25CS305
Database Management Systems

M. NARMADHA,

Assistant Professor,

Computer Science and Engineering

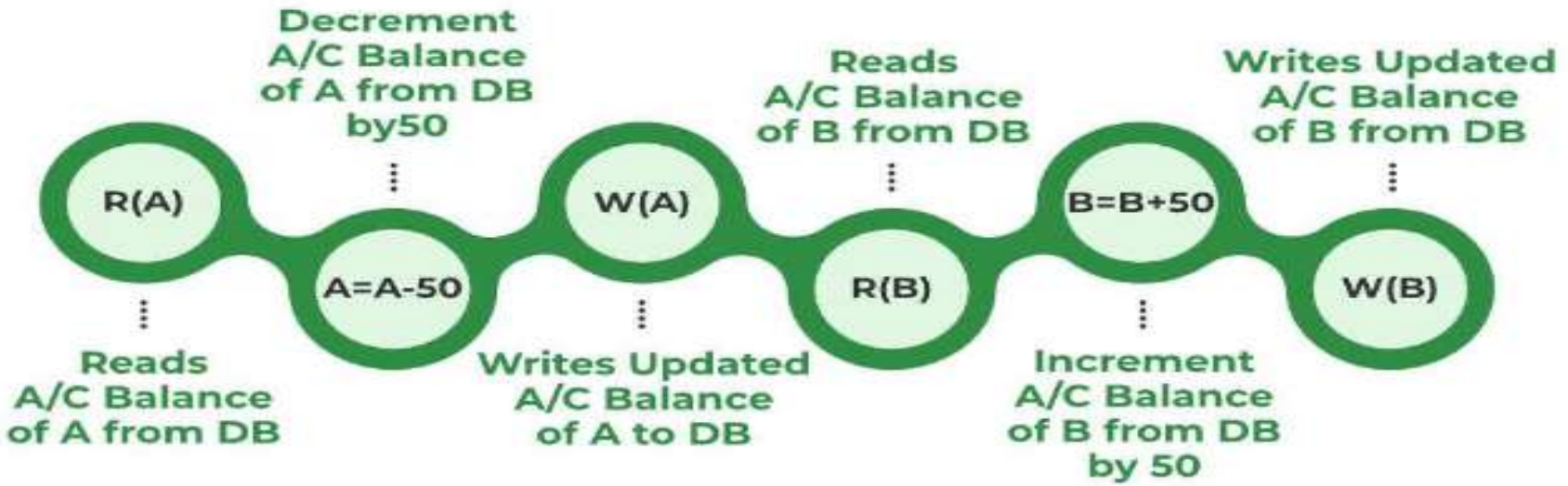
NARSIMHA REDDY ENGINEERING COLLEGE
UGC AUTONOMOUS INSTITUTION
Maisammaguda (V), Kompally - 500100, Secunderabad, Telangana State, India
Accredited by NBA & NAAC with 'A' Grade
Approved by AICTE
Permanently affiliated to JNTUH

UNIT-V

Transaction Concept

- A transaction can be defined as a group of tasks. A single task is the minimum processing unit which cannot be divided further.

- Let's take an example of a simple transaction. Suppose a bank employee transfers Rs 50 from A's account to B's account. This very simple and small transaction involves several low-level tasks.


Decrement A/C Balance of A from DB by50
Reads A/C Balance of B from DB
Writes Updated A/C Balance of B from DB
R(A)
A=A-50
W(A)
R(B)
B=B+50
W(B)
Reads A/C Balance of A from DB
Writes Updated A/C Balance of A to DB
Increment A/C Balance of B from DB by 50

Desirable Properties of Transaction (ACID Properties)

For a transaction to be performed in DBMS, it must possess several properties often called ACID properties.

- **A** – Atomicity
- **C** – Consistency
- **I** – Isolation
- **D** – Durability

Atomicity. Either all operations of the transaction are reflected properly in the database, or none .

- **Consistency**. Execution of a transaction in isolation preserves the consistency of the database.

- **Isolation**. Even though multiple transactions may execute concurrently, the system guarantees that,

for every pair of transactions Ti and Tj,

it appears to Ti that either Tj finished execution before Ti started, or

Tj started execution after Ti finished.

Thus, each transaction is unaware of other transactions executing concurrently in the system.

- Durability. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction Atomicity and Durability or Transaction States

Transaction States

Transaction must be in one of the following states:

- Active : the initial state; the transaction stays in this state while it is executing
- Partially committed : after the final statement has been executed
- Failed : after the discovery that normal execution can no longer proceed
- Aborted : after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction
- Committed : after successful completion

