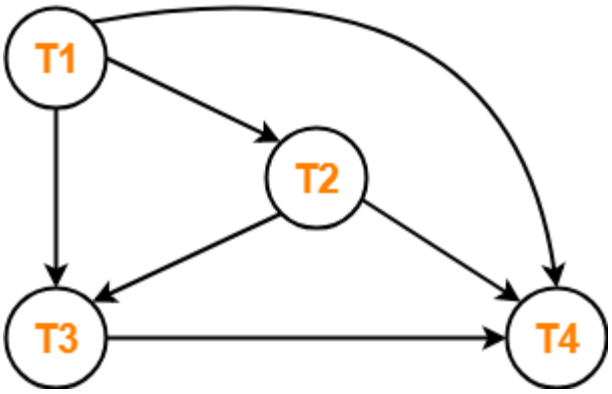
Step-01:

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

- $W_1(B)$, $W_2(B)$ ($T_1 \rightarrow T_2$)
- $W_1(B)$, $W_3(B)$ ($T_1 \rightarrow T_3$)
- $W_1(B)$, $W_4(B)$ ($T_1 \rightarrow T_4$)
- $W_2(B)$, $W_3(B)$ ($T_2 \rightarrow T_3$)
- $W_2(B)$, $W_4(B)$ ($T_2 \rightarrow T_4$)
- $W_3(B)$, $W_4(B)$ ($T_3 \rightarrow T_4$)

Step-02:

Draw the precedence graph-



- Clearly, there exists no cycle in the precedence graph.
- Therefore, the given schedule S is conflict serializable.
- Thus, we conclude that the given schedule is also view serializable.

Problem-02:

Check whether the given schedule S is view serializable or not-

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

Solution-

- We know, if a schedule is conflict serializable, then it is surely view serializable.
- So, let us check whether the given schedule is conflict serializable or not.

Checking Whether S is Conflict Serializable Or Not-

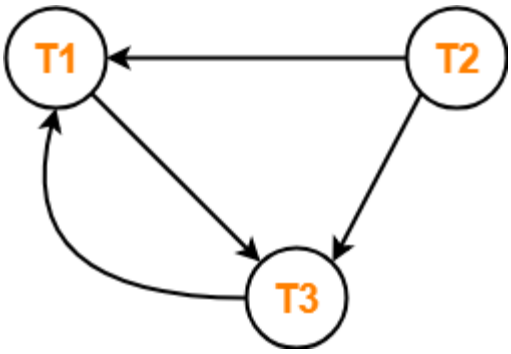
Step-01:

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

- $R_1(A)$, $W_3(A)$ ($T_1 \rightarrow T_3$)
- $R_2(A)$, $W_3(A)$ ($T_2 \rightarrow T_3$)
- $R_2(A)$, $W_1(A)$ ($T_2 \rightarrow T_1$)
- $W_3(A)$, $W_1(A)$ ($T_3 \rightarrow T_1$)

Step-02:

Draw the precedence graph-



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

Now,

- Since, the given schedule S is not conflict serializable, so, it may or may not be view serializable.
- To check whether S is view serializable or not, let us use another method.
- Let us check for blind writes.

Checking for Blind Writes-

- There exists a blind write $W_3(A)$ in the given schedule S.
- Therefore, the given schedule S may or may not be view serializable.

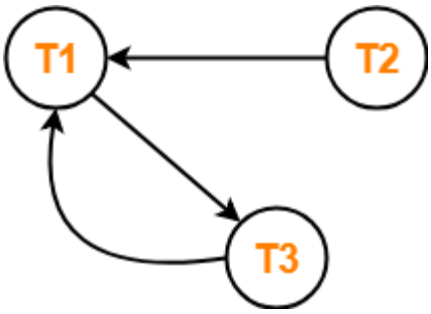
Now,

- To check whether S is view serializable or not, let us use another method.
- Let us derive the dependencies and then draw a dependency graph.

Drawing a Dependency Graph-

- T1 firstly reads A and T3 firstly updates A.
- So, T1 must execute before T3.
- Thus, we get the dependency **T1 → T3**.
- Final updation on A is made by the transaction T1.
- So, T1 must execute after all other transactions.
- Thus, we get the dependency **(T2, T3) → T1**.
- There exists no write-read sequence.

Now, let us draw a dependency graph using these dependencies-



- Clearly, there exists a cycle in the dependency graph.
- Thus, we conclude that the given schedule S is not view serializable.

Problem-03:

Check whether the given schedule S is view serializable or not-

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 x 1.1
W (B)	
	W (B)

Solution-

- We know, if a schedule is conflict serializable, then it is surely view serializable.
- So, let us check whether the given schedule is conflict serializable or not.

Checking Whether S is Conflict Serializable Or Not-

Step-01:

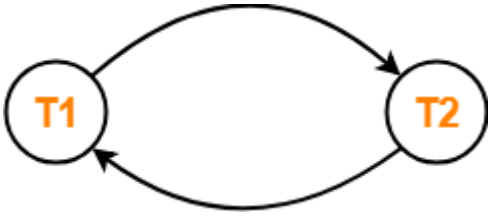
List all the conflicting operations and determine the dependency between the transactions-

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

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

Step-02:

Draw the precedence graph-



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

Now,

- Since, the given schedule S is not conflict serializable, so, it may or may not be view serializable.
- To check whether S is view serializable or not, let us use another method.
- Let us check for blind writes.

Checking for Blind Writes-

- There exists no blind write in the given schedule S.
- Therefore, it is surely not view serializable.

Alternatively,

- You could directly declare that the given schedule S is not view serializable.
- This is because there exists no blind write in the schedule.
- You need not check for conflict serializability.

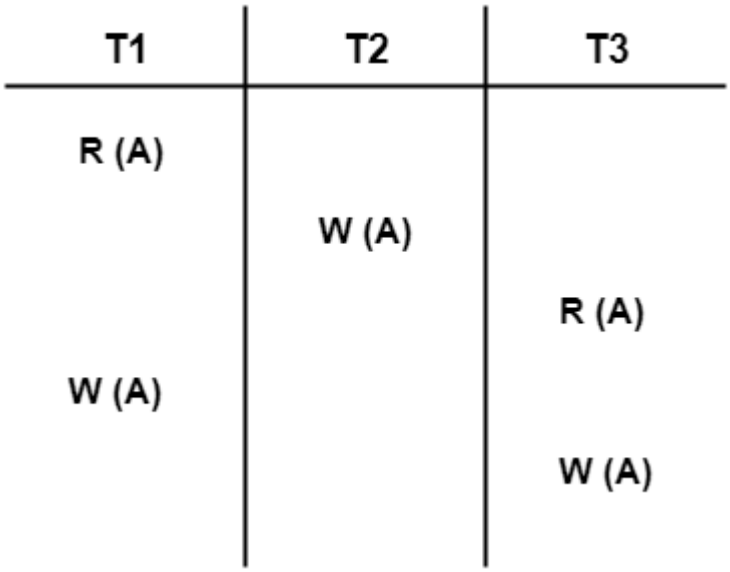
Problem-04:

Check whether the given schedule S is view serializable or not. If yes, then give the serial schedule.

