UNIT I

What is Data?
Data is a collection of a distinct unit of information. Data is basically information that can be translated into a particular form for efficient movement and processing. Data consists of raw facts and figures - it does not have any meaning until it is processed and given a context. Data comes in many forms.

What is information?
Data is basically information that can be translated into a particular form for efficient movement and processing.
Data consists of raw facts and figures - it does not have any meaning until it is processed and given a context.
Data comes in many forms.

Database System Concepts
1. Database
2. DBMS (Database Management System)
3. View of Data: Data Abstraction, Instances and Schemas
4. Data Model

Database

What is database?
A database is an organized collection of data that is stored and managed to ensure easy access, retrieval, and updating.Databases are designed to store large amounts of information efficiently and allow users or applications to query and manipulate the data.

Why Database?
Manages large amounts of data
Easy to update data
Access
Security of data
Easy to research data

Data Base Management System (DBMS)

A Database Management System (DBMS) is a software system that enables users to create, manage, and interact with databases efficiently. It serves as an interface between the database and the end-users or applications, ensuring that data is organized, stored, and retrieved effectively while maintaining security and consistency.

Advantage of DBMS
- Controls redundancy : It stores all the data in a single database file, so it can control
- data redundancy.
- Data sharing
- Backup
- Multiple user interfaces

Disadvantage of DBMS
- Size : It occupies large disk space and large memory to run efficiently.
- Cost : DBMS requires a high-speed data processor and larger memory to run DBMS
- software, so it is costly.

- Complexity : DBMS creates additional complexity and requirements.

Features of DBMS

Data Storage: Organizes data in structured formats (e.g., tables, files, or documents).
- Data Manipulation: Allows CRUD operations: Create, Read, Update, Delete.
- Querying: Provides query languages like SQL (Structured Query Language) to retrieve and manipulate data.
- Data Integrity: Ensures data accuracy and consistency.
- Security: Controls access through authentication and permissions.
- Concurrency Control: Manages multiple user access simultaneously without conflicts.
- Backup and Recovery: Protects data through regular backups and restores in case of failures.
- Data Independence: Separates data from applications, allowing modifications without impacting programs.

Problem With File System
- Data redundancy and inconsistency.
- Difficulty in Accessing Data:
- Data isolation.
- Integrity problems.
- Atomicity problems.
- Concurrent-access anomalies
- Security problems

Database System Concepts and Architecture

Database System Concepts and Architecture refer to the fundamental principles, components, and structures that define how a database system is designed, implemented, and operated.

View of Data

Data Abstraction

Data abstraction in databases hides complex data structures from users by providing different levels of abstraction, making data retrieval and interaction simple and efficient.

Three Levels of Data Abstraction:
- Physical Level: How data is physically stored.
- Logical Level: What data is stored and relationships.
- View Level: User-specific views of data.

Instances and Schema:

Instances: An instance refers to the actual data stored in the database at any given point in time. It is the content of the database as it exists for a specific moment.

•Schema: The schema defines the logical structure of the database, including the tables, relationships, and constraints. It is a blueprint of how data is organized.

• Physical Schema: The physical schema describes the database design at the physical level. Describes how data is stored on storage media (e.g., disk).

•Logical Schema: Describes the logical view of data

Data Model

A data model defines how data is represented, stored, and organized in a database system.

Common data models include:

Relational Model

Network model

Hierarchical model

Object oriented model

Client-Server Architecture:

- *Two-tier*: Direct client-server interaction.

- *Three-tier*: Includes an application server layer

A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database system can be broadly divided into the storage manager and the query processor components.

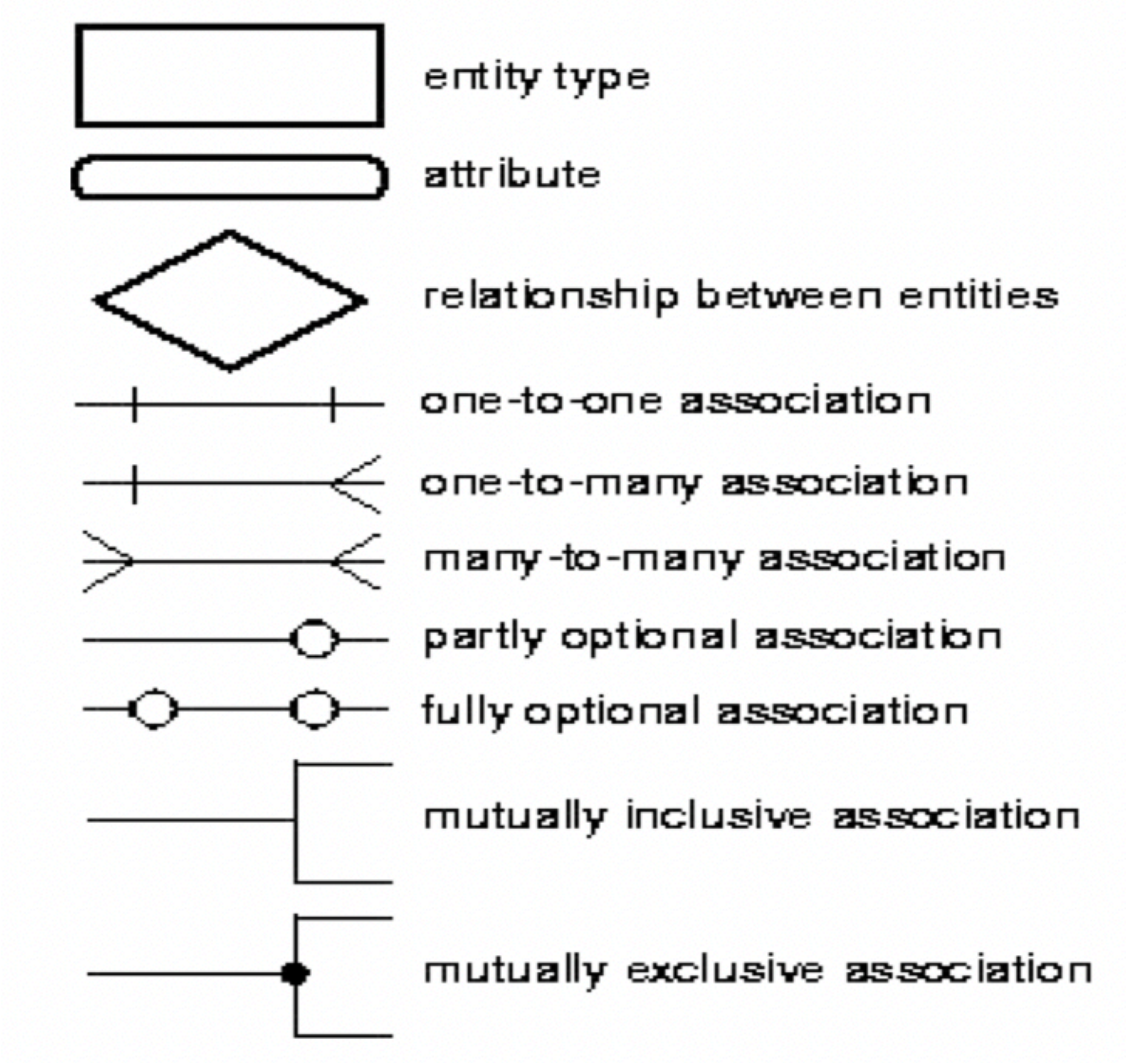
- *Storage Manager*: Handles data storage, integrity, transactions, buffering, and indexing.
- *Query Processor*: Interprets and optimizes DDL/DML queries and executes them.

Database Design Process

- Requirements Collection and Analysis
- Conceptual Database Design (using ER/EER models)
- Choice of DBMS
- Logical Database Design (mapping conceptual to logical schema)
- Physical Database Design (storage structures, indexing)
- Implementation and Tuning

Strategies for Schema Design

- Top Down Strategy: Start with a schema containing high-leve abstractions and then apply successive top-down refinements
- Bottom-Up Strategy: Start with a schema containing basics abstractions and then combine or add to these abstractions.



Entity-Relationship (ER) Model

- Entities: Real-world objects (e.g., Customer, Product, Student).
- Attributes: Properties describing entities, classified as:
  Simple, Composite, Single-Valued, Multi-Valued, Derived, Key, Stored, Complex.

Types of Attributes in ER Models

1. Simple Attribute: Cannot be divided into smaller components.
2. Composite Attribute: Can be divided into smaller sub-parts, each with its own meaning.
3. Single-Valued Attribute: Holds only one value for each entity instance.
4. Multi-Valued Attribute: Can hold multiple values for a single entity instance.
5. Derived Attribute: Values are derived from other attributes.
6. Key Attribute: A unique attribute that identifies an entity instance uniquely.
7. Stored Attribute: An attribute whose value is stored in the database, not derived from other attributes.
8. Complex Attribute: A combination of multi-valued and composite attributes.

Strong entity set	Weak entity set
A single rectangle is used for the representation of a strong entity set.	A double rectangle is used for the representation of a weak entity set.
It contains sufficient attributes to form its primary key.	It does not contain sufficient attributes to form its primary key.
A diamond symbol is used for the representation of the relationship that exists between the two strong entity sets.	A double diamond symbol is used for the representation of the identifying relationship that exists between the strong and weak entity set.
A single line is used for the representation of the connection between the strong entity set and the relationship.	A double line is used for the representation of the connection between the weak entity set and the relationship set.
Total participation may or may not exist in the relationship.	Total participation always exists in the identifying relationship.

Relations

In the Entity-Relationship (ER) model, relations describe the associations or interactions between entities. These relationships are crucial in defining how entities are connected and interact with one another in a database system.

Total Participation – Each entity is involved in the relationship. Total participation is represented by double lines.

- Partial participation – Not all entities are involved in the relationship. Partial participation is represented by single lines.

Entity set
• An entity set is a set of entities of the same type that share the same properties, or attributes. The set of all persons who are customers at a given bank, for example, can be defined as the entity set customer.
• The individual entities that constitute a set are said to be the extension of the entity set. Thus, all the individual bank customers are the extension of the entity set custome Keys in ER and Relational Models

Key Type	Unique?	Allows NULL?	Purpose/Description
Super Key	Yes	Yes	The super key is a set of one or more columns whose combined values can uniquely identify each record within a table
Candidate Key	Yes	No	Minimal super key, potential primary key
Primary Key	Yes	No	uniquely identifies each record within a table. It cannot contain duplicate values or NULL values.
Alternate Key	Yes	No	Candidate keys not chosen as primary key but is unique(ex: mobile no)
Foreign Key	No	Yes	Establishes relationships between tables
Unique Key	Yes	Yes (one)	Ensures uniqueness, allows one NULL
Composite Key	Yes	No	Combination of attributes uniquely identifying record may not be unique on their own, their combination ensures uniqueness.
Surrogate Key	Yes	No	System-generated artificial key (e.g., auto-increment)
Secondary Key	No	Yes	Used for searching/indexing, not unique

Super key → Any key that can identify a row.

Candidate key → The simplest form of a super key.

Advantages of ER Model

• Conceptually it is very simple: ER model is very simple because if we know relationship between entities and
attributes, then we can easily draw an ER diagram.
• Better visual representation: ER model is a diagrammatic representation of any logical structure of database. By
seeing ER diagram, we can easily understand relationship among entities and relationship.

Super Key	Candidate Key
Super Key is an attribute (or set of attributes) that is used to uniquely identifies all attributes in a relation.	Candidate Key is a subset of a super key.
All super keys can't be candidate keys.	But all candidate keys are super keys.
Various super keys together makes the criteria to select the candidate keys.	Various candidate keys together makes the criteria to select the primary keys.
In a relation, number of super keys is more than number of candidate keys.	While in a relation, number of candidate keys are less than number of super keys.

• Effective communication tool: It is an effective communication tool for database designer.
• Highly integrated with relational model: ER model can be easily converted into relational model by simply
converting ER model into tables.
• Easy conversion to any data model: ER model can be easily converted into another data model like hierarchical
data model, network data model and so on.

