UNIT-4(Transaction Processing Concept)

Transaction:

- The transaction is a set of logically related operation. It contains a group of tasks.

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

Open_Account(X)
Old_Balance = X.balance
New_Balance = Old_Balance - 800
X.balance = New_Balance
Close_Account(X)

Y's Account

Open_Account(Y)
Old_Balance = Y.balance
New_Balance = Old_Balance + 800
Y.balance = New_Balance
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. R(X);
2. X = X - 500;
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.

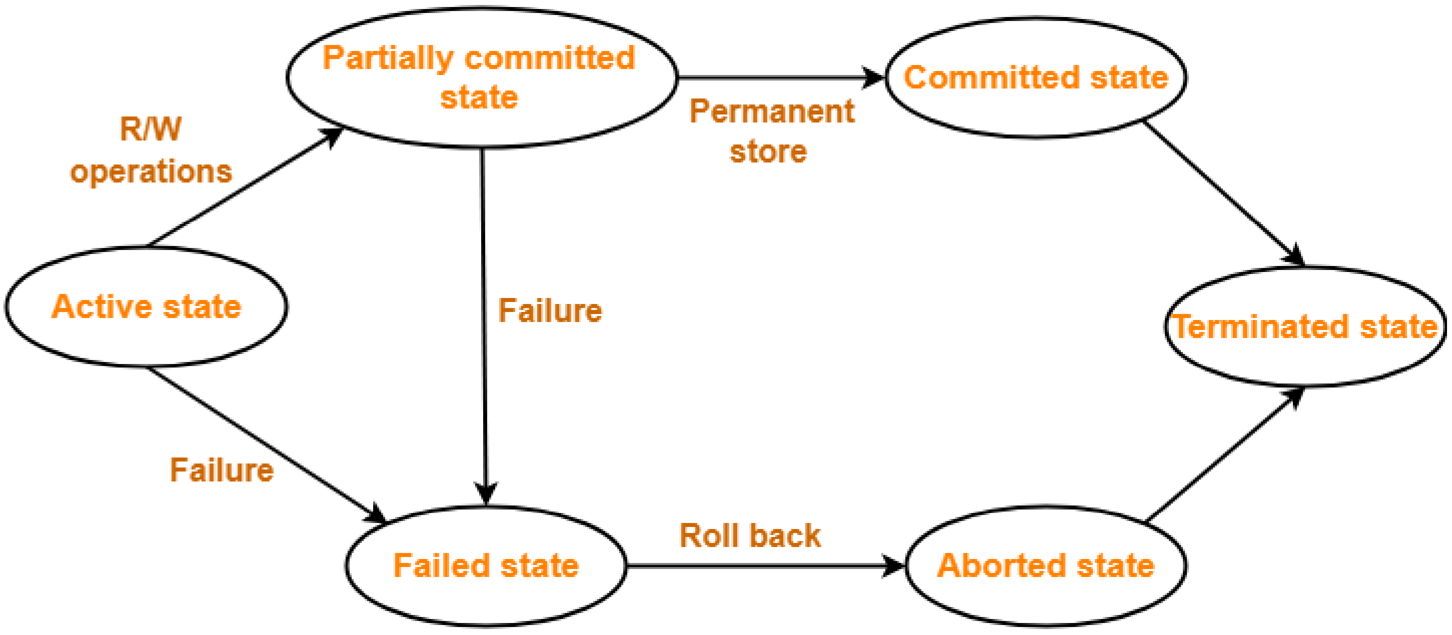
Transaction States-

A transaction goes through many different states throughout its life cycle.

These states are called as **transaction states**.

Transaction states are as follows-

1. Active state
2. Partially committed state
3. Committed state
4. Failed state
5. Aborted state
6. Terminated state



Transaction States in DBMS

Active State-

This is the first state in the life cycle of a transaction.

- A transaction is called in an **active state** as long as its instructions are getting executed.
- All the changes made by the transaction now are stored in the buffer in main memory.

2. Partially Committed State-

After the last instruction of transaction has executed, it enters into a **partially committed state**.

- After entering this state, the transaction is considered to be partially committed.
- It is not considered fully committed because all the changes made by the transaction are still stored in the buffer in main memory.

3. Committed State-

After all the changes made by the transaction have been successfully stored into the database, it enters into a **committed state**.

- Now, the transaction is considered to be fully committed.

NOTE-

After a transaction has entered the committed state, it is not possible to roll back the transaction.

- In other words, it is not possible to undo the changes that has been made by the transaction.
- This is because the system is updated into a new consistent state.
- The only way to undo the changes is by carrying out another transaction called as **compensating transaction** that performs the reverse operations.

4. 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**.

5. 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**.

6. 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.

ACID Properties | ACID Properties in DBMS

ACID Properties-

- It is important to ensure that the database remains consistent before and after the transaction.
- To ensure the consistency of database, certain properties are followed by all the transactions occurring in the system.
- These properties are called as **ACID Properties** of a transaction.

A = Atomicity

C = Consistency

I = Isolation

D = Durability

1. Atomicity-

- This property ensures that either the transaction occurs completely or it does not occur at all.
- In other words, it ensures that no transaction occurs partially.
- That is why, it is also referred to as "**All or nothing rule**".
- It is the responsibility of Transaction Control Manager to ensure atomicity of the transactions.

2. Consistency-

- This property ensures that integrity constraints are maintained.
- In other words, it ensures that the database remains consistent before and after the transaction.
- It is the responsibility of DBMS and application programmer to ensure consistency of the database.

3. Isolation-

- This property ensures that multiple transactions can occur simultaneously without causing any inconsistency.
- During execution, each transaction feels as if it is getting executed alone in the system.
- A transaction does not realize that there are other transactions as well getting executed parallely.
- Changes made by a transaction becomes visible to other transactions only after they are written in the memory.
- The resultant state of the system after executing all the transactions is same as the state that would be achieved if the transactions were executed serially one after the other.
- It is the responsibility of concurrency control manager to ensure isolation for all the transactions.

