Aktu | Database Management System | BCS-501

Unit-4

Syllabus

Transaction Processing Concept: Transaction System, Testing of Serializability, Serializability of Schedules, Conflict & View Serializable Schedule, Recoverability, Recovery from Transaction Failures, Log Based Recovery, Checkpoints, Deadlock Handling. Distributed Database: Distributed Data Storage, Concurrency Control, Directory System.

Content Unit-4

- Transaction System
- ACID properties - (2021-22)
- Operations During a Transaction
- State diagram - (2022-23)
- Schedule & Its Types - (2022-23)
- Serializability
- Testing of Serializability
- Conflict & View Serializable
- Recoverability
- Recoverable , Cascadeless & Strict
- Recovery System
- Types of Failures
- Log-Based Recovery - 2023-24
- Deferred Database Modification 2023-24
- Immediate Database Modification
- Checkpoints
- Deadlock
- Characteristics of Deadlock / Necessary Conditions
- Deadlock Handling -2023-24 & 2021-22
- Distributed Database & its Components

Transaction System

A transaction is a series of database operations performed as a single logical unit of work, ensuring the database stays consistent even if something goes wrong. It guarantees all-or-nothing execution, which means either all the steps of the transaction are completed, or none are.

- **Example:** Imagine you are transferring ₹500 from Account A to Account B:

1. Deduct ₹500 from Account A.
2. Add ₹500 to Account B.

If either step fails (e.g., insufficient balance in Account A), the transaction should rollback to avoid inconsistent states (like ₹500 disappearing).

AKTU- 2021-22

List ACID properties of a transaction. Explain the usefulness of each. What is the importance of log?

The ACID properties ensure reliability and consistency for all database transactions.

1. Atomicity:

- **Meaning:** A transaction is treated as a single unit. It either completes entirely or fails entirely.
- **Example:** In a train booking, deducting a seat and assigning it to a passenger must both succeed; otherwise, neither action is taken.
- **Why Useful?:** Prevents incomplete changes that leave the database in an inconsistent state.

2. Consistency:

- **Meaning:** A transaction must take the database from one valid state to another, maintaining all integrity rules.
- **Example:** If a student's marks are updated, total marks must still reflect correctly.
- **Why Useful?:** Ensures database rules (like constraints or relationships) are always followed.

3. Isolation:

- **Meaning:** Multiple transactions can occur simultaneously, but they should not interfere with each other.
- **Example:** Two people trying to book the last flight seat will not both succeed; one will complete first.
- **Why Useful?:** Prevents data corruption or incorrect results in a concurrent environment.

4. Durability:

- **Meaning:** Once a transaction is committed, the changes are permanent, even if the system crashes.
- **Example:** After transferring money, the balance update remains saved despite power failure.
- **Why Useful?:** Provides reliability and trust in the system.

Importance of Log:

A log is a record of all operations performed by transactions. It is critical for failure recovery.

How it works:

Logs contain:

- **UNDO information:** Used to reverse incomplete transactions.
- **REDO information:** Used to reapply committed transactions.

Why important?

1. **If a transaction fails midway, the log ensures the database can roll back changes to its previous state.**
2. **After a system crash, logs are used to restore the database to a consistent state.**

Operations During a Transaction

1. Read (R)

- The transaction reads data from the database into memory.
- Example: Read(A) copies the value of A from the database into the transaction's local memory.

2. Write (W)

- The transaction writes or updates the value in the database.
- Example: Write(A) updates the value of A in the database after making changes in memory.

3. Update

- The transaction modifies the value of a variable in memory.
- Example: A = A - 100 reduces the value of A in the transaction's memory.

4. Commit

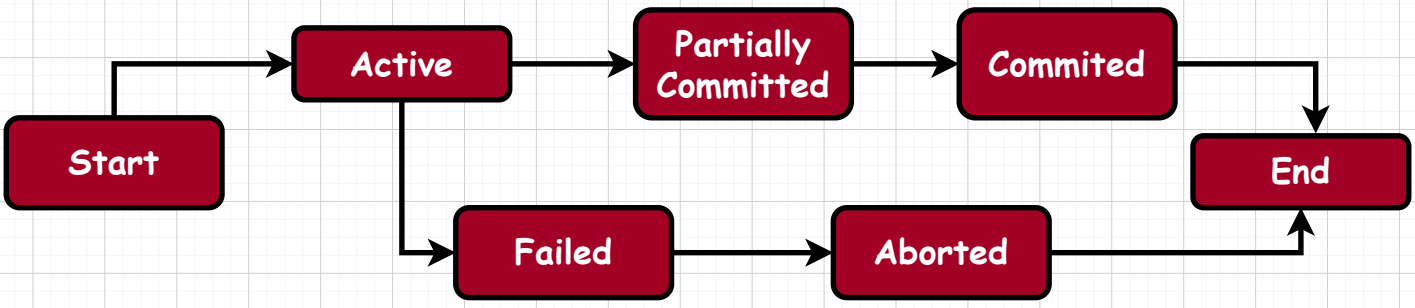
- The transaction's changes are saved permanently to the database.
- After the COMMIT operation, the changes made by the transaction become visible to other transactions.
- Example: COMMIT ensures all updates to A and B are finalized in the database.

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

AKTU- 2022-23

Q. Draw a state diagram and discuss the typical states that a transaction goes through during execution

A transaction moves through various states during its execution. Here's a step-by-step explanation with a diagram:



Explanation of States:

1. Active State:

- The transaction is actively executing its operations (e.g., reading or writing to the

database).
- **Example:** Deducting ₹500 from Account A.

2. Partially Committed State:

- The transaction has completed its final operation but hasn't yet saved the changes permanently.
- **Example:** Both ₹500 deduction and addition operations are completed but not yet saved to the database.

3. Committed State:

- The transaction is successfully completed, and its changes are permanently saved.
- **Example:** Account A and Account B balances are updated and stored in the database.

4. Failed State:

- Errors or issues (like system failure, invalid input, or insufficient funds) prevent the transaction from completing.
- **Example:** ₹500 deduction fails due to insufficient balance in Account A.

5. Aborted State:

- The database rolls back the transaction, undoing any changes made during execution.
- **Example:** If the money transfer fails, Account A and B are restored to their original balances.

AKTU- 2022-23

Q.When is a Transaction Rolled Back?

A rollback occurs when a transaction fails, undoing all changes made so far to maintain the database's consistency.