S : $R_1(A)$, $W_2(A)$, $R_3(A)$, $W_1(A)$, $W_3(A)$

Solution-

For simplicity and better understanding, we can represent the given schedule pictorially as-



- We know, if a schedule is conflict serializable, then it is surely view serializable.
- So, let us check whether the given schedule is conflict serializable or not.

Checking Whether S is Conflict Serializable Or Not-

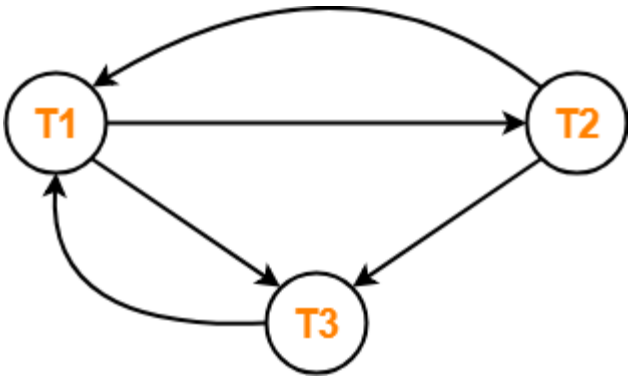
Step-01:

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

- $R_1(A)$, $W_2(A)$ ($T_1 \rightarrow T_2$)
- $R_1(A)$, $W_3(A)$ ($T_1 \rightarrow T_3$)
- $W_2(A)$, $R_3(A)$ ($T_2 \rightarrow T_3$)
- $W_2(A)$, $W_1(A)$ ($T_2 \rightarrow T_1$)
- $W_2(A)$, $W_3(A)$ ($T_2 \rightarrow T_3$)
- $R_3(A)$, $W_1(A)$ ($T_3 \rightarrow T_1$)
- $W_1(A)$, $W_3(A)$ ($T_1 \rightarrow T_3$)

Step-02:

Draw the precedence graph-



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

Now,

- Since, the given schedule S is not conflict serializable, so, it may or may not be view serializable.
- To check whether S is view serializable or not, let us use another method.
- Let us check for blind writes.

Checking for Blind Writes-

- There exists a blind write $W_2(A)$ in the given schedule S.
- Therefore, the given schedule S may or may not be view serializable.

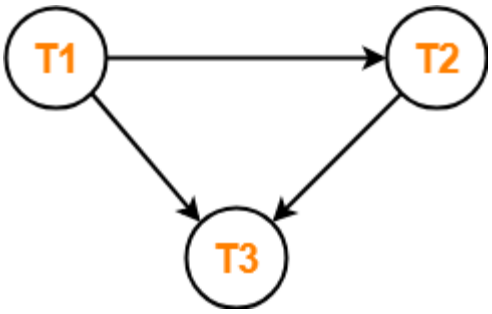
Now,

- To check whether S is view serializable or not, let us use another method.
- Let us derive the dependencies and then draw a dependency graph.

Drawing a Dependency Graph-

- T1 firstly reads A and T2 firstly updates A.
- So, T1 must execute before T2.
- Thus, we get the dependency **T1 → T2**.
- Final updation on A is made by the transaction T3.
- So, T3 must execute after all other transactions.
- Thus, we get the dependency **(T1, T2) → T3**.
- From write-read sequence, we get the dependency **T2 → T3**

Now, let us draw a dependency graph using these dependencies-



- Clearly, there exists no cycle in the dependency graph.
- Therefore, the given schedule S is view serializable.
- The serialization order T1 → T2 → T3.

Irrecoverable Schedules-

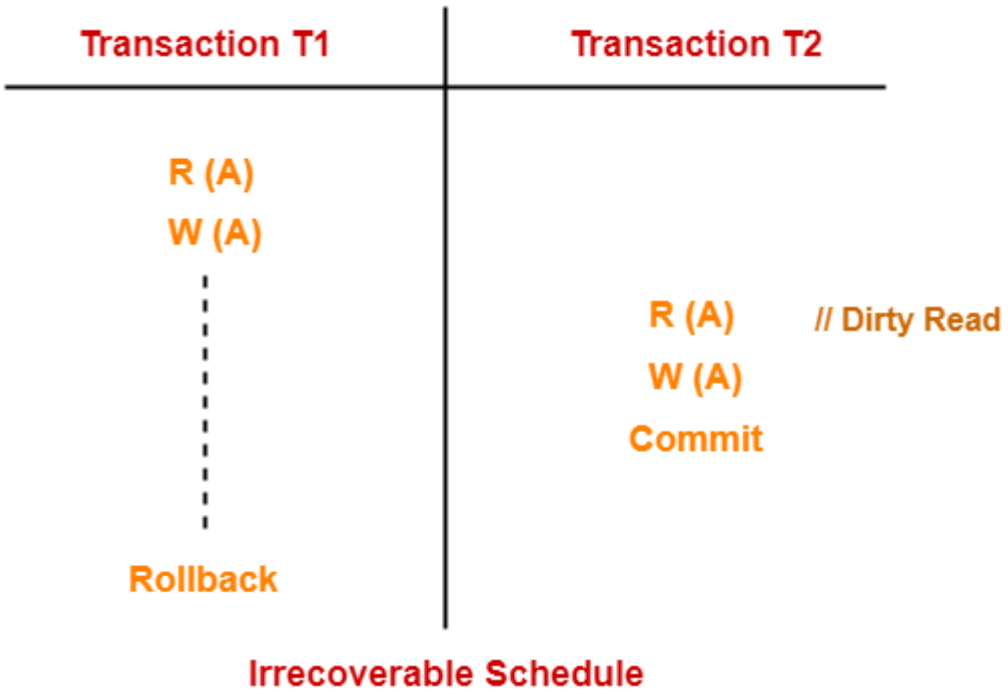
If in a schedule,

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

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

Example-

Consider the following schedule-



Irrecoverable Schedule

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 can not recover since it has already committed.

