Prathamesh Arvind Jadhav

DBMS for Placement

Chapter-1 : Basics

1. Data & Information

➢ Data

- **Definition**:
  Data refers to **raw facts and figures** that are collected but have **no meaning by themselves**. These are simply values or observations that are not yet processed or interpreted.
- **Key Characteristics**:
  - Unorganized
  - Can be in the form of numbers, text, symbols, or audio/video
  - No direct meaning or context
- **Examples**:
  - ‘Prathamesh’, ‘90’, ‘Computer Science’
    These values are just individual facts and don’t convey any meaningful message when isolated.

➢ Information

- **Definition**:
  Information is the **processed, organized, or structured form of data** that makes it **meaningful and useful** for decision-making or understanding.
- **Key Characteristics**:
  - Organized and structured
  - Contextual and meaningful
  - Helps in understanding or analysis
- **Example**:
  - **“Prathamesh scored 90 in Computer Science.”**
    This statement provides a complete, meaningful insight — who did what, and how well — transforming raw data into information.

---

2. Database System vs File System

Understanding the difference between a **File System** and a **Database Management System (DBMS)** is essential to appreciate the advantages that a DBMS offers in handling large, complex, and structured data.

□ **File System**

- A file system is a traditional way to store and manage data using files (e.g., .txt, .csv, .xls).
- Each application defines and manages its own data format.
- There is **no built-in mechanism for relationships**, indexing, or concurrent access.

□ **Key Characteristics**:

- Stores data in **flat files**
- Lacks centralized control
- High **data redundancy** (duplicate data)
- No automated way to maintain consistency

□ **Database System (DBMS)**

- A DBMS is software that helps users **store, manage, and retrieve data** efficiently.
- It offers a structured environment with features like **indexing, constraints, relational integrity, and query language (SQL)**.
- Centralized management helps maintain consistency and security.

□ **Key Characteristics**:

- Stores data in **tables (relations)**
- Supports **data normalization** to reduce redundancy
- Ensures **data consistency** and **integrity**
- Provides **access control**, **backup**, and **concurrent user support**

Comparison Table: File System vs Database System

Feature	File System	Database System (DBMS)
Data Redundancy	High – duplicate data across files	Low – normalization reduces duplication
Data Consistency	Difficult to maintain	Maintained via integrity constraints
Security	Limited – file-level protection	High – user roles, authentication, encryption
Querying	Manual – requires programming or search	SQL – easy and powerful querying
Concurrency Control	Not supported – prone to data conflict	Supported – multiple users access safely
Data Integrity	Must be handled manually	Enforced automatically via DBMS rules
Data Relationships	Hard to define and maintain	Easily managed through relational models
Backup & Recovery	Manual or through OS tools	Automated and efficient
Example	Text files, Excel spreadsheets	MySQL, Oracle, PostgreSQL

□ **Real-Life Analogy**

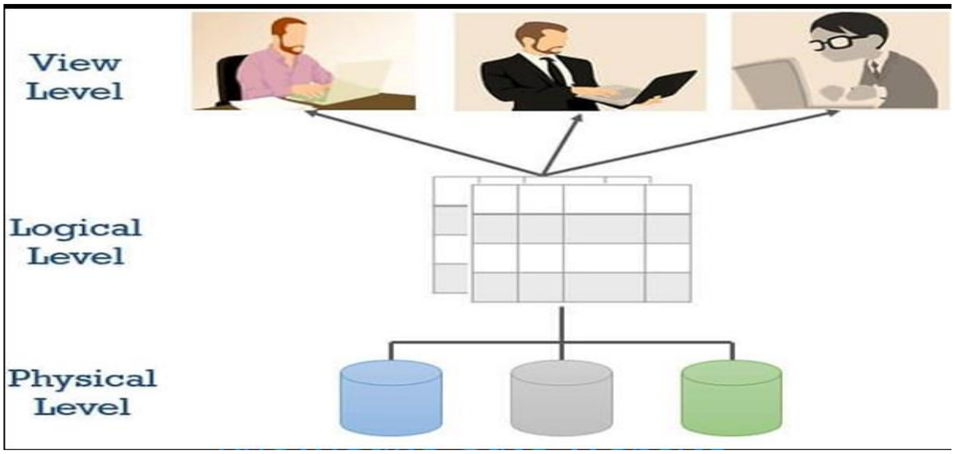
- **File System** is like using a bunch of paper files in a cabinet — harder to search, update, or secure.
- **Database System** is like using a smart digital filing system with search, filters, and access control — fast and efficient.

---

3. Views of Database (Three-Level Architecture of DBMS)

The **three-level architecture** of a Database Management System (DBMS) is designed to separate the user's view from the physical structure of the database. This separation helps ensure **data abstraction**, **independence**, and **security**.

Three Levels of Database Architecture



1.Internal Level (Physical View)

- **Definition**:
  This is the **lowest level** of the DBMS architecture. It defines **how the data is actually stored** on storage devices (like hard drives).
- **Focus**:
  - Efficiency
  - Data compression
  - Indexing
  - File structure
  - Physical record placement
- **Key Points**:
  - Completely hidden from users
  - Managed by database administrators (DBAs)
  - Deals with **binary format**, **storage blocks**, **access methods**
- **Analogy**: Like the engine of a car — users don't see it, but it powers everything.

2.Conceptual Level (Logical View)

- **Definition**:
  The **middle level** that describes the **logical structure of the entire database**. It shows **what data is stored** and **how the data is related**.
- **Focus**:
  - Entities and relationships
  - Data types
  - Integrity constraints
  - Logical schema (ER diagrams, tables)

- **Key Points**:
  - Independent of physical storage
  - Only one conceptual schema per database
  - Shared across all users and applications
- **Example**: A student table with attributes: StudentID, Name, Email, Marks, Attendance.

3.External Level (User View)

- **Definition**:
  This is the **highest level** of abstraction. It defines **how individual users or applications interact** with the database.
- **Focus**:
  - Customized views
  - Data hiding and security
  - User access control
- **Key Points**:
  - Different users can have different views of the same data.
  - Multiple **external schemas** (views) can exist for different users.
- **Example**:
  - A **student** may see: Name, Grades, Courses.
  - A **teacher** may see: Name, Attendance, Performance.
  - An **admin** may see: Fees, Admission details, etc.

□ Why Use the Three-Level Architecture?

- **Data Abstraction**: Hides details users don’t need.
- **Security**: Restrict access to sensitive data.
- **Data Independence**: Changes at one level don’t affect others (explained in next topic).
- **Scalability and Modularity**: Easier to manage and evolve.

□ Real-Life Analogy:

Imagine a **banking app**:

- You (user) only see your **account balance** and transactions (External Level).
- The bank software knows how your data relates to accounts and branches (Conceptual Level).
- Internally, this data is stored in blocks and indexes on a server (Internal Level).

---

4. Data Independence

Definition:
Data Independence refers to the **ability to change the schema at one level of a database system without affecting the schema at the next higher level**.

This concept ensures **flexibility**, **maintainability**, and **robustness** of database applications.