Super classes and Subclasses- Inheritance relationship

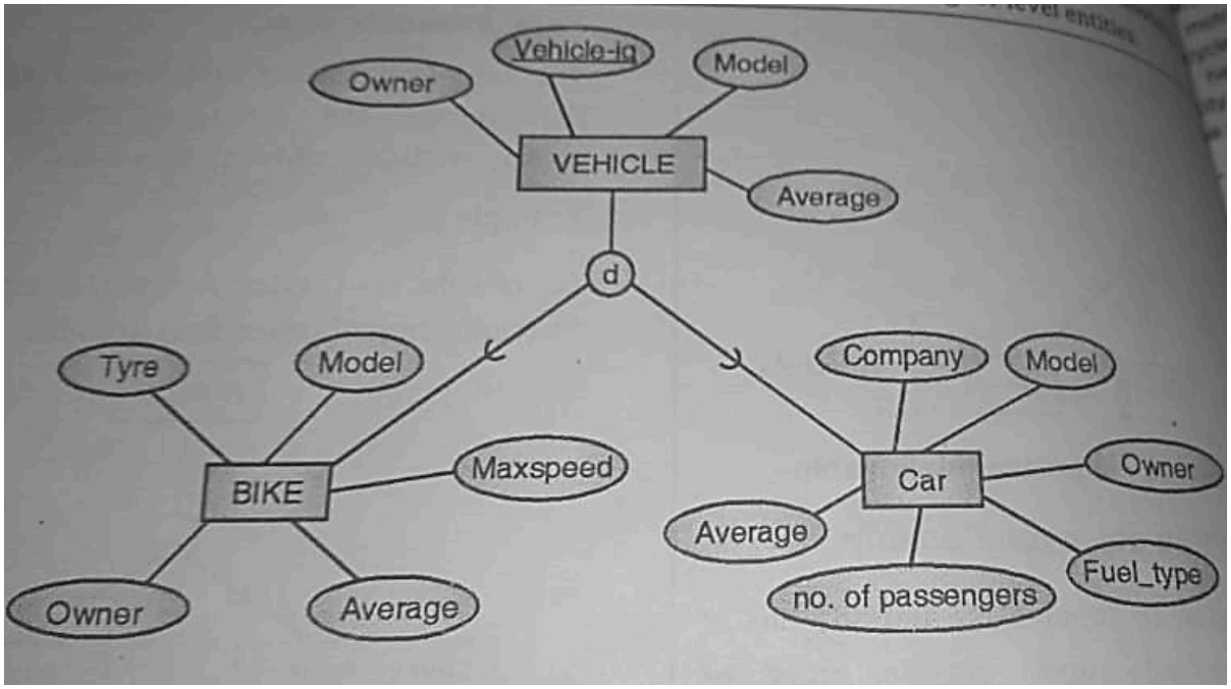
Super class and sub class relationship leads to concept of inheritance.
The relationship between sub class and super class is denoted with d symbol.

Specialization

Specialization is a process that defines a entities which is divided into sub groups based on their characteristic. It is a top down approach, in which one higher entity can be broken down into two lower level entity. Ex: Employee is a developer and tester.

Generalization

Generalization is the process of generalizing the entities which contain the properties of all the generalized entities, i.e., subclasses are combined to make a superclass. It is a bottom approach, in which two lower level entities combine to form a higher level entity Ex: Tiger Lion and Elephant is a animal.



Aggregation

Aggregation is a process in which a single entity alone is not able to make sense in a relationship so the relationship of two entities acts as one entity. Aggregation can be defined as a procedure for combining multiple entities into a single one.

In real world, we know that a manager not only manages the employee working under them but he has to manage the project as well. In suchscenario if entity “Manager” makes a “manages” relationship with either “Employee” or “Project” entity alone then it will not make any sense because he has to manage both. In these cases the relationship of two entities acts as one entity. In our example, the relationship “Works-On” between “Employee” & “Project” acts as one entity that has a relationship “Manages” with the entity “Manager”.

Relational Model

The Relational Model explains how data is stored in relational databases — data is stored in tables (relations) where each table represents an entity and each row represents a record (tuple).

Constraints in Relational Model

Constraints are rules applied to data to maintain accuracy and consistency.
They are checked before any operation (insertion, deletion, or update) in the database.
If a constraint is violated, the operation fails.

1. Domain Constraints

- These are attribute-level constraints.
- They define what values are allowed for an attribute.
- Example: If the constraint AGE > 0 is applied on the AGE field, then inserting a negative value like AGE = -5 will fail.

2. Key Constraints (Key Integrity)

- Each relation (table) must have a unique identifier, called a key.

- The key uniquely identifies every record (tuple) in a table.
- Example: ROLL_NO is the key in the STUDENT table.
    - No two students can have the same roll number.
    - Key values cannot be NULL.

Referential Integrity: When one attribute of a relation can only take values from other attribute of any other relation, it is called referential integrity.

Anomalies in Database

1. Insertion Anomaly (in Referencing Relation)

This occurs when you cannot insert a record in a table because it violates a foreign key constraint.

Example:

- Suppose you have two tables:
    - BRANCH (Referenced Relation) — contains BRANCH_CODE
    - STUDENT (Referencing Relation) — also has a BRANCH_CODE that references BRANCH

If you try to insert a student with BRANCH_CODE = ‘ME' but 'ME' does not exist in the BRANCH table, the database will not allow the insertion.

2. Deletion / Update Anomaly (in Referenced Relation)

This happens when you try to delete or update a record from the referenced table that is still being used by another table. Example: If you try to delete a branch with BRANCH_CODE = ‘CS' while there are students in the STUDENT table using 'CS' as their branch, the database will not allow the deletion because it would leave “orphan” records in STUDENT.

Codd’s 12 Rules:

Rule No.	Name	In Simple Words
0	Foundation Rule	Must follow all other rules
1	Information Rule	Store everything in tables
2	Guaranteed Access	Every piece of data is easily accessible
3	Nulls	Handle missing values properly
4	Catalog	Database structure also stored in tables
5	Language	SQL or similar must support all operations
6	View Update	Should allow updating through views
7	Set-level Ops	Allow bulk operations
8	Physical Data Independence	Internal storage changes shouldn't affect users
9	Logical Data Independence	Adding fields shouldn't break apps
10	Integrity Independence	Constraints defined in DB, not apps
11	Distribution Independence	Works same on one or many computers
12	Non-subversion	No bypassing the rules

UNIT II

Tuple:A row in a relation is called a tuple.

Attribute:A column in a relation is called an attribute. It is also termed as field or data item.

Degree: Number of attributes in a relation is called degree of a relation.

Cardinality: Number of tuples in a relation is called cardinality of a relation.

Core Concepts of Normalization

- Normalization is a systematic approach to minimize redundancy and avoid anomalies in database tables.
- A poorly designed database can lead to anomalies such as update, deletion, and insertion problems, making data management complex and error-prone.
- The goal of normalization is to bring the database to a consistent state by removing these anomalies.

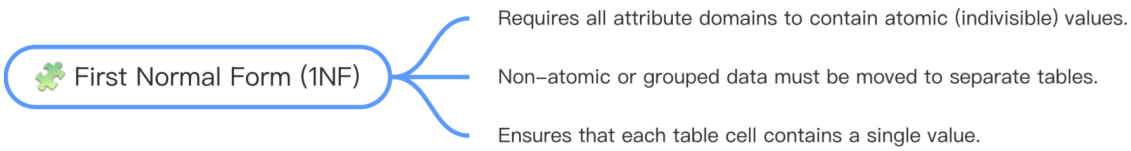
Problems with Unnormalized Data

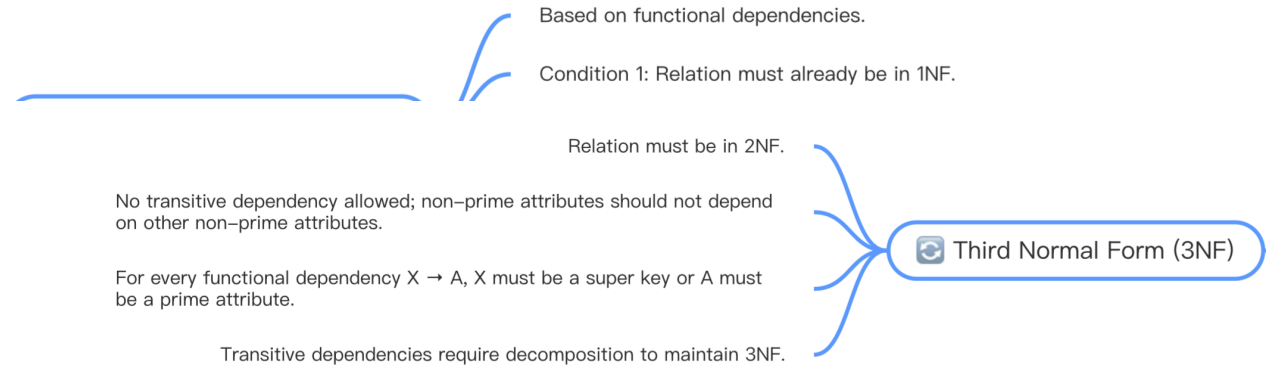
Unnormalized data can cause several issues, notably:

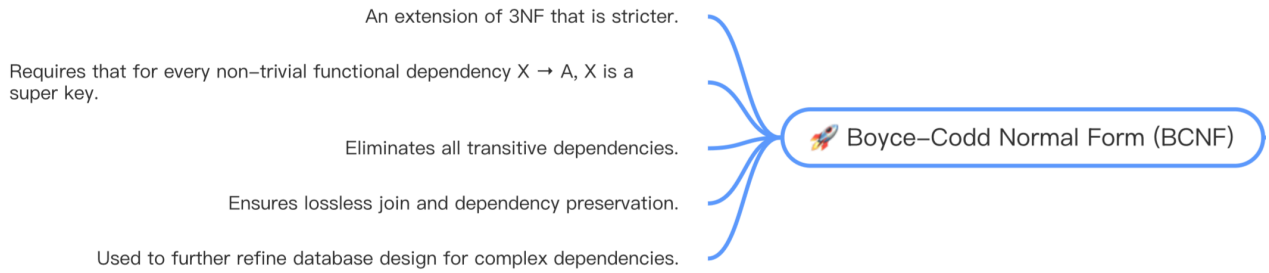
- Update Anomalies: Data items scattered across multiple places make it difficult to update records consistently.
- Deletion Anomalies: Deleting data in one relation might inadvertently remove related data, leading to loss of important information.
- Insertion Anomalies: Inability to insert data due to missing related data in the record, causing incomplete or inconsistent data entries.

Normalization addresses these by ensuring data is stored logically and consistently.

Types of Normal Form







❌ Not in 1NF:

Student_ID	Name	Subjects
1	Ravi	Math, Science
2	Neha	English, Math

👉 “Subjects” contains multiple values → not atomic.

✅ In 1NF:

Student_ID	Name	Subject
1	Ravi	Math
1	Ravi	Science
2	Neha	English
2	Neha	Math

🧩 2NF (Second Normal Form)

Rule: Must be in 1NF + **no partial dependency** (non-key attribute depends on full key).

Example:
❌ Not in 2NF

Emp_ID	Proj_ID	Emp_Name	Proj_Name
1	P1	Ravi	Alpha

✅ In 2NF
Employee: (Emp_ID, Emp_Name)
Project: (Proj_ID, Proj_Name)
Works_On: (Emp_ID, Proj_ID)

🧩 3NF (Third Normal Form)

Rule: Must be in 2NF + **no transitive dependency** (non-key depends only on key).

Example:
❌ Not in 3NF

Student_ID	Student_Name	Dept_ID	Dept_Name
1	Ravi	D1	Computer

✅ In 3NF
Student: (Student_ID, Student_Name, Dept_ID)
Department: (Dept_ID, Dept_Name)

🧩 BCNF (Boyce–Codd Normal Form)

Rule: For every dependency X → Y, **X must be a superkey**.

Example:
❌ Not in BCNF

Student	Course	Instructor
Ravi	DBMS	Prof. A
Neha	DBMS	Prof. A

FDs:
(Student, Course) → Instructor
Instructor → Course ❌ (Instructor not a key)

✅ In BCNF
Instructor: (Instructor, Course)
Student_Instructor: (Student, Instructor)

Structured Query Language (SQL)

- **Characteristics**:
  - Non-procedural (focuses on *what* to do, not *how*).
  - Easy to learn/use.
  - Handles large databases efficiently.
  - Compatible with various DBMS (Oracle, MySQL, MS SQL Server, etc.).
- **Advantages**:
  - High-speed data retrieval.
  - Standardization (ISO).
  - Minimal coding for database operations.
  -

SQL Commands and Processing Capabilities

- DDL (Data Definition Language): Create/modify database objects.
  - `CREATE`, `ALTER`, `DROP` (tables, views, indexes).
- DML (Data Manipulation Language): Modify/query data.
  - `SELECT`, `INSERT`, `UPDATE`, `DELETE`.
- DCL (Data Control Language): Manage access/permissions.

    - `GRANT`, `REVOKE`.
- TCL (Transaction Control Language): Manage transactions.
    - `COMMIT`, `ROLLBACK`, `SAVEPOINT`.