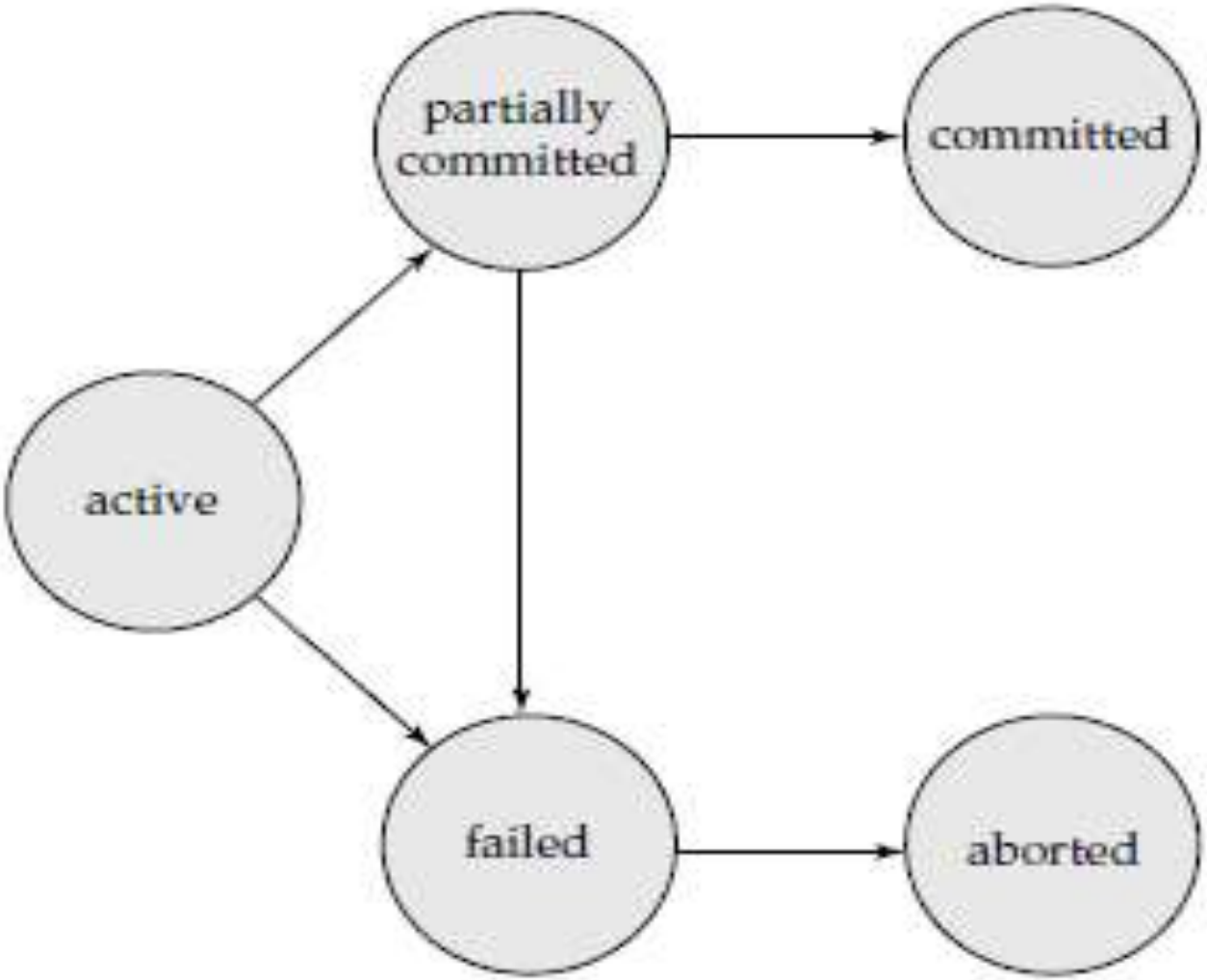


Figure State diagram of a transaction.

Serializability

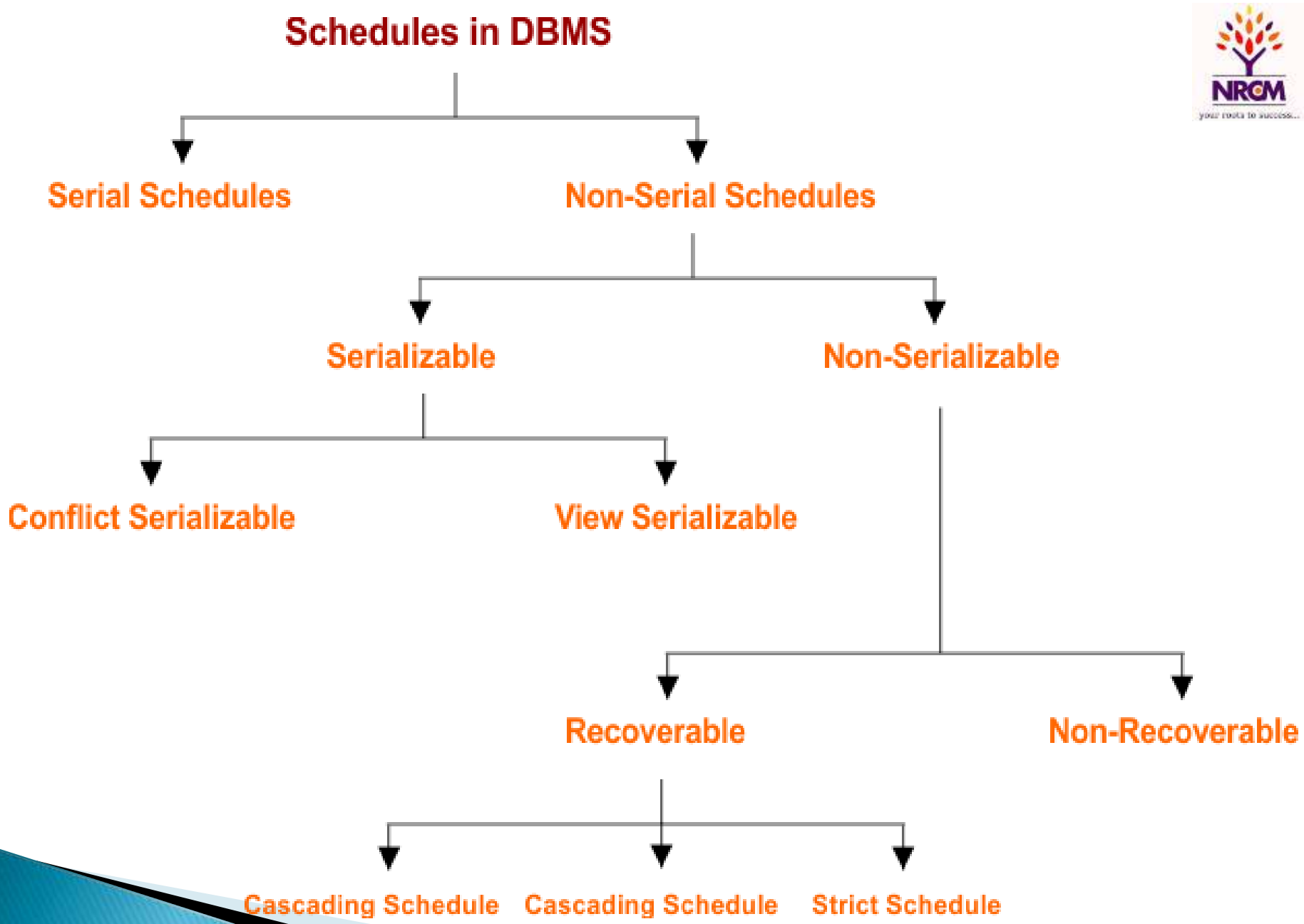
Serializability

❖Serializability is a concept in database management systems (DBMS) that **ensures the correctness of concurrent transactions**.

❖Serializability in DBMS guarantees that the execution of multiple transactions in parallel does not produce any unexpected or incorrect results.