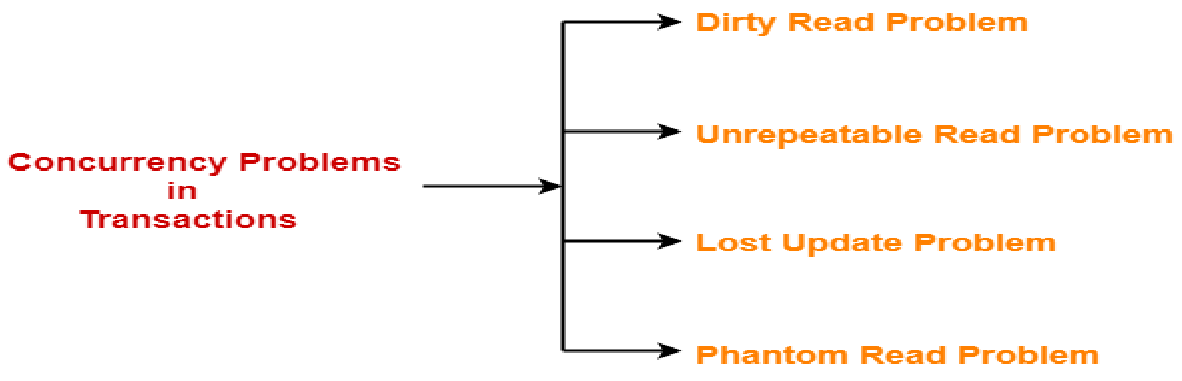
4. Durability-

- This property ensures that all the changes made by a transaction after its successful execution are written successfully to the disk.
- It also ensures that these changes exist permanently and are never lost even if there occurs a failure of any kind.
- It is the responsibility of recovery manager to ensure durability in the database.

Concurrency Problems in DBMS-

When multiple transactions execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems. Such problems are called as **concurrency problems**.

The concurrency problems are-


Concurrency Problems in Transactions
Dirty Read Problem
Unrepeatable Read Problem
Lost Update Problem
Phantom Read Problem

1. Dirty Read Problem
2. Unrepeatable Read Problem
3. Lost Update Problem
4. Phantom Read Problem

1. Dirty Read Problem-

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

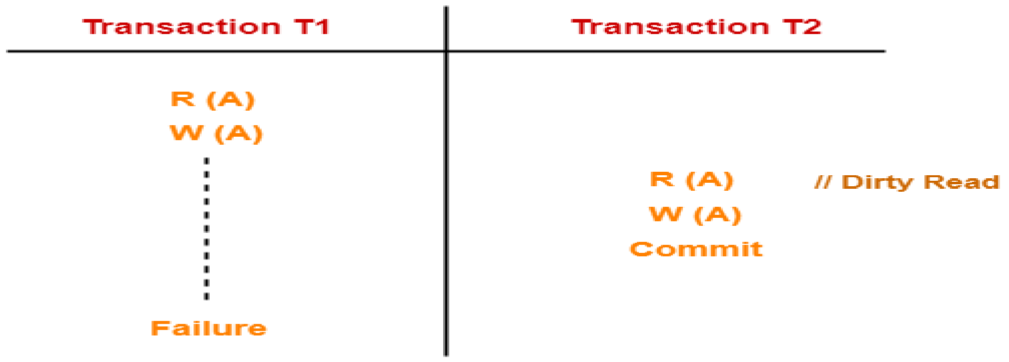
This read is called as dirty read because-

- There is always a chance that the uncommitted transaction might roll back later.
- Thus, uncommitted transaction might make other transactions read a value that does not even exist.
- This leads to inconsistency of the database.

NOTE-

- Dirty read does not lead to inconsistency always.
- It becomes problematic only when the uncommitted transaction fails and roll backs later due to some reason.

Example-



Here,

1. T1 reads the value of A.
2. T1 updates the value of A in the buffer.
3. T2 reads the value of A from the buffer.
4. T2 writes the updated the value of A.
5. T2 commits.
6. T1 fails in later stages and rolls back.

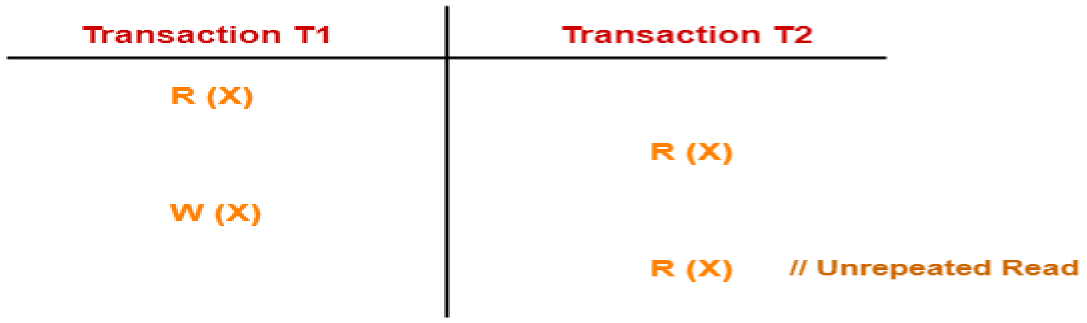
In this example,

- T2 reads the dirty value of A written by the uncommitted transaction T1.
- T1 fails in later stages and roll backs.
- Thus, the value that T2 read now stands to be incorrect.
- Therefore, database becomes inconsistent.

2. Unrepeatable Read Problem-

This problem occurs when a transaction gets to read unrepeated i.e. different values of the same variable in its different read operations even when it has not updated its value.

Example-



Here,

1. T1 reads the value of X (= 10 say).
2. T2 reads the value of X (= 10).
3. T1 updates the value of X (from 10 to 15 say) in the buffer.
4. T2 again reads the value of X (but = 15).