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

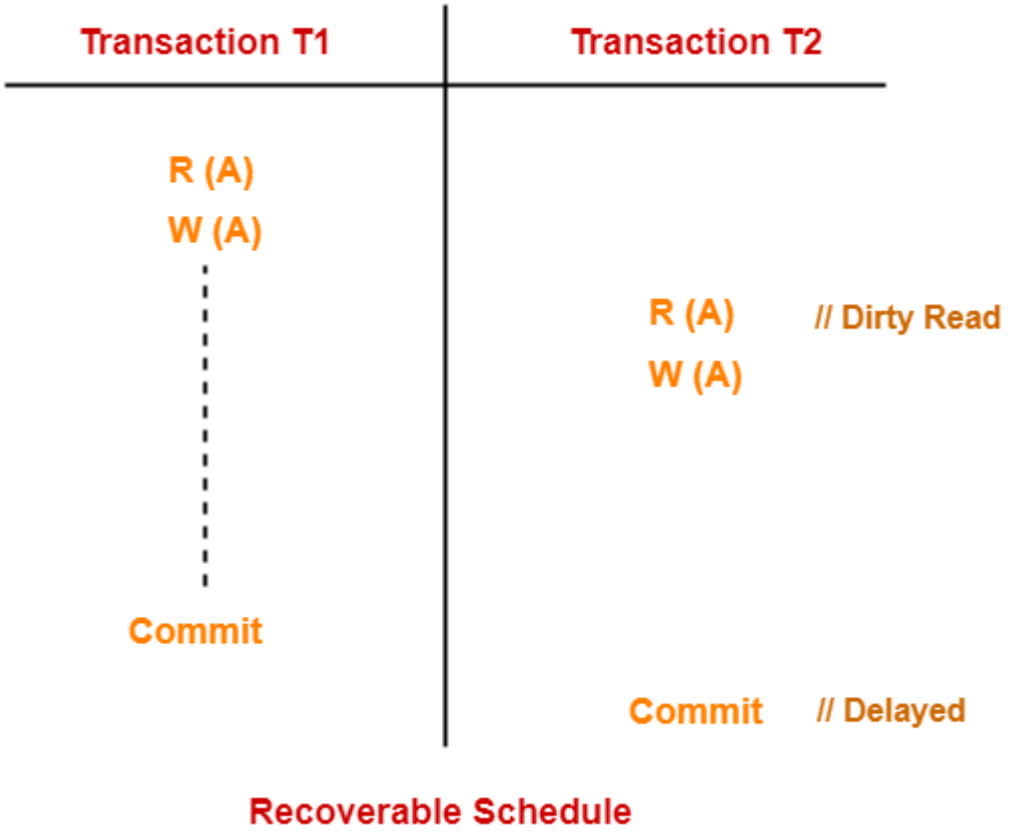
then such a schedule is known as a **Recoverable Schedule**.

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.

Example-

Consider the following schedule-



Recoverable Schedule

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-

Method-01:

Check whether the given schedule is conflict serializable or not.

- If the given schedule is conflict serializable, then it is surely recoverable. Stop and report your answer.
- If the given schedule is not conflict serializable, then it may or may not be recoverable. Go and check using other methods.

Thumb Rules

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

Method-02:

Check if there exists any dirty read operation.

(Reading from an uncommitted transaction is called as a dirty read)

- If there does not exist any dirty read operation, then the schedule is surely recoverable. Stop and report your answer.
- If there exists any dirty read operation, then the schedule may or may not be recoverable.

If there exists a dirty read operation, then follow the following cases-

Case-01:

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.

Case-02:

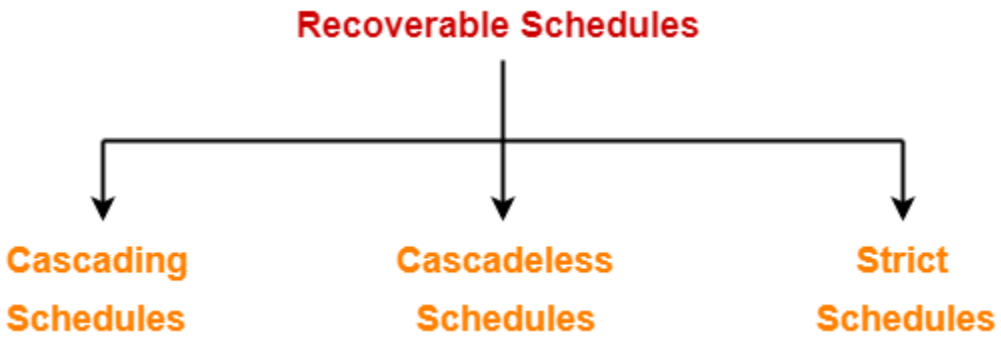
If the commit operation of the transaction performing the dirty read is delayed till the commit or abort operation of the transaction which updated the value, then the schedule is recoverable.

Thumb Rule

No dirty read means a recoverable schedule.