Types of Data Independence

1.Logical Data Independence

- **Definition**: The ability to change the **conceptual schema** without altering the **external schema** or application programs.
- **Example**:
  Suppose you **add a new column Email to a table** in the database. This **should not affect** the user interface (forms, queries) that don't use that column.
- **Use Cases**:
  - Adding/removing attributes in a table.
  - Merging two tables logically.
- **Important for**: Application programmers and end-users.

2.Physical Data Independence

- **Definition**: The ability to change the **internal schema** (physical storage details) without affecting the **conceptual schema**.
- **Example**:
  Suppose you **move data from an HDD to an SSD** or **reorganize indexes** for better performance. These changes should **not affect how the data is logically represented**.
- **Use Cases**:
  - Changing data format (row-based to column-based).
  - Modifying file structures or access methods.
- **Important for**: Database administrators and performance tuners.

Goal of Data Independence

- **Ease of modification**
- **Long-term system sustainability**

- **Cost-effective maintenance**
- **Separation of concerns** between logical and physical aspects

---

5. Instances & Schema

Understanding the difference between **Schema** and **Instance** is essential in grasping how data is stored and evolves in a database.

□ Schema

- **Definition**:
  A **schema** is the **overall design or blueprint** of the database.
- **Characteristics**:
  - Defined during database creation
  - **Rarely changes**
  - Describes structure, data types, relationships, and constraints
- **Example**:

```
CREATE TABLE Student (
   Name VARCHAR(50),
   Roll_No INT,
   Marks INT
);
```

  This is the **schema** of the table.

- **Analogy**: Like the architectural blueprint of a building.

□ Instance

- **Definition**:
  An **instance** is the **actual data present in the database at a particular point in time**.
- **Characteristics**:
  - **Dynamic and frequently changes**
  - Can differ from moment to moment
  - Reflects current state of the database

- **Example**:

Name	Roll_No	Marks
Prathamesh	101	90
Sneha	102	85

- This is an **instance** of the Student table.
- **Analogy**: Like the people currently living in a building — they may change, but the blueprint remains the same.

---

6. OLAP vs OLTP

Feature	OLTP (Online Transaction Processing)	OLAP (Online Analytical Processing)
Purpose	Manage day-to-day **transactional tasks**	Perform **complex analysis and reporting**
Users	Clerks, cashiers, database admins (DBAs)	Executives, data analysts, decision makers
Data	**Current, real-time data**	**Historical and aggregated data**
Operations	Read, Insert, Update, Delete (CRUD operations)	Complex read-only queries (e.g., summaries, trends)
Speed	Very **fast for short transactions**	Slower but optimized for **bulk data analysis**
Database Design	Highly **normalized** (to remove redundancy)	**Denormalized** for faster querying
Example Use Cases	ATM transactions, order processing, flight bookings	Sales forecasting, customer segmentation, BI tools

---

Use Case Comparison

Use Case	OLTP	OLAP
Bank System	Deposit, withdraw, transfer money	Analyze monthly withdrawal trends
E-Commerce	Add item to cart, checkout	Generate reports on seasonal product sales
Education	Record student grades	Compare performance of batches over 5 years

---

Summary

- OLTP is **optimized for speed and reliability** in routine operations.
- OLAP is **optimized for deep insights, trends, and analysis.**

---

7. Types of Databases

1.Hierarchical Database

- **Structure**: Tree-like (one parent, many children)
- **Relationship**: One-to-many
- **Navigation**: Top-down
- **Advantages**:
  - Fast access if relationships are fixed
  - Simple and easy to implement
- **Disadvantages**:
  - Rigid structure; hard to reorganize
- **Example**: IBM IMS, XML document storage

2.Network Database

- **Structure**: Graph (many-to-many relationships)
- **Relationship**: A child can have multiple parents
- **Advantages**:
  - More flexible than hierarchical DBs
- **Disadvantages**:
  - Complex to design and maintain
- **Example**: Integrated Data Store (IDS)

3.Relational Database (RDBMS)

- **Structure**: Tables (also called relations)
- **Key Features**:
  - Uses **SQL (Structured Query Language)**
  - Data is organized in **rows and columns**
  - Supports **primary and foreign keys**
- **Advantages**:
  - High data integrity and consistency
  - Easy to maintain relationships
- **Example**: MySQL, PostgreSQL, Oracle, SQLite

4.Object-Oriented Database

- **Structure**: Stores data as **objects** (like OOP languages)
- **Key Features**:
  - Supports **inheritance**, **encapsulation**, **polymorphism**
  - Suitable for **multimedia**, **CAD**, and **complex applications**
- **Advantages**:
  - Seamless integration with object-oriented programming
- **Example**: db4o, ObjectDB

5.NoSQL Database

- **Structure**: Non-relational (document, key-value, graph, column)
- **Use Case**: Big Data, real-time analytics, unstructured data
- **Types**:
  - Document-based (MongoDB)
  - Column-based (Cassandra)
  - Graph-based (Neo4j)
  - Key-value store (Redis)
- **Advantages**:
  - High scalability and performance
  - Schema-less (flexible structure)
- **Disadvantages**:
  - Less mature than RDBMS
  - May lack standardization
- **Examples**: MongoDB, Cassandra, Couchbase, DynamoDB

---

8. DBA (Database Administrator)

A **Database Administrator (DBA)** is a professional responsible for managing, maintaining, and securing a database system. The DBA ensures that data is **available, consistent, secure**, and **efficiently managed**.

□ **Key Roles and Responsibilities of a DBA**

Responsibility	Details
1. Designing Database Schemas	- Create tables, indexes, constraints - Plan relationships between data entities

Responsibility	Details
2. Installing & Upgrading DBMS Software	- Set up DBMS software (e.g., MySQL, Oracle) - Apply patches and upgrades
3. User Access and Security Management	- Create and manage user roles - Set privileges and prevent unauthorized access
4. Performance Tuning and Optimization	- Optimize SQL queries - Monitor and tune DB performance to avoid bottlenecks
5. Backup and Recovery Planning	- Schedule regular backups - Implement recovery strategies in case of data loss
6. Monitoring & Troubleshooting	- Use monitoring tools to detect issues - Resolve crashes, slow queries, etc.
7. Ensuring Data Integrity	- Enforce data validation rules - Prevent corruption or duplication of data
8. Data Migration	- Move data between systems or formats during upgrades or cloud migration
9. Documentation	- Maintain documentation of configurations, user roles, backup policies, etc.

□ DBA as the Guardian of the Database

A DBA acts like a **guardian or security chief** for a database system, balancing access, speed, and protection. They serve multiple roles, such as:

- **Data Engineer**: Ensuring smooth data flow and structure.
- **Security Expert**: Protecting sensitive information.
- **System Admin**: Handling technical backend tasks.
- **Disaster Recovery Lead**: Restoring operations after data loss.

□ Example Scenario

Problem: Users report the app is slow.

DBA Tasks:

- Check for **long-running queries**
- Identify **locked tables or deadlocks**
- Use **indexing** to speed up lookups
- Optimize queries or restructure database design

---

9. DBMS Architecture

DBMS Architecture refers to the design structure of a database system that defines how users interact with the system and how different components communicate. The architecture ensures **efficiency**, **security**, **scalability**, and **maintainability** of database operations.