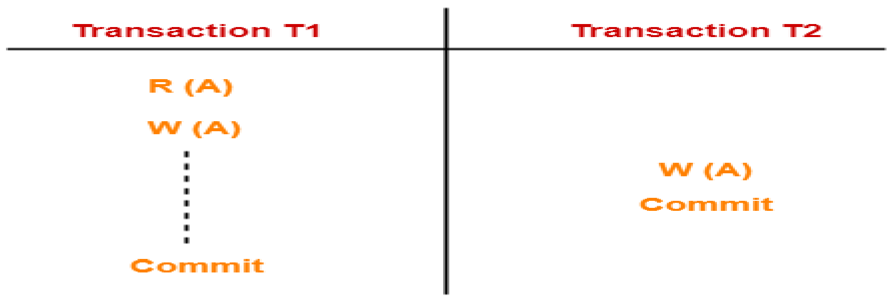
In this example,

- T2 gets to read a different value of X in its second reading.
- T2 wonders how the value of X got changed because according to it, it is running in isolation.

3. Lost Update Problem-

This problem occurs when multiple transactions execute concurrently and updates from one or more transactions get lost.

Example-



Here,

1. T1 reads the value of A (= 10 say).
2. T1 updates the value to A (= 15 say) in the buffer.
3. T2 does blind write A = 25 (write without read) in the buffer.
4. T2 commits.
5. When T1 commits, it writes A = 25 in the database.

In this example,

- T1 writes the over written value of A in the database.
- Thus, update from T1 gets lost.

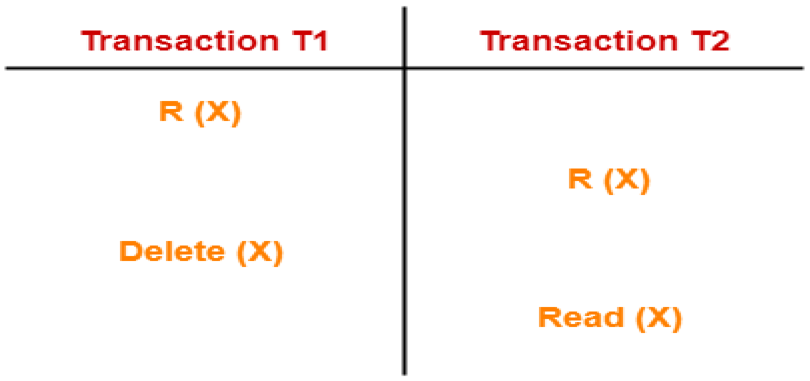
NOTE-

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

4. Phantom Read Problem-

This problem occurs when a transaction reads some variable from the buffer and when it reads the same variable later, it finds that the variable does not exist.

Example-



Transaction T1	Transaction T2
R (X)	
	R (X)
Delete (X)	
	Read (X)

Here,

1. T1 reads X.
2. T2 reads X.
3. T1 deletes X.
4. T2 tries reading X but does not find it.

In this example,

- T2 finds that there does not exist any variable X when it tries reading X again.
- T2 wonders who deleted the variable X because according to it, it is running in isolation.

Avoiding Concurrency Problems-

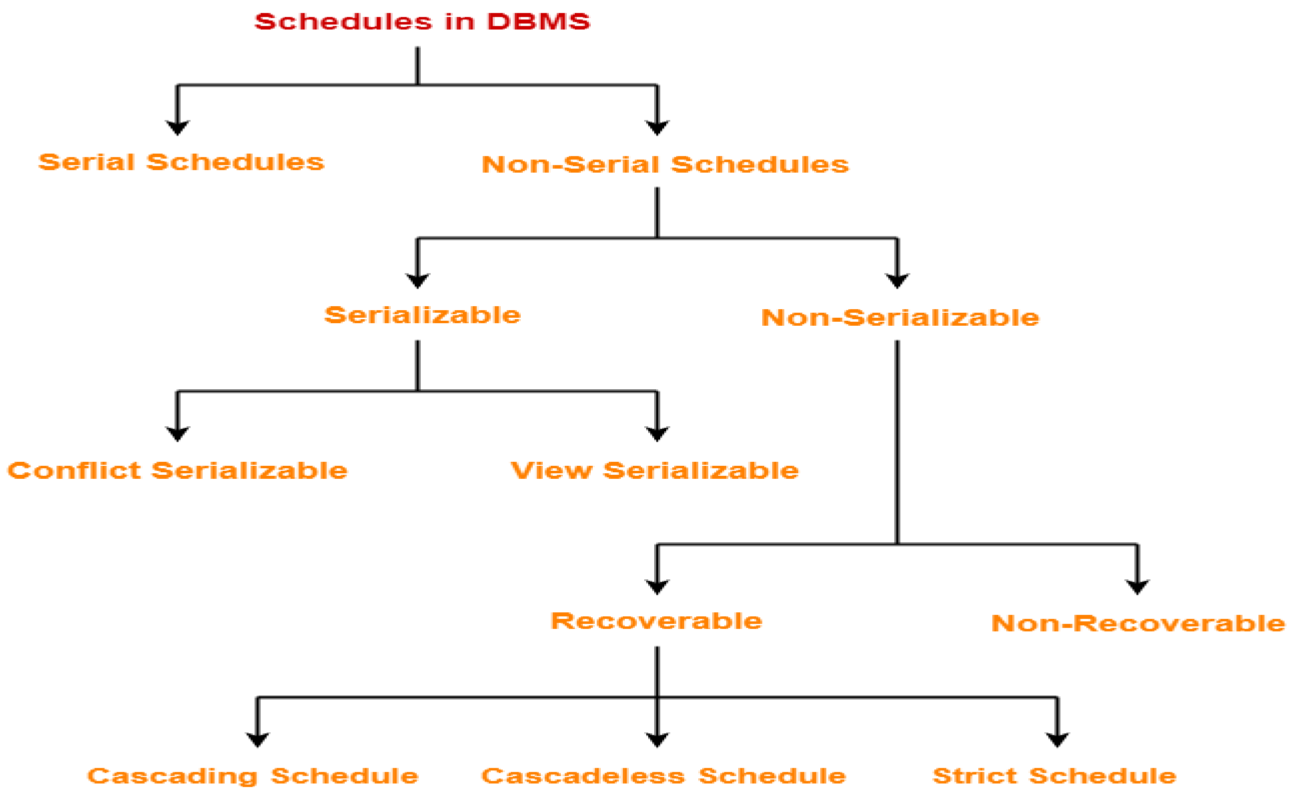
- To ensure consistency of the database, it is very important to prevent the occurrence of above problems.
- Concurrency Control Protocols help to prevent the occurrence of above problems and maintain the consistency of the database.