Scenarios for Rollback:

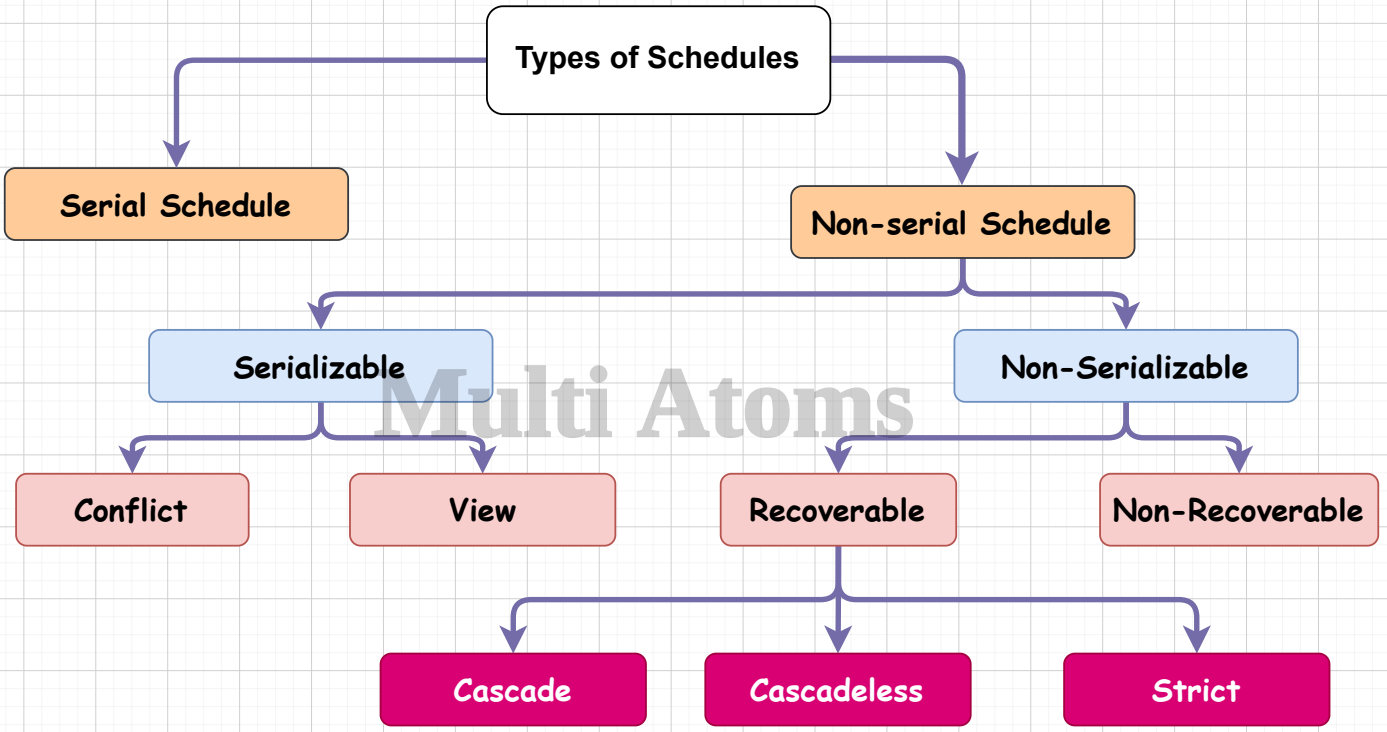
1. **System Failure:** Power outages or software crashes interrupt the transaction midway.
2. **Database Constraint Violation:** The transaction tries to break a rule (e.g., inserting a duplicate primary key).
3. **Deadlock:** Two transactions are stuck waiting for resources held by each other, blocking progress.
4. **Explicit Cancellation:** The user or system aborts the transaction manually.

AKTU- 2022-23

Schedule

A Schedule is the order in which the operations (like Read, Write, Commit, etc.) of multiple transactions are executed.

- A schedule is valid if it preserves the sequence of operations within each transaction.
- Schedules determine how concurrent transactions affect the database.



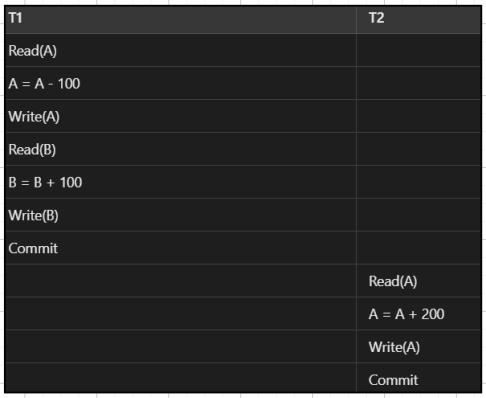
Types of Schedules

1. Serial Schedule

- Transactions are executed one after another with no interleaving.
- Ensures database consistency.
- **Example:** If two transactions T1 and T2 exist:
- Execute all operations of T1 first, then T2.
- Or execute all operations of T2 first, then T1.

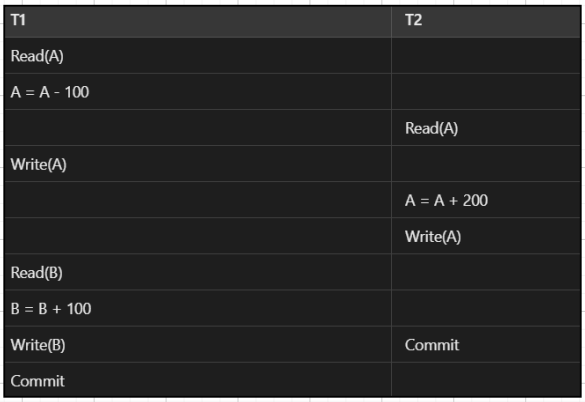
2. Non-Serial Schedule

- Allows interleaving of operations from different transactions.
- Provides better concurrency.
- Example: Operations from T1 and T2 are mixed together.



T1	T2
Read(A)	
A = A - 100	
Write(A)	
Read(B)	
B = B + 100	
Write(B)	
Commit	
	Read(A)
	A = A + 200
	Write(A)
	Commit

Serial



T1	T2
Read(A)	
A = A - 100	
	Read(A)
Write(A)	
	A = A + 200
	Write(A)
Read(B)	
B = B + 100	
Write(B)	Commit
Commit	

Non- Serial

Serializability in Non-Serial Schedules

Non-Serial schedules can be:

1. Serializable

- Behaves as if the transactions were executed in a serial order.
- **Includes two types:**

1. **Conflict Serializability:** Based on conflicts (Read-Write, Write-Write).
2. **View Serializability:** Based on the final result.