Integrity Constraints

- **Purpose**: Ensure data consistency and prevent invalid data entry.
- Types:

Constraint Type	Description
`NOT NULL`	Column cannot have NULL values
`UNIQUE`	All values in a column must be distinct
`PRIMARY KEY`	Combination of `NOT NULL` and `UNIQUE`, uniquely identifies rows
`FOREIGN KEY`	Enforces referential integrity by linking to primary key in another table
`CHECK`	Ensures values satisfy a specific condition
`DEFAULT`	Provides default value if none is specified
`INDEX`	Improves the speed of data retrieval

Feature	DELETE	TRUNCATE	DROP
Command Type	DML (Data Manipulation Language)	DDL (Data Definition Language)	DDL
Purpose	Removes specific rows	Removes **all rows**	Removes **entire table**
Syntax	`DELETE FROM table WHERE condition;`	`TRUNCATE TABLE table;`	`DROP TABLE table;`
Removes Data?	✅ Yes	✅ Yes (all)	✅ Yes (and structure)
Removes Structure?	❌ No	❌ No	✅ Yes
WHERE Clause	✅ Allowed	❌ Not allowed	❌ Not applicable
Auto Increment Reset	❌ No	✅ Yes	✅ Yes (table recreated if needed)

Rollback (Undo)	✅ Possible	❌ Usually not (depends on engine)	❌ No
Speed	⏳ Slower	⚡ Very Fast	⚡ Fastest
Triggers Activated?	✅ Yes	❌ No	❌ No
Use Case	Delete some data	Empty a table quickly	Remove table completely

UNIT IV

Transaction: A unit of program execution that accesses and possibly updates multiple data items in a database.

- Example: Transferring $50 from account A to account B involves reading, updating, and writing data items atomically.
- Must address two main issues: **failures** (hardware/software) and **concurrent execution** of multiple transactions.

transaction to transfer $50 from account A to account B:
1.read(A)
2.A := A – 50
3.write(A)
4.read(B)
5.B := B + 50
6.write(B)

ACID Properties: Essential properties of transactions ensuring correctness and reliability.

- **Atomicity**: All operations of a transaction are fully completed or none are reflected.
- **Consistency**: Transactions preserve database integrity and integrity constraints.
- **Isolation**: Concurrent transactions appear as if executed serially, hiding intermediate states.
- **Durability**: Once committed, changes persist despite system failures.

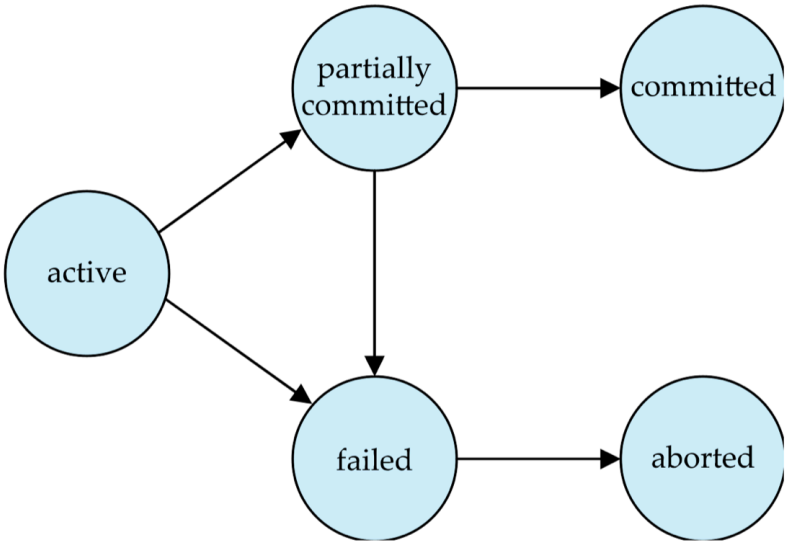
Transaction Execution and States

- **Transaction States**:

State	Description
Active	Transaction is executing.

• Partially Committed	• Final statement executed, but commit not yet done.
• Failed	• Transaction cannot continue normal execution.
• Aborted	• Transaction rolled back, database restored to pre-transaction state.
• Committed	• Transaction successfully completed and changes permanently saved.

- After aborting, a transaction may be **restarted** (if no logical errors) or **killed**.



Concurrent Executions

Multiple transactions are allowed to run concurrently in the system. Advantages are:

Increased processor and disk utilization, leading to better transaction throughput

Reduced average response time for transactions: short transactions need not wait behind long ones.

five concurrency problems

The five concurrency problems that can occur in the database are:

- Temporary Update Problem(Dirty Read)
  Temporary update or dirty read problem occurs when one transaction updates an item and fails. But the updated item is used by another transaction before the item is changed or reverted back to its last value.

  Transaction T1 updates a value but has not committed yet.
  Transaction T2 reads this uncommitted (dirty) value.
  Later, **T1 fails or rolls back**, and its update is undone.
  Now T2 has used a value that never actually existed in the database.
-

👉 **Why is it called a dirty read?**
Because T2 reads "dirty" (uncommitted, temporary, incorrect) data.

- Incorrect Summary Problem
  Aggregate functions compute inconsistent results due to concurrent updates. Consider a situation, where one transaction is applying the aggregate function on some records while another transaction is updating these records. The aggregate function may calculate some values

- Lost Update Problem
  Lost Update Problem (Simple Explanation)

The **Lost Update Problem** happens when:

1. **Two transactions read the same data item at the same time**.
2. Both calculate new values independently.
3. Each writes back its own result.
4. **The update of one transaction overwrites the update of the other**, causing one update to be *lost*.

🔷 Example (Easy to Understand)

Assume **X = 100**

Transaction T1

- Reads X = 100
- Adds 50 → new value = 150
- Will write X = 150

Transaction T2

- Reads X = 100
- Subtracts 30 → new value = 70
- Writes X = 70

What happens?

If **T1 writes first**, then **T2 writes after**, the final value becomes:

👉 **X = 70**

The update made by T1 (X = 150) is **lost**, because T2's write overwritten it.

- Unrepeatable Read Problem

An **unrepeatable read** happens when:

1. A transaction **T2 reads a value X**.
2. Another transaction **T1 updates X and commits**.
3. When **T2 reads X again**, it gets a **different value** than before.

So T2 **cannot repeat the same read** and get the same result — the value has changed in between.

- Phantom Read Problem

The phantom read problem occurs when a transaction reads a variable once but when it tries to read that same variable again, an error occurs saying that the variable does not exist.
In the above example, once transaction 2 reads the variable X, transaction 1 deletes the variable X without transaction two knowledge, thus, when transaction two tries to read X, it is not able to do it

Schedules

- Schedule: A sequence that defines the chronological order of instructions from concurrent transactions.
- Serial Schedule: Transactions executed one after another without interleaving.
- Concurrent Schedule: Transactions interleave, potentially improving performance but risking consistency.
- A schedule is serializable if it is equivalent to some serial schedule, preserving consistency.

Recoverability and Cascades

- Recoverable Schedule: If a transaction reads data written by another, the writer must commit before the reader commits.
- Cascading Rollbacks: Occur when a single transaction failure causes rollback of multiple dependent transactions.
- Cascadeless Schedule: Prevents cascading rollbacks by allowing transactions to read only committed data.
- Irrecoverable Schedule: Schedules where changes from aborted transactions are reflected, leading to inconsistency.

Serializability

- A schedule is serializable if it is equivalent to some serial schedule, preserving consistency.

Types of Serializability

Type	Definition	Notes
Conflict Serializability	Two schedules are conflict equivalent if one can be transformed into the other by swapping non-conflicting operations.	Easier to test using precedence graphs.
View Serializability	Two schedules are view equivalent if they produce the same read and write sequences and final database state.	More general; testing is NP-complete.

- Conflict occurs when operations from different transactions access the same data item and at least one is a write.
- Conflict serializability can be tested via precedence graphs:
  - Vertices represent transactions.
  - Directed edges represent conflicts.
  - Acyclic graph → schedule is conflict serializable.

Isolation Levels in SQL-92

Level	Description
Serializable	Default level; ensures full serializability.
Repeatable Read	Only committed data is read; repeated reads yield consistent results but may not guarantee serializability.
Read Committed	Only committed data is read; successive reads may differ.
Read Uncommitted	Even uncommitted data can be read, allowing dirty reads.

- Lower isolation levels trade consistency for performance and are suitable for approximate queries or read-only transactions.

- Some databases (e.g., Oracle) implement snapshot isolation, which is not part of SQL standard but widely used.

Implementation of Isolation

- Lock-based protocols:
  - Lock granularity: database-wide vs. item-level.
  - Lock types: shared (read) vs. exclusive (write).
  - Lock duration affects concurrency and overhead.
- Timestamp-based protocols:
  - Each transaction has a timestamp.
  - Data items track read and write timestamps.
  - Used to detect and prevent out-of-order access.
- Multi-version Concurrency Control (MVCC):
  - Maintains multiple versions of data items.
  - Transactions read from consistent snapshots.

Transaction Management in SQL

- Transactions begin implicitly in SQL.
- End with:
  - `COMMIT WORK`: commits current transaction.
  - `ROLLBACK WORK`: aborts current transaction.
- By default, individual SQL statements commit implicitly (auto-commit).
- Auto-commit can be disabled (e.g., JDBC `connection.setAutoCommit(false)`).
- Isolation levels can be set globally or per transaction (e.g., `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE`).

Precedence Graph and Serializability Testing

- Construction of the precedence graph is used to test conflict serializability.
- Acyclic graph means schedule is conflict serializable.
- Cycle detection algorithms exist with complexity $O(n^2)$ or better.
- Topological sorting of the graph yields a possible serial order equivalent to the concurrent schedule.

Concurrency control

Concurrency control is the procedure in DBMS for managing simultaneous operations without conflicting with each another

Concurrency Control Protocols

Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose

1. Lock-Based Protocol
2. Two Phase Locking Protocol
3. Timestamp-Based Protocols
4. Validation-Based Protocol
5. Graph-Based Protocols
6. Tree-Based Protocols

NOTE: maximum parallelism (high throughput)

Lock-Based Protocols

A lock is a mechanism to control concurrent access to a data item

Data items can be locked in two modes :

- exclusive (X) mode.
  Data item can be both read as well as written
  X-lock is requested using lock-X instruction
- shared (S) mode.
  Data item can only be read.
  S-lock is requested using lock-S instruction

Lock requests are made to concurrency-control manager
Transaction do not access data items before having acquired a lock on that data item
Transactions release their locks o**n a data item only after they have accessed a data item**

Rules for locks in transaction:

I. A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
II. Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive lock on the item no other transaction may hold any lock on the item
III. If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted

Lock compatibility matrix:

	S	X
S	true	false
X	false	false

This means that, for example, we have two transaction, T! and T2. For T1, we have shared lock on one item which is yet to be unlocked, and we want to have a shared lock for T2 also. So it is allowed since two read, don't generate a conflict. While others read-write, write-read , write-write generate conflict. That's why if an exclusive lock exist on the item, no other transactions may hold any lock on that item.

In a Lock-Based Concurrency Control Protocol, what is the primary purpose of locks to prevent data from being modified by multiple transactions simultaneously.

Pitfalls of Lock-Based Protocols

Consider the partial schedule

- Neither T3 nor T4 can make progress -executing lock-S(B) causes T4 to wait for T3to release its lock on B, while executing lock-X(A) causes T3to wait for T4 to release its lock on A.
- **Such a situation is called a deadlock.**

- To handle a deadlock one of T3or T4 must be rolled back and its locks released.

T3	T4
lock-x(B) read (B) B:= B - 50 write (B) lock-X(A)	 lock-s(A) read (A) lock-s(B)

Starvation occurs when a transaction waits indefinitely because it is repeatedly denied the lock it needs, even though the system is running normally.

2. How Starvation Happens

Starvation typically occurs when:

- A transaction waits for an exclusive lock (X-lock)
- Meanwhile, a continuous stream of other transactions request shared locks (S-locks)
- Since shared locks are compatible, they keep getting granted
- The waiting transaction never gets the exclusive lock

Example:
Transaction T1 requests an X-lock, but T2, T3, T4… keep getting S-locks first → T1 is starved.

3. Causes

- Poorly designed lock manager
- No fairness in lock granting
- New lock requests bypass earlier pending ones
- Repeated rollbacks due to deadlocks

4. Prevention of Starvation

To avoid starvation, the lock manager should follow fairness policies, such as:

1. FIFO ordering of lock requests
   → The lock is granted based on the order of arrival.
2. A lock is granted only if:
   - No conflicting lock is held
   - No earlier transaction is waiting for the same lock

Ensures every transaction eventually progresses.