There are mainly two types of schedules.

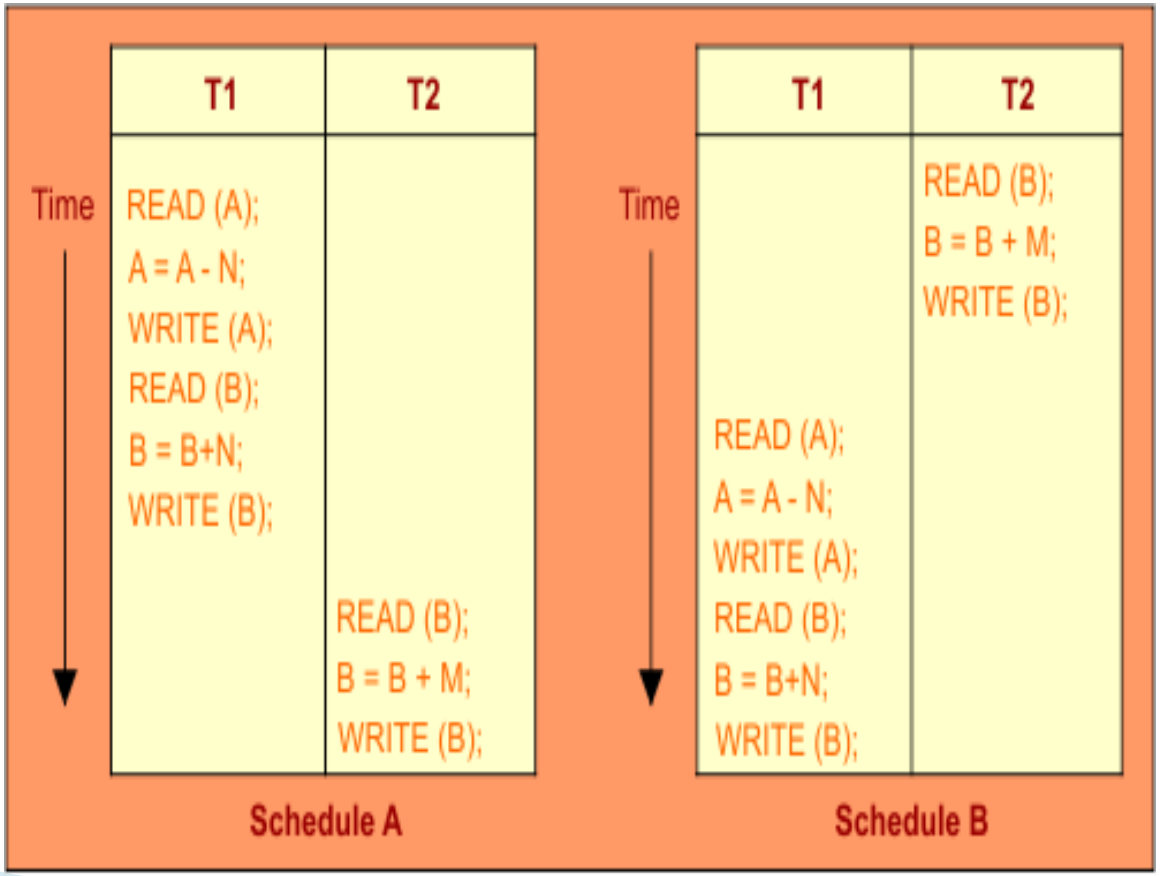
1. Serial Schedule
2. Non-Serial Schedule


Schedules in DBMS
Serial Schedules
Non-Serial Schedules
Serializable
Non-Serializable
Conflict Serializable
View Serializable
Recoverable
Non-Recoverable
Cascading Schedule
Cascading Schedule
Strict Schedule

1. 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.



Schedules Both A and B are Serial

Advantage of Serial Schedule

Main benefit of serial schedule is that there is no concurrency problem.

Disadvantage of Serial Schedule

It is time wasting approach because if T1,T2,T3 arrive at same time then T2,T3 has to wait until T1 is completed.

2. Non-Serial (Parallel) Schedule

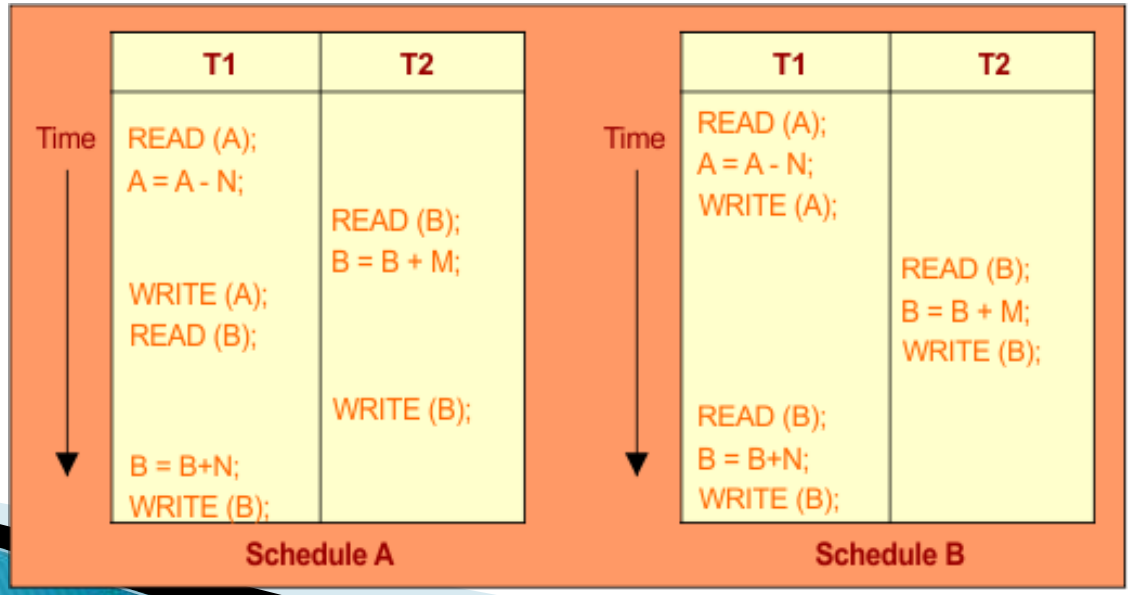
In the non-serial schedule, the other transaction proceeds without waiting for the previous transaction to complete.