2. Non-Serializable

- **Does not guarantee consistency.**

Recoverability in Non-Serializable Schedules

Recoverability ensures no data inconsistencies occur when transactions fail. Types include:

1. Recoverable Schedule

- Ensures that if one transaction depends on another, the dependent transaction commits only after the first transaction commits.

2. Cascading Schedule

2. Cascading Schedule

- Failure in one transaction causes rollback of multiple transactions.

3. Cascadeless Schedule

- Prevents cascading rollbacks by ensuring a transaction reads only committed data.

4. Strict Schedule

- Prevents both cascading rollbacks and dirty reads.

Concurrency Problems

Concurrency problems occur when multiple transactions execute simultaneously and interact with the same data, leading to unexpected or incorrect outcomes. The most common concurrency problems are:

1. Dirty Read

- **Definition:** A transaction reads uncommitted changes made by another transaction.
- **Impact:** Inconsistent and unreliable data.
- **Prevention:** Use the Read Committed isolation level or higher.

T1 (Uncommitted Write)	T2 (Reads Dirty Data)
`Read(A=100)`	
`A = A - 50 (A=50)`	
`Write(A=50)` (Uncommitted)	`Read(A=50)` (Dirty Read)
`ROLLBACK` (Reverts A to 100)	Uses incorrect `A=50`

3. Phantom Read

- **Definition:** A transaction re-executes a query and gets a different result set because another transaction has added or removed rows that match the query criteria.
- **Impact:** The result set changes unexpectedly.
- **Prevention:** Use the Serializable isolation level.

T1 (Reads Rows)	T2 (Inserts Rows)
`SELECT * WHERE salary > 5000`	
(Returns 2 rows: Emp1, Emp2)	
	`INSERT Emp3 (salary=6000)`
	`COMMIT`
`SELECT * WHERE salary > 5000`	(Now returns 3 rows: Emp1, Emp2, Emp3)

4. Lost Update

- **Definition:** Two transactions read the same data and update it concurrently, but one update overwrites the other, leading to a loss of data.
- **Impact:** One update is lost, leading to incorrect results.
- **Prevention:** Use locks or the Serializable isolation level.

T1 (Updates A)	T2 (Updates A)
`Read(A=100)`	`Read(A=100)`
`A = A + 20 (A=120)`	
	`A = A - 30 (A=70)`
`Write(A=120)`	
	`Write(A=70)` (Overwrites A=120)

5. Deadlock

- **Definition:** Two or more transactions wait indefinitely for resources locked by each other, creating a circular wait.
- **Impact:** Transactions are unable to proceed, leading to a system stall.
- **Prevention:** Deadlock detection and recovery mechanisms (e.g., timeout, resource ordering).

T1 (Locks Resource A)	T2 (Locks Resource B)
`Lock(A)`	
	`Lock(B)`
`Waits for Lock(B)`	`Waits for Lock(A)`

Types of Read-Write Conflicts

1. Write-Read Conflict (Uncommitted Read)

- Occurs when one transaction writes to a data item, and another transaction reads it before the first transaction commits.
- Problem: The second transaction may read incorrect or uncommitted data.

T1: Write(X)
T2: Read(X) // Before T1 commits

2. Read-Write Conflict (Dirty Write)

- Happens when a transaction reads a data item, and another transaction writes to it before the first transaction completes.
- **Problem:** The written data might overwrite uncommitted changes, causing inconsistency.

T1: Read(X)
T2: Write(X) // Before T1 commits

T2: Write(X) // Before T1 commits

3. Write-Write Conflict (Overwriting)

- Occurs when two transactions write to the same data item without considering the order of execution.
- **Problem:** One write operation overwrites the other's changes, leading to loss of data.

T1: Write(X)
T2: Write(X) // Without checking T1's commit

What is Serializability?

- A schedule is serializable if its result is the same as a serial schedule (i.e., transactions execute without overlapping).
- Serializable schedules ensure database consistency is maintained even when transactions are executed concurrently.

Importance of Serializability

1. Ensures database consistency.
2. Avoids unexpected behaviors or conflicts between transactions.
3. Is a key property for concurrency control mechanisms in DBMS.

Types of Serializability

AKTU- 2023-24, 2021-22

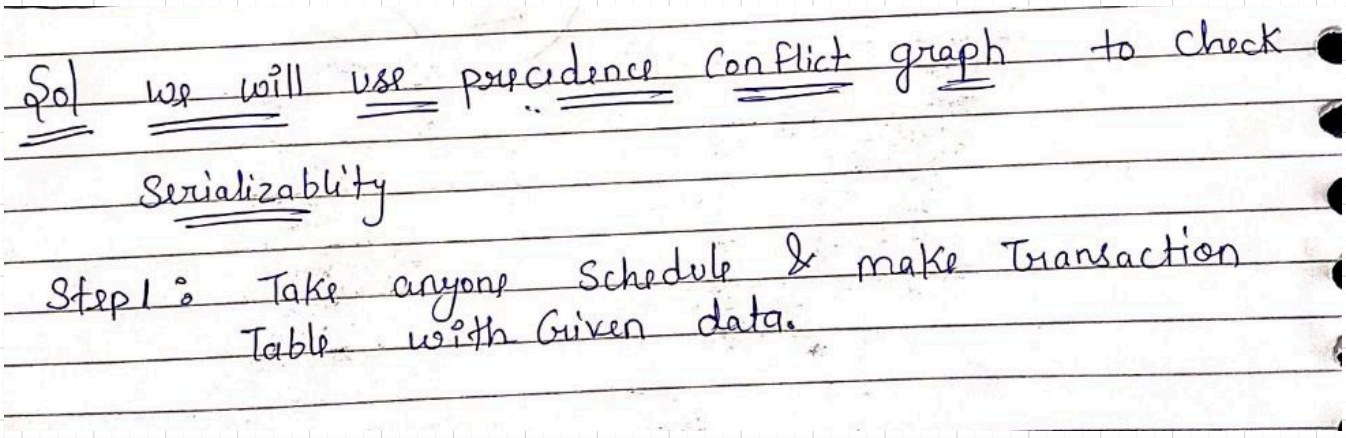
1. Conflict Serializability

- A schedule is conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations (read/write on different data items).
- Conflict serializability is determined using a precedence graph
- A schedule is conflict-serializable if its precedence graph (serialization graph) has no cycles.