5. Key Point

Starvation ≠ Deadlock

- Deadlock: transactions wait for each other
- Starvation: a transaction keeps waiting because others repeatedly get priority

The Two Phase Locking Protocol

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

- Growing phase. A transaction may obtain locks, but may not release any lock.
- Shrinking phase. A transaction may release locks, but may not obtain any new locks.
- The protocol assures serializability.
- Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.
- Cascading rollback may occur under two-phase locking.

Strict Two-Phase Locking (Strict 2PL)

Strict 2PL is a locking protocol that:

- Follows the Two-Phase Locking rule (growing phase + shrinking phase)
- Requires all exclusive locks (X-locks) to be held until the transaction commits

Key Properties

- Exclusive locks are not released early
- Prevents any other transaction from reading uncommitted data
- Therefore avoids the problem of Cascading Rollbacks

Also strict recoverable

Why it prevents cascading rollbacks?

Because any data written by an uncommitted transaction remains X-locked, so:

- No other transaction can read dirty data
- Even if a transaction aborts, no other transaction is affected

Rigorous Two-Phase Locking

A stronger version of strict 2PL.

- All locks (Shared + Exclusive) are held until commit/abort
- Ensures the highest level of safety

Serialization property

With Rigorous 2PL:

- Transactions appear to execute in the order they commit
- Produces strict schedules

Lock Conversion Mechanism

In basic 2PL, a transaction that will write a data item must take an **exclusive lock (X-lock)** from the beginning.

This reduces concurrency because other transactions cannot even **read** the item.

Lock conversion allows a transaction to:

- Start with a Shared lock (S-lock) while reading
- Later upgrade to an Exclusive lock (X-lock) when it needs to write

This increases concurrency while still maintaining serializability.

Timestamp Ordering Protocol

Timestamp Assignment
Each transaction Ti entering the system is given a unique timestamp TS(Ti).
A smaller timestamp means an older transaction.

Timestamps can be generated by:

1. System Clock → TS(Ti) = current time
2. Logical Counter → counter increments for each new transaction

If Ti comes before Tj, then TS(Ti) < TS(Tj).

The Timestamp Ordering (TO) protocol ensures:

- Conflict serializability
- Transactions are executed in the order of their timestamps
- Older transactions get priority over newer ones

It avoids deadlocks because no locks are used and transactions never wait; they are simply rolled back if they violate rules.

To check ordering, the system stores two timestamps for each item Q:

- R-timestamp(Q)
  The largest timestamp of any transaction that successfully read Q.
- W-timestamp(Q)
  The largest timestamp of any transaction that successfully wrote Q.

These timestamps help the protocol decide whether a read or write should be allowed or rejected.

A) Read Operation — Ti issues *read(Q)*

- Case 1: TS(Ti) < W-timestamp(Q)
  - Ti is trying to read an old value (a newer transaction already wrote Q)
  - This violates timestamp order
    → Read is rejected and Ti is rolled back
- Case 2: TS(Ti) ≥ W-timestamp(Q)
  - Read is allowed
  - Update:
    R-timestamp(Q) = max(R-timestamp(Q), TS(Ti))

B) Write Operation — Ti issues *write(Q)*

Write is allowed only if both conditions hold:

1. TS(Ti) ≥ R-timestamp(Q)
   - Ensures Ti is not overwriting a value needed by a newer reader.
2. TS(Ti) ≥ W-timestamp(Q)
   - Ensures Ti is not writing an old value.

If either condition fails:

- Ti is attempting to write an obsolete or unnecessary value
- This violates timestamp order
  → Write is rejected and Ti is rolled back

If write succeeds:

- Update: W-timestamp(Q) = TS(Ti)

Rollback and Restart

If any read or write violates the timestamp rules:

- The transaction Ti is rolled back
- It is restarted with a new timestamp, making it the newest transaction

This ensures the system always maintains timestamp order.

Advantages	Description
Prevents Conflicts	By using timestamps, the protocol prevents conflicts between transactions that might arise due to simultaneous reads or writes on the same data item.
Ensures Serializability	The timestamp ordering protocol ensures serializability because transactions are executed in timestamp order, which corresponds to some serial execution.
No Deadlock	Timestamp ordering inherently avoids deadlocks since transactions are not waiting for each other to release locks or resources. Transactions are either allowed to proceed or aborted and restarted.
Simple Concept	The use of timestamps makes it easy to implement and reason about, as the system can rely on logical time to enforce the execution order.
Disadvantages	**Description**
Rollback Overhead	If a transaction frequently violates the timestamp ordering rules, it might be rolled back and restarted multiple times. This can result in significant overhead.
Starvation	A transaction with an earlier timestamp may be starved if it is continually rolled back due to conflicts with later transactions, especially under high contention.
No Locking	Unlike other concurrency control mechanisms like Two-Phase Locking, the timestamp ordering protocol does not use locks, meaning it might face challenges in some scenarios requiring fine-grained control over resource usage.

Timestamp Ordering & Thomas' Write Rule

Thomas Rule: If a transaction attempts to write a value that is already outdated, instead of rolling it back, the system simply ignores the write.

Thomas' Write Rule — Conditions:

Case 1 — Reject
If TS(Ti) < R-timestamp(Q)

→ The value Ti wants to write was previously needed but is now too late.
→ Write is rejected, Ti rolled back.

Case 2 — Ignore (obsolete write) If

`TS(Ti) < W-timestamp(Q)`
→ Ti wants to write an old, outdated version of Q.
→ No need to rollback, because no future transaction will use this value.
→ The write is ignored. (No rollback)

Case 3 — Accept

If TS(Ti) ≥ R-timestamp(Q) an TS(Ti) ≥ W-timestamp(Q)

→ Ti is writing a fresh, valid value
→ Perform the write and update W-timestamp(Q)

See the example below:

- Since *T16 starts* before *T17, we shall assume that TS(T16) < TS(T17). The read(Q) operation of T16 succeeds,* as does the write(*Q) operation of T17.*

T_{16}	T_{17}
read(Q)	
	write(Q)
write(Q)	

- *When T16 attempts its write(Q) operation,* we find that TS(*T16) < W-timestamp(Q), since W-timestamp(Q) = TS(T17).*
- *Thus, the* write(*Q) by T16 is rejected and transaction T16 must be rolled back.*

In timestamp ordering, every transaction gets a timestamp when it starts.
If T16 starts before T17, then:

`TS(T16) < TS(T17)`
This fixes the *logical (serial) order* of transactions.

T16 reads Q → Allowed

T17 writes Q → Allowed

- Now W-timestamp(Q) = TS(T17)

T16 attempts write(Q)

- Check rule: `TS(T16) < W-timestamp(Q)` which means: `older transaction trying to overwrite value written by a younger one Therefore → Write is rejected, T16 is rolled back`

Even though timestamp-ordering protocol forces rollback, logically:

- T17 already wrote a *newer* value of Q.
- T16 (older) is trying to write an outdated value.
- This value will never be used by any future transaction.

Why?

Case 1: A transaction Ti with TS < TS(T17) wants to read Q

It will be rolled back anyway because:

`TS(Ti) < W-timestamp(Q)`

Case 2: A transaction Tj with TS > TS(T17)

It must read the value written by T17, not T16.

So the value T16 tries to write is obsolete and irrelevant.

Recovery System

Storage Types

1. Volatile Storage

- Loses contents on system crash.
- Examples: main memory (RAM), cache memory.

2. Non-Volatile Storage

- Survives system crashes.
- Examples: disk, SSD, tape, flash memory, non-volatile RAM.

3. Stable Storage

- Idealized storage that survives all failures.
- Achieved by keeping multiple copies of each block on different disks or remote sites for protection against disasters.

Failures During Block Transfer

During writing of a block to disk, three outcomes are possible:

1. Successful completion – data correctly stored.
2. Partial failure – block is updated with corrupted data.
3. Total failure – block never updated.

Protecting Against Transfer Failure

(Assuming two copies of each block)

1. Write to first block.
2. If success → write to second block.
3. Output considered complete only if both writes succeed.

Types of Failures

Transaction Failure

- Logical errors → invalid input, impossible operations.
- System errors → DBMS aborts transaction (e.g., deadlock).

System Crash

- Due to power/hardware/software failure → contents of volatile memory lost.
- *Fail-stop assumption*: non-volatile storage is not corrupted.

Disk Failure

- Head crash, physical damage → loss of disk data.

Recovery Algorithms

Recovery ensures ACID properties despite failures.

Recovery consists of:

A. Actions during normal processing → Save enough information to allow recovery.

B. Actions after failure → Restore database to a consistent state ensuring atomicity, consistency, durability.

Log-Based Recovery

A log is kept on stable storage.
It records all updates for crash recovery.

Log Records:

1. Start of transaction. <Ti start>
2. Before and after values for writes <Ti, X, V1, V2>
    - V1 = old value
    - V2 = new value
5. Commit record. <Ti commit>

These log entries let the system redo committed transactions and undo uncommitted ones after a crash.

Approaches Using Logs

1) Deferred Database Modification

- Actual database updates are performed after commit.
- During execution, only log writes are made.

2) Immediate Database Modification

- Database may be updated before commit.

- Requires both undo (for uncommitted) and redo (for committed) operations.

Deferred Database Modification

All log records are written to stable storage, but actual database updates are delayed (deferred) until the transaction commits.

The system relies on the log to redo transactions after a crash.

How Deferred Modification Works

(Assume transactions execute serially)

1. Transaction Start

When a transaction starts: <Ti start> is written to the log.

2. Write Operation

For a write write(X): <Ti, X, V> is written to the log
→ V = new value of X
→ Old value is not needed (because we never undo in this scheme).

Important:
No actual update is applied to the database yet — all writes are deferred.

3. Partial Commit

When the transaction finishes execution: <Ti commit> is recorded in the log.

4. Actual Database Update

After commit, the log is read and all updates are applied (REDO phase):

- Apply new values of all X modified by Ti.
- Only committed transactions are redone.

Recovery Rules

During crash recovery:

A transaction Ti is redone ONLY IF:

- <Ti start> is present, and
- <Ti commit> is present

Redo(Ti) means:

Set all data items to their new values recorded in the log.

No UNDO is needed because the database was never updated before commit.

Crash Possibilities

Crashes may occur:

1. While the transaction was still executing
2. While deferred updates are being applied during recovery

Example: T0 and T1 (serial execution)

Transactions:

T0:

read(A)
A = A – 50
write(A)
read(B)
B = B + 50
write(B)

T1

read(C)
C = C – 100
write(C)

Logs at Different Times

Assume crashes at different points:

Case (a): Crash before any commit

Log contains:

<T0 start>
<T0, A, newA>
<T0, B, newB>
<T1 start>
<T1, C, newC>
📌 No commit records found → NO transaction is redone.

✔ No redo needed

Case (b): Crash after T0 commits, before T1 commits

Log contains:

<T0 start>
<T0, A, newA>
<T0, B, newB>
<T0 commit>

<T1 start>
<T1, C, newC>

📌 <T0 commit> present → redo(T0) is required
📌 <T1 commit> not present → do NOT redo(T1)

✔ Redo(T0) only

Case (c): Crash after both T0 and T1 commit

Log contains:

<T0 start>
<T0, A, newA>
<T0, B, newB>
<T0 commit>
<T1 start>
<T1, C, newC>
<T1 commit>
Both transactions committed → both must be redone

✔ Redo(T0)
✔ Redo(T1)

NOTE: REDO means applying the "after-values" from the log to the database, ensuring committed transactions are reflected in the DB.

Immediate Database Modification

In this scheme, the database is updated immediately after every write operation — even before the transaction commits. Because uncommitted changes may be written to the database, UNDO is required during recovery.

1. Logs written before the actual DB update (Write-Ahead Logging – WAL)

For every write(X), the log record must be written first, before the database item X is updated.

A log entry is of the form: <Ti, X, old_value, new_value>

Both values are needed because:

- REDO → uses new_value
- UNDO → uses old_value

2. DB may be updated before commit

As soon as log is safely stored on stable storage, the system may write: X = new_value even if the transaction has not committed.

3. Output order is flexible

Actual disk writes (output of updated blocks like BA, BB, BC) may happen:

- before commit
- after commit

- in any order

The only rule:

Log records for block B must be flushed BEFORE writing block B to disk.

🔷 Recovery of Immediate Modification

Because uncommitted data *may already be in DB*, recovery performs both UNDO and REDO.

✔ UNDO(Ti)

Restores affected data items to their old values. Used when the log has: <Ti start> but NO <Ti commit>

✔ REDO(Ti)

Reapplies new values of committed transactions.
Used when the log has: <Ti start> and <Ti commit>

✔ Order of Actions

Recovery always performs:

1️⃣ UNDO all uncommitted transactions

(go backwards through log)

2️⃣ REDO all committed transactions

(go forwards through log)

✔ Both UNDO and REDO must be idempotent

Meaning:

If executed multiple times, they must produce the same result as executing them once.