□ Types of DBMS Architecture

---

1. One-Tier Architecture (1-Tier)

□ ***Description:***

- The database and the user interface reside on the **same machine**.
- The user interacts **directly** with the database without any intermediary.

□ ***Diagram:***

[User Interface + DBMS + Database]
(Single System)

Use Case:

- Useful in **standalone** applications like **MS Access** or **SQLite-based desktop apps**.

□ ***Limitations:***

- Not suitable for multi-user or web-based environments.
- Poor security and scalability.

---

2. Two-Tier Architecture (2-Tier)

□ ***Description:***

- Based on the **Client-Server Model**.
- The client (user) communicates directly with the **DB server** over a network.
- The application logic may reside on the client.

□ ***Diagram:***

[Client Application] <---> [Database Server (DBMS + Database)]

Use Case:

- Used in small-scale applications.
- Example: **VB.NET with SQL Server**, **Java + MySQL desktop apps**.

□ ***Limitations:***

- Difficult to manage if clients increase.
- Application logic and data access logic are tightly coupled.

---

3. Three-Tier Architecture (3-Tier)

□ ***Description:***

- Divides the system into **three layers** for better modularity.
- Promotes separation of concerns and makes maintenance easier.

□ ***Structure:***

[Presentation Tier (Client/UI)]
⇅
[Logic Tier (Application Server)]
⇅
[Data Tier (Database Server)]

□ ***Components Breakdown:***

Tier	Description
Presentation Tier	- User interface - Handles input/output - Examples: Web browser, mobile app
Logic Tier	- Contains business logic - Validates input, processes data - Examples: Java/Python backend
Data Tier	- Stores data using DBMS - Examples: MySQL, Oracle, PostgreSQL

Advantages:

- **Separation of Concerns**: Each tier is independent and modular.
- **Security**: The user doesn’t have direct access to the database.
- **Scalability**: Easier to scale each layer separately.
- **Maintainability**: Changes in one layer don’t affect others.

□ Real-World Analogy

Tier	Analogy Example
Presentation	Waiter taking your order
Logic	Kitchen deciding how to prepare it
Data	Storage room where ingredients are kept

□ Example: Online Bookstore

- **Presentation Tier**: User opens the website to search for a book.
- **Logic Tier**: The backend server processes the request, checks inventory, applies discounts.
- **Data Tier**: The database fetches book details and availability.

Chapter-2 : ER Diagram

ER Diagram Concepts

An **ER Diagram (Entity-Relationship Diagram)** is a visual representation of data and how it relates to other data in a database. It is a crucial step in **database design**.

---

1. Entity

□ **Definition**:
An **Entity** is a real-world object or concept that can be **uniquely identified** and has a **distinct existence** in a database. Entities are the fundamental building blocks in the **Entity-Relationship (ER) model**.

□ **Types of Entities**:

1. Strong Entity:

- An entity that **exists independently** of other entities.
- Has a **primary key** to uniquely identify each instance.
- Represented by a **single rectangle** in ER diagrams.

□ **Example**:
Student, Employee, Car, etc.

- Student can exist independently with its own unique ID.
- ER Representation:

Student

---

2. Weak Entity:

- An entity that **cannot exist on its own**.
- Has **no sufficient attribute(s)** to form a primary key.
- Identified through a **relationship with a strong entity**.
- Uses a **partial key (discriminator)**.
- Represented by a **double rectangle**.

□ **Example**:
Dependent (Child/Spouse) of an Employee.

- A Dependent cannot exist in the database without linking it to an Employee.
- ER Representation:

Dependent

□ **Real-life Example**:
Let's say you are designing a **Student Management System**.

- Student is an entity because:
  - It has distinct data like Name, Roll No, and Department.
  - Each student is independently identifiable.

Name	Roll_No	Department
Prathamesh	101	Computer Sci.
Raj	102	AI & DS

- Each row here is an **instance** of the Student entity.

---

2. Attributes

Definition:
Attributes are the **properties or characteristics** that describe an entity. Each attribute holds some information about the entity.

Types of Attributes:

Type	Description	Example
Simple/Atomic	Cannot be divided further into sub-parts.	Name, Age
Composite	Can be broken down into smaller sub-parts.	Name → First Name, Last Name
Derived	Values can be calculated or derived from other attributes.	Age derived from Date of Birth (DOB)
Multi-valued	Can have multiple values for a single entity.	Phone Numbers (home, mobile)
Key Attribute	Uniquely identifies each entity instance.	Student_ID, Employee_ID

Explanation with examples:

- **Simple Attribute:**
  Age cannot be further subdivided; it's a single atomic value like 20, 30, etc.
- **Composite Attribute:**
  Name can be divided into First Name and Last Name.
  Example:

  ```
  Name
   ├── First Name: Prathamesh
   └── Last Name: Jadhav
  ```

- **Derived Attribute:**
  Age can be calculated using the Date of Birth (DOB) attribute and the current date. So, Age is *derived* from DOB.

- **Multi-valued Attribute:**
  A person can have multiple phone numbers — home, office, mobile — all related to the same entity Person.
- **Key Attribute:**
  Student_ID uniquely identifies every student in the database; no two students can share the same ID.

ER Diagram Notation:

- Simple and Composite attributes: **Oval shape**
- Derived attributes: **Dashed oval**
- Multi-valued attributes: **Double oval**
- Key attributes: **Underlined attribute name**

---

3. Relationship

□ **Definition:**
A **Relationship** represents an **association or connection** between two or more entities in a database. It shows how entities are linked to each other.

□ **Example:**

- A **"Works_For"** relationship between the entities Employee and Department indicates that an employee works in a particular department.

□ **Types of Relationships:**

Type	Description	Example
One-to-One (1:1)	Each entity in set A is related to **at most one** entity in set B, and vice versa.	Each Employee has one Passport and each Passport belongs to one Employee.
One-to-Many (1:N)	Each entity in set A can relate to **many** entities in set B, but each entity in B relates to only one entity in A.	A Department has many Employees, but an Employee works in only one Department.
Many-to-Many (M:N)	Many entities in set A relate to many entities in set B.	Students enroll in many Courses, and each Course has many Students.

Visualization:

- **One-to-One (1:1):**

  Employee 1 -------- 1 Passport

- **One-to-Many (1:N):**

  Department 1 -------- N Employees

- **Many-to-Many (M:N):**

  Students M -------- N Courses

Additional Notes:

- Relationships can also have **attributes**, called **relationship attributes**.
  For example, in a Works_For relationship, an attribute could be Date_Joined.
- In ER diagrams, relationships are shown as **diamonds**, connected by lines to the entities.

---

4. Degree of Relationship

 Definition:
The **degree of a relationship** refers to the **number of different entity types** involved in that relationship.

 Types of Degree:

Type	Number of Entities Involved	Description & Example
Unary	1	Relationship involves only one entity type.
		Example: An Employee **manages** another Employee.
Binary	2	Relationship involves two different entity types.
		Example: An Employee **works in** a Department.
Ternary	3	Relationship involves three different entity types.
		Example: A Doctor **prescribes** Medicine to a Patient.