- **Steps to Create a Precedence Graph:**
  1. Create a node for each transaction in the schedule.
     T_i
     $\rightarrow T$ T T
  2. Draw a directed edge $_j$ if a conflicting operation in $_i$ precedes $_j$.
     (Conflicts: Read-Write, Write-Read, Write-Write on the same data item.)
  3. If the graph contains a **cycle**, the schedule is **not conflict serializable**. Otherwise, it is conflict serializable.

AKTU- 2022-23

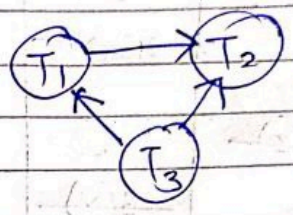
Consider the three transactions T1, T2, and T3, and the schedules S1 and S2given below. State whether each schedule is serializable or not. If a schedule is serializable, write down the equivalent serial schedule(s).
T1: r1 (X); r1 (Z); w1 (X);
T2: r2 (Z); r2 (Y); w2 (Z); w2 (Y);
T3: r3 (X); r3 (Y); w3 (Y);
S1: r1 (X); r2 (Z); r1 (Z); r3 (X); r3 (Y); w1 (X); w3 (Y); r2 (Y); w2 (Z); w2 (Y);
S2: r1 (X); r2 (Z); r3 (X); r1 (Z); r2 (Y); r3 (Y); w1 (X); w2 (Z); w3 (Y); w2 (Y);



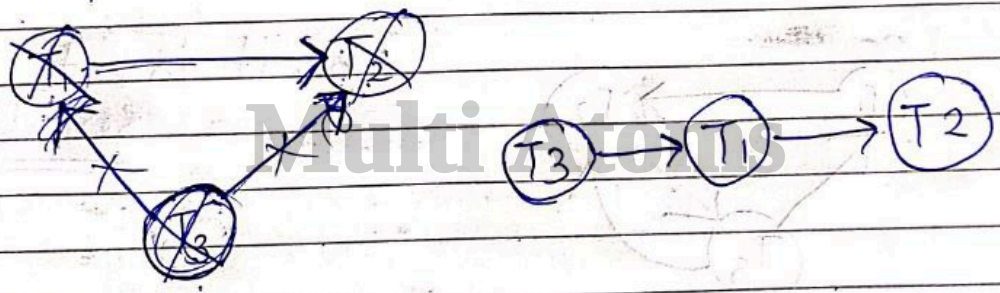
① S1 Schedule.

T1	T2	T3
r1(x)		
	r2(z)	
r1(z)		
		r3(x)
		r3(y)
w1(x)		
		w3(y)
	r2(x)	
	w2(z)	
	w2(y)	

check.
read - write
write - read
write - write.



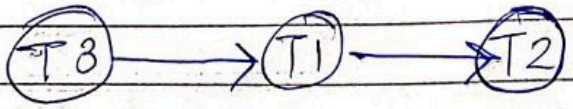
yes → it is Serializable



→ check cycle if there is cycle this is not Serializable

→ If not. ① remove zero degree edges. and repeat.

② Track the node & make Serializable Schedule.

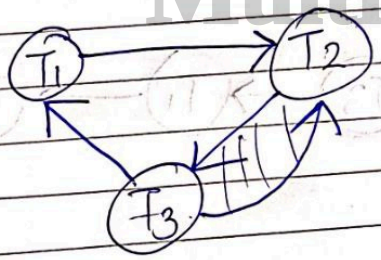


(11)

S2

T1	T2	T3
r1(x)		
	r2(z)	
		r3(x)
r1(z)		
	r2(y)	
		r3(y)
w1(x)		
	w2(z)	
		w3(y)
	w2(y)	

check.
r - w
w - r
w - w.



No, it is not serializable.

View Serializability

View serializability ensures that a non-serial schedule produces the same results as a serial schedule by maintaining the consistency of database operations.

A schedule is view serializable if it is view equivalent to a serial schedule.

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

Steps to Check View Serializability

1. **Check Initial Read:** Compare the initial reads for each data item in the non-serial and serial schedules.
2. **Check Updated Read:** Verify that each transaction reads data updated by the same transaction in both schedules.
3. **Check Final Write**: Ensure the final writes for each data item are performed by the same transaction.

If all three conditions are satisfied, the schedule is view equivalent and hence view serializable.

S1

T1	T2
$R(A)$	
$A = A + 10$	
$W(A)$	
$R(B)$	
$B = B + 20$	
$W(B)$	
	$R(A)$
	$A = A + 10$
	$W(A)$
	$R(B)$
	$B = B \times 1.1$
	$W(B)$

S2

T1	T2
$R(A)$	
$A = A + 10$	
$W(A)$	
	$R(A)$
	$A = A + 10$
	$W(A)$
$R(B)$	
$B = B + 20$	
$W(B)$	
	$R(B)$
	$B = B \times 1.1$
	$W(B)$

1. Initial Read

1. Initial Read

The first transaction that reads a data item in S_1 and S_2 should be the same.

- For A:
  - In S_1, T_1 performs $R(A)$ first.
  - In S_2, T_1 performs $R(A)$ first.
    ✅ **Condition is satisfied for A.**
- For B:
  - In S_1, T_1 performs $R(B)$ first.
  - In S_2, T_1 performs $R(B)$ first.
    ✅ **Condition is satisfied for B.**
- In $2, 1$ performs $R(B)$ first.
  ✅ **Condition is satisfied for B.**

✅ Condition is satisfied for B.

2. Updated Read

If a transaction T_i reads a value X updated by another transaction T_j, then the same order of updates must be preserved in both schedules.

- For A:
  - In S_1, T_2 reads A updated by T_1 (via $W(A)$).
  - In S_2, T_2 reads A updated by T_1 (via $W(A)$).
    ✅ **Condition is satisfied for A.**
- For B:
  - In S_1, T_2 reads B updated by T_1 (via $W(B)$).
  - In S_2, T_2 reads B updated by T_1 (via $W(B)$).
    ✅ **Condition is satisfied for B.**

3. Final Write

The transaction that performs the last write on a data item in S_1 and S_2 should be the same.