Non-serial schedule can be of two types

- Serializable
- Non-Serializable Schedule.

Example: following schedules are non-serial where T1 and T2 are running concurrently.

- The operations of T1 and T2 are interleaved.
- So, the following schedules are an example of a Non-Serial Schedule.



Schedules Both A and B are Parallel

Types of Serializability in DBMS

There are mainly two types of Serializability in DBMS.

1. Conflict Serializability

2. View Serializability

1. 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

Let's consider two different transactions, *Ti* and *Tj*. Then considering the above conditions, the following table follows:

Transaction i	Transaction j	IsConflicting
Readi (X)	Readj (X)	Non-conflicting
Readi (X)	Writej (X)	Conflicting
Writei (X)	Readj (X)	Conflicting
Writei (X)	Writej (X)	Conflicting

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Before swapping

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

After swapping

2. View Serializable

If a schedule is view equivalent to its serial schedule then the given schedule is said to be View Serializable. Lets take an example.

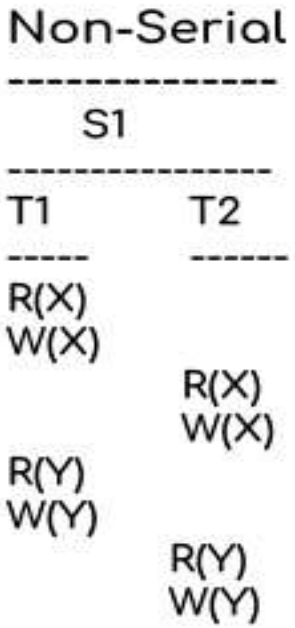
Two schedules T1 and T2 are said to be view equivalent, if they satisfy all the following conditions:

1. **Initial Read:** Initial read of each data item in transactions must match in both schedules.

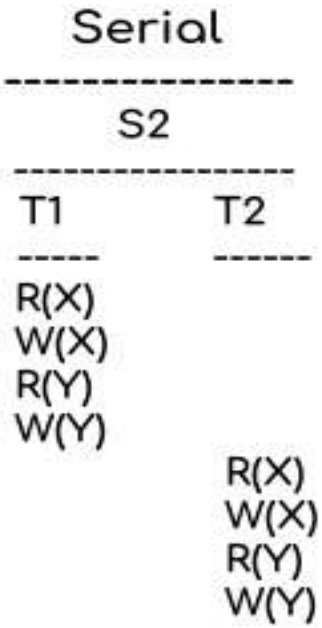
2. **Final Write:** Final write operations on each data item must match in both the schedules.

3. **Update Read:** If in schedule S1, the transaction T1 is reading a data item updated by T2 then in schedule S2, T1 should read the value after the write operation of T2 on same data item.

View Serializable Example



Non-Serial S1	
T1	T2
R(X)	
W(X)	
	R(X)
	W(X)
R(Y)	
W(Y)	
	R(Y)
	W(Y)



Serial S2	
T1	T2
R(X)	
W(X)	
R(Y)	
W(Y)	
	R(X)
	W(X)
	R(Y)
	W(Y)

S2 is the serial schedule of S1. If we can prove that they are view equivalent then we we can say that given schedule S1 is view Serializable

Initial Read

- In schedule S1, transaction T1 first reads the data item X. In S2 also transaction T1 first reads the data item X.
- Lets check for Y. In schedule S1, transaction T1 first reads the data item Y. In S2 also the first read operation on Y is performed by T1.
- We checked for both data items X & Y and the **initial read** condition is satisfied in S1 & S2.

Final Write

- In schedule S1, the final write operation on X is done by transaction T2. In S2 also transaction T2 performs the final write on X.
- Lets check for Y. In schedule S1, the final write operation on Y is done by transaction T2. In schedule S2, final write on Y is done by T2.
- We checked for both data items X & Y and the **final write** condition is satisfied in S1 & S2.

Update Read

- In S1, transaction T2 reads the value of X, written by T1. In S2, the same transaction T2 reads the X after it is written by T1.
- In S1, transaction T2 reads the value of Y, written by T1. In S2, the same transaction T2 reads the value of Y after it is updated by T1.
- The update read condition is also satisfied for both the schedules.

Transaction Isolation and Atomicity

Transaction Isolation and Atomicity or Recoverability

Data recoverability is the process of restoring data that has been lost, accidentally deleted, corrupted or made inaccessible for any reason.

Recoverability Schedules

A recoverable schedule is one where, for each pair of Transaction T_i and T_j such that T_j reads data item previously written by T_i the commit operation of T_i appears before the commit operation T_j.

T8	T9
read(A)	
write(A)	
	read(A)
read(B)	

Cascadeless schedules

T10	T11	T12
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

A single transaction failure leads to a series of transaction rollbacks is called **Cascading rollback**.

- Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work.
- It is desirable to restrict the schedules to those where cascading rollbacks cannot occur, Such schedules are called Cascadeless Schedules.

A cascadeless schedule is one where for each pair of transaction T_i and T_j such that T_j reads data item, previously written by T_i the commit operation of T_i appears before the read operation of T_j.

Every Cascadeless schedule is also recoverable schedule.

Transaction Isolation Levels

We can define transaction isolation by the following phenomena.

Database anomalies

- Database anomalies are generated results that seem incorrect when looked at from the scope of a single transaction, but are correct when looked at from the scope of all transactions. The different types of database anomalies are described as follows:

- **Dirty** reads occur when:
  - Transaction A inserts a row into a table.
  - Transaction B reads the new row.
  - Transaction A rolls back.

  Transaction B may have done work to the system based on the row inserted by transaction A, but that row never became a permanent part of the database.

- **Nonrepeatable** reads occur when:
  - Transaction A reads a row.
  - Transaction B changes the row.
  - Transaction A reads the same row a second time and gets the new results.

- **Phantom** reads occur when:
  - Transaction A reads all rows that satisfy a WHERE clause on an SQL query.
  - Transaction B inserts an additional row that satisfies the WHERE clause.
  - Transaction A re-evaluates the WHERE condition and picks up the additional row.

Transaction Isolation Levels

Read Uncommitted is the lowest level of isolation, allowing access to data before modifications are performed.