Schedules in DBMS-

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

Types of Schedules-

In DBMS, schedules may be classified as-

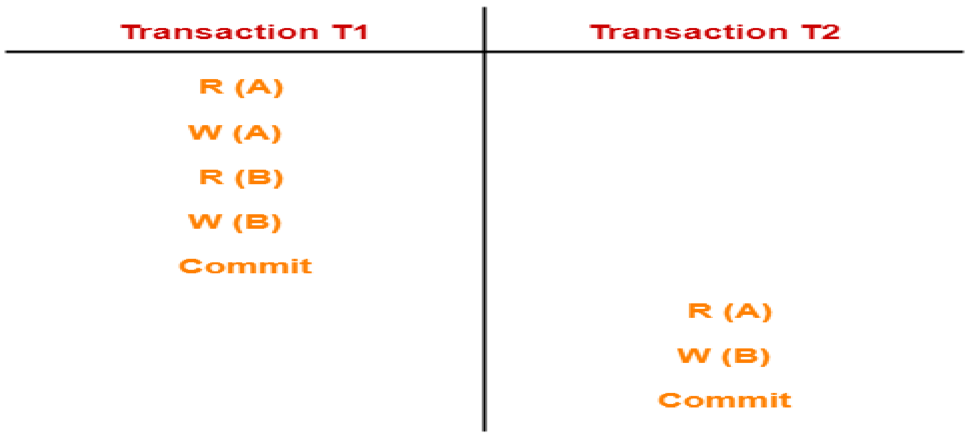


Serial Schedules-

In serial schedules,

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

Example-01:



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

In this schedule,

- There are two transactions T1 and T2 executing serially one after the other.
- Transaction T1 executes first.
- After T1 completes its execution, transaction T2 executes.
- So, this schedule is an example of a Serial Schedule.

Example-02:

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

In this schedule,

- There are two transactions T1 and T2 executing serially one after the other.
- Transaction T2 executes first.
- After T2 completes its execution, transaction T1 executes.
- So, this schedule is an example of a Serial Schedule.

Non-Serial Schedules-

In non-serial schedules,

- Multiple transactions execute concurrently.
- Operations of all the transactions are inter leaved or mixed with each other.

Example-01:

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

In this schedule,

- There are two transactions T1 and T2 executing concurrently.
- The operations of T1 and T2 are interleaved.
- So, this schedule is an example of a Non-Serial Schedule.

Example-02:

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

In this schedule,

- There are two transactions T1 and T2 executing concurrently.
- The operations of T1 and T2 are interleaved.
- So, this schedule is an example of a **Non-Serial Schedule**.

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-

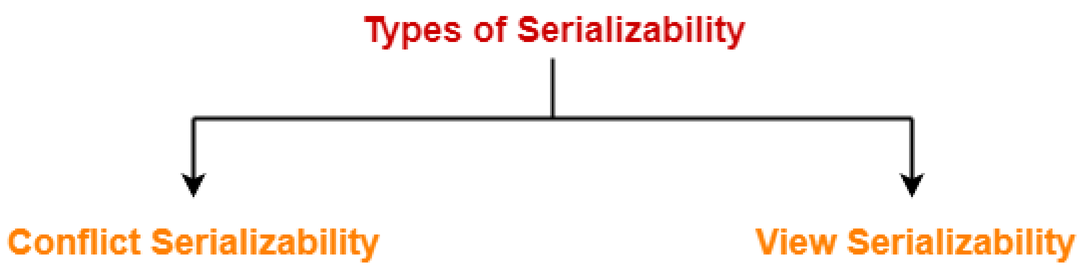
If a given non-serial schedule of 'n' transactions is equivalent to some serial schedule of 'n' transactions, then it is called as a **serializable schedule**.

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.

Exampe-01:

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

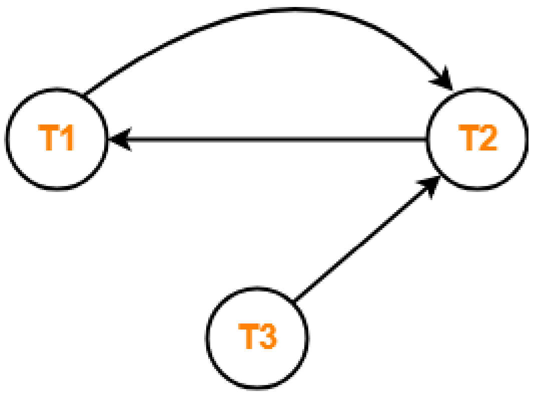
$$S : R_1(A) , R_2(A) , R_1(B) , R_2(B) , R_3(B) , W_1(A) , W_2(B)$$

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.

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

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-

$$\begin{tikzcd}
T1 \arrow[rr, bend left] & T2 \arrow[l] \arrow[r] \arrow[d] & T4 \\
 & T3 \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

Clearly, there exists no cycle in the precedence graph.

- Therefore, the given schedule S is conflict serializable.

View Serializability

- A schedule will view serializable if it is view equivalent to a serial schedule.
- If a schedule is conflict serializable, then it will be view serializable.
- The view serializable which does not conflict serializable contains blind writes.

View Equivalent

Two schedules S1 and S2 are said to be view equivalent if they satisfy the following conditions:

1. Initial Read

An initial read of both schedules must be the same. Suppose two schedule S1 and S2. In schedule S1, if a transaction T1 is reading the data item A, then in S2, transaction T1 should also read A.