Explanation:

- **Unary Relationship (also called recursive):**
  When an entity is related to itself.
  Example: In an organization, an employee may manage another employee.
- **Binary Relationship:**
  The most common relationship type. It connects two entities.
  Example: Employee and Department.
- **Ternary Relationship:**
  Involves three entities together in a single relationship.
  Example: Doctor, Patient, and Medicine involved in a prescription.

ER Diagram Notation:

- The relationship diamond connects the involved entities, with lines equal to the degree number.
- Unary: One entity connected twice (with a recursive loop).
- Binary: Two entities connected.
- Ternary: Three entities connected.

---

5. Mapping Cardinality

□ **Definition:**
Mapping cardinality defines **how many instances of one entity** can or must be associated with **instances of another entity** in a relationship.

It answers questions like:

- For each entity A, how many entities B can it be linked to?
- Is it just one, many, or optional?

□ Types of Mapping Cardinality:

Type	Description	Example
One-to-One (1:1)	Each instance of Entity A is related to **one and only one** instance of Entity B, and vice versa.	A person has one passport; a passport belongs to one person.
One-to-Many (1:N)	Each instance of Entity A can be related to **many instances** of Entity B, but each instance of B is related to **only one** instance of A.	A department has many employees; each employee works for one department.

Type	Description	Example
Many-to-Many (M:N)	Each instance of Entity A can be related to **many instances** of Entity B, and vice versa.	Students enroll in many courses; courses have many students.

Visual Example:

- **One-to-One:**
  Person 1 -------- 1 Passport
- **One-to-Many:**
  Department 1 -------- N Employees
- **Many-to-Many:**
  Students M -------- N Courses

Important Notes:

- Mapping cardinality helps define **constraints** on the relationship.
- It affects how the database schema is designed — for example, many-to-many relationships often require a **junction table** in relational databases.
- Cardinality is shown on ER diagrams as numbers or symbols near the relationship lines (e.g., 1, N, M).

---

6. Weak Entity Set

□ **Definition:**
A **Weak Entity** is an entity that **cannot be uniquely identified by its own attributes alone**. It depends on a related **strong entity** for its identification.

□ **Characteristics of Weak Entity:**

- **No Primary Key:**
  It does not have a unique identifier (primary key) by itself.
- **Partial Key (Discriminator):**
  It has some attributes called a **partial key** that can uniquely identify weak entities **only when combined with the key of the related strong entity**.
- **Existence Dependency:**
  It **must be associated** with a strong entity through a **relationship** (called an identifying relationship).

- **Double Rectangle and Double Diamond:**
  In ER diagrams, a weak entity is shown with a **double rectangle**, and the identifying relationship is shown with a **double diamond**.

 Example:

- Dependent is a weak entity. It cannot be uniquely identified without knowing the Employee it depends on.
- Attributes of Dependent might include: Dependent_Name (partial key) and Relationship.
- Each dependent is uniquely identified by the combination of Employee_ID (from the strong entity) and Dependent_Name.

ER Diagram Representation:

- **Strong Entity:** Single rectangle (e.g., Employee)
- **Weak Entity:** Double rectangle (e.g., Dependent)
- **Identifying Relationship:** Double diamond connecting weak entity to strong entity

---

Why use Weak Entities?

- They model entities that exist only in relation to other entities (like dependents, rooms in a building, or installments of a loan).

---

7. Conversion from ER Diagram to Relational Model

 Steps to convert ER Diagram elements into Relational Database tables:

ER Element	Relational Model Conversion
Entity	Convert to a **table** where attributes become columns.
Simple Attribute	Directly becomes a **column** in the table.
Composite Attribute	Break down into **individual simple attributes** and create columns for each.
Multi-valued Attribute	Create a **separate table** to store multiple values linked back to the main entity using a foreign key.
Relationship (1:N)	Add a **foreign key** in the table on the “many” side pointing to the “one” side.

ER Element	Relational Model Conversion
Relationship (M:N)	Create a **new table** (junction table) with foreign keys referencing both entity tables.
Weak Entity	Create a table for the weak entity; include the **primary key of the strong entity as a foreign key** along with the weak entity's partial key.

Explanation with examples:

- **Entity to Table:**
  Example: Entity Student with attributes StudentID, Name, Age becomes a Student table with these columns.
- **Composite Attribute:**
  If Name is composite (First, Last), create two columns: FirstName, LastName.
- **Multi-valued Attribute:**
  Example: If a Student can have multiple phone numbers, create a separate StudentPhone table with columns: StudentID (foreign key), PhoneNumber.
- **One-to-Many Relationship:**
  Example: Department (1) and Employee (N). Add DepartmentID as a foreign key in the Employee table.
- **Many-to-Many Relationship:**
  Example: Student and Course have M:N relationship. Create StudentCourse table with foreign keys StudentID and CourseID.
- **Weak Entity:**
  Example: Dependent table will include EmployeeID (foreign key) and DependentName (partial key), combined to form the primary key.

---

8. Generalization

□ **Definition:**
Generalization is the process of **abstracting common features** (attributes and relationships) from two or more entities and combining them into a **higher-level generalized entity (superclass)**.

□ **Purpose:**
To reduce redundancy by identifying shared properties among entities.

□ **Example:**

- Entities: Car, Truck
- Common features: Vehicle_ID, Engine_Type, Manufacturer

- Generalized entity: Vehicle (superclass) with common attributes
- Car and Truck become subclasses inheriting from Vehicle.

□ **Diagrammatically:**

```
  Vehicle
  /    \
Car    Truck
```

---

9. Specialization

□ **Definition:**
Specialization is the opposite of generalization. It is the process of **dividing a broad entity** into **more specific subclasses** based on distinguishing characteristics.

□ **Purpose:**
To capture specific attributes or behaviors for subclasses that are not relevant to the whole superclass.

□ **Example:**

- Entity: Employee
- Specialized into: Manager, Technician
- Manager might have attributes like Budget and Team_Size
- Technician might have attributes like Skill_Set and Shift

□ **Key Points:**

- **Generalization:** Bottom-up (from specific to general)
- **Specialization:** Top-down (from general to specific)

---

10. Aggregation

Definition:
Aggregation is used when a **relationship itself needs to be treated as an entity** because it participates in another relationship.

Use Case:
When there is a need to express a relationship involving another relationship, aggregation helps to model this more complex association.

Example:

- Supervisor monitors a Project (relationship: Monitors)
- The Monitors relationship itself is related to a Department (belongs to)
- Here, Monitors (the relationship) is treated as an entity to relate with Department.

Diagram:

```
[Supervisor] ----- Monitors ----- [Project]
     \                         /
      ------- Belongs To -------
             [Department]
```

Summary:

- **Generalization** merges common aspects into a parent class.
- **Specialization** breaks a general class into more specific sub-classes.
- **Aggregation** treats a relationship as an entity to link it with other entities.

Chapter-3: Relational Model

1. Definition of Relational Model

- The **Relational Model (RM)** organizes data into **relations**, which are more commonly referred to as **tables**.
- A **Relational Database** is a collection of such tables, where each table has a unique name and stores data in **rows (tuples)** and **columns (attributes)**.
- The **relation** is mathematically a **set of tuples** sharing the same attributes.