Read Committed allows you access to information only after it has been committed to the database.

Repeatable Reads allow transactions to be accessed after they have begun, even if they have not completed. This level enables phantom reads or the awareness of inserted or deleted rows even when changes to existing rows are not readable.

"Serializable," the highest level, denotes that one transaction must be completed before another can start.

Isolation Level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	May occur	May occur	May occur
Read Committed	Don't occur	May occur	May occur
Repeatable Read	Don't occur	Don't occur	May occur
Serializable	Don't occur	Don't occur	Don't occur

Concurrency Control

- **Lock Based Protocols**
- **Deadlock Handling**
- ***Multiple granularities***
- ***Timestamp based Protocols***
- ***Validation based Protocols***

Concurrency Control

It is process of managing simultaneous / parallel execution of transactions in a shared database.

Ex:

Purpose of concurrency control

- To enforce isolation
- To preserve database consistency
- To resolve conflicts (RW, WR, WW)

Lock - Based Protocols

- It is a mechanism in which a transaction cannot read or write data unless the appropriate lock is acquired.

- This helps in eliminating the concurrency problem by locking a particular transaction to a particular user.

- The lock is a variable that denotes those operations that can be executed on the particular data item.

There are mainly **two** modes in which a data item may be locked.

1. Shared. It is also called as **read lock** If a transaction Ti has obtained a shared-mode lock (denoted by S) on item Q, then Ti can **read, but cannot write, Q.**

2. Exclusive. This lock is also called an **update or write lock**. If a transaction Ti has obtained an exclusive-mode lock (denoted by X) on item Q, then Ti can **both read and write Q.**

T_1: lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

T_2: lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display($A + B$).

Granting of Locks

When a transaction *Ti* requests a lock on a data item *Q* in a particular mode *M*, the concurrency-control manager grants the lock provided that

1. There is no other transaction holding a lock on *Q* in a mode that conflicts with *M*.

2. There is no other transaction that is waiting for a lock on *Q*, and that made its lock request before *Ti*.

The Two-Phase Locking Protocol

The two-phase locking protocol ensures Serializability.

This protocol requires that each transaction issue lock and unlock requests in **two** phases:

1. **Growing phase**. Locks are obtained but not released.
2. **Shrinking phase.** Locks are released but no new locks are acquired .

A modification of two-phase locking is called the **Strict Two-Phase Locking Protocols.**

The most widely used locking protocol is the Strict Two-Phase Locking Protocol, also called as **Strict 2PL**, has following two rules.

Rule 1 : If a transaction T wants to read an object, it first requests a Shared lock. But, if a transaction T wants to modify an object, it first requests an exclusive lock on the object.

Rule 2 : All locks held by a transaction are released when the transaction is completed.

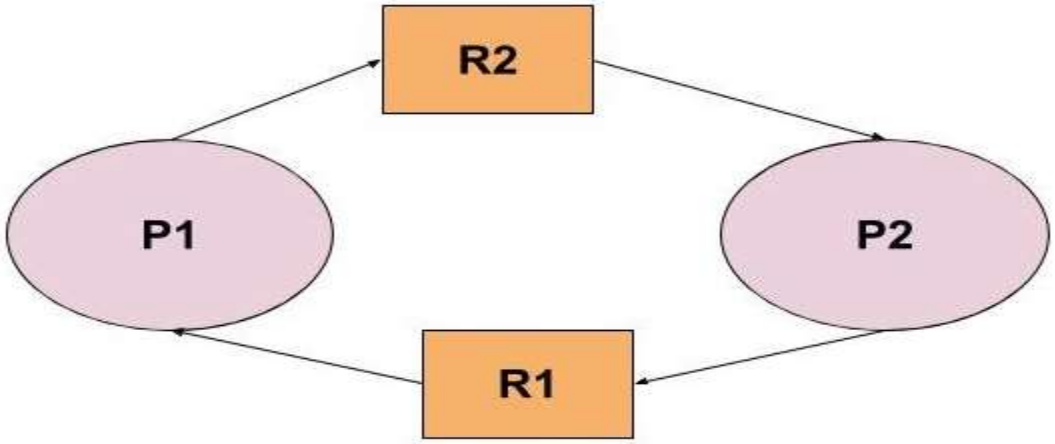
The locking protocol can have **two** cases as follows :

1. If two transactions access completely different parts of the database, then they proceed without interruption on their ways.
2. If two transactions access same object of the database, then their actions are ordered serially i.e., all actions of locked transactions are completed first, then this lock is releases and the other transaction can now proceed.