This ensures safety even if recovery is interrupted and restarted.

🔷 Example (T0 and T1)

Log + Database Writes Overview

<T0 start>
<T0, A, 1000, 950>
<T0, B, 2000, 2050>
A = 950
B = 2050
<T0 commit>
<T1 start>
<T1, C, 700, 600>
C = 600
(BB, BC output)

<T1 commit>
(BA output)

- BA, BB, BC represent the disk blocks containing A, B, C.

⭐ Recovery Cases

🟩 Case (a): Crash before T0 commits

Log ends at:

<T0 start>
<T0, A, 1000, 950>
<T0, B, 2000, 2050>
✔ Only T0 started → undo(T0)
Restore:

A = 1000
B = 2000

🟩 Case (b): Crash after T0 commit but before T1 commit

Log ends at:

... T0 commit
<T1 start>
<T1, C, 700, 600>
✔ T1 uncommitted → undo(T1)
Restore:

C = 700
✔ T0 committed → redo(T0)
Set:

A = 950
B = 2050

🟩 Case (c): Crash after both T0 and T1 commit

Log ends at:

<T0 commit>
<T1 commit>
✔ Both committed → redo both

A = 950
B = 2050
C = 600

Checkpointing in DBMS (Recovery)

A checkpoint is a recovery mechanism used to limit the amount of log that must be scanned after a system crash. It marks a time when the database is in a consistent state, and all previous updates have been safely written to disk.

Why Checkpoints Are Needed

- Recovery becomes slow if the entire log has to be read.
- Log files keep growing very large over time.
- Some transactions may be unnecessarily redone even though their updates are already stored on disk.

What Happens During a Checkpoint

1. All log records in memory are flushed to stable storage.
2. All modified (dirty) buffer blocks are written to disk.
3. A special log record is written:

   <checkpoint L>

4.

   where L is the list of transactions active at the time of checkpoint.

Everything before this point is guaranteed to be stable.

Recovery Using Checkpoint

Recovery uses two lists:

- Redo-list → transactions that committed
- Undo-list → active or incomplete transactions

Steps:

1. Scan the log backwards until the latest <checkpoint L> is found.
2. While scanning backward:
   - If <Ti commit> is found → add Ti to redo-list
   - If <Ti start> is found and Ti is not in redo-list → add Ti to undo-list
3. For each Ti in L (active at checkpoint):
   - If Ti is not in redo-list → add Ti to undo-list
4. Perform:
   - UNDO all transactions in undo-list
   - REDO all transactions in redo-list

Meaning of Undo & Redo

- Undo(Ti): Restore old values for uncommitted transactions.
- Redo(Ti): Reapply new values for committed transactions.

Benefits of Checkpointing

- Reduces recovery time drastically.
- Prevents reading the full log.
- Keeps log file size manageable.
- Ensures efficient crash recovery in immediate modification systems.

Shadow Paging

Shadow paging is an alternative to log-based recovery. It is mainly useful when transactions execute serially (one at a time).

The main idea is to maintain two page tables:

1. Shadow Page Table
    - Stored in non-volatile storage
    - Represents the database *before* the transaction starts
    - Never modified during the transaction
2. Current Page Table
    - Used during transaction execution
    - Contains updated page mappings
    - Can be modified

Initially, both page tables are identical.

How Shadow Paging Works

1. During page update

When a page is written for the first time:

- Make a copy of that page to a free disk page
- Update the current page table to point to this copy
- Perform the update on the copied page

➡ The original page (pointed by shadow page table) remains unchanged.

2. Commit Procedure

To commit a transaction:

1. Flush all modified pages to disk
2. Write the current page table to disk
3. Make it the new shadow page table
    - This is done by simply updating a pointer stored at a fixed disk location
    - Once this pointer is updated, the transaction is considered committed

Key point:

After commit, no recovery is needed—the shadow page table always reflects a consistent database.

3. Failures

If a crash occurs before commit:

- Shadow page table is unchanged
- Database is immediately restored to the pre-transaction state
- No undo or redo required

4. Garbage Collection

Pages that are no longer referenced by the current/shadow page table must be freed.

This is required because new versions create old unused copies.

Advantages of Shadow Paging

- ✔ No log records needed → reduced overhead
- ✔ Very fast recovery (no redo/undo).
- ✔ Consistent state always available through shadow table.

Disadvantages of Shadow Paging

- ✘ Copying the full page table is expensive
- ✘ High commit overhead (all modified pages + page table must be flushed)
- ✘ Page fragmentation (related pages may get scattered on disk)
- ✘ Requires garbage collection to remove old versions
- ✘ Not suitable for concurrent transactions

Deadlocks

A deadlock occurs when two or more transactions are waiting for each other to release locks, and none can proceed.

Deadlock Detection

1. Wait-For Graph

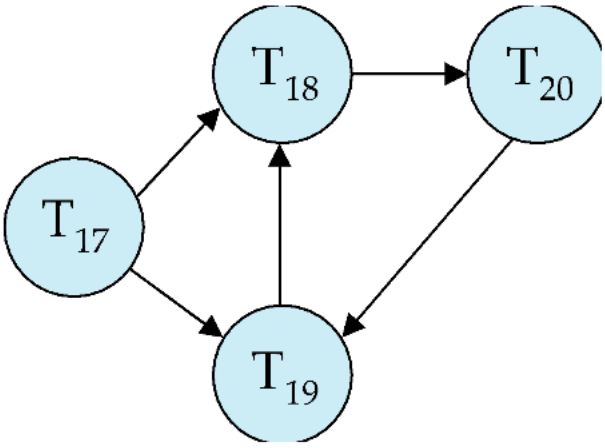
A directed graph G = (V, E)

- V = Transactions
- E = Ti → Tj means Ti is waiting for Tj

Deadlock = cycle in the wait-for graph

Process

- System regularly checks the graph
- If cycle exists → deadlock detected



Wait-for graph with a cycle

Deadlock Recovery

After detecting a deadlock:

1. Choose a victim

Rollback the transaction with minimum cost:

- Work done
- Number of data items used
- Number of rollbacks already suffered (to avoid starvation)

2. Rollback Strategy

- Total rollback: Abort completely and restart
- Partial rollback: Roll back only enough to break deadlock

Partial rollback reduces wasted work.

Deadlock Prevention

Goal: Ensure system never enters deadlock.

1. Pre-declaration

Transaction must request all locks at start. (Not practical, reduces concurrency)

2. Lock Ordering

All data items have a predefined order; transactions must acquire locks in that order only.
Prevents cycles → prevents deadlocks.

Timestamp-Based Prevention

1. Wait–Die (Non-preemptive)

- Older transaction may wait
- Younger transaction dies (is rolled back)

Rule:
If TS(Ti) < TS(Tj) → Ti (older) waits
Else → Ti (younger) dies

Example timestamps:
T22=5 (oldest), T23=10, T24=15 (youngest)

- T22 waits for T23
- T24 dies if it requests item held by T23

Younger ones keep dying → may suffer multiple rollbacks.

2. Wound–Wait (Preemptive)

- Older transaction wounds younger → younger rolls back
- Younger transaction waits for older

Rule:
If TS(Ti) < TS(Tj) → Ti wounds Tj
Else → Ti waits

Fewer rollbacks compared to wait-die.

Examples:

- T22 requests item held by T23 → T23 is rolled back
- T24 requests item held by T23 → T24 waits

Timeout-Based Scheme

- A transaction waits for only a fixed time
- If timeout expires → it rolls back and restarts
- No deadlocks possible
- But may cause starvation
- Difficult to choose optimal timeout period

UNIT V

Relational Database Management System (RDBMS)

- An RDBMS organizes data into structured tables with rows and columns, based on the relational model.
- Data is stored in relations (tables), with unique rows (records) and columns (attributes).

Key Features:

- Structured Data Storage: Uses tables to store data systematically.
- ACID Compliance: Ensures reliable transactions with Atomicity, Consistency, Isolation, and Durability.
- Keys Usage: Primary keys uniquely identify records, foreign keys establish relationships.
- SQL Support: Utilizes Structured Query Language for data manipulation and querying.

Ideal Use:

- Best suited for structured data requiring strong consistency and integrity.

Issues with RDBMS:

- Scalability: Vertical scaling is costly and limited as data grows.
- Handling Unstructured Data: Difficulty managing semi-structured or unstructured data like JSON.
- Rigid Schema: Schema changes demand downtime and complex restructuring.
- Performance Bottlenecks: JOIN operations slow down with large datasets.
- Distributed Architecture Limitations: Not originally designed for distributed or cloud environments.

NoSQL Databases

- NoSQL databases are non-relational systems designed to handle diverse data models and massive data volumes.

Key Features:

- Flexible Schema: Allows dynamic schema design, adapting to evolving data structures.
- Horizontal Scalability: Can scale out across multiple servers or nodes.
- Support for Various Data Models: Includes key-value, document, column-family, and graph databases.
- Suitable for Unstructured Data: Efficient in handling big data and web applications.

Need for NoSQL:

- Big data: massive amount of data generated by model application require more efficient processing
  Flexibility: NoSQL provides a flexible schema designed to table for evolving application
- Supports high availability and scalability through replication and partitioning.

NoSQL Database Types:

Type	Data Model Description	Use Cases	Examples
Key-Value	Data stored as unique keys mapped to values	Caching, session management	Redis, DynamoDB, Riak
Column-Based	Stores data in flexible column-families, compressible	Analytical apps, large-scale storage	Apache Cassandra, HBase

Document-Based	Stores semi-structured data in JSON, BSON, XML formats	CMS, e-commerce, real-time analytics	MongoDB, Couchbase
Graph-Based	Nodes and edges to represent entities and relationships	Social networks, fraud detection	Neo4j, Amazon Neptune
Multi-model	Combines features of multiple NoSQL types	*Not specified*	*Not specified*

Advantages and Limitations of NoSQL

Advantages:
- Flexibility: Dynamic schema adjustments without downtime.
- Scalability: Horizontal scaling supports large data volumes and traffic.
- High Availability: Replication and partitioning enhance uptime.
- Performance: Optimized for specific queries like key-value lookups and document retrieval.

Limitations:
- Lack of standardized protocols; NoSQL databases vary significantly.
- Absence of stored procedures in some systems (e.g., MongoDB).
- Limited availability of GUI tools for database management.
- Scarcity of experienced NoSQL developers due to its relative novelty.

Data Types in Databases

Data Type	Description	Examples
Structured Data	Organized in rows and columns, easily managed	SQL Databases (Employee table)
Semi-Structured Data	Contains tags or markers to separate elements	JSON, XML files, emails with metadata
Unstructured Data	No predefined format, difficult to manage	Text files, Social media posts

Use Cases for NoSQL

NoSQL excels in scenarios demanding flexibility, scalability, and high availability:
- Big Data applications managing massive distributed datasets.
- Real-time analytics requiring fast data processing.
- Social networks and graph data emphasizing complex relationships.
- Content Management Systems requiring flexible handling of unstructured content.

BASE Properties in Distributed Systems

BASE is an alternative to ACID, designed for distributed environments prioritizing availability and scalability over strict consistency.
BASE Acronym:

Term	Description
Basically Available	System guarantees availability of data and services
Soft State	Data state can change over time, may be temporarily inconsistent
Eventual Consistency	System will eventually reach a consistent state across all nodes

Detailed Explanation:

- Updates are first applied to a single replica.
- Propagation to other replicas occurs asynchronously, causing temporary inconsistency.
- The system ensures all replicas eventually reflect the same data.
- Prioritizes availability over immediate consistency, suitable for applications like social media and online shopping.

Implementing BASE in Distributed Systems

ACID	BASE
ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee the integrity and consistency of data in a traditional database.	The BASE properties are a more relaxed version of ACID properties by trading some consistency and providing more availability.
The primary difference between the two is that ACID requires immediate consistency,	BASE only requires eventual consistency.
ACID is better suited to traditional transactional databases.	The BASE is more suitable for use in large-scale, highly-available systems.

Key techniques include:

- Data Replication: Copies data across multiple nodes for fault tolerance.
- Conflict Resolution: Mechanisms to handle conflicting updates.
- Eventual Consistency: Partial consistency accepted temporarily, with guaranteed convergence.]

CAP Theorem in Distributed Systems

Statement:

- A distributed system can provide only two of the following three guarantees simultaneously:
  - Consistency ©: All nodes see the same data at the same time. Every read receives the most recent write
  - Availability (A): Every request receives a response, regardless of node failures.
  - Partition Tolerance §: System continues to operate despite network partitions.

The Trade-off:

- Systems must choose which two properties to prioritize when partition occurs.
- No system can guarantee all three simultaneously.