---

2. Core Concepts in RM

1. **Relation**: A table with columns and rows.
2. **Tuple**: A row in a table. Each row is a unique **data record**.
3. **Attribute**: A column in a table. Each attribute describes a property of the entity represented.
4. **Domain**: The **set of allowed values** for an attribute. For example, for an attribute "Age", the domain might be integers from 1 to 100.
5. **Relation Schema**: Defines the structure of a relation. It includes:
   - Name of the relation
   - Name and domain of each attribute
6. **Instance**: The actual content or data stored in a relation at a given time.

---

3. Examples of RDBMS

Some of the most commonly used **Relational Database Management Systems (RDBMS)** are:

- **Oracle**
- **MySQL**
- **IBM DB2**
- **Microsoft Access**

---

4. Degree and Cardinality

- **Degree**: The **number of attributes/columns** in a relation.
- **Cardinality**: The **number of tuples/rows** in a relation.

---

5. Relational Keys

Keys are vital to ensure that each tuple is uniquely identifiable and maintain referential integrity:

1. **Super Key**:
   - Any set of attributes that can uniquely identify a tuple.
   - Can be a single attribute or combination.
2. **Candidate Key (CK)**:
   - A **minimal super key**.
   - It does not contain any redundant attribute.
   - A relation can have multiple candidate keys.

3. **Primary Key (PK)**:
    - A chosen candidate key to uniquely identify tuples.
    - Must be **unique** and **not NULL**.
4. **Alternate Key (AK)**:
    - Candidate keys that are not chosen as the primary key.
5. **Foreign Key (FK)**:
    - An attribute in one table that refers to the **primary key** in another table.
    - Used to **establish a relationship** between two tables.
    - **Referencing table** (child) and **Referenced table** (parent).
6. **Composite Key**:
    - A **primary key** formed by combining **two or more attributes**.
7. **Compound Key**:
    - A primary key made by combining **two foreign keys**.
8. **Surrogate Key**:
    - A system-generated key (like auto-incremented numbers).
    - Acts as a synthetic primary key.

---

6. Properties of Relations

To ensure data consistency and integrity, relations must follow these rules:

1. Each relation must have a **unique name**.
2. **Attributes** must have **unique names** (no duplicate column names).
3. The **values in attributes must be atomic** (indivisible).
4. Each **tuple must be unique** (no duplicate rows).
5. The **order of tuples (rows)** and **attributes (columns)** has no significance.
6. Must follow **integrity constraints**.

---

7. Integrity Constraints

Integrity constraints are rules to ensure the **accuracy and consistency** of the data in a relational database.

Types of Integrity Constraints:

A. **Domain Constraints**:

- Define the **permissible values** for an attribute.
- Example: Age must be a number between 1 and 100.
- Ensures the data entered is within a valid domain.

B. **Entity Integrity Constraints**:

- **Primary Key** of a relation cannot be **NULL**.
- Ensures that each record has a unique and valid identifier.

C. **Referential Integrity Constraints**:

- Maintain consistency among tuples in two related relations.
- A **foreign key** must either:
  - Be **NULL**
  - Or match a **primary key** value in the referenced table.
- Prevents orphaned records and maintains logical relationships.

D. **Key Constraints** (Detailed):

1. **NOT NULL**:
   - Attribute must have a value (can't be left blank).
   - Enforces mandatory data input.
2. **UNIQUE**:
   - All values in the column must be unique.
   - No duplicates allowed.
3. **DEFAULT**:
   - Automatically assigns a default value if no value is provided.
4. **CHECK**:
   - Validates data before it is inserted or updated.
   - Example: CHECK (salary > 0)
5. **PRIMARY KEY**:
   - Unique and not null.
   - Only one PK per table.
6. **FOREIGN KEY**:
   - Connects child table to parent table.
   - Ensures referential integrity.

Chapter-4 :RDBMS & Functional Dependency

1. Basics & Properties of RDBMS

- **RDBMS** (Relational Database Management System) stores data in **tables (relations)**.
- Each table consists of **rows (tuples)** and **columns (attributes)**.
- Tables are related using **keys** like Primary Key and Foreign Key.

Properties:

- **Atomicity**: Every cell must hold a single value (no multivalued attributes).
- **Uniqueness**: Each tuple must be unique.
- **Column Uniqueness**: Each attribute/column name must be distinct.

- **Order Independence**: Row or column order doesn't matter.
- **Integrity Constraints**: Must maintain domain, entity, and referential integrity.

---

2. Update Anomalies

Update anomalies arise when a database is **poorly normalized**. These are:

1. **Insertion Anomaly**:
    - You can't insert data unless other unrelated data is also provided.
    - *Example*: Can't insert a new course unless at least one student is enrolled.
2. **Deletion Anomaly**:
    - Deleting a record may delete critical unrelated information.
    - *Example*: Deleting the only student enrolled in a course deletes course data.
3. **Updation Anomaly**:
    - Updating one field requires updating it in multiple places, risking inconsistency.
    - *Example*: Changing a professor's name requires changing in all course records.

---

3. Purpose of Normalization

Normalization is a step-by-step process to **minimize redundancy and avoid anomalies**.

Goals:

- Remove duplicate data.
- Ensure data dependencies make sense (functional dependency).
- Organize data into logical groupings.
- Ensure data integrity and consistency.

Normalization uses **normal forms (1NF, 2NF, 3NF, BCNF...)** to structure data.

---

4. Functional Dependency (FD)

Functional Dependency (FD) is a **relationship between attributes** in a table.

If **A → B**, then for each unique value of A, there is **only one** corresponding value of B.

- **A → B** means **A functionally determines B**.
- It helps to identify **redundancy** and normalize relations.

Example:
If RollNo → Name, then each RollNo maps to exactly one Name.

---

5. Closure of Set of Attributes

The **closure of an attribute set X**, denoted as **X^+**, is the **set of all attributes** functionally determined by X.

 Steps to find closure X^+:

1. Start with $X^+ = X$.
2. Apply FDs where LHS $\subseteq X^+$.
3. Add RHS to X^+.
4. Repeat until no more attributes can be added.

Use of Closure:

- To find candidate keys.
- To test FD implication (does X → Y follow from FDs?).

---

6. Armstrong's Axioms

Armstrong's Axioms are **inference rules** used to derive **all valid functional dependencies** from a given set.

Axioms:

1. **Reflexivity**: If $Y \subseteq X$, then $X \to Y$
2. **Augmentation**: If $X \to Y$, then $XZ \to YZ$
3. **Transitivity**: If $X \to Y$ and $Y \to Z$, then $X \to Z$

Additional (Derived) Rules:

4. **Union**: If $X \to Y$ and $X \to Z$, then $X \to YZ$
5. **Decomposition**: If $X \to YZ$, then $X \to Y$ and $X \to Z$
6. **Pseudo-transitivity**: If $X \to Y$ and $WY \to Z$, then $WX \to Z$

These axioms help in proving or deriving functional dependencies.

---

7. Equivalence of Two Sets of FDs

Two sets of FDs **F and G are equivalent** if:

- Every dependency in **F can be derived from G**.
- Every dependency in **G can be derived from F**.