Deadlock Handling

Deadlock

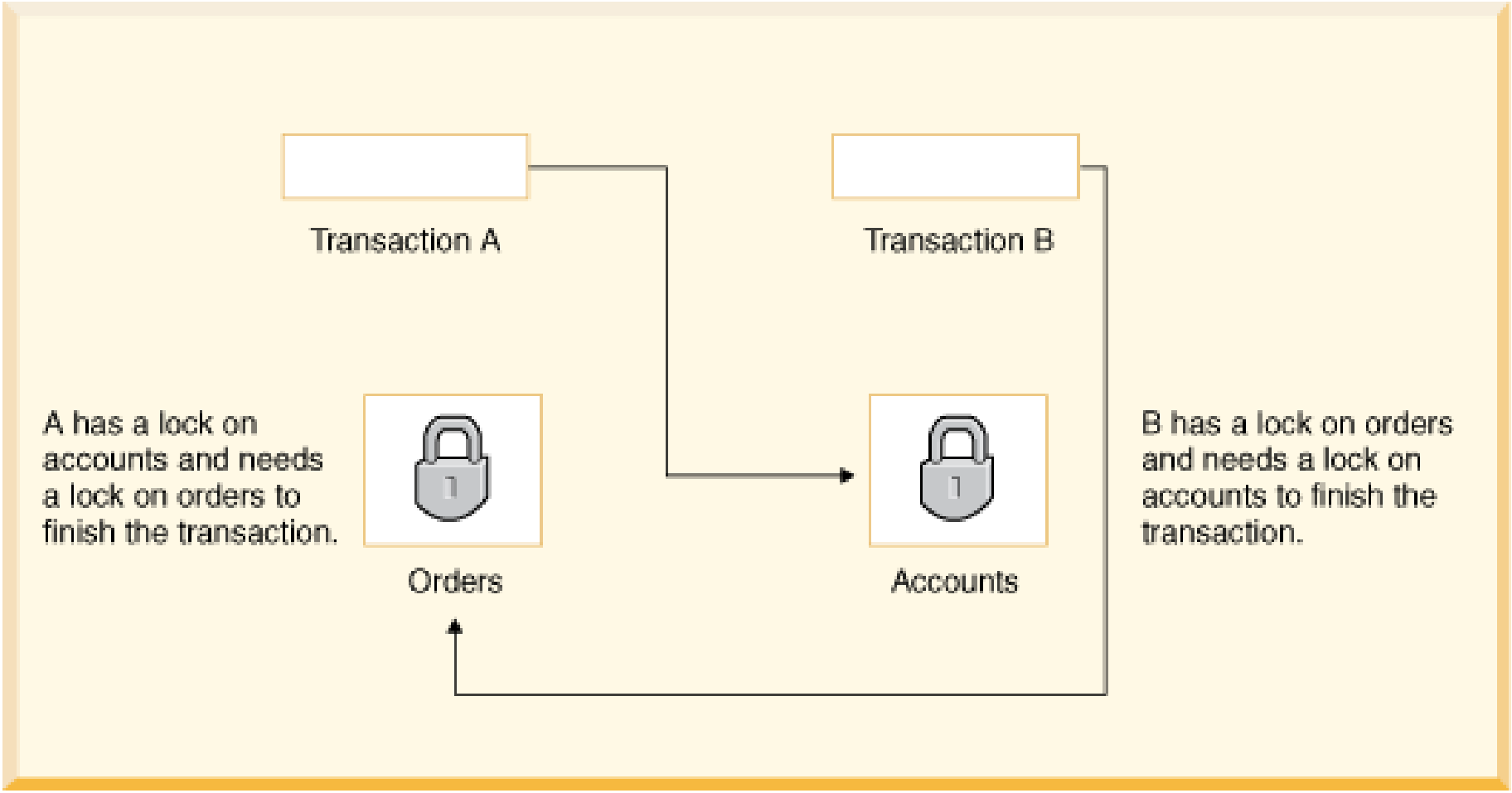
- Deadlock is a situation where two or more transactions are waiting indefinitely for each other to g



Process P1 is holding R1 and waiting for R2 while Process P2 is holding R2 and waiting for R1.

Example: Consider a situation where there are two processes P1 and P2 executing simultaneously. The process P1 is holding the resource R1 and needs R2 to complete its execution. The process P2 is holding the resource R2 and needs R1 to complete its execution. So, here one process is waiting for the other process to release its resource and hence both the process is unable to complete its execution and the system goes into a halt.

A deadlock where two transactions are waiting for one another to give up locks.



Deadlock Handling Methods

- Deadlock Avoidance
- Deadlock Detection
- Deadlock Prevention

Deadlock avoidance

- Whenever a database is *stuck in a deadlock* it's *better to abort the transactions* that are *resulting deadlock* but this is a waste of resources and time.

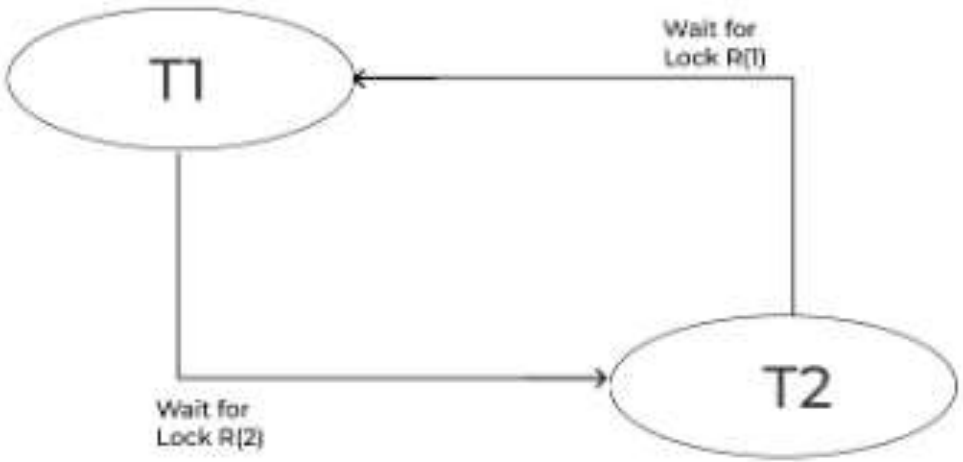
- Deadlock avoidance mechanism used *to detect any deadlock situation in advance* by using some techniques like WEIGHTS FOR GRAPH but this wait for graph method is suitable only for smaller databases, for large databases other prevention techniques are used.

Deadlock detection (Wait for Graph)

- Wait for graph method is *suitable only for smaller databases*

Steps :

- In this method, *a graph is created based on transactions and their locks.*
- While Creating the graph *if a cycle or closed loop is encoun* k.



Deadlock prevention

- Prevention techniques are *used for larger databases* .
- Resource *allocation is done in such a way that deadlock never occurs.*
- DBMS ***analyse*** *the transactions to determine whether any deadlock situation can arise or not*, if there is any possibility then DBMS never allows such transaction for execution

.

Timestamp-Based Protocols

Timestamp-Based Protocols

Timestamps

Each transaction Ti in the system associated with a unique fixed timestamp, denoted by **TS(Ti)**.

If a transaction Ti has been assigned timestamp TS(Ti), and a new transaction Tj enters the system, then TS(Ti) < TS(Tj).

If TS(Ti) < TS(Tj), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction Ti appears before transaction Tj.

To implement this scheme, we associate with each data item Q **two** timestamp values:

- **W-timestamp(Q)** denotes the largest timestamp of any transaction that executed write(Q) successfully.

- **R-timestamp(Q)** denotes the largest timestamp of any transaction that executed read(Q) successfully.

Validation-Based Protocols

Validation-Based Protocols

Validation phase is also known as optimistic concurrency control technique. In the validation based protocol, the transaction is executed in the following three phases:

Read phase: In this phase, the transaction T is read and executed. It is used to read the value of various data items and stores them in temporary local variables. It can perform all the write operations on temporary variables without an update to the actual database.

Validation phase: In this phase, the temporary variable value will be validated against the actual data to see if it violates the Serializability.