Types of Recoverable Schedules-

A recoverable schedule may be any one of these kinds-

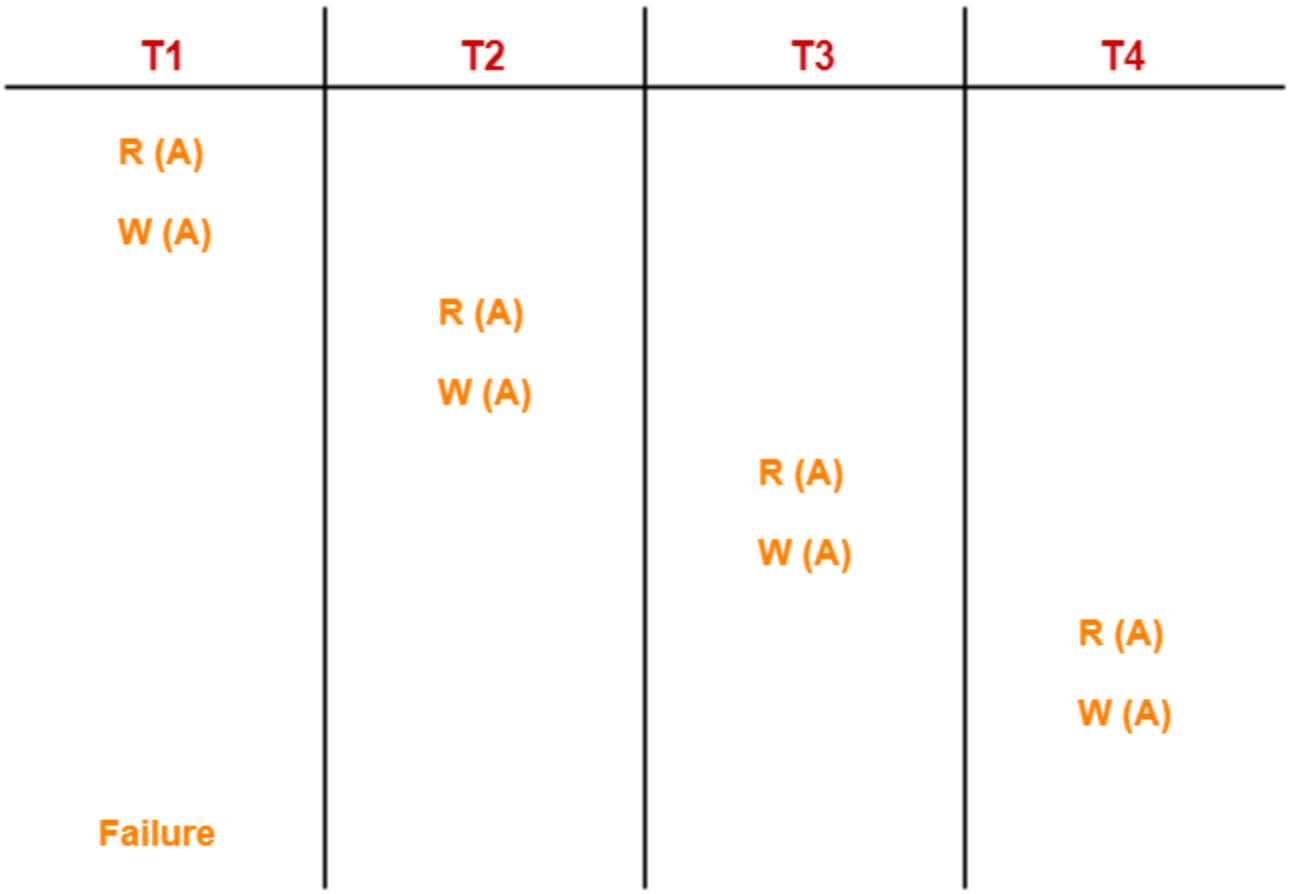


1. Cascading Schedule
2. Cascadeless Schedule
3. Strict Schedule

Cascading Schedule-

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

Example-



Cascading Recoverable Schedule

Here,

- Transaction T2 depends on transaction T1.
- Transaction T3 depends on transaction T2.
- Transaction T4 depends on transaction T3.

In this schedule,

- The failure of transaction T1 causes the transaction T2 to rollback.
- The rollback of transaction T2 causes the transaction T3 to rollback.
- The rollback of transaction T3 causes the transaction T4 to rollback.

Such a rollback is called as a **Cascading Rollback**.

NOTE-

If the transactions T2, T3 and T4 would have committed before the failure of transaction T1, then the schedule would have been irrecoverable.

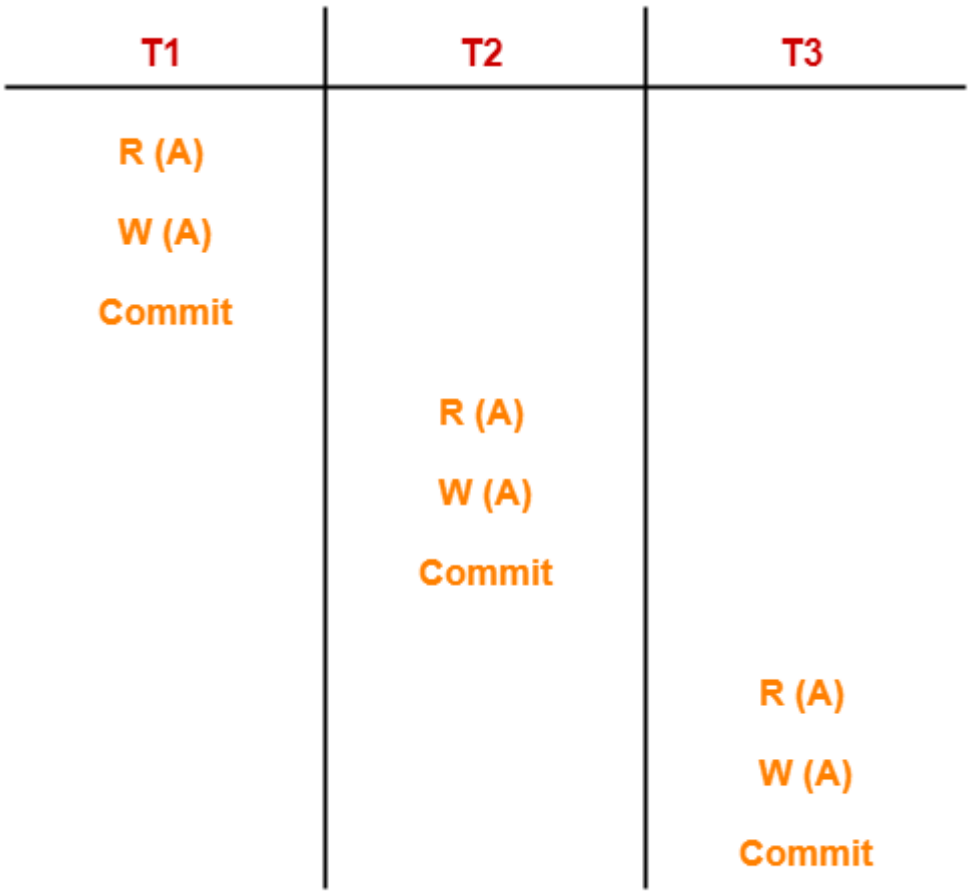
Cascadeless Schedule-

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

In other words,

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

Example-

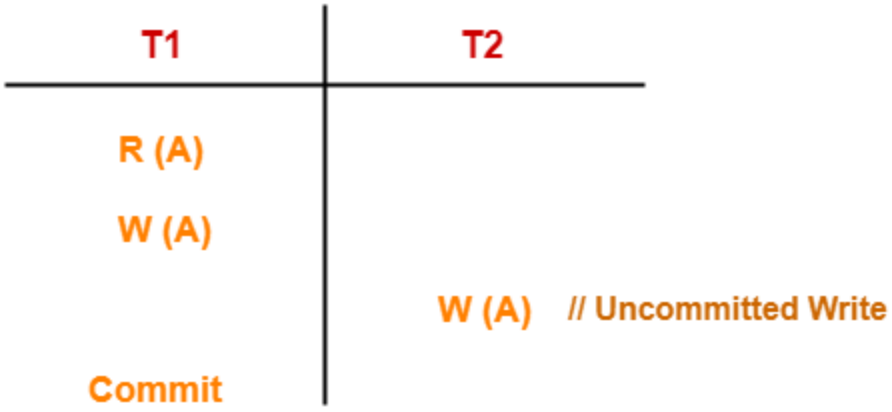


Cascadeless Schedule

NOTE-

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

Example-



Cascadeless Schedule

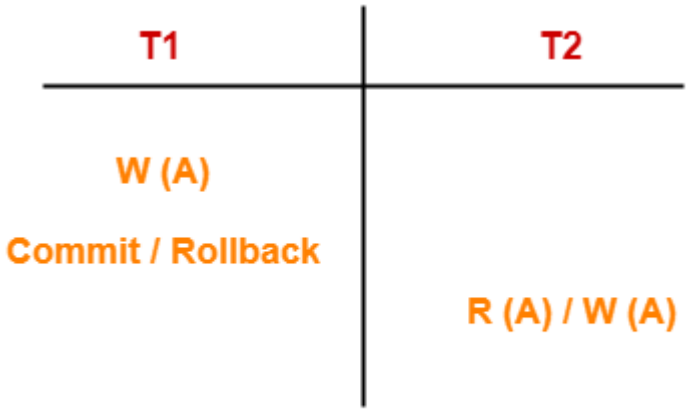
Strict Schedule-

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

In other words,

- Strict schedule allows only committed read and write operations.
- Clearly, strict schedule implements more restrictions than cascadeless schedule.