F ≡ G if:

- **$F^+ = G^+$**

This helps in minimizing and optimizing FD sets for normalization.

---

8. Canonical Cover (Minimal Cover)

A **Canonical Cover** is a minimal set of FDs that is equivalent to the original set.

It removes:

- Redundant attributes on the left or right side.
- Redundant FDs.
- Combines multiple FDs where applicable.

Steps to find Canonical Cover:

1. **Split**: Ensure all FDs are of the form $X \rightarrow A$.
2. **Remove extraneous attributes** from LHS or RHS.
3. **Remove redundant FDs** (check if it can be derived from others).
4. **Combine** FDs with the same LHS: $X \rightarrow A, X \rightarrow B \rightarrow X \rightarrow AB$

Canonical cover is useful for **efficient normalization and schema design**.

---

9. Keys in RDBMS

Keys are attributes or sets of attributes that help in **uniquely identifying tuples**.

Types of Keys:

1. **Super Key**:
   - Any set of attributes that uniquely identifies a row.
   - May contain extra (redundant) attributes.
2. **Candidate Key**:

  - Minimal Super Key (no unnecessary attributes).
  - One table may have multiple candidate keys.
3. **Primary Key**:
   - One of the candidate keys chosen as the **main key**.
   - Must be **NOT NULL** and **UNIQUE**.
4. **Alternate Key**:
   - Remaining candidate keys which are not primary.
5. **Foreign Key**:
   - Attribute in one table that refers to the **primary key of another table**.
   - Used to **maintain referential integrity**.
6. **Composite Key**:
   - Primary key made using more than one attribute.
7. **Surrogate Key**:
   - System-generated unique key (e.g., auto-increment ID).
   - Used when natural key is not available or complex.

Chapter-4 : Normalization

Normalization in DBMS

Normalization is a **systematic approach of decomposing complex tables** into simpler ones to **eliminate redundancy**, **prevent update anomalies**, and **ensure data integrity**.

---

1. First Normal Form (1NF)

A table is in **1NF** if:

- All attributes contain only **atomic (indivisible)** values.
- There are **no repeating groups or arrays**.

 Example (Not in 1NF):

RollNo	Name	Courses
1	Ram	Math, Science
2	Shyam	English, History

➡ Here, "Courses" is multivalued.

 After 1NF:

RollNo	Name	Course
1	Ram	Math

RollNo	Name	Course
1	Ram	Science
2	Shyam	English
2	Shyam	History

---

2. Second Normal Form (2NF)

A table is in **2NF** if:

- It is already in **1NF**.
- **Every non-prime attribute** is **fully functionally dependent** on the **whole primary key**.

Problem in Partial Dependency:
If a table has a **composite key (A, B)** and a non-key attribute **C** depends only on **A**, then it's not in 2NF.

Example (Not in 2NF):

RollNo	CourseID	StudentName	CourseName

Here, RollNo + CourseID is the primary key. But:

- StudentName depends on RollNo
- CourseName depends on CourseID

➡ **Solution**: Split into:

- Student(RollNo, StudentName)
- Course(CourseID, CourseName)
- Enrollment(RollNo, CourseID)

---

3. Third Normal Form (3NF)

A table is in **3NF** if:

- It is already in **2NF**.
- There is **no transitive dependency** between **non-prime attributes** and the **primary key**.

Transitive Dependency:
A → B → C, then A → C is transitive.

□ **Example (Not in 3NF)**:

EmpID	EmpName	DeptID	DeptName

Here:

- EmpID → DeptID (Direct)
- DeptID → DeptName (Transitive)

➡□ **Solution**: Split into:

- Employee(EmpID, EmpName, DeptID)
- Department(DeptID, DeptName)

---

4. Boyce-Codd Normal Form (BCNF)

A stricter version of 3NF.

A table is in **BCNF** if:

- It is in **3NF**, and
- For every non-trivial functional dependency **X → Y**, **X should be a superkey**.

□ **Example (Not in BCNF)**:

Course	Instructor	Room

Assume:

- Course → Instructor
- Room → Instructor

Neither **Course** nor **Room** is a superkey.

➡□ **Solution**: Decompose the table to make determinants superkeys.

---

5. Multivalued Dependency (MVD)

A **Multivalued Dependency** exists when:

If $A \rightarrow\rightarrow B$, then for each value of A, there is a **set of values** of B, **independent** of other attributes.

□ **Notation**: $A \rightarrow\rightarrow B$ (read as: A multivalued determines B)

□ **Example**:

Student	Course	Hobby
John	Math	Singing
John	Math	Reading
John	Science	Singing
John	Science	Reading

Here, Student $\rightarrow\rightarrow$ Course and Student $\rightarrow\rightarrow$ Hobby

---

6. Fourth Normal Form (4NF)

A table is in **4NF** if:

- It is in **BCNF**, and
- **No multivalued dependency** exists unless it is a **trivial dependency**.

□ **Trivial MVD**: $A \rightarrow\rightarrow B$ is trivial if $B \subseteq A$ or $A \cup B =$ all attributes.

□ **Solution**:
Decompose into separate tables to eliminate independent MVDs.

---

7. Lossy vs. Lossless Decomposition in DBMS

Lossless (Non-Lossy) Decomposition

Definition:
A decomposition is **lossless** if we can reconstruct the original relation **exactly** from the decomposed tables **using a natural join**.

Key Properties:

- No **data is lost or added** in the process.
- **Ensures correctness** and integrity of the database.
- It is also called **non-additive join decomposition**.

□ Condition for Lossless Join:

Given: A relation **R** is decomposed into **R1** and **R2**.
The decomposition is **lossless** if:

$(R1 \cap R2) \to R1$ or $(R1 \cap R2) \to R2$

That means: The **common attributes** must be a **superkey** in at least one of the decomposed relations.

□ Example of Lossless Decomposition:

Let us consider a relation:

Student(RollNo, Name, Dept)

We decompose it into:

R1 = Student1(RollNo, Name)
R2 = Student2(RollNo, Dept)

□ Common attribute = **RollNo**

Check:
If **RollNo → Name** and **RollNo → Dept**
Then RollNo is a **superkey** in both, so the decomposition is **lossless**.

JOIN of R1 and R2 will give the original Student table back.

Lossy Decomposition

Definition:
A decomposition is **lossy** if we **cannot** reconstruct the original relation **without data loss or spurious tuples**.

□ Issues in Lossy Decomposition:

- Introduces **extra rows (spurious tuples)** when performing joins.
- Can cause **inconsistencies** and **incorrect data retrieval**.

□ **Example of Lossy Decomposition:**

Original relation:

Employee(EmpID, EmpName, DeptName)

We decompose it into:

R1 = (EmpID, EmpName)
R2 = (EmpName, DeptName)

□ Common attribute = **EmpName**

But EmpName is **not a key** (multiple employees can have the same name).

When we join:

- We may get **incorrect combinations** of EmpID and DeptName for the same EmpName.
- Thus, it's a **lossy decomposition**.

---

8. Fifth Normal Form (5NF)

Also known as **Project-Join Normal Form (PJNF)**.

A table is in **5NF** if:

- It is in **4NF**, and
- It cannot be **decomposed further** without loss,
- All join dependencies are **implied by candidate keys**.