Write phase: If the validation of the transaction is validated, then the temporary results are written to the database or system otherwise the transaction is rolled back.

Validation (T_i): It contains the time when Ti finishes its read phase and starts its validation phase.

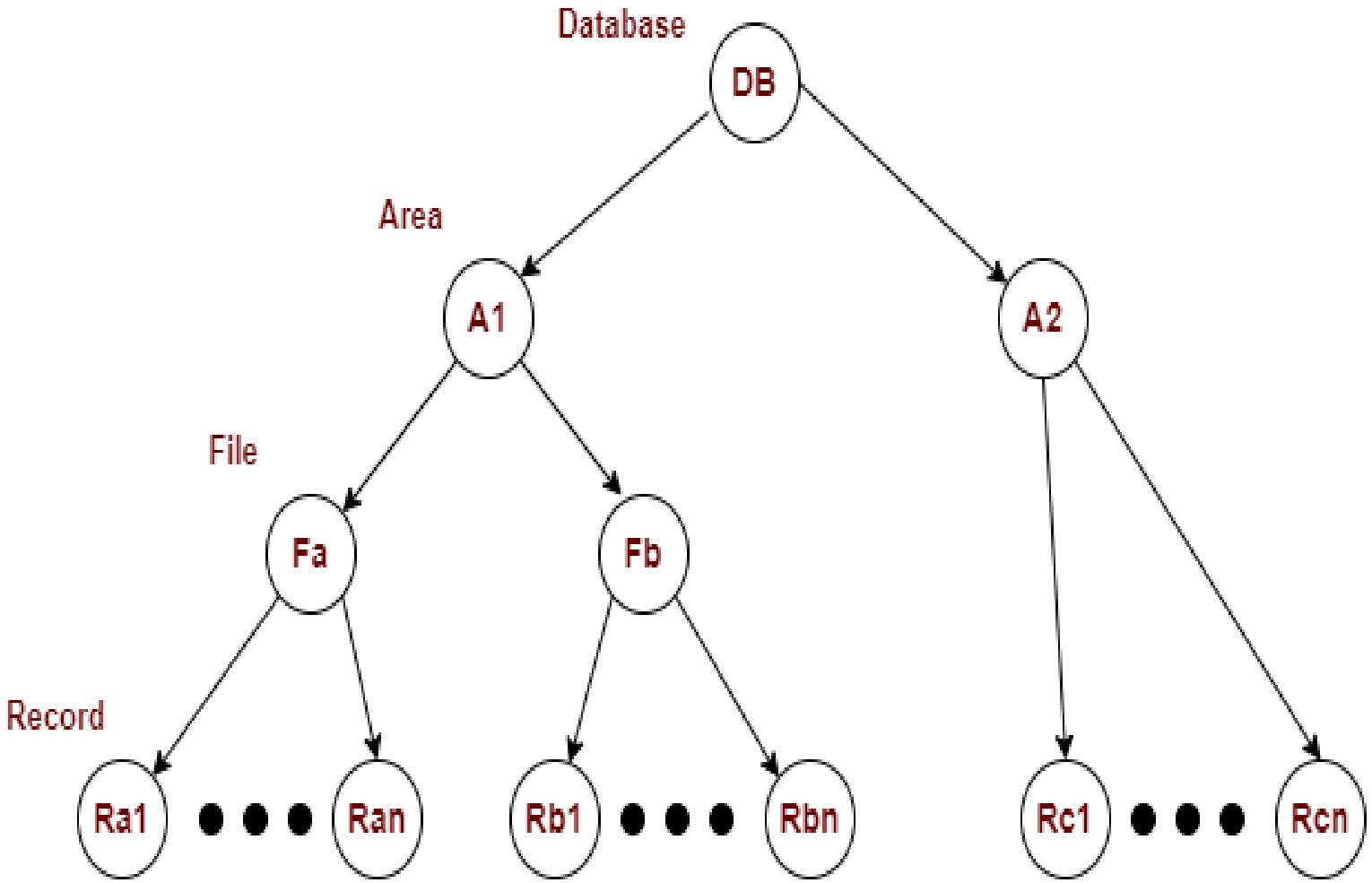
Finish(Ti): It contains the time when Ti finishes its write phase.

This protocol is used to determine the time stamp for the transaction for serialization using the time stamp of the validation phase, as it is the actual phase that determines if the transaction will commit or rollback.

- Hence TS(T) = validation(T).
- The serializability is determined during the validation process. It can't be decided in advance.
- While executing the transaction, it ensures a

Multiple Granularity

Multiple Granularity



Multi Granularity tree Hierarchy

The **multiple-granularity locking protocol**, which ensures serializability, in this:

Each transaction *Ti* can lock a node *Q* by following these rules:

1. It must observe the lock-compatibility function.

2. It must lock the root of the tree first, and can lock it in any mode.

3. It can lock a node *Q* in S or IS mode only if it currently has the parent of *Q* locked in either IX or IS mode.

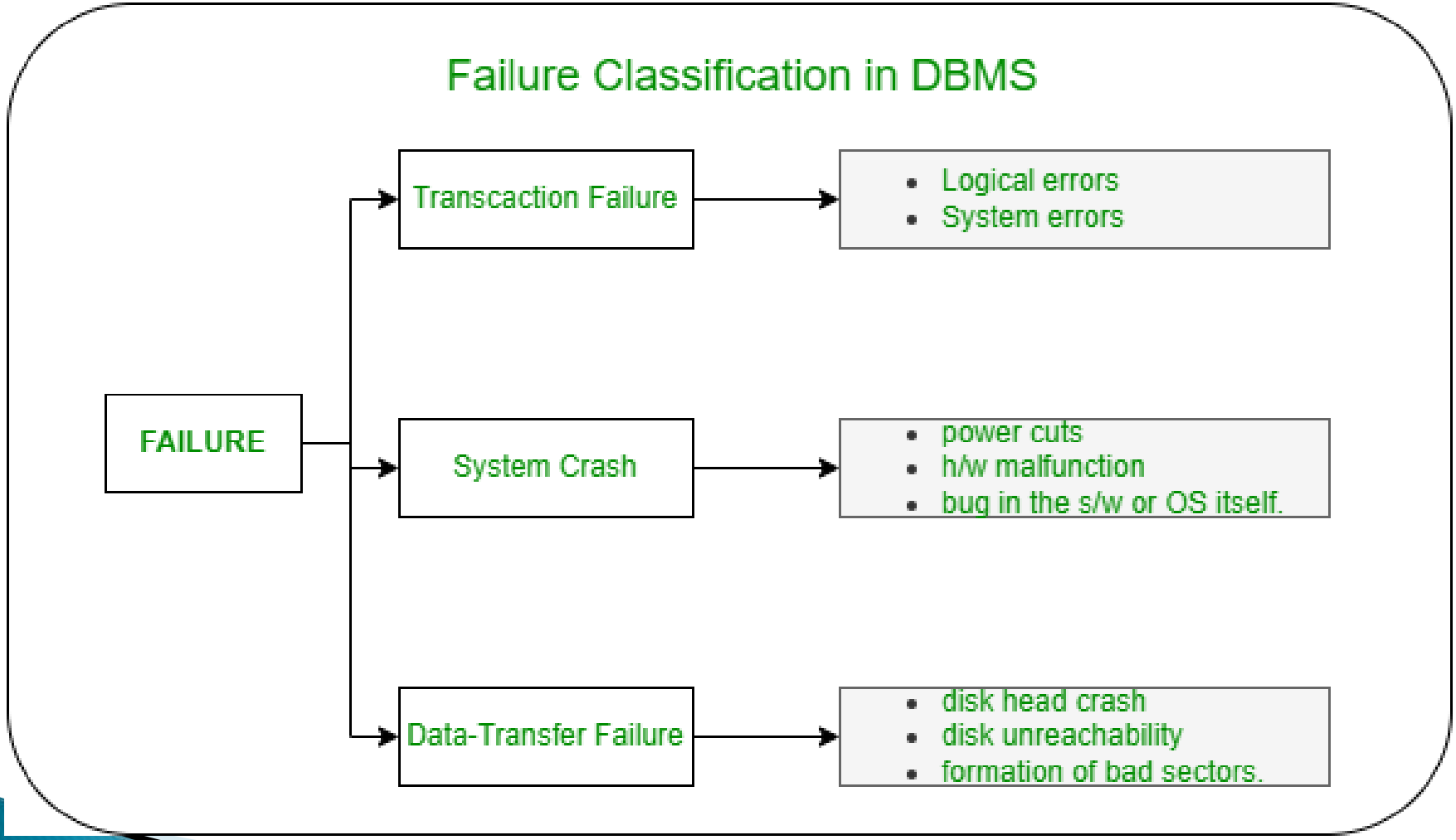
4. It can lock a node *Q* in X, **SIX, or IX** mode only if it currently has the parent of *Q* locked in either IX or SIX mode.

Observe that the multiple-granularity protocol requires that locks be acquired in ***top- down*** **(root-to-leaf)** order, whereas locks must be released in ***bottom-up*** **(leaf-to-root)** order

Recovery System

Failure Classification

A failure in DBMS can be classified as:



Failure Classification : In DBMS there are several transactions running in a specified schedule. However sometimes these transactions fail due to several reasons.

1. Transaction Failure

A transaction is a set of statements, if a transaction fails it means there is a statement in the transaction which is not able to execute. This can happen due to various reasons such as:

- **Logical Error:** If the logic used in the statement itself is wrong, it can be fail.
- **System Error:** When the transaction is executing but due to a fault in system, the transaction fails abruptly. For example: Deadlock condition in transaction can result in System error.

2. System Crash

System can crash due to various reasons such as:

- Power supply disruptions
- Software issues such as Operating system issues
- Hardware issues

The non-volatile memory does not get affected in a system crash (Fail-stop assumption).

3. Data Transfer failure

Disk failure occurs due to frequent failure of hard disks and storage drives.

There can be several reasons of a disk failure such as: formation of bad sectors in disk, corruption of disk, viruses, not enough resources available on disk.

Recovery and Atomicity

Recovery and Atomicity

Log Records:

- Log is nothing but a file which contains a sequence of records, each log record refers to a write operation.
- All the log records are recorded step by step in the log file. We can say, log files store the history of all updates activities.
- Log contains start of transaction, transaction number, record number, old value, new value, end of transaction etc.
- When the system crashes, then by using log files, we can return back to the previous state.
- The log is kept on disk so that it is not affected by failures except disk and failures

There are four types of log records:

- When the transaction is initiated, then it writes 'start' log.

  <Tn, Start>

- When the transaction is updated

  <Ti, Xi, V1, V2> − update log record, where Ti=transaction, Xi=data, V1=old data, V2=new value.

- When the transaction is finished, then it writes another log to indicate the end of the transaction.

  <Tn, Commit>

- When a Transaction has aborted.

  < Tn abort >

Example-1: Consider the following transactions :

Let **T1** is a transaction that transfers 50 from A to B.

Let **T2** is a transaction that withdraws 100 from C.

Transaction's	Log File

T1 : read(A);
A = A -50;
write(A)
read(B);
B = B + 50;
write(B);
 	
T2 : read(C);
C = C – 100;
write(C);
	LOG File : < T1 start >
	<T1, A, 1000, 950>
	<T1, B, 2000,2050>
	< T1 commit>
	<T2 start>
	<T2, C, 700,600>
	< T2 commit >

Buffer Management

Log-record buffering :

Here, it is required to know how the buffer management has to function which is essential to the implementation of a crash-recovery scheme that ensures data consistency and imposes a minimal amount of overhead on interactions.

a) Transaction **Ti** goes into the commit state after the <**Ti commit**> log record has been output to the stable storage.

b) Before the <**Ti commit**> log record can be output to stable storage, all log records concerning transaction **Ti** must have been output to the stable storage.

c) Before a block of data in the main memory can be output to the database, all log records concerned with the data in that block must have been output to the stable storage.

This rule is said to be the 'write-ahead logging' (WAL) rule.

Database buffering :

The system stores the database in non-volatile storage and brings the blocks of data into the main memory as required. In this process, if any block is modified in the main memory, it should be stored in disk and then that block can be used for other blocks to be overwritten.

One might expect that transactions would force-output all modified blocks to disk when they commit. Such a policy is called **'force policy'**. When a data block B1 wants to be output to the disk, all log records concerned with the data in B1 must be output to stable storage.