- For A:
  - In S_1, the final write on A is performed by T_2.
  - In S_2, the final write on A is performed by T_2.
    ✅ **Condition is satisfied for A.**
- For B:
  - In S_1, the final write on B is performed by T_2.
  - In S_2, the final write on B is performed by T_2.
    ✅ **Condition is satisfied for B.**

1. **Conflict Serializable = View Serializable:**
   - If a schedule is conflict serializable, it is always view serializable.
2. **View Serializable ≠ Conflict Serializable:**
   - A view serializable schedule may not be conflict serializable if it involves blind writes (a write operation without a preceding read).

Aspect	Conflict Serializability	View Serializability
Definition	A schedule is conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations.	A schedule is view serializable if it is view equivalent to a serial schedule.
Focus	Focuses on **conflicting operations** (read/write, write/write) and their order.	Focuses on the **logical outcome** of the transactions (initial read, updated read, and final write).
Condition	Checks for **conflict equivalence** between schedules.	Checks for **view equivalence** between schedules.
Practicality	Easier to check and implement as it only involves swapping operations and checking conflicts.	More general and complex as it involves checking all three conditions: initial read, updated read, and final write.
Includes Blind Writes	Does **not include blind writes** (write operations with no preceding read).	Includes **blind writes** in its validation process.
Complexity	Relatively simpler as it involves identifying and reordering conflicts.	More complex as it evaluates logical equivalence beyond conflicts.
Relation	All conflict-serializable schedules are view serializable.	Not all view-serializable schedules are conflict serializable.
Example of Exclusion	Schedules with **blind writes** are not conflict serializable but may be view serializable.	Schedules that are conflict serializable are also view serializable.

Recoverability

AKTU- 2022-23

Recoverability ensures that the database remains consistent even if a transaction fails or rolls back. It means one transaction should commit only if all the other transactions it depends on have committed successfully.

Non-Recoverable Schedule (Bad)

T1	T2	Explanation
Write(A)		T1 writes data to A.
	Read(A)	T2 reads the data written by T1.
	Commit	T2 commits its transaction (dependent on T1's data).
Rollback		T1 fails and rolls back, making the data used by T2 invalid.

1. **Problem**: T2 committed before T1 committed, and then T1 failed.
2. **Result**: Database is in an inconsistent state because T2 used invalid data.

Recoverable Schedule (Good)

T1	T2	Explanation
Write(A)		T1 writes data to A.
	Read(A)	T2 reads the data written by T1.
Commit		T1 commits its transaction.
	Commit	T2 commits after ensuring T1's data is valid.

1. **Why Good?:** T2 commits after T1 commits, ensuring consistency.

Types of Schedules in Terms of Recoverability

1. Recoverable Schedule:

- A schedule is recoverable if a transaction commits only after the transactions it depends on have committed.
- **Example:** If T2 reads a value written by T1, T2 should not commit until T1 has committed.
- Why important? Prevents inconsistency caused by committing a transaction that depends on another uncommitted transaction.

2. Cascadeless Schedule:

- A stricter type of schedule where a transaction is not allowed to read uncommitted data from another transaction.
- **Example**: T2 cannot read A until T1 has committed its changes to A.
- Why important? Prevents cascading rollbacks where one failure causes many transactions to fail.

T1	T2	Explanation
Write(A)		T1 writes data to A.
Commit		T1 commits **before** any other transaction reads its data.
	Read(A)	T2 reads data only after T1 commits.
	Commit	T2 commits.

1. **Why Better?:** T2 waits for T1 to commit before reading its data, avoiding cascading rollbacks.

3. Strict Schedule:

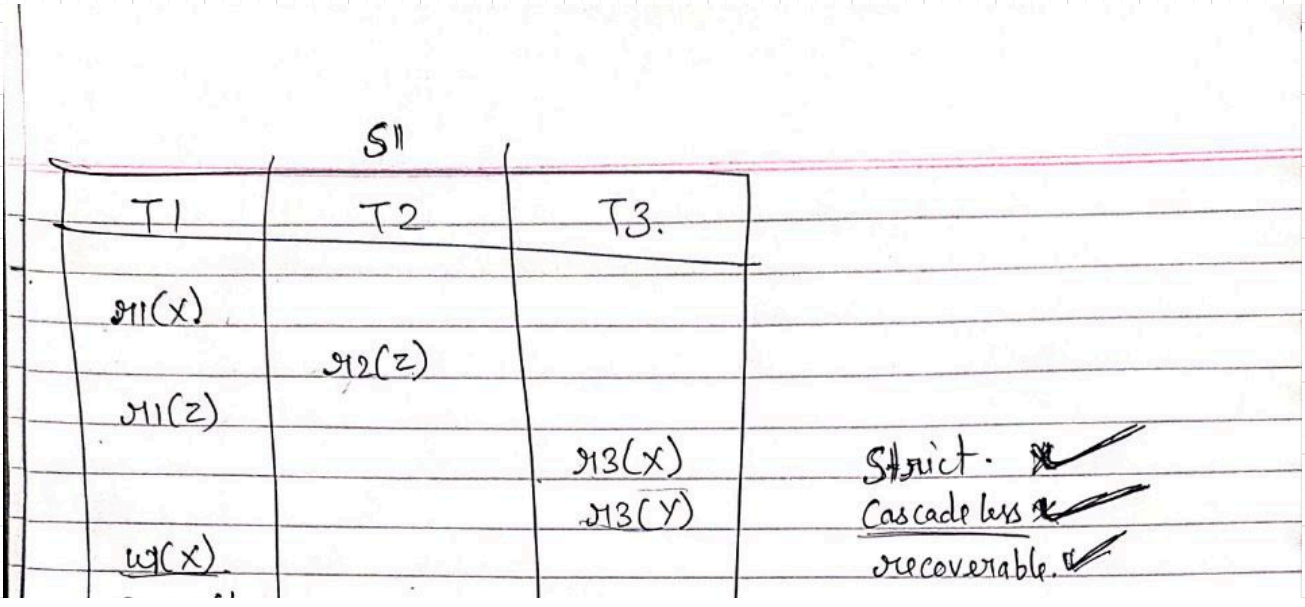
- The strictest type where no transaction can read or write a value modified by another transaction until that transaction has committed or rolled back.
- Why important? Makes recovery easier because no uncommitted changes are accessed by other transactions.

T1	T2	Explanation
Write(A)		T1 writes data to A.
	Wait	T2 is not allowed to read or write A until T1 commits.
Commit		T1 commits its transaction.
	Read(A)	T2 reads A only after T1 commits.
	Commit	T2 commits.