T1	T2
Read(A)	
	Write(A)

Schedule S1

T1	T2
	Write(A)
Read(A)	

Schedule S2

Above two schedules are view equivalent because Initial read operation in S1 is done by T1 and in S2 it is also done by T1.

2. Updated Read

In schedule S1, if Ti is reading A which is updated by Tj then in S2 also, Ti should read A which is updated by Tj.

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

Schedule S1

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

Schedule S2

3. Final Write

A final write must be the same between both the schedules. In schedule S1, if a transaction T1 updates A at last then in S2, final writes operations should also be done by T1.

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

Schedule S1

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

Schedule S2

Above two schedules is view equal because Final write operation in S1 is done by T3 and in S2, the final write operation is also done by T3.

Recoverability of Schedule

Irrecoverable Schedules-

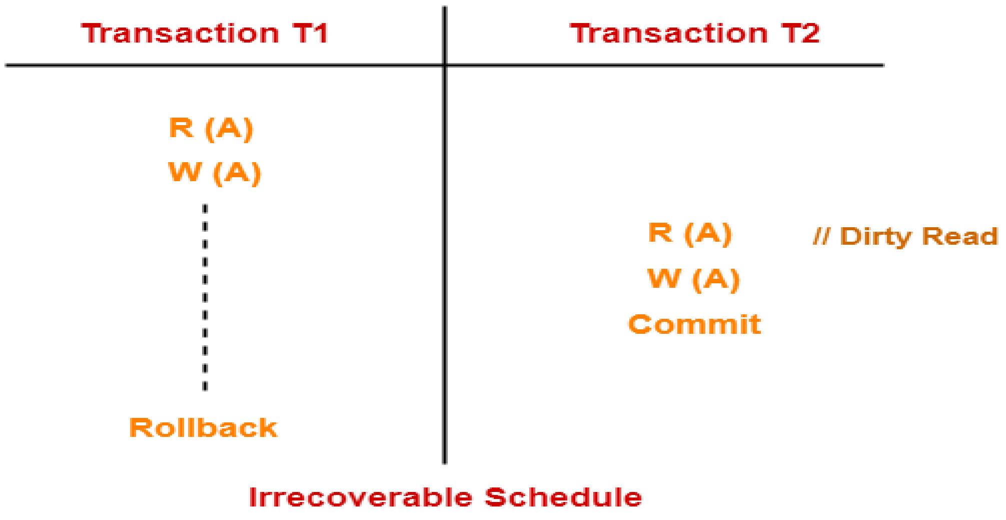
If in a schedule,

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

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

Example-

Consider the following 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

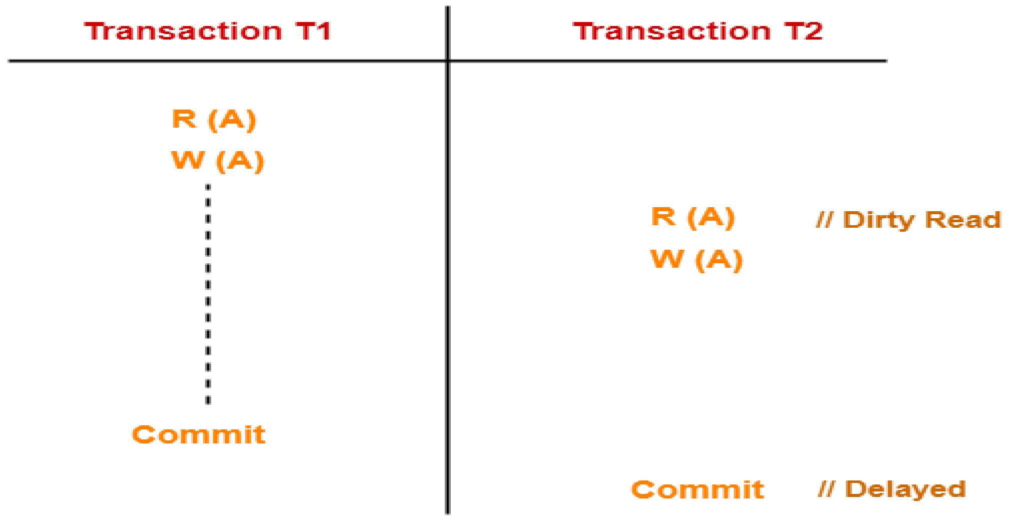
then such a schedule is known as a **Recoverable Schedule**.

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-



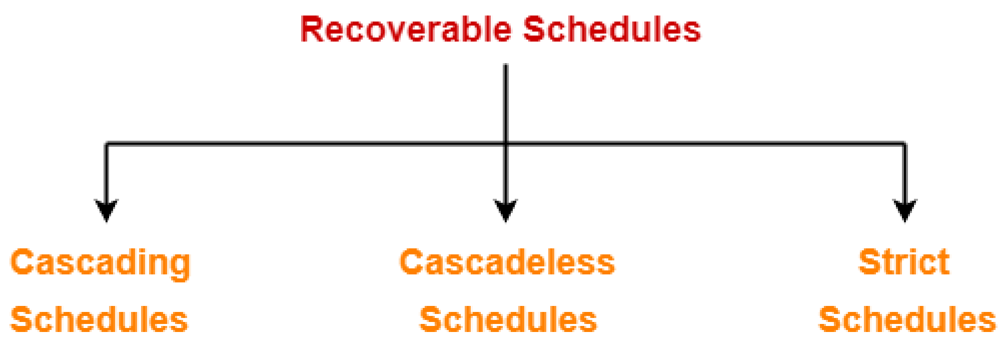
Recoverable Schedule

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.