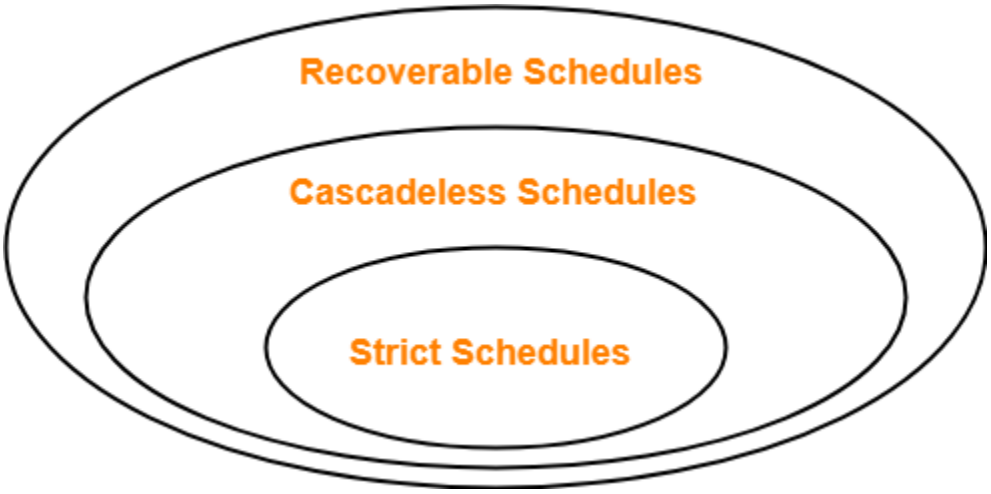
Example-



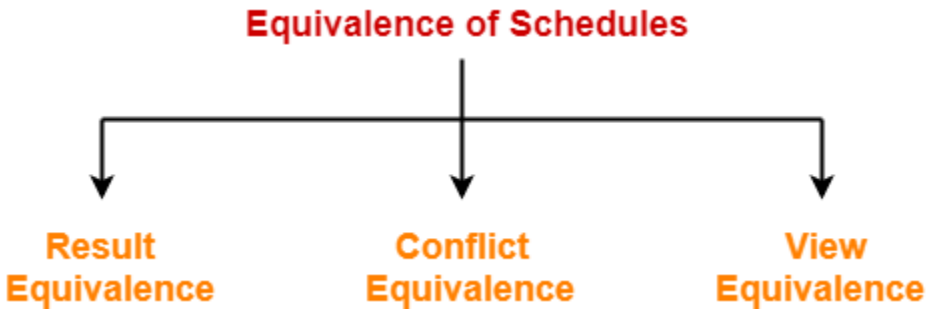
Strict Schedule

Remember-

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



Equivalence of Schedules-



1. Result Equivalence
2. Conflict Equivalence
3. View Equivalence

1. Result Equivalent Schedules-

- If any two schedules generate the same result after their execution, then they are called as result equivalent schedules.

- This equivalence relation is considered of least significance.
- This is because some schedules might produce same results for some set of values and different results for some other set of values.

2. Conflict Equivalent Schedules-

If any two schedules satisfy the following two conditions, then they are called as conflict equivalent schedules-

1. The set of transactions present in both the schedules is same.
2. The order of pairs of conflicting operations of both the schedules is same.

3. View Equivalent Schedules-

We have already discussed about View Equivalent Schedules**.**

PRACTICE PROBLEMS BASED ON EQUIVALENCE OF SCHEDULES-

Problem-01:

Are the following three schedules result equivalent?

Schedule S1

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

Schedule S2

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

Schedule S3

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

Solution-

To check whether the given schedules are result equivalent or not,

- We will consider some arbitrary values of X and Y.
- Then, we will compare the results produced by each schedule.
- Those schedules which produce the same results will be result equivalent.

Let X = 2 and Y = 5.

On substituting these values, the results produced by each schedule are-

Results by Schedule S1- X = 21 and Y = 10

Results by Schedule S2- X = 21 and Y = 10

Results by Schedule S3- X = 11 and Y = 10

- Clearly, the results produced by schedules S1 and S2 are same.
- Thus, we conclude that S1 and S2 are result equivalent schedules.

Problem-02:

Are the following two schedules conflict equivalent?

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

Schedule S1

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

Schedule S2

Solution-

To check whether the given schedules are conflict equivalent or not,

- We will write their order of pairs of conflicting operations.
- Then, we will compare the order of both the schedules.
- If both the schedules are found to have the same order, then they will be conflict equivalent.

For schedule S1-

The required order is-

- $R_1(A)$, $W_2(A)$
- $W_1(A)$, $R_2(A)$

- $W_1(A)$, $W_2(A)$

For schedule S2-

The required order is-

- $R_1(A)$, $W_2(A)$
- $W_1(A)$, $R_2(A)$
- $W_1(A)$, $W_2(A)$

- Clearly, both the given schedules have the same order.
- Thus, we conclude that S1 and S2 are conflict equivalent schedules.

  Concurrency Control is the management procedure that is required for controlling concurrent execution of the operations that take place on a database.

  But before knowing about concurrency control, we should know about concurrent execution.

  Concurrent Execution in DBMS

  - In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database. It means that the same database is executed simultaneously on a multi-user system by different users.
  - While working on the database transactions, there occurs the requirement of using the database by multiple users for performing different operations, and in that case, concurrent execution of the database is performed.
  - The thing is that the simultaneous execution that is performed should be done in an interleaved manner, and no operation should affect the other executing operations, thus maintaining the consistency of the database. Thus, on making the concurrent execution of the transaction operations, there occur several challenging problems that need to be solved.

  Problems with Concurrent Execution

  In a database transaction, the two main operations are **READ** and **WRITE** operations. So, there is a need to manage these two operations in the concurrent execution of the transactions as if these operations are not performed in an interleaved manner, and the data may become inconsistent. So, the following problems occur with the Concurrent Execution of the operations:

  Problem 1: Lost Update Problems (W - W Conflict)

  The problem occurs *when two different database transactions perform the read/write operations on the same database items in an interleaved manner (i.e., concurrent execution) that makes the values of the items incorrect hence making the database inconsistent*.

  For example:

  Consider the below diagram where two transactions T_x and T_y, are performed on the same account A where the balance of account A is $300.

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A - 50	
t_3	—	READ (A)
t_4	—	A = A + 100
t_5	—	—
t_6	WRITE (A)	—
t_7		WRITE (A)

LOST UPDATE PROBLEM

- At time t1, transaction T_x reads the value of account A, i.e., $300 (only read).
- At time t2, transaction T_x deducts $50 from account A that becomes $250 (only deducted and not updated/write).
- Alternately, at time t3, transaction T_y reads the value of account A that will be $300 only because T_x didn't update the value yet.
- At time t4, transaction T_y adds $100 to account A that becomes $400 (only added but not updated/write).
- At time t6, transaction T_x writes the value of account A that will be updated as $250 only, as T_y didn't update the value yet.
- Similarly, at time t7, transaction T_y writes the values of account A, so it will write as done at time t4 that will be $400. It means the value written by T_x is lost, i.e., $250 is lost.