□ 5NF handles **join dependency** where data is reconstructed from **multiple joins** and not just FDs or MVDs.

□ **Example (Advanced)**:
An employee can work on multiple projects for multiple clients. Proper decomposition ensures that **cross-product combinations** are valid and no redundancy or anomaly occurs.

---

9. Dependency Preserving Decomposition

Definition:

A **decomposition** of a relation schema **R** into sub-relations **R_1, R_2, ..., Rn** is said to be **dependency preserving** if:

All the **functional dependencies (FDs)** defined on **R** can be **enforced** by simply enforcing them **on the individual decomposed relations (R_1, R_2, ..., Rn) — without needing to recombine them using joins.**

Why is Dependency Preservation Important?

In real-world databases:

- Constraints (i.e., functional dependencies) must always hold true to maintain **data integrity**.
- **Enforcing constraints through joins** is **costly and inefficient**.
- If we lose a dependency in decomposition, we might need to recombine relations just to validate or enforce a constraint.

This goes against **efficient DBMS operation**.

So, **dependency preservation** ensures:

- **All constraints are locally enforceable**, i.e., in the individual tables themselves.
- **Better performance**.
- **Simpler query validation** and updates.

Real-World Motivation:

Imagine if we had to **join multiple large tables frequently** just to:

- Check data validity
- Maintain a constraint

That would result in:

- Higher **CPU/memory** usage
- Poor **performance**
- More complex **maintenance**

Ideal Scenario:

A decomposition should ideally be:
Lossless
Dependency Preserving

This ensures:

- No data is lost or added
- Constraints can be enforced locally and efficiently

 Quick Recap:

Dependency Preserving Decomposition means:

- **No FD is lost**.
- All constraints can be enforced **within individual decomposed relations**.
- Avoids costly **joins** for enforcing constraints.

Ideal Normalization should aim for:

- **Lossless join**
- **Dependency preservation**

Together, they ensure **data integrity**, **query efficiency**, and **constraint enforcement**.

Chapter-6 : Indexing

1. Overview of Indexing

Indexing in DBMS is a **data structure** technique that improves the **speed of data retrieval** operations on a database table at the cost of **additional storage space** and **slightly slower write operations**.

 Purpose of Indexing:

- To allow **faster search** of rows in a table.
- To **reduce the number of disk I/O operations** needed.
- To enable **efficient query execution** for large datasets.

 Think of it like a book index:

- Instead of reading every page, you go to the index to find the page number for a topic.

---

2. Types of Indexing

There are various types of indexing based on how the index is constructed and used. Let’s go through the major ones.

2.1. Primary Indexing

□ **Definition:**

- Built on a **sorted** column that is also the **primary key** of the table.
- There is **only one primary index** per table.

□ **Characteristics:**

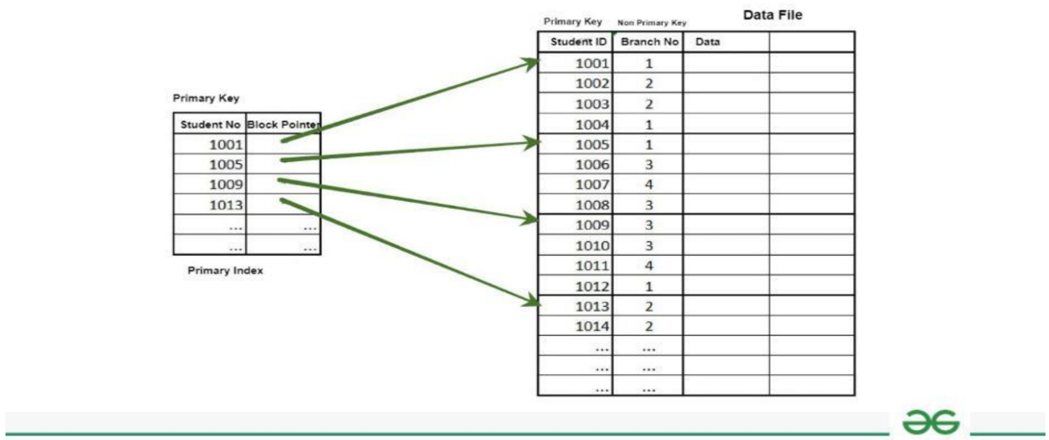
- It is **sparse** (not every record has an index entry).
- It is **ordered**.
- Based on **primary key**, which is **unique** and **not null**.

□ **Example:**

Assume a student table:

Roll_No (PK)	Name	Marks
101	Raj	89
102	Seema	92
103	Aman	85

- An index is built on **Roll_No**, the primary key.
- The DBMS stores the disk block address of each key.
- The search uses **binary search** on the index file.



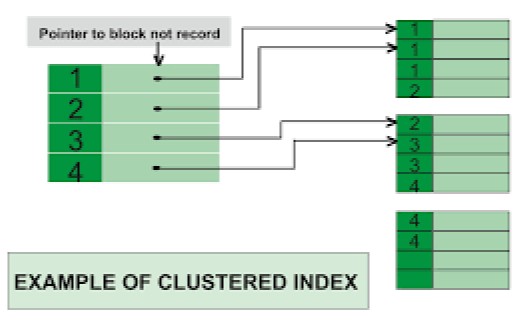
2.2. Clustered Indexing

□ **Definition:**

- Data is **physically stored** on the disk in the **same order** as the index.
- One table can have **only one clustered index**, because the data rows can be sorted only once.

□ **Characteristics:**

- A **dense index** (every record has an index entry).
- Sorting happens **physically** on disk, not just logically.



□ **Example:**

- Suppose you have an **Employee** table sorted by **Department_ID**.
- A clustered index on **Department_ID** means the rows will be stored together per department on disk.

Very useful when queries frequently access data in a range (e.g., department-wise).

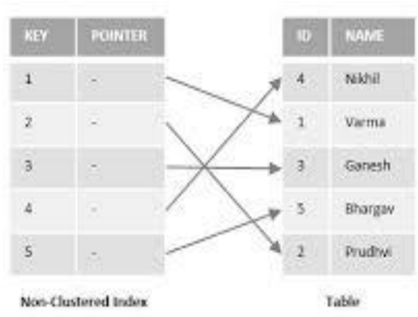
2.3. Secondary Indexing (Non-clustered Index)

 Definition:

- Index is built on **non-primary key attributes** (i.e., non-unique columns).
- Data is **not stored** in the order of this index.
- A table can have **multiple** secondary indexes.

 Characteristics:

- Usually a **dense index**.
- Helps in quick search on **non-primary** fields.



 Example:

For a **Student table**, if you create an index on **Name**, which is **not unique**, the system creates a secondary index file with:

[Name] => [List of addresses of matching records]

 Summary of Types:

Type	Based On	Storage Order	Dense/Sparse	Unique Required	Count per Table
Primary Index	Primary Key	Sorted	Sparse	Yes	One

Type	Based On	Storage Order	Dense/Sparse	Unique Required	Count per Table
Clustered Index	Any column	Sorted	Dense	No	One
Secondary Index	Any non-PK column	Unsorted	Dense	No	Multiple

---

3. B-Tree Indexing

□ **Definition:**

- A **balanced tree data structure** used to implement **multilevel indexes**.
- Commonly used in **relational databases** to improve query performance.