Types of Recoverable Schedules-

A recoverable schedule may be any one of these kinds-



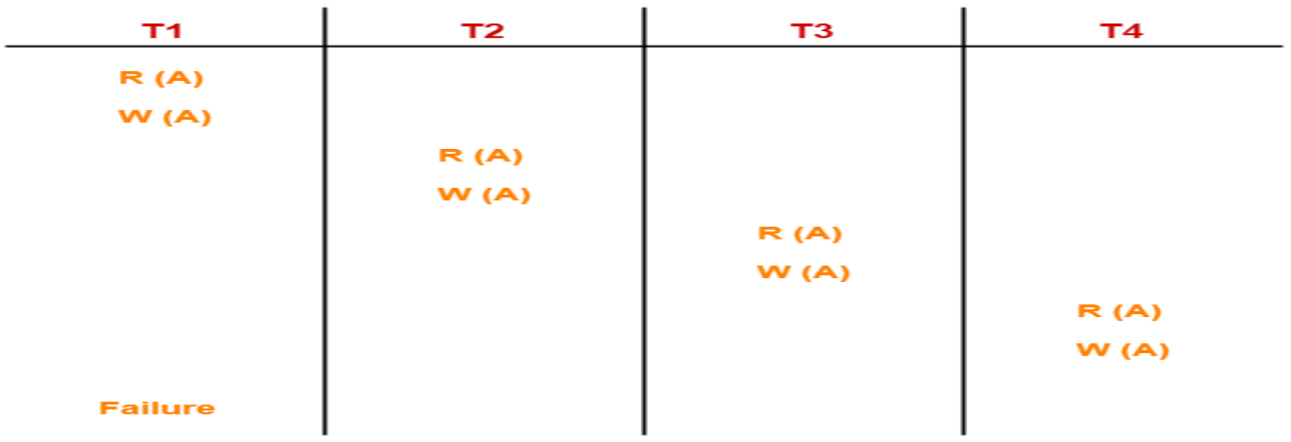
1. Cascading Schedule
2. Cascadeless Schedule
3. Strict Schedule

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-



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

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.

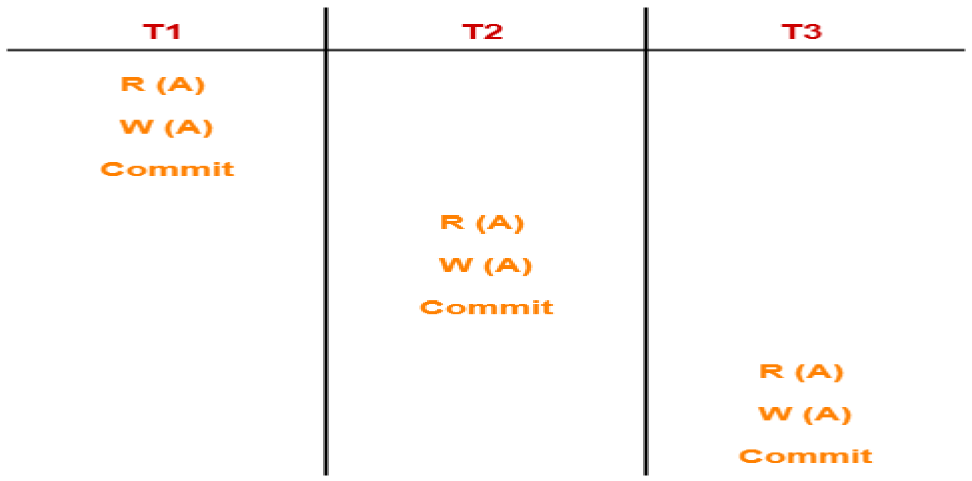
Cascadeless Schedule-

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-



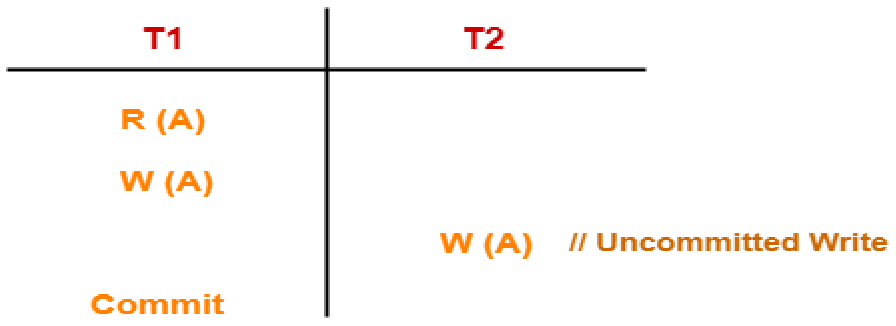
T1	T2	T3
R (A)		
W (A)		
Commit		
	R (A)	
	W (A)	
	Commit	
		R (A)
		W (A)
		Commit