Hence data becomes incorrect, and database sets to inconsistent.

Dirty Read Problems (W-R Conflict)

The dirty read problem occurs *when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rollback, the updated database item is accessed by another transaction. There comes the Read-Write Conflict between both transactions.*

For example:

Consider two transactions T_x and T_y in the below diagram performing read/write operations on account A where the available balance in account A is $300:

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A + 50	—
t_3	WRITE (A)	—
t_4	—	READ (A)
t_5	SERVER DOWN ROLLBACK	—

DIRTY READ PROBLEM

- At time t1, transaction T_x reads the value of account A, i.e., $300.
- At time t2, transaction T_x adds $50 to account A that becomes $350.
- At time t3, transaction T_x writes the updated value in account A, i.e., $350.
- Then at time t4, transaction T_y reads account A that will be read as $350.
- Then at time t5, transaction T_x rollbacks due to server problem, and the value changes back to $300 (as initially).
- But the value for account A remains $350 for transaction T_y as committed, which is the dirty read and therefore known as the Dirty Read Problem.

Unrepeatable Read Problem (W-R Conflict)

Also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.

For example:

Consider two transactions, T_x and T_y, performing the read/write operations on account A, having an available balance = $300. The diagram is shown below:

Time	T_x	T_y
t_1	READ (A)	—
t_2	—	READ (A)
t_3	—	A = A + 100
t_4	—	WRITE (A)
t_5	READ (A)	—

UNREPEATABLE READ PROBLEM

- At time t1, transaction T_x reads the value from account A, i.e., $300.
- At time t2, transaction T_y reads the value from account A, i.e., $300.
- At time t3, transaction T_y updates the value of account A by adding $100 to the available balance, and then it becomes $400.
- At time t4, transaction T_y writes the updated value, i.e., $400.
- After that, at time t5, transaction T_x reads the available value of account A, and that will be read as $400.
- It means that within the same transaction T_x, it reads two different values of account A, i.e., $ 300 initially, and after updation made by transaction T_y, it reads $400. It is an unrepeatable read and is therefore known as the Unrepeatable read problem.

Thus, in order to maintain consistency in the database and avoid such problems that take place in concurrent execution, management is needed, and that is where the concept of Concurrency Control comes into role.

Concurrency Control

Concurrency Control is the working concept that is required for controlling and managing the concurrent execution of database operations and thus avoiding the inconsistencies in the database. Thus, for maintaining the concurrency of the database, we have the concurrency control protocols.

Concurrency Control Protocols

The concurrency control protocols ensure the *atomicity, consistency, isolation, durability* and *serializability* of the concurrent execution of the database transactions. Therefore, these protocols are categorized as:

- Lock Based Concurrency Control Protocol
- Time Stamp Concurrency Control Protocol

Lock Based Protocol

- Basic 2-PL
- Conservative 2-PL
- Strict 2-PL
- Rigorous 2-PL

Lock Based Protocols –

A lock is a variable associated with a data item that describes a status of data item with respect to possible operation that can be applied to it. They synchronize the access by concurrent transactions to the database items. It is required in this protocol that all the data items must be accessed in a mutually exclusive manner. Let me introduce you to two common locks which are used and some terminology followed in this protocol.

1. **Shared Lock (S):** also known as Read-only lock. As the name suggests it can be shared between transactions because while holding this lock the transaction does not have the permission to update data on the data item. S-lock is requested using lock-S instruction.
2. **Exclusive Lock (X):** Data item can be both read as well as written.This is Exclusive and cannot be held simultaneously on the same data item. X-lock is requested using lock-X instruction.

Lock Compatibility Matrix –

	S	X
S	✔	✘
X	✘	✘

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other
  transactions.
- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive(X) on the item no other transaction may hold any lock on the item.

If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. Then the lock is granted.

Problem With Simple Locking…

Consider the Partial Schedule:

	T_1	T_2
1	lock-X(B)	
2	read(B)	
3	B:=B-50	
4	write(B)	
5		lock-S(A)
6		read(A)
7		lock-S(B)
8	lock-X(A)	
9	……	……

Deadlock – consider the above execution phase. Now, T_1 holds an Exclusive lock over B, and T_2 holds a Shared lock over A. Consider Statement 7, T_2 requests for lock on B, while in Statement 8 T_1 requests lock on A. This as you may notice imposes a **Deadlock** as none can proceed with their execution.

Starvation – is also possible if concurrency control manager is badly designed. For example: A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item. This may be avoided if the concurrency control manager is properly designed.

2PL(Two Phase Locking Protocol)

Now, recalling where we last left off, there are two types of Locks available **Shared S(a)** and **Exclusive X(a)**. Implementing this lock system without any restrictions gives us the Simple Lock-based protocol (or *Binary Locking*), but it has its own disadvantages, they do **not guarantee Serializability**. Schedules may follow the preceding rules but a non-serializable schedule may result.
To guarantee serializability, we must follow some additional protocol *concerning the positioning of locking and unlocking operations* in every transaction. This is where the concept of Two-Phase Locking(2-PL) comes into the picture, 2-PL ensures serializability. Now, let's dig deep!

Two-Phase Locking –

A transaction is said to follow the Two-Phase Locking protocol if Locking and Unlocking can be done in two phases.

1. **Growing Phase:** New locks on data items may be acquired but none can be released.
2. **Shrinking Phase:** Existing locks may be released but no new locks can be acquired.

Note – If lock conversion is allowed, then upgrading of lock(from S(a) to X(a)) is allowed in the Growing Phase, and downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

Let's see a transaction implementing 2-PL.

	T_1	T_2
1	lock-S(A)	
2		lock-S(A)
3	lock-X(B)	
4	…….	……
5	Unlock(A)	
6		Lock-X(C)
7	Unlock(B)	
8		Unlock(A)
9		Unlock(C)
10	…….	……

This is just a skeleton transaction that shows how unlocking and locking work with 2-PL. Note for:

Transaction T_1:

- The growing Phase is from steps 1-3.
- The shrinking Phase is from steps 5-7.
- Lock Point at 3

Transaction T_2:

- The growing Phase is from steps 2-6.
- The shrinking Phase is from steps 8-9.
- Lock Point at 6