CAP Theorem Database Classifications

System Type	Consistency	Availability	Partition Tolerance	Typical Use Cases	Examples
CA	Yes	Yes	No	Apps with strong network, requiring matched data	MySQL, PostgreSQL
CP	Yes	No	Yes	Apps needing strong consistency	MongoDB, Google's Chubby, Apache ZooKeeper
AP	No	Yes	Yes	Apps requiring high uptime, tolerating stale data	Amazon DynamoDB, Apache Cassandra, Riak

Notes:

- CA databases lack fault tolerance in distributed environments.
- CP databases sacrifice availability during partitions to maintain consistency.
- AP databases prioritize availability, accepting eventual consistency.

Detailed CAP Theorem Concepts

Concept	Description
Strong Consistency	All nodes always reflect the latest write simultaneously; requires coordination.
Eventual Consistency	Nodes may temporarily differ but converge over time; favors availability and partition tolerance.
Availability	System operational at all times; responses guaranteed even if data is stale.
Partition Tolerance	System operates correctly despite communication breakdowns between nodes.

Practical Use Cases for BASE and CAP Systems

- Social Media Platforms: Prioritize availability (BASE, AP) over strict consistency.
- E-commerce: Often tolerate eventual consistency for better availability and responsiveness.
- Gaming: Requires high responsiveness; temporary inconsistencies acceptable.

- Banking Systems: Opt for CP systems, prioritizing consistency over availability.

Data warehouse

A data warehouse is a centralized repository designed to store large volumes of data collected from multiple heterogeneous sources under a unified schema to support management decision-making. Unlike operational databases, which focus on current transactional data, data warehouses emphasize subject-oriented, integrated, time-variant, and nonvolatile data storage, facilitating long-term historical analysis and strategic business insights. The process of data warehousing involves extracting data from diverse sources, cleaning and transforming it, and loading it into a warehouse designed for efficient querying and analysis.

Characteristics of a Data Warehouse

Defined by **Bill Inmon**, the father of data warehousing:

Characteristic	Description
—————————	—————————
Subject-Oriented	Organized around key business areas (e.g., sales, finance).
Integrated	Combines data from multiple heterogeneous sources.
Time-Variant	Contains historical data over long time periods.
Non-Volatile	Data is read-only; not frequently updated or deleted.

Data warehouses employ multi-tiered architectures with backend servers managing data storage and OLAP (Online Analytical Processing) engines providing multidimensional data analysis capabilities. They support different data models, including enterprise warehouses (comprehensive organizational data), data marts (department-specific data subsets), and virtual warehouses (views over operational data). Schemas such as star, snowflake, and fact constellation (galaxy) organize data into fact and dimension tables, enabling flexible, high-performance multidimensional queries.

Data Warehouse Architecture

A typical DW architecture has **three layers**:

1. **Data Source Layer**
- Contains operational databases, logs, CRM, ERP systems, etc.
- Data extracted through **ETL (Extract, Transform, Load)**.

2. **Data Storage Layer**
- Central warehouse storing cleaned, transformed data.
- Organized using **schemas** (Star, Snowflake, Galaxy).

3. **Data Presentation Layer**
- Provides user access via OLAP tools, dashboards, and reporting systems.

ETL Process
Extract → Transform → Load** is the pipeline that populates the data warehouse.

1. **Extract:** Pull data from multiple sources.
2. **Transform:** Clean, format, and apply business rules.
3. **Load:** Store into the warehouse for querying and analysis.

OLTP vs OLAP

Feature	OLTP (Operational)	OLAP (Analytical)
Purpose	Day-to-day operations	Decision support

Data	Current, short-term	Historical, long-term
Users	Clerks, developers	Analysts, executives
Queries	Simple, frequent	Complex, infrequent
Updates	Continuous	Periodic batch loads
Example	Banking system	Business intelligence dashboard

Data Warehouse Schemas – Notes

1. Multidimensional Data

- Used for analysis, not transactions
- Two types of attributes:
  ✔ Dimension → describes data (item, date, store)
  ✔ Measure → numbers (sales, quantity)

2. Fact Table

- Stores measure values
- Very large
- Contains foreign keys to all dimensions
- Example: Sales fact → item_id, store_id, date, customer_id, number, price

3. Dimension Tables

- Store descriptive details
- Fact table refers to them using IDs
- Examples:
  ✔ store (city, state, country)
  ✔ item (name, category)
  ✔ customer (name, address)
  ✔ date (month, quarter, year)

4. Star Schema

- 1 fact table in center
- Multiple dimension tables around it
- Looks like a star
- Simple, fast for queries

5. Snowflake Schema

- Normalized star schema
- Dimensions split into sub-dimensions
- Example: item → manufacturer table

- Less redundancy, more joins

6. Galaxy Schema (Fact Constellation)

- Multiple fact tables
- Shared dimensions
- Looks like many stars together
- Used in large enterprise warehouses

One-Line Memory Hooks

- Fact table → numbers
- Dimension table → details
- Star → simple
- Snowflake → normalized
- Galaxy → many fact tables

OLAP Operations (Very Important!)

1. Roll-Up (Aggregation)

- Go from detail → summary
- Example: day → month → quarter → year

2. Drill-Down (De-aggregation)

- Go from summary → detail
- Year → quarter → month → week → day

3. Slice

- Fix one dimension
- Example: Sales for 2024 only

4. Dice

- Select a sub-cube using multiple conditions
- Example: Region = Asia AND Product = Laptops AND Year = 2024

5. Pivot (Rotate)

- Change the axis/view of cube
- Rows ↔ Columns

6. Drill-Across

- Access related data across different fact tables

7. Drill-Through

- Go from cube summary → actual transaction records
- Example: From total sales → actual invoice list

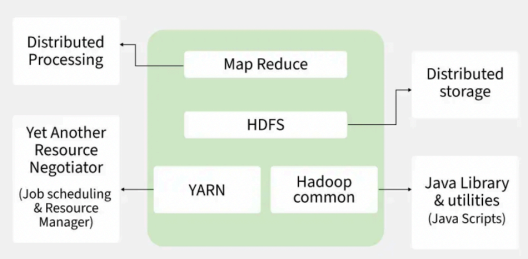
Hadoop Architecture

Hadoop is an open-source Java framework that stores and processes massive data using clusters of inexpensive (commodity) hardware

- Based on Google's MapReduce programming model, it enables distributed, parallel processing
- Big companies like Facebook, Yahoo, Netflix and eBay use Hadoop to handle large-scale data efficiently.

Components of Hadoop Architecture

- MapReduce
- HDFS (Hadoop Distributed File System)
- YARN (Yet Another Resource Negotiator)
- Common Utilities or Hadoop Common



Hadoop Distributed File System

HDFS (Hadoop Distributed File System) is Hadoop's primary storage system, built for high-throughput access to large datasets. It runs on inexpensive commodity hardware and stores data in large blocks to optimize performance.

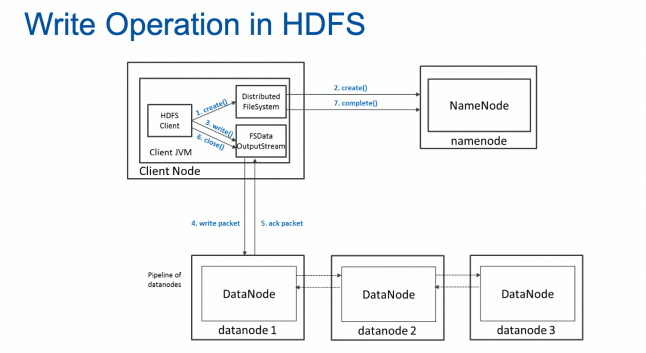
- HDFS ensures fault tolerance and high availability across the cluster.

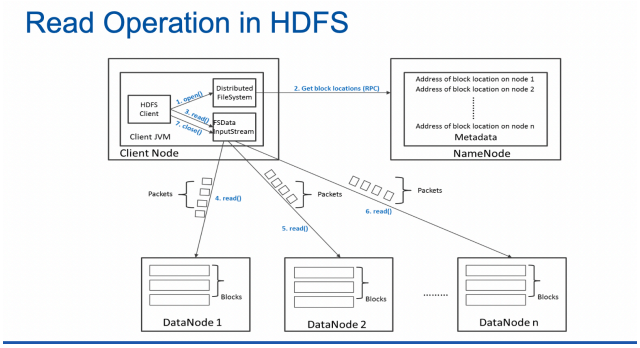
HDFS Architecture Components:

- NameNode (Master Node): The master node in HDFS that stores metadata (not actual data), manages file operations and directs clients to nearest DataNode for efficient access.
- DataNode (Slave Node): Stores actual data blocks, serves read/write requests and reports to NameNode. Supports replication (default 3) for fault tolerance and scales storage and performance with more nodes.
- File Block In HDFS: In HDFS data is always stored in the form of blocks. By default, each block is 128MB in size, although this value can be manually configured depending on the use case (commonly increased to 256MB or more in modern systems).

  Block placement policy
  - First replica is placed on the local node
  - Second replica is placed in a different rack
  - Third replica is placed in the same rack as the second replica

Suppose you upload a file of 400MB to HDFS. Hadoop will divide this file into blocks as follows:
128MB + 128MB + 128MB + 16MB = 400MB
This creates four blocks three of 128MB and one of 16MB. Hadoop splits files purely by size, not content, so a single record can span across two blocks.



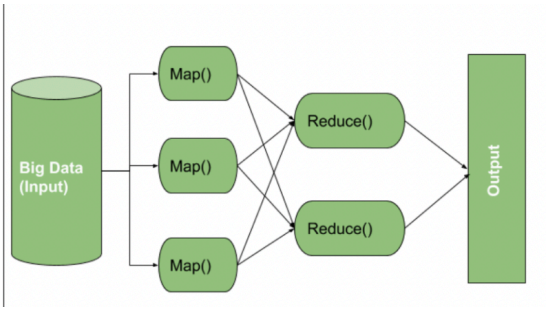


MapReduce

- MapReduce is a data processing model in Hadoop that runs on YARN. It enables fast, distributed and parallel processing by dividing tasks into two phases Map and Reduce making it efficient for handling large-scale data.
- **MapReduce Workflow**: workflow begins when input data is split into key-value pairs by Map() function. These are then grouped by key and processed by Reduce() function for tasks like sorting or aggregation. The final output is written to HDFS.

Map Task Components:

- RecordReader: reads input data and converts it into key-value pairs, with keys as location info and values as actual data.
- Mapper: processes each pair and outputs zero or more new key-value pairs.
- Combiner (optional): acts as a mini-reducer to group Mapper output and reduce data transfer before shuffling.
- Partitioner: assigns key-value pairs to Reducers using key.hashCode() % numberOfReducers.



Reduce Task Components:

- Shuffle and Sort: transfers intermediate key-value pairs from Mappers to Reducers and sorts them by key. Shuffling begins as soon as some Mappers finish.
- Reducer: processes grouped key-value pairs, performing tasks like aggregation or filtering based on logic.
- OutputFormat: writes final results to HDFS using a RecordWriter, typically storing each record as a key-value line.

1. Comparison with additional file system

Traditional file systems use small blocks (e.g., 4KB), while HDFS uses large blocks (128MB or more).

- Larger blocks in HDFS reduce metadata and I/O overhead, improving scalability and efficiency for big data processing.

2. Replication in HDFS

HDFS replication ensures data availability and fault tolerance by storing multiple copies of each block.

- Default Replication Factor: 3 (configurable in hdfs-site.xml)
- If a file is split into 4 blocks, with a replication factor of 3: 4 blocks × 3 replicas = 12 total blocks
- Designed for commodity hardware, where failures are common—replication prevents data loss. While it increases storage usage, reliability is prioritized over space efficiency.

3. Rack awareness.

- A rack is a group of machines (typically 30–40) in a Hadoop cluster. Large clusters have many racks. Rack Awareness helps NameNode to:
- Choose the nearest DataNode for faster read/write operations.
- Reduce network traffic by minimizing inter-rack data transfer.
- This improves overall performance and efficiency in data access.

YARN (Yet Another Resource Negotiator)

- YARN is resource management layer in Hadoop ecosystem. It allows multiple data processing engines like MapReduce, Spark and others to run and share cluster resources efficiently.
- It handles two core responsibilities:
- Job Scheduling: Splits large tasks into smaller jobs, assigns them to nodes and manages priorities, dependencies and execution.
- Resource Management: Allocates and monitors cluster resources (CPU, memory, etc.) needed for job execution.

Components of Yarn:

- ResourceManager: Master node that manages global resource allocation.
- NodeManager: Slave node that monitors resources on individual nodes.
- ApplicationMaster: Manages lifecycle of each individual application/job.