1. **Why Best?:** T2 neither reads nor writes A until T1 commits, ensuring maximum safety.

AKTU- 2022-23

Consider schedules S1, S2, and S3 below. Determine whether each schedule is strict, cascade less, recoverable, or non recoverable. (Determine the strictest recoverability condition that each schedule satisfies.)
S1: r1 (X); r2 (Z); r1 (Z); r3 (X); r3 (Y); w1 (X); c1; w3 (Y); c3; r2 (Y); w2 (Z); w2 (Y); c2;
S2: r1 (X); r2 (Z); r1 (Z); r3 (X); r3 (Y); w1 (X); w3 (Y); r2 (Y); w2 (Z); w2 (Y); c1;c2; c3;
S3: r1 (X); r2 (Z); r3 (X); r1 (Z); r2 (Y); r3 (Y); w1 (X); c1; w2 (Z); w3 (Y); w2 (Y);c3; c2;



Commit		
	W3(Y)	
	Commit.	Strictest Condition for S1 : Strict.
R2(Y)		
W2(Y)		
W2(Z)		
W2(Y)		
Commit		

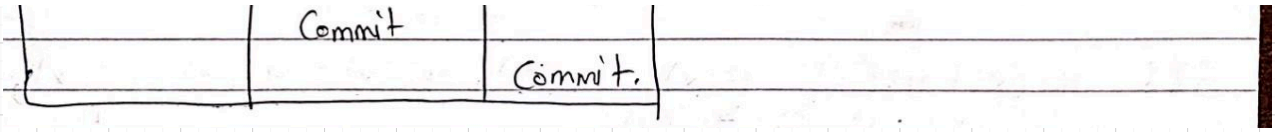
Recoverability → Commit of Modified (write of operation) Transaction is first before the commit of another transaction.

Cascadeless Schedule → Commit operation of T1 is Just (No dirty read) before the Read operation of T2

Strict Schedule → Data Modified (write) by one Transaction is allowed to perform R/W Operation after commit by T1 transaction.

S2

T1	T2	T3	
R1(X)			
	R2(Z)		Recoverable ✗
R1(Z)			Cascadeless ✗
		R3(X)	Strict ✗
		R3(Y)	
W1(X)			Strictest Condition for S2 : Non-Recoverable.
-		W3(Y)	
	R2(Y)		
	W2(Z)		
	W2(Y)		
Commit			



S3

T1	T2	T3
r1(x)		
	r2(z)	
		r3(x)
r1(z)		
	r2(y)	
		r3(y)
w1(x)		
Commit		
	w2(z)	
		w3(y)
	w2(y)	
		Commit
	Commit	

Recoverable ✓
Cascadeless ✓
Strict X

Strictest Condition for S3 : Cascadeless

Recovery System

It is responsible for ensuring the database's consistency and integrity after failures. It restores the database to its last consistent state using recovery techniques.

Types of Failures

Failures in DBMS can occur at various levels and are classified to identify the cause and restore the database to a consistent state.

1. Transaction Failure

Occurs when a transaction cannot complete or reach a consistent state.

Reasons for Transaction Failure

- **Logical Errors:**

  Errors in the transaction's logic (e.g., division by zero, constraint violations).
  Example: A transaction tries to withdraw more money than available in an account.

- **Syntax Errors:**

- Caused by system-imposed restrictions (e.g., deadlocks, resource unavailability).
- The DBMS may terminate the transaction automatically.
- Example: A transaction fails due to a deadlock.

2. System Crash

Occurs due to issues at the system level, like hardware or software failures.

Causes of System Crash

- **Power Failure:** Sudden loss of power halts the system.
- **Hardware Failure:** Malfunctioning components like RAM or processor.
- **Software Failure:** Errors in the operating system or DBMS.

Fail-Stop Assumption

- Non-volatile storage (e.g., hard drives) is assumed to remain unaffected during system crashes.

3. Disk Failure

Related to physical storage media.

Causes of Disk Failure

- **Bad Sectors:** Corruption in parts of the disk.
- **Head Crashes:** Physical damage to the disk's read/write head.
- **Disk Unreachability:** System fails to access the disk due to connectivity issues.

Impact of Disk Failure

- Partial or complete loss of stored data.
- Requires backup restoration or specialized recovery tools.

1. Undo Operation

- **Purpose:** Reverts changes made by uncommitted transactions.
- **When Used:** If a transaction fails or aborts, its changes to the database must be rolled back to maintain consistency.

Example of Undo
Transaction T1:

1. Write(A = 50)
2. Write(B = 30)

If T1 fails before committing:

Undo changes:

- Restore A to its original value.
- Restore B to its original value.

Multi Atoms

2. Redo Operation

- **Purpose:** Reapplies changes made by committed transactions to ensure durability.
- **When Used:** If a system crash occurs after a transaction commits but before the changes are written to the database, those changes need to be reapplied.

Transaction T2:

1. Write(A = 100)
2. Write(B = 200)
3. Commit

If a crash occurs after the commit but before changes are fully applied:

Redo changes:

- Reapply A = 100.
- Reapply B = 200.

Log-Based Recovery

Aktu 2023-24

A log is a sequence of records stored in stable storage to enable recovery of the database after a failure. Each database operation is logged before it is applied.

How Logs Are Maintained

1. **Start of Transaction**
   - **<Tn, Start>:** Indicates the transaction Tn has started.
2. **Modification Log**
   - **<Tn, Account_Balance, 1000, 1200>:** Logs the old and new values after a transaction modifies the Account_Balance.
3. **Commit Log**
   - **<Tn, Commit>:** Indicates that the transaction Tn has been successfully completed.
4. Abort Log
   - **<Tn,abort>:** Indicates Any failure occur

Database Modification Approaches

Aktu 2023-24

1. Deferred Database Modification

- **Definition:** Database changes are applied only after the transaction commits.
- **Log Requirement:**

1. Operations are logged, but changes are deferred until the transaction commits.
2. No undo is required, as changes are applied only after ensuring transaction success.

<T0. Start>
<T0. A. 850. 800>
<T0, B, 1000, 1050>
<T0. Commit>
<T1. Start>
<T1, C, 600, 500>

T0 (Transaction)	T1 (Transaction)
Start	
Write(A: 850 → 800)	
Write(B: 1000 → 1050)	
Commit	
	Start
	Write(C: 600 → 500)

Recovery Process:

- If the system crashes after <T0, Commit>, redo T0 as its changes are in the log.
- If the system crashes before <T1, Commit>, ignore T1's changes, as they are not applied.