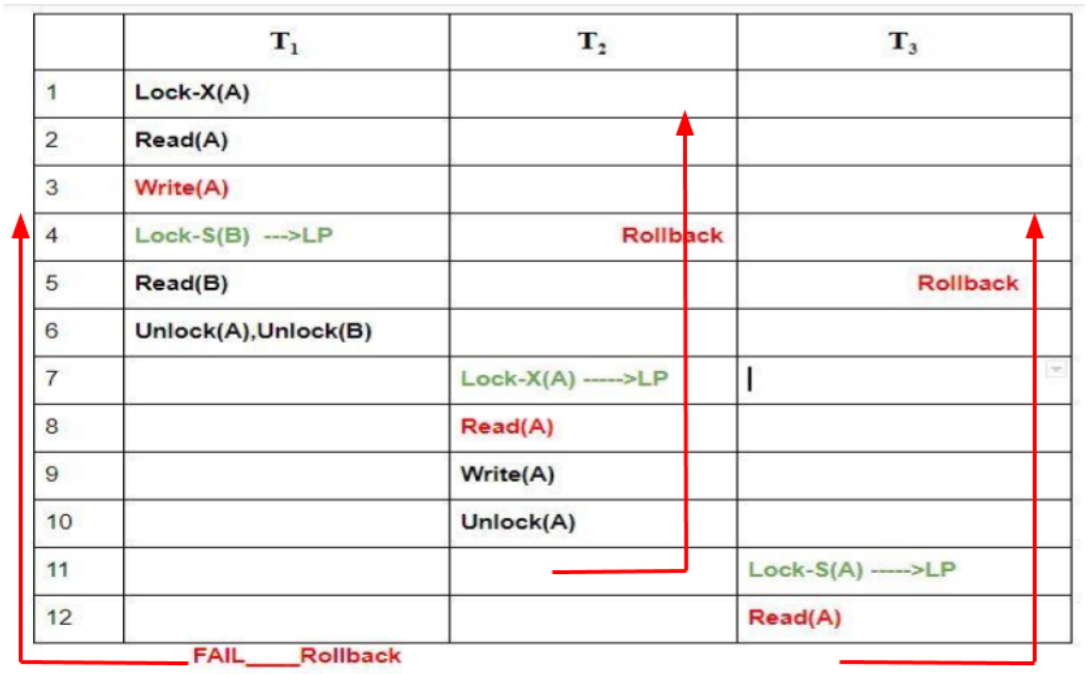
Hey, wait!

What is **LOCK POINT?** The Point at which the growing phase ends, i.e., when a transaction takes the final lock it needs to carry on its work. Now look at the schedule, you'll surely understand.

I have said that 2-PL ensures serializability, but there are still some drawbacks of 2-PL. Let's glance at the drawbacks:

- Cascading Rollback is possible under 2-PL.
- Deadlocks and Starvation are possible.

Cascading Rollbacks in 2-PL –
Let's see the following Schedule:



LP - Lock Point

Read(A) in T_2 and T_3 denotes Dirty Read because of Write(A) in T_1 .

Take a moment to analyze the schedule. Yes, you're correct, because of Dirty Read in T_2 and T_3 in lines 8 and 12 respectively, when T_1 failed we have to roll back others also. Hence, **Cascading Rollbacks are possible in 2-PL.** I have taken skeleton schedules as examples because it's easy to understand when it's kept simple. When explained with real-time transaction problems with many variables, it becomes very complex.

Deadlock in 2-PL –
Consider this simple example, it will be easy to understand. Say we have two transactions T_1 and T_2.

Schedule: Lock-X_1(A) Lock-X_2(B) Lock-X_1(B) Lock-X_2(A)

Drawing the precedence graph, you may detect the loop. So Deadlock is also possible in 2-PL.

Two-phase locking may also limit the amount of concurrency that occurs in a schedule because a Transaction may not be able to release an item after it has used it. This may be because of the protocols and other restrictions we may put on the schedule to ensure serializability, deadlock freedom, and other factors. This is the price we have to pay to ensure serializability and other factors, hence it can be considered as a bargain between concurrency and maintaining the ACID properties.

The above-mentioned type of 2-PL is called **Basic 2PL**. To sum it up it ensures Conflict Serializability but ***does not*** prevent Cascading Rollback and Deadlock. Further, we will study three other types of 2PL, Strict 2PL, Conservative 2PL, and Rigorous 2PL.

Rigorous two-phase locking is even stricter: here all locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

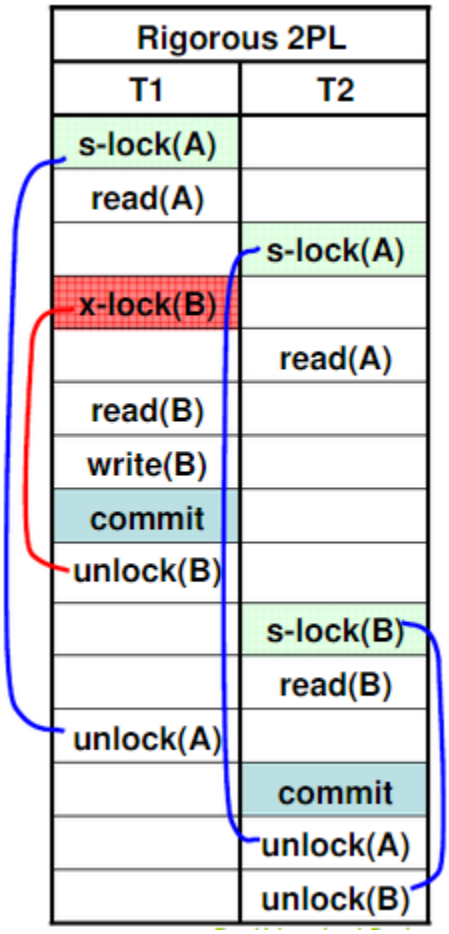
Much deeper :

Strict 2PL :

Strict 2PL	
T1	T2
s-lock(A)	
read(A)	
	s-lock(A)
x-lock(B)	
unlock(A)	
read(B)	
write(B)	
	read(A)
	unlock(A)
commit	
unlock(B)	
	s-lock(B)
	read(B)
	unlock(B)
	commit

Same as 2PL but Hold all exclusive locks until the transaction has already successfully committed or aborted. –It guarantees cascadeless recoverability

Rigorous 2PL :

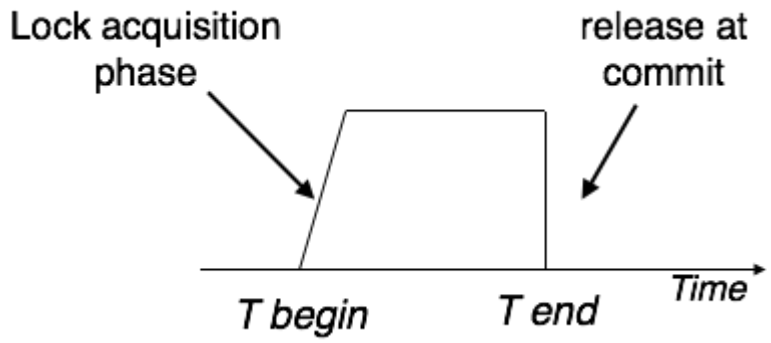


Same as Strict 2PL but Hold all locks until the transaction has already successfully committed or aborted. –It is used in dynamic environments where data access patterns are not known before hand.