Cascadeless Schedule

NOTE-

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

Example-



T1	T2
R (A)	
W (A)	
	W (A) // Uncommitted Write
Commit	

Cascadeless Schedule

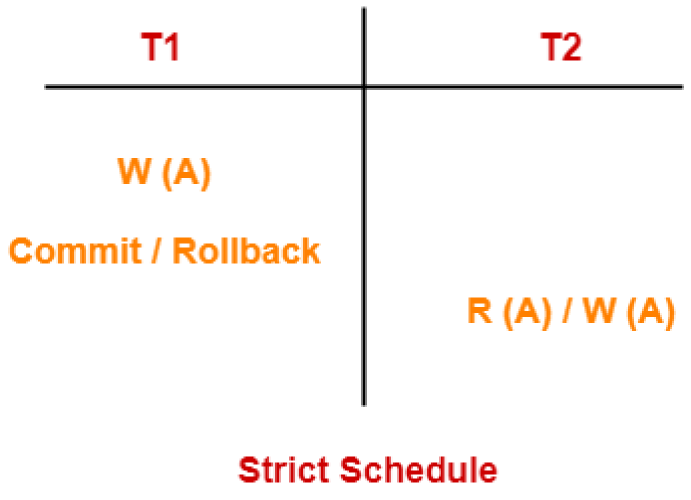
Strict Schedule-

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,

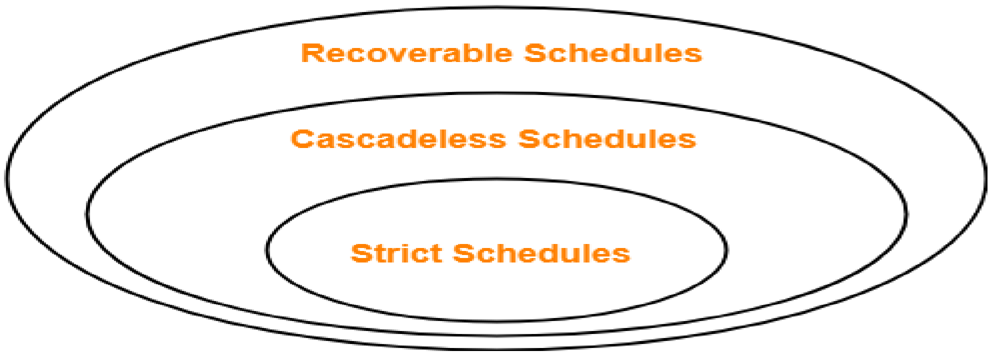
- Strict schedule allows only committed read and write operations.
- Clearly, strict schedule implements more restrictions than cascadeless schedule.

Example-



Remember-

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



Failure Classification

To find that where the problem has occurred, we generalize a failure into the following categories:

1. Transaction failure
2. System crash
3. Disk failure

1. Transaction failure

The transaction failure occurs when it fails to execute or when it reaches a point from where it can't go any further. If a few transaction or process is hurt, then this is called as transaction failure.

Reasons for a transaction failure could be -

1. **Logical errors:** If a transaction cannot complete due to some code error or an internal error condition, then the logical error occurs.
2. **Syntax error:** It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it. **For example,** The system aborts an active transaction, in case of deadlock or resource unavailability.

2. System Crash

   - System failure can occur due to power failure or other hardware or software failure. **Example:** Operating system error.

     Fail-stop assumption: In the system crash, non-volatile storage is assumed not to be corrupted.

3. Disk Failure

   - It occurs where hard-disk drives or storage drives used to fail frequently. It was a common problem in the early days of technology evolution.
   - Disk failure occurs due to the formation of bad sectors, disk head crash, and unreachability to the disk or any other failure, which destroy all or part of disk storage.

Log-Based Recovery

- The log is a sequence of records. Log of each transaction is maintained in some stable storage so that if any failure occurs, then it can be recovered from there.
- If any operation is performed on the database, then it will be recorded in the log.
- But the process of storing the logs should be done before the actual transaction is applied in the database.

For example, there is a transaction to modify the City of a student. The following logs are written for this transaction.

- When the transaction is initiated, then it writes 'start' log.
  1. <Tn, Start>
- When the transaction modifies the City from 'Noida' to 'Bangalore', then another log is written to the file.

  1. <Tn, City, 'Noida', 'Bangalore' >
- When the transaction is finished, then it writes another log to indicate the end of the transaction.

  1. <Tn, Commit>

There are two approaches to modify the database:

1. Deferred database modification:

- The deferred modification technique occurs if the transaction does not modify the database until it has committed.
- In this method, all the logs are created and stored in the stable storage, and the database is updated when a transaction commits.

2. Immediate database modification:

- The Immediate modification technique occurs if database modification occurs while the transaction is still active.
- In this technique, the database is modified immediately after every operation. It follows an actual database modification.

Recovery using Log records

When the system is crashed, then the system consults the log to find which transactions need to be undone and which need to be redone.

1. If the log contains the record <Ti, Start> and <Ti, Commit> or <Ti, Commit>, then the Transaction Ti needs to be redone.

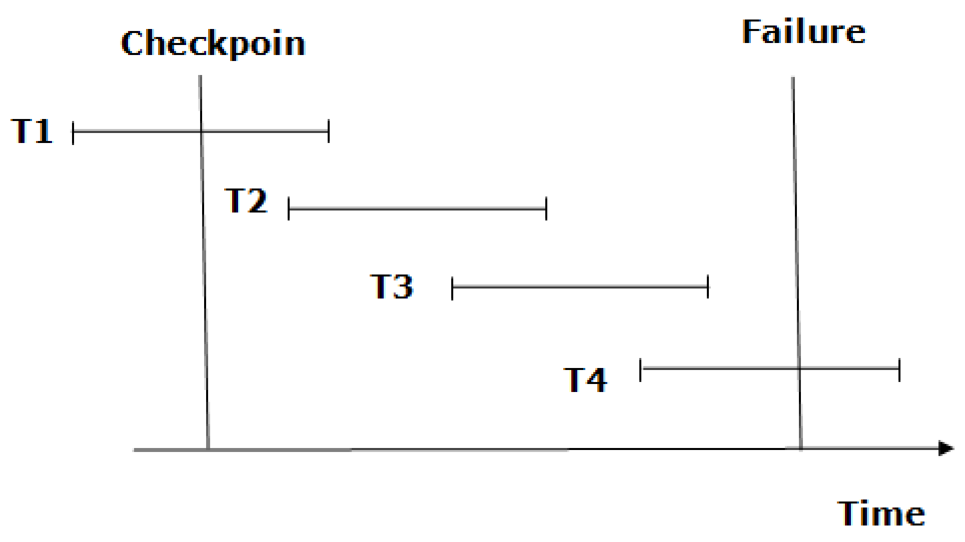
2. If log contains record<T_n, Start> but does not contain the record either <Ti, commit> or <Ti, abort>, then the Transaction Ti needs to be undone.

Checkpoint

- The checkpoint is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk.
- The checkpoint is like a bookmark. While the execution of the transaction, such checkpoints are marked, and the transaction is executed then using the steps of the transaction, the log files will be created.
- When it reaches to the checkpoint, then the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction till next checkpoint and so on.
- The checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transactions were committed.