2. Immediate Database Modification

- **Definition:** Changes to the database are applied immediately, even before the transaction commits.
- **Log Requirement:**

1. Every operation writes to the log before the actual database modification.
2. Both undo and redo operations are needed in case of failure.

Advantages:
Faster updates during transaction execution.

<T0, Start>
<T0, A, 850, 800>
<T0, B, 1000, 1050>
<T0, Commit>
<T1, Start>
<T1, C, 600, 500>

T0 (Transaction)	T1 (Transaction)
Start	
Write(A: 850 → 800)	
Write(B: 1000 → 1050)	
Commit	
	Start
	Write(C: 600 → 500)

Recovery Process:

- **If the system crashes after <T0, Commit>, redo T0 since it is committed.**
- **If the system crashes before <T1, Commit>, undo T1 since it is not committed.**

Checkpoints

A checkpoint is a mechanism that marks a point in the transaction log where the system was in a consistent state. It ensures efficient log management by discarding older logs after the checkpoint and recording new ones after the checkpoint.

Recovery Process:

1. **Log Scanning:** The recovery system scans the logs in reverse order, starting from the most recent transaction logs and going back to the checkpoint.

- **Redo List:** Transactions that were committed (i.e., <Tn, Start> and <Tn, Commit>) and need to be reapplied.
- **Undo List:** Transactions that were incomplete (i.e., <Tn, Start> but no <Tn, Commit> or <Tn, Abort>) and need to be rolled back.

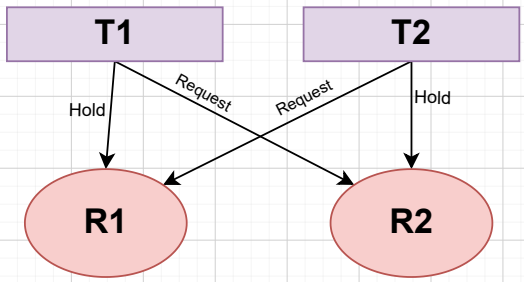
Example of Recovery:
Let's assume we have the following log after a checkpoint:

Log Record	Transaction T1	Transaction T2	Transaction T3	Transaction T4
<T1, Start>	X			
<T2, Start>		X		
<T3, Start>			X	
<T2, Commit>		X		
<T4, Start>				X
<T3, Commit>			X	

Redo transactions: T2 and T3 (both have <Tn, Start> and <Tn, Commit>).
Undo transactions: T1 (only <Tn, Start>) and T4 (only <T4, Start>).

Deadlock in DBMS

A deadlock occurs in a database when **two or more transactions are stuck because each is waiting for the other to release a resource. This creates a cycle where no transaction can proceed, halting the system.** Deadlocks are a significant challenge in multi-user environments and can severely impact the system's performance and reliability.



Characteristics of Deadlock / Necessary Conditions

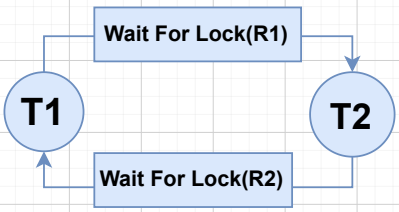
1. **Mutual Exclusion:** Only one transaction can hold a specific resource at a time.
2. **Hold and Wait:** A transaction holding resources may request additional resources held by others.
3. **No Preemption:** Resources cannot be forcibly taken from a transaction; they must be released voluntarily.
4. **Circular Wait:** A set of transactions exists where each is waiting for the next to release a resource.

Deadlock Handling

1. Deadlock Detection

Wait-For Graph:

- Transactions are represented as nodes. If a cycle is detected in the graph, a deadlock exists.
- **Action:** Abort one transaction in the cycle to break the deadlock.



2. Deadlock Avoidance

- **Resource Ordering**: Always access resources in a predefined order.
- **Release Quickly:** Release resources immediately after use.
- **Lock Granularity**: Use finer locks (e.g., row-level instead of table-level) to reduce conflicts.

1. **Example:**
   - For a **Students** and **Exams** table, always access Students first, then Exams. This consistency avoids scenarios where transactions hold conflicting locks.

3. Deadlock Prevention

Deadlock prevention ensures that the system allocates resources in a way that avoids circular waits, which are the main cause of deadlocks. Two common schemes for prevention are Wait-Die and Wound-Wait.

Wait-Die Scheme

- If an older transaction requests a resource held by a younger transaction, it waits.
- If a younger transaction requests a resource held by an older transaction, the younger one is aborted and restarted later.
- Key Idea: Older transactions have higher priority, and younger ones are rolled back when conflicts occur.

Example:

- Transaction T1 (older) requests a resource held by T2 (younger) → T1 waits.
- Transaction T2 (younger) requests a resource held by T1 (older) → T2 is aborted and restarted.

Wound-Wait Scheme

- If an older transaction requests a resource held by a younger transaction, the younger transaction is aborted (wounded) and restarted later.
- If a younger transaction requests a resource held by an older transaction, it waits until the resource is released.
- Key Idea: Older transactions can preempt younger ones to avoid deadlocks.

Example:

- Transaction T1 (older) requests a resource held by T2 (younger) → T2 is aborted, and T1 proceeds.
- Transaction T2 (younger) requests a resource held by T1 (older) → T2 waits.

Aspect	Wait-Die Scheme	Wound-Wait Scheme
Technique	Non-preemptive	Preemptive
Older Transaction	Waits for younger transactions.	Forces younger transactions to abort.
Younger Transaction	Aborted if it requests older transactions.	Waits for older transactions to release.
Number of Rollbacks	Higher due to frequent aborts of younger transactions.	Lower as younger transactions wait instead of rolling back often.
Resource Allocation	Older transactions are more patient.	Older transactions are more aggressive.
System Overhead	Higher due to increased rollbacks.	Lower as fewer rollbacks occur.

Q. Discuss the procedure of deadlock detection and recovery in transaction?

AKTU- 2021-22 & 2023-24

In multi-user database systems, deadlocks can occur when two or more transactions wait indefinitely for resources held by each other. To ensure the system operates smoothly, it is crucial to detect and recover from deadlocks efficiently.