Key Features of YARN:

- Multi-Tenancy: Supports multiple users and applications.
- Scalability: Efficiently scales to handle thousands of nodes and jobs.
- Better Cluster Utilization: Maximizes resource usage across the cluster.
- Compatibility: Works with MapReduce and other processing models like Spark.

Hadoop Common (Common Utilities)

- Hadoop Common, also known as Common Utilities, includes core Java libraries and scripts required by all components in a Hadoop ecosystem such as HDFS, YARN and MapReduce.
- These libraries offer core functionalities such as:
- File system and I/O operations
- Configuration and logging
- Security and authentication
- Network communication
- Hadoop Common provides shared libraries and utilities that help all Hadoop components work together. It handles hardware failures automatically and includes tools like Hadoop Archive, native library support and RPC mechanisms.

UNIT VI

Centralized Database Systems

Centralized systems are the simplest form of database architectures, running on a single computer system without interaction with other systems.

- General-purpose centralized systems consist of one to a few CPUs connected through a common bus to shared memory.
- They vary between single-user systems (e.g., personal computers with one CPU and limited disks) and multi-user systems (servers with multiple CPUs, disks, and memory serving numerous users via terminals).
- Single-user systems often lack critical features such as concurrency control and sophisticated crash recovery, relying on primitive backup methods.
- Multi-user systems support full transactional features, enabling multiple concurrent updates with mechanisms to handle conflicts and data integrity.
- Modern centralized machines feature coarse-granularity parallelism, where multiple processors share memory but do not typically parallelize individual queries; instead, multiple queries run concurrently on separate processors.
- Multitasking and time-sharing allow single-processor machines to handle multiple processes, making them functionally similar to coarse-grained parallel machines.
- In contrast, fine-grained parallelism involves many processors working simultaneously on parts of a single query to speed up processing.

Client-Server Database Systems

Client-server systems distribute database functionality between clients and servers, improving modularity and scalability.

- The server (back-end) manages core database functions like query evaluation, access methods, concurrency control, and recovery.
- The client (front-end) handles user interaction via tools such as forms and graphical interfaces.
- Communication between front-end and back-end occurs through SQL or APIs like ODBC (Open Database Connectivity) for C and JDBC for Java.

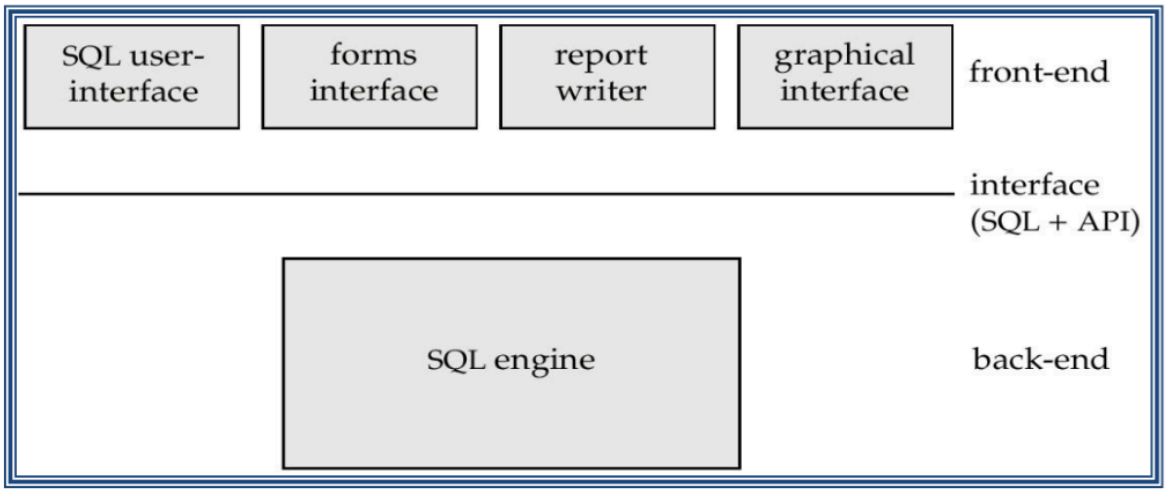
Transaction Servers

- Transaction servers execute client-submitted SQL requests and return results, often supporting transactional remote procedure calls (RPC), allowing multiple RPCs to be grouped into a single transaction, ensuring atomicity.
- Typical server processes include:
  - Server processes handling query execution and communication.
  - Lock manager process for managing locks and deadlock detection.
  - Database writer process for writing modified data back to disk.
  - Log writer process for persisting transaction logs.
  - Checkpoint process for periodic system state saving.
  - Process monitor for handling process failures and recovery.
- Shared memory is used for common data structures like buffer pools, lock tables, and log buffers, protected by operating system semaphores to ensure mutual exclusion.
- Lock management involves monitoring requests, granting or waiting for locks, and deadlock detection, critical for maintaining consistency across concurrent transactions.

Data Servers and Data Shipping Mechanisms

In environments with high-speed local-area networks, **data-server systems** ship data to clients for local processing, reducing server load.

- Two primary methods for data transmission are:
  - **Page shipping**: Transmits a page containing multiple data items, serving as a form of **prefetching** that anticipates future requests.
  - **Item shipping**: Sends individual tuples or objects but incurs higher communication overhead.



- Page shipping locks entire pages, potentially blocking other clients unnecessarily. To mitigate this, **lock de-escalation** allows clients to return unneeded locks to the server for redistribution.
- **Data caching** at clients allows reuse of data across transactions; however, clients must verify data freshness and acquire locks to maintain consistency.
- **Lock caching** permits clients to reuse locks locally when data access patterns are partitioned, reducing communication with the server. The server tracks cached locks and invalidates conflicting ones when necessary.
- Challenges arise when machines fail, complicating lock consistency and requiring robust protocols

Parallel Database Systems

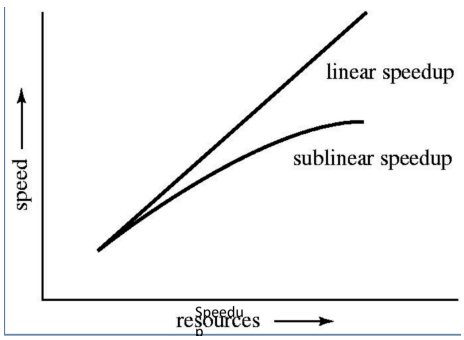
Parallel systems use multiple processors and disks connected by fast networks to improve throughput and response time.

- Types of parallelism:
  - **Coarse-grain parallelism**: Few powerful processors.
  - **Fine-grain (massively parallel)**: Thousands of small processors.
- Performance metrics:
  - **Throughput**: Number of tasks completed per time unit.
  - **Response time**: Time to complete a single task.
- Key concepts:
  - **Speedup**: Reduction in execution time of a fixed-size task by increasing parallelism, ideally linear with the number of processors.

Consider the execution time of a task on the larger machine is TL and the execution time of a same task on the smaller machine is TS.

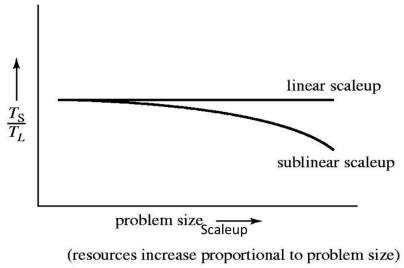
•The speed up is measured due to parallelism as TS/ TL.

•The parallel system is said to demonstrate linear speedup if the speedup is N. Where N is the number of resources used in larger systems.If the speedup is less than N,the system is said to demonstrate sublinear speedup.



Scaleup: Ability to handle larger tasks proportionally with increased resources, maintaining constant execution time.
Suppose Q be a task and Qn is the task i.e. N times bigger than Q.

- Execution time of Q on machine is TS while execution time of Qn on parallel machine is TL, then the scaleup is calculated as TS/TL
- The parallel system is said to demonstrate linear scaleup on task Q if TL=TS.
- If TL>TS the system is said to demonstrate sublinear scaleup.



- Limitations to ideal speedup and scaleup include:
  - Startup costs: Overhead of initiating many processes.
  - Interference: Contention for shared resources leading to waiting.
  - Skew: Unequal division of tasks causing delays due to longer processing on some processors.

Interconnection Network Architectures

•Bus
. System components send data on and receive data from a single communication bus;

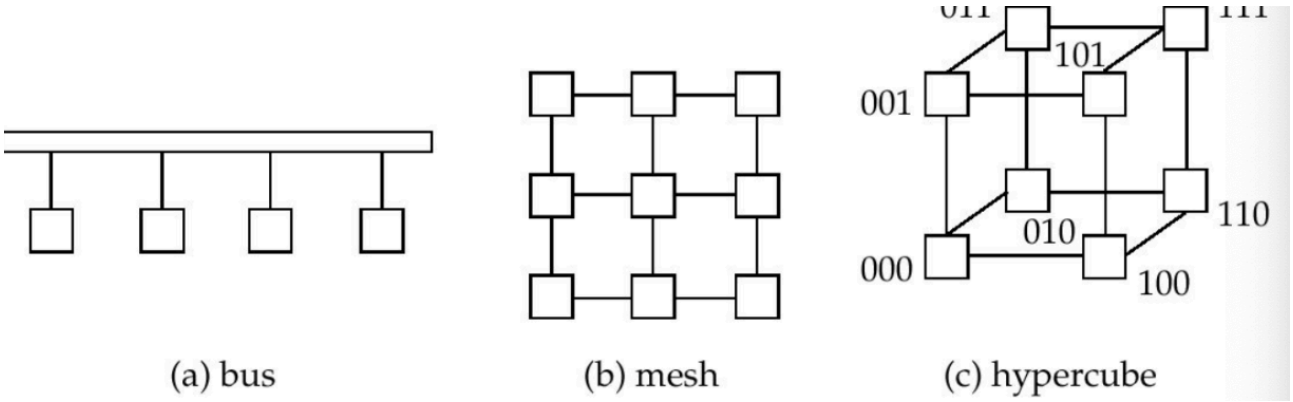
–Does not scale well with increasing parallelism.

•Mesh
. Components are arranged as nodes in a grid, and each component is connected to all adjacent components

–Communication links grow with growing number of components, and so scales better.

•Hypercube
. Components are numbered in binary; components are connected to one another if their binary representations differ in exactly one bit.

–n components are connected to log(n) other components and can reach each other via at most log(n) links; reduces communication delays.



Parallel architectures:

Shared Memory

•In a shared-memory architecture, the processors and disks have access to a common memory, typically via a bus or through an interconnection network.
• The benefit of shared memory is extremely efficient communication between processors—data in shared memory can be accessed by any processor .
•A processor can send messages to other processors much faster by using memory writes than by sending a message through a communication mechanism.

•The downside of shared-memory machines is that the architecture is not scalable beyond 32 or 64 processors because the bus or the interconnection network becomes a bottleneck (since it is shared by all processors).
•Adding more processors does not help after a point, since the processors will spend most of their time waiting for their turn on the bus to access memory.

Shared-Disk

Architecture Setup

- All processors can access all disks through an interconnection network.
- Each processor has its own private memory (no shared RAM).

Advantages

1. No Memory Bus Bottleneck

- Each processor uses its own memory →
  No single memory bus that everyone fights for (unlike shared-memory systems).

2. Fault Tolerance

- Database is stored on disks accessible to all processors.
- If one processor or its memory fails → another processor can take over because the disks are still accessible.

Main Problem

Scalability Issues

- The interconnection network between processors and disks becomes a new bottleneck.
- Especially problematic when the database performs many disk accesses.

Shared-Nothing Architecture

- Each node has its own:
  - Processor
  - Memory
  - Disk(s)
- Nodes talk to each other through a high-speed network.

How It Works

- Each node is the server for its own disk data (data ownership).

- Local data access = fast, because it's done on local disk + local memory.
- Network is used only for:
    - Queries that need data from other nodes
    - Access to nonlocal disks
    - Sending final results

Advantages

1. Highly Scalable

- No shared memory.
- No shared disk.
- No central bottleneck → can scale to many processors easily.

2. Fast Local I/O

- Most disk operations are handled locally → reduces network load.

Drawbacks

1. Expensive Communication

- When data is needed from another node → higher cost.
- Requires coordination software at both sending and receiving nodes.

2. Slow Non-Local Disk Access

- Accessing data stored on another node's disk is slower than in shared-memory or shared-disk architectures.

Hierarchical Architecture

- A hybrid architecture that combines:
    - Shared-Memory
    - Shared-Disk
    - Shared-Nothing

How It Works

Top Level → Shared-Nothing

- Nodes do not share memory or disks.
- Connected by an interconnection network.