Recovery using Checkpoint

In the following manner, a recovery system recovers the database from this failure:



- The recovery system reads log files from the end to start. It reads log files from T4 to T1.
- Recovery system maintains two lists, a redo-list, and an undo-list.
- The transaction is put into redo state if the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just <Tn, Commit>. In the redo-list and their previous list, all the transactions are removed and then redone before saving their logs.
- **For example:** In the log file, transaction T2 and T3 will have <Tn, Start> and <Tn, Commit>. The T1 transaction will have only <Tn, commit> in the log file. That's why the transaction is committed after the checkpoint is crossed. Hence it puts T1, T2 and T3 transaction into redo list.
- The transaction is put into undo state if the recovery system sees a log with <Tn, Start> but no commit or abort log found. In the undo-list, all the transactions are undone, and their logs are removed.
- **For example:** Transaction T4 will have <Tn, Start>. So T4 will be put into undo list since this transaction is not yet complete and failed amid.

Deadlock in DBMS:

A deadlock is a condition where 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 no task ever gets finished and is in waiting state forever.

For example: In the student table, transaction T1 holds a lock on some rows and needs to update some rows in the grade table. Simultaneously, transaction T2 holds locks on some rows in the grade table and needs to update the rows in the Student table held by Transaction T1.

Now, the main problem arises. Now Transaction T1 is waiting for T2 to release its lock and similarly, transaction T2 is waiting for T1 to release its lock. All activities come to a halt state and remain at a standstill. It will remain in a standstill until the DBMS detects the deadlock and aborts one of the transactions.

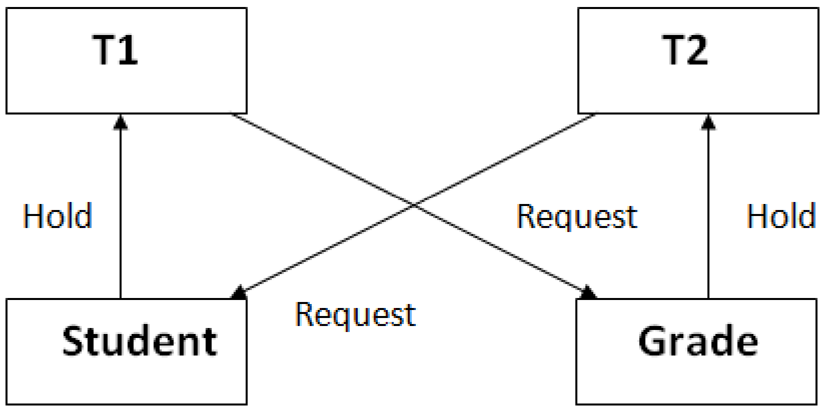


Figure: Deadlock in DBMS

Deadlock Avoidance

- When a database is stuck in a deadlock state, then it is better to avoid the database rather than aborting or restating the database. This is a waste of time and resource.
- Deadlock avoidance mechanism is used to detect any deadlock situation in advance. A method like "wait for graph" is used for detecting the deadlock situation but this method is suitable only for the smaller database. For the larger database, deadlock prevention method can be used.

Deadlock Detection

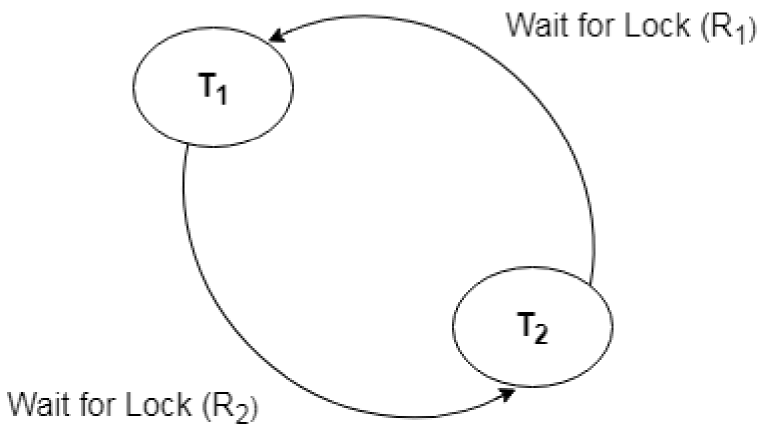
In a database, when a transaction waits indefinitely to obtain a lock, then the DBMS should detect whether the transaction is involved in a deadlock or not. The lock manager maintains a Wait for the graph to detect the deadlock cycle in the database.

Wait for Graph

- This is the suitable method for deadlock detection. In this method, a graph is created based on the transaction and their lock. If the created graph has a cycle or closed loop, then there is a deadlock.

- The wait for the graph is maintained by the system for every transaction which is waiting for some data held by the others. The system keeps checking the graph if there is any cycle in the graph.

The wait for a graph for the above scenario is shown below:



Deadlock Prevention

- Deadlock prevention method is suitable for a large database. If the resources are allocated in such a way that deadlock never occurs, then the deadlock can be prevented.
- The Database management system analyzes the operations of the transaction whether they can create a deadlock situation or not. If they do, then the DBMS never allowed that transaction to be executed.

Wait-Die scheme

In this scheme, if a transaction requests for a resource which is already held with a conflicting lock by another transaction then the DBMS simply checks the timestamp of both transactions. It allows the older transaction to wait until the resource is available for execution.

Let's assume there are two transactions Ti and Tj and let TS(T) is a timestamp of any transaction T. If T2 holds a lock by some other transaction and T1 is requesting for resources held by T2 then the following actions are performed by DBMS:

1. Check if TS(Ti) < TS(Tj) - If Ti is the older transaction and Tj has held some resource, then Ti is allowed to wait until the data-item is available for execution. That means if the older transaction is waiting for a resource which is locked by the younger transaction, then the older transaction is allowed to wait for resource until it is available.
2. Check if TS(T_i) < TS(Tj) - If Ti is older transaction and has held some resource and if Tj is waiting for it, then Tj is killed and restarted later with the random delay but with the same timestamp.

Wound wait scheme

- In wound wait scheme, if the older transaction requests for a resource which is held by the younger transaction, then older transaction forces younger one to kill the transaction and release the resource. After the minute delay, the younger transaction is restarted but with the same timestamp.
- If the older transaction has held a resource which is requested by the Younger transaction, then the younger transaction is asked to wait until older releases it.