Deadlock Detection

Deadlock detection identifies cycles of waiting transactions that prevent further progress. The primary approach is the Wait-for Graph.

Wait-for Graph (WFG)

- **Definition:** A directed graph where each node represents a transaction, an edge T1 -> T2 indicates that transaction T1 is waiting for a resource held by transaction T2.
- **Cycle:** If the graph contains a cycle, a deadlock is present.

Detection Steps:

1. **Monitor Resources:** The DBMS tracks transactions and their resource requests.
2. **Graph Construction:** The system constructs a wait-for graph using transaction states and resource allocations.
3. **Cycle Detection:** Algorithms such as depth-first search (DFS) are used to find cycles in the graph.
4. **Deadlock Confirmation:** If a cycle exists, the transactions involved are declared

deadlocked.

Deadlock Recovery

Once a deadlock is detected, the system must resolve it to allow progress. Common recovery strategies include:

1. Transaction Abortion

Key Idea: Abort one or more transactions involved in the deadlock to break the cycle.
Criteria for Selection:

- **Priority:** Abort younger or less critical transactions first.
- **Resource Usage:** Choose transactions holding fewer resources.
- **Progress:** Prefer aborting transactions with minimal progress to avoid wasting work.

2. Rollback Transactions

Key Idea: Undo the actions of the aborted transactions.
Steps:

- Use log-based recovery techniques to undo changes made by the transaction.
- Ensure data consistency and integrity by restoring the database to its previous state.

3. Timeout Mechanism

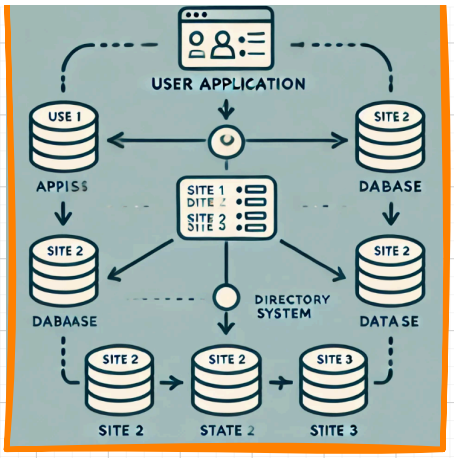
Key Idea: Automatically terminate transactions waiting too long for resources.

- **Implementation:** A transaction is forcefully aborted if it exceeds the timeout threshold.

Distributed Database

It is a collection of data spread across multiple locations, interconnected via a network. Each site in a distributed database system functions independently, but together they form a unified database system.

Key components of distributed databases include **Distributed Data Storage, Concurrency Control, and Directory System**, explained below:



1. Distributed Data Storage

Distributed Data Storage refers to the practice of splitting and storing a database across multiple physical locations, which could be on different servers or geographical regions. This allows for better performance, scalability, and fault tolerance.

Techniques:

1. Fragmentation:

- The database is divided into smaller pieces called fragments. These fragments can be stored across multiple locations.
- Horizontal Fragmentation: Divides a table by rows (e.g., all customer data for a specific region).
- Vertical Fragmentation: Divides a table by columns (e.g., only storing certain fields like customer names or addresses).

2. Replication:

- Copies of the same data are stored at multiple sites to ensure availability and faster access.
- Full Replication: All data is copied to every site.
- Partial Replication: Only some data is copied across sites, based on access patterns or other criteria.

3. Hybrid Approach:

- Combines both fragmentation and replication to ensure that data is divided efficiently and replicated for fault tolerance.

Advantages:

- **Faster local access to data:** Data can be stored closer to users or applications, reducing latency.
- **Improved reliability and fault tolerance:** Even if one site fails, data is still available from other sites that store replicas.

Challenges:

- **Data Synchronization across Sites:** Keeping data consistent across multiple locations can be complex, especially in cases of updates or changes.
- **Increased Storage Requirements:** Replicating data across multiple sites requires additional storage space, which can increase costs.

2. Concurrency Control

Concurrency Control ensures that multiple transactions running simultaneously across different locations in a distributed database do not cause inconsistencies or violations of data integrity.

Goals:

- **Maintain Data Integrity:** Ensures that concurrent transactions do not interfere with each other, keeping the database consistent.
- **Prevent Conflicts during Concurrent Updates:** Prevents issues like lost updates, temporary inconsistency, or conflicting updates.
- **Preserve Transaction Isolation and Consistency:** Ensures that transactions are isolated from one another and the system remains in a consistent state even when multiple transactions are executed concurrently.

Techniques:

1. **Lock-Based Protocols:**

   - **Distributed Two-Phase Locking (2PL):** A protocol where each transaction locks resources before it starts and releases locks after completing. It ensures consistency by maintaining a consistent order of locking across sites, ensuring that all required locks are acquired before a transaction can be executed.

2. **Time-Stamp Ordering:**

   - Each transaction is assigned a unique global timestamp, and conflicts are resolved by the order of their timestamps. This ensures transactions follow the correct sequence without interfering with one another.

3. **Optimistic Concurrency Control:**

   - Transactions execute without locks or restrictions, but before committing, they are validated to check if any conflicts occurred during their execution. If no conflicts are found, the transaction is committed; otherwise, it is rolled back and retried.

4. **Quorum-Based Protocols:**

   - In this method, a transaction requires approval from a majority (quorum) of the nodes before it can proceed. This ensures that data is not modified by transactions that are not fully validated by the majority.

3. Directory System

The directory system in a distributed database maintains metadata about the database's structure, data locations, fragmentation, and replication. It functions like a "map" that tracks where data resides and how it's organized, enabling efficient data retrieval and management across multiple sites.

Responsibilities:

- **Locate Data or Fragments:** It helps in finding where specific data or its fragments are stored across the distributed system.
- **Manage Replication:** It tracks duplicate copies of data to ensure they are available and up to date across different sites.
- **Transparent Access:** The directory system ensures that users and applications can access data without needing to know the physical location of the data or where it is replicated.

Types:

1. **Centralized Directory:**
   - A single directory that holds all the metadata. It's simple to implement but creates a single point of failure, which could be problematic for reliability and availability.
2. **Distributed Directory:**
   - Metadata is distributed across multiple locations or nodes. This improves reliability, fault tolerance, and access speed, as it eliminates the risks associated with a single point of failure.
3. **Hierarchical Directory:**
   - This approach combines both centralized and distributed systems, organizing the metadata in a tree-like structure to balance efficiency and scalability.