Inside Each Node

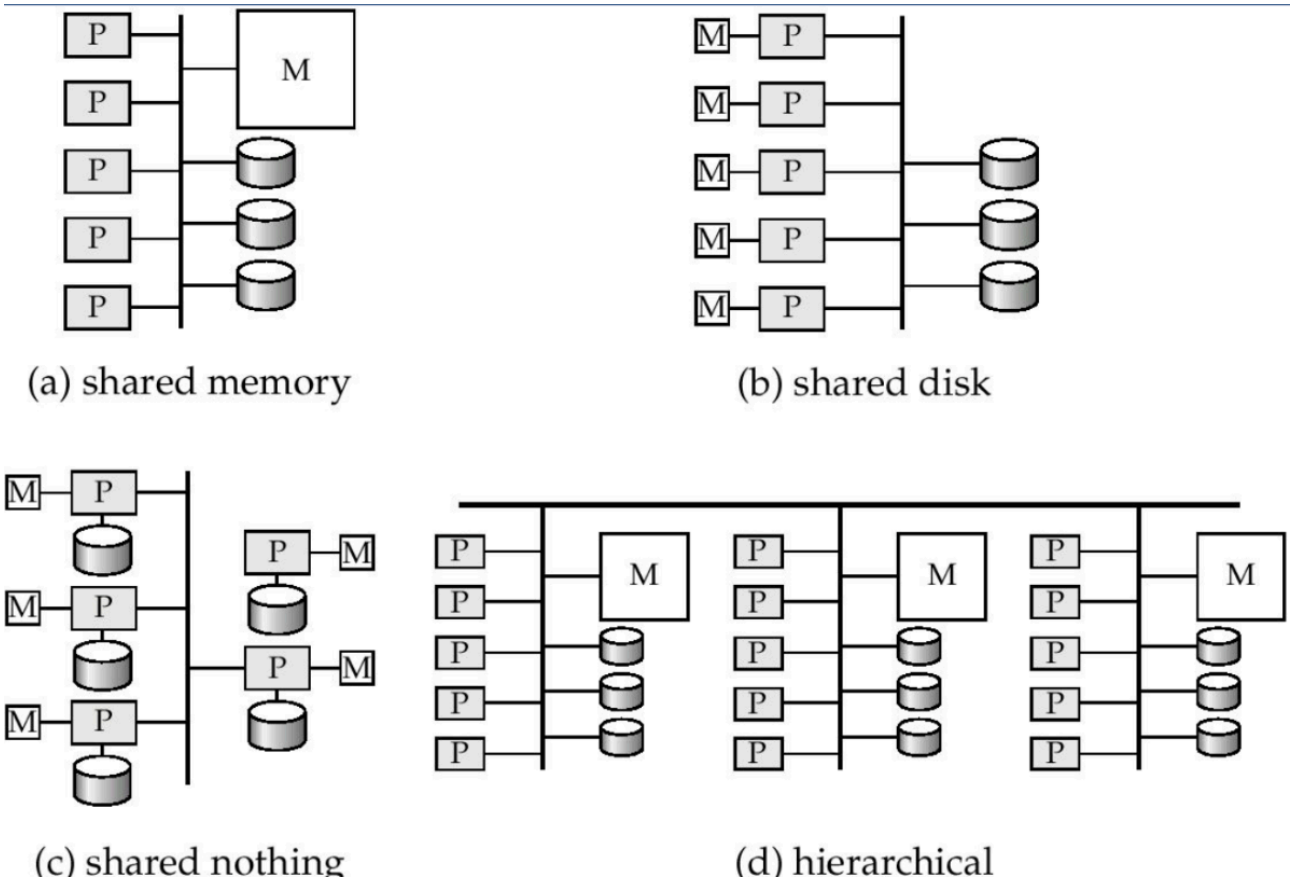
- Each node can be:
  - Shared-Memory system (multiple processors sharing memory), or
  - Shared-Disk system, where disks are shared and each shared-disk subsystem may have its own shared-memory base.

Overall Structure

- System forms a hierarchy:
  - Bottom layer → Shared-Memory (small processor groups)
  - Middle layer → Shared-Disk (optional)
  - Top layer → Shared-Nothing (clusters of nodes)

Advantages

- Combines the strengths of all architectures:
  - Speed of shared-memory within nodes
  - Fault tolerance of shared-disk
  - Scalability of shared-nothing



Data Distribution: Replication and Fragmentation

Two fundamental approaches for storing data in distributed databases are replication and fragmentation.

- Replication involves storing identical copies of a relation at multiple sites.
  - Advantages:
    - Availability: Failure at one site does not prevent access.
    - Increased parallelism: Multiple sites can serve read queries independently.
  - Disadvantages:
    - Update overhead: Changes must propagate to all replicas to maintain consistency, increasing communication and processing costs.
- Fragmentation partitions a relation into parts stored at different sites.
  - Types:
    - Horizontal fragmentation: Divides tuples based on predicates, e.g., accounts by branch.
    - Vertical fragmentation: Splits relation schema into subsets of attributes, ensuring a common key (or tuple-id) exists in all fragments to enable reconstruction.
  - Fragmentation supports:
    - Placement of data closer to where it is accessed.
    - Parallel processing on fragments.
    - Combining horizontal and vertical fragmentation provides flexible data distribution.
- Data transparency concepts ensure users do not need to be aware of fragmentation or replication details:
  - Fragmentation transparency: Users access relations without knowing their internal partitions.
  - Replication transparency: Users see a single logical copy regardless of physical replicas.
  - Location transparency: Users need not know the physical site of data.

Distributed Database System

- Database stored across multiple computers (sites/nodes).
- Sites communicate via networks (LAN, WAN, telephone lines).
- No shared memory or shared disk between sites.
- Sites can be small (workstation) or large (mainframe).

2. Difference: Parallel DB (Shared-Nothing) vs Distributed DB

- Distributed DBs are usually:
  - Geographically separated
  - Independently administered
  - Slower interconnection
- Supports:

    - Local transactions → Access only local site data
    - Global transactions → Access data from multiple sites

3. Why Use Distributed Databases?

✔ Sharing of Data

- Users at one site can access data stored at another site.
- Example: A bank branch can access accounts in another branch.

✔ Autonomy

- Each site controls its own local data.
- Global DBA + Local DBAs → controlled independence.

✔ Availability

- If one site fails, others continue working.
- Replication allows reading data from other sites.
- Requires failure detection + smooth reintegration on recovery.

5. Implementation Issues

✔ Atomicity (2-Phase Commit Protocol - 2PC)

- Ensures transaction either commits everywhere or aborts everywhere.

✔ Concurrency Control

- Transactions across sites need coordination.
- Deadlocks across multiple sites → need global deadlock detection.

✔ Disadvantages

- Complex software, expensive development
- More bugs due to parallel operations
- Higher overhead (messages + coordination)

6. Types of Distributed Databases

i) Homogeneous

- All sites use same DBMS,
- Share same global schema,
- Fully aware of each other.

ii) Heterogeneous

- Sites use different DBMS / schemas
- Limited cooperation
- Mix of relational, object, hierarchical models, etc.

⭐ 7. Distributed Data Storage Methods

A) Data Replication

- Relation stored at multiple sites
- Full replication → copy at every site

Advantages

1. High Availability
2. Increased parallelism (multiple sites handle read queries)

Disadvantages

- High update overhead → must update all replicas
- Risk of inconsistency if not updated everywhere

B) Data Fragmentation

Relation r divided into pieces r_1, r_2, …, rn such that r can be reconstructed.

1. Horizontal Fragmentation

- Fragment by rows (tuples)
- Each fragment = selection:
  - r1 = σbranch = “Hillside”(r)
  - r2 = σbranch = “Valleyview”(r)

Used when:

- Tuples are accessed mostly at the same site
- Minimizes data transfer

Reconstruction:
r = r1 ∪ r2 ∪ … ∪ rn

2. Vertical Fragmentation

- Fragment by columns (attributes)

- Each fragment = projection:
    - Must include a superkey / tuple-id to ensure lossless join

Example:

- employee-privateinfo (employee-id, salary)
- employee-public-info (employee-id, name, designation)

Reconstruction:
r = r1 ⋈ r2 ⋈ … ⋈ rn

Advantages of Fragmentation

Horizontal:

- Parallel processing
- Stores tuples where they are most used

Vertical:

- Stores attributes where frequently accessed
- Efficient joins (using tuple-id)
- Parallel processing

Mixed:

- Can fragment → sub-fragment → replicate → re-fragment
- Very flexible

⭐ 8. Data Transparency

Users need not know how data is stored or distributed.

Types:

- Fragmentation Transparency → User doesn't know data is fragmented
- Replication Transparency → User sees one logical copy
- Location Transparency → User doesn't know where data physically lives

DISTRIBUTED TRANSACTIONS

- Operations that access data on multiple sites in a distributed database.
- Must preserve ACID across all sites.
- Local transaction → only one site.

- Global transaction → multiple sites involved.
- Harder because:
  - Many sites participate
  - Failure of a site or communication link can cause inconsistency.

🔷 Transaction Components at Each Site

1. Transaction Manager (TM)

- Executes local + subtransactions.
- Maintains log for recovery.
- Ensures concurrency control locally.
- Ensures ACID for its site.

2. Transaction Coordinator (TC)

- Starts transactions.
- Breaks the transaction into subtransactions.
- Sends them to relevant sites.
- Coordinates commit/abort across all sites.

📌 SYSTEM FAILURE MODES

- Site failure (system crash)
- Message loss (handled by TCP/IP)
- Communication link failure → messages rerouted if possible.
- Network partition → network split into isolated groups.

📌 COMMIT PROTOCOLS

Used to ensure atomicity across sites:
➡️ Either ALL commit or ALL abort.

⭐ TWO-PHASE COMMIT (2PC)

PHASE 1: Prepare Phase

Coordinator (Ci):

- Logs <prepare T>
- Sends prepare message to all participants

Participant:

- If cannot commit → logs <no T> → sends abort
- If can commit → logs <ready T> → forces log → sends ready

PHASE 2: Commit/Abort Phase

Coordinator:

- If *all* replied ready → logs <commit T>
- Else → logs <abort T>
- Sends commit/abort decision to all participants

Participants:

- On receiving decision → commit or abort locally → log the result

📌 IMPORTANT TERMS

Ready State

- After participant sends ready T
- It MUST follow the coordinator's decision
- Must retain locks until final outcome

Blocking Problem

- If coordinator fails after participants send ready,
  → participants wait indefinitely.

📌 FAILURE HANDLING IN 2PC

1️⃣ Participant Site Failure

Coordinator sees:

- Fails before ready → treat as abort
- Fails after ready → ignore failure and continue protocol

On Recovery, site checks log:

- <commit T> → redo

- <abort T> → undo
- <ready T> → ask coordinator for outcome
- No records → undo (site failed before responding)

2️⃣ Coordinator Failure

Participants check their logs:

1. If any site has <commit T> → commit
2. If any site has <abort T> → abort
3. If any site has no ready record → coordinator couldn't have committed → abort
4. Else → all are ready but no decision → wait
   (This is the blocking problem)

3️⃣ Network Partition

- Sites in same partition as coordinator proceed normally.
- Others assume coordinator failed → follow recovery rules.
- Safe, but may still wait for coordinator.

⭐ THREE-PHASE COMMIT (3PC)

Designed to avoid blocking (unlike 2PC).

Assumptions

- No network partition
- At least one site always alive
- Max K site failures allowed

PHASE 1 – Prepare (Same as 2PC)

- Coordinator sends prepare
- Participants reply ready or abort

PHASE 2 – Pre-commit Phase

Coordinator:

- Writes <precommit T> or <abort T>

- Sends precommit/abort to participants
  Participants:
- Log decision
- If precommit, send ack

PHASE 3 – Commit Phase

Coordinator:

- After receiving required ack messages
- Logs <commit T>
- Sends commit to all sites

Participants:

- Commit transaction

📌 FAILURE HANDLING IN 3PC

Participant Recovery

Check log:

- <commit T> → redo
- <abort T> → undo
- <ready T> but no precommit → ask coordinator
- <precommit T> → send ack and continue
- No ready record → undo

Coordinator Failure

- Remaining sites elect new coordinator
- Each sends its local status:
  - Committed / Aborted / Ready / Precommitted / Not Ready
- New coordinator decides final outcome

📌 SUPER-SHORT MEMORY TRICKS

⭐ 2PC = Prepare → Commit

- Phase 1 → Ask

- Phase 2 → Decide

⭐ 3PC adds Pre-Commit

- Prepare → Precommit → Commit
- Removes blocking by adding a middle "safe point"

⭐ Ready = Promise

- Once a site sends ready, it must obey final decision.