There is no deadlock. Also, a younger transaction requesting an item held by an older transaction is aborted and restart with the same timestamp, starvation is avoided.

Strict Two-Phase Locking

The first phase of Strict-2PL is same as 2PL. After acquiring all the locks in the first phase, the transaction continues to execute normally. But in contrast to 2PL, Strict-2PL does not release a lock after using it. Strict-2PL holds all the locks until the commit point and releases all the locks at a time.



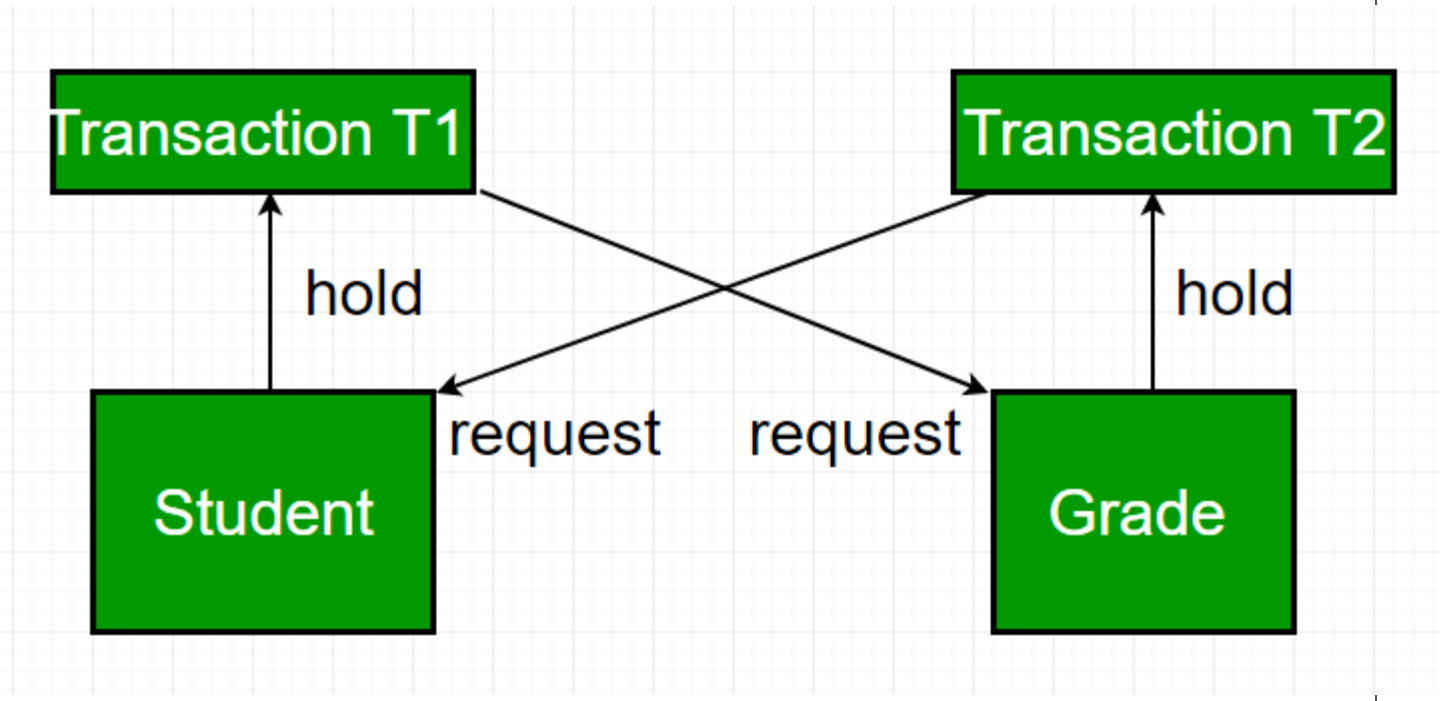
Strict-2PL does not have cascading abort as 2PL does.

Dead Lock:

In a database, a deadlock is an unwanted situation in which two or more transactions are waiting indefinitely for one another to give up locks. Deadlock is said to be one of the most feared complications in DBMS as it brings the whole system to a Halt.

Example – let us understand the concept of Deadlock with an example :
Suppose, Transaction T1 holds a lock on some rows in the Students table and **needs to update** some rows in the Grades table. Simultaneously, Transaction **T2 holds** locks on those very rows (Which T1 needs to update) in the Grades table **but needs** to update the rows in the Student table **held by Transaction T1**.

Now, the main problem arises. Transaction T1 will wait for transaction T2 to give up lock, and similarly, transaction T2 will wait for transaction T1 to give up the lock. As a consequence, All activity comes to a halt and remains at a standstill forever unless the DBMS detects the deadlock and aborts one of the transactions.



Deadlock in DBMS

Deadlock Avoidance –
When a database is stuck in a deadlock, It is always better to avoid the deadlock rather than restarting or aborting the database. Deadlock avoidance method is suitable for smaller databases whereas deadlock prevention method is suitable for larger databases.
One method of avoiding deadlock is using application-consistent logic. In the above given example, Transactions that access Students and Grades should always access the tables in the same order. In this way, in the scenario described above, Transaction T1 simply waits for transaction T2 to release the lock on Grades before it begins. When transaction T2 releases the lock, Transaction T1 can proceed freely.
Another method for avoiding deadlock is to apply both row-level locking mechanism and READ COMMITTED isolation level. However, It does not guarantee to remove deadlocks completely.

Timestamp Ordering Protocol

- The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
- The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.
- The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.
- Let's assume there are two transactions T1 and T2. Suppose the transaction T1 has entered the system at 007 times and transaction T2 has entered the system at 009 times. T1 has the higher priority, so it executes first as it is entered the system first.
- The timestamp ordering protocol also maintains the timestamp of last 'read' and 'write' operation on a data.

Basic Timestamp ordering protocol works as follows:

1. Check the following condition whenever a transaction Ti issues a **Read (X)** operation:

- If W_TS(X) >TS(Ti) then the operation is rejected.
- If W_TS(X) <= TS(Ti) then the operation is executed.
- Timestamps of all the data items are updated.

2. Check the following condition whenever a transaction Ti issues a **Write(X)** operation:

- If TS(Ti) < R_TS(X) then the operation is rejected.
- If TS(Ti) < W_TS(X) then the operation is rejected and Ti is rolled back otherwise the operation is executed.

Where,

TS(TI) denotes the timestamp of the transaction Ti.

R_TS(X) denotes the Read time-stamp of data-item X.

W_TS(X) denotes the Write time-stamp of data-item X.

Advantages and Disadvantages of TO protocol:

- TO protocol ensures serializability since the precedence graph is as follows:

Image: Precedence Graph for TS ordering

- TS protocol ensures freedom from deadlock that means no transaction ever waits.
- But the schedule may not be recoverable and may not even be cascade- free.

Example