Properties of B-Tree:

1. Balanced tree: all leaf nodes are at the **same level**.
2. Each node has:
   - Multiple **keys**
   - Multiple **pointers/children**
3. Tree **automatically balances** itself on insert/delete.
4. Height of tree is kept **low**, so **search time is log(n)**.

□ **B-Tree Usage:**

- Indexing large databases.
- Handling frequent insert/delete operations.
- File systems and operating systems.

□ **Working of B-Tree Indexing:**

Suppose we insert:

10, 20, 30, 40, 50, 60, 70, ...

A B-Tree of order 3 would create a structure like:

```
     [30]
    /    \
[10,20]  [40,50,60]
```

- Searching for 50? Traverse:
  - Root: 30 → right → node [40,50,60] → found □
- Time complexity: **O(log n)**

B-Tree vs Binary Tree:

Feature	B-Tree	Binary Tree
Children	Multiple	At most 2
Balanced	Always	Not guaranteed
Search Time	log(n)	Up to O(n)
Disk I/O	Optimized	Not optimized
Used In DBMS?	Yes	No

Key Advantages of Indexing

1. **Faster SELECT queries**.
2. **Efficient sorting & filtering**.
3. **Improves performance** of joins and search.
4. Slightly **slower INSERT/DELETE/UPDATE**, due to index updates.
5. Consumes **extra storage**.

1. What is Indexing?

Indexing is a **database optimization technique** used to speed up the retrieval of data from a table. It **reduces the number of disk accesses** during query processing by providing a quicker path to the data.

- **Analogy**: Like the index in a book, it lets you jump directly to the page you need.
- **Use cases**: Particularly helpful in **read-heavy operations** like:
  - SELECT queries
  - Queries using WHERE, JOIN, and sorting (ORDER BY)

---

2. Structure of an Index

An **index** consists of two main components:

- **Search Key**: A copy of the **Primary Key**, **Candidate Key**, or any other column used for searching.
- **Data Reference**: A **pointer** (address) to the actual data block on the disk where the corresponding value is stored.

Note: Indexing provides a **secondary** access path to records (the primary is by row scanning or hashing).

---

3. Properties of Indexing

- Indexing is **optional**, but **improves query speed**.
- **Index files are always sorted** by the search key.
- Indexes **consume extra memory** but reduce I/O cost for read operations.

---

4. Indexing Methods in DBMS

A. Primary Index (a.k.a. Clustering Index)

A **Primary Index** is created when the data file is **sorted** according to the **search key**. This search key could be a **primary key** or a **non-primary key**.

Key Features:

- One primary index per table.
- Data is **physically ordered** based on the search key.
- More efficient since data is already arranged.

 Note: “Primary index” does **not** always mean index on primary key. This is a **common misconception**.

B. Dense Index vs Sparse Index

1. Dense Index:

- **Every** search key value has an entry in the index file.
- Contains:
  - Key value
  - Pointer to the **first** matching record
- Requires **more space**, but lookup is **faster**.

2. Sparse Index:

- **Only some** search key values are stored in the index.
- Each index entry points to a **block** of records.
- Reduces space, but retrieval may need to **scan within block**.

C. Types of Primary Indexing Based on Key

1. Based on Primary Key:

- Data file is sorted based on **primary key**.
- Index uses **primary key as search key**.
- Forms a **sparse index**, i.e., one entry per block.

2. Based on Non-Key Attribute:

- File is sorted on a **non-primary key**.
- Index contains all **unique values** of that attribute.
- Forms a **dense index**, i.e., one entry per unique value.
- Example: Employees sorted by department_id.

D. Multi-Level Index

- Used when the **single-level index** becomes too large.
- The index is split into **multiple levels**.
- Works like a **hierarchical tree** of indices:

  - Level 1 → Level 2 → ... → Actual data
- Reduces time even for binary searches.

---

5. Secondary Index (Non-Clustering Index)

Used when the **data file is unsorted**. In this case, **Primary Indexing cannot be applied**.

Key Features:

- Can be created on **key** or **non-key attributes**.
- Called "secondary" because another index (usually primary) already exists.
- Number of entries = number of records in the data file.
- It is typically a **dense index**.

Example: Indexing Email column in an unsorted employee table.

---

6. B-Tree Index (Conceptual Overview)

While not covered in deep detail in the lecture, a **B-Tree** is commonly used to implement indexing.

B-Tree Characteristics:

- A **balanced tree structure** used in most RDBMSs (like MySQL, PostgreSQL).
- Keeps data sorted and allows searches, sequential access, insertions, and deletions in **logarithmic time**.
- Each node contains multiple keys.
- **Self-balancing**: Ensures the tree does not become skewed.

---

7. Advantages of Indexing

1. **Faster retrieval** of records.
2. Reduces **I/O operations**.
3. Improves performance of:
   - SELECT queries
   - JOINs
   - WHERE and ORDER BY clauses

---

8. Limitations of Indexing

1. **Consumes extra memory** to store index tables.
2. **Slows down write operations**:
    - INSERT: Index must be updated with the new value.
    - DELETE: Index entry must be removed.
    - UPDATE: May need to update the index (especially if indexed column is changed).

Chapter-7: Transaction

1. What is a Transaction?

A **transaction** is a **logical unit of work** in a database that includes one or more operations like read, write, update, or delete.

- Example: Transferring ₹1000 from Account A to Account B involves:
    - Read(A)
    - A = A – 1000
    - Write(A)
    - Read(B)
    - B = B + 1000
    - Write(B)

 These 6 steps together make **one transaction**. Either **all must succeed**, or **none** should.

---

2. ACID Properties

ACID ensures reliable processing of database transactions.

A – Atomicity

- **All-or-nothing**.
- A transaction either **completes fully** or **has no effect**.
- If any part fails, the entire transaction is rolled back.

C – Consistency

- Database moves from **one valid state to another**.
- All integrity constraints must be maintained.

I – Isolation

- Transactions are **executed independently**.

- Intermediate states of one transaction are **invisible** to others.

D – Durability

- Once a transaction is committed, its effects are **permanent**.
- Even in case of a **system crash**, data changes remain.

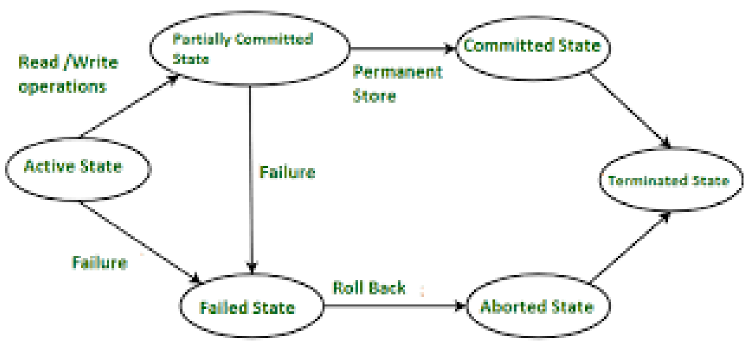
---

3. Transaction States

A transaction passes through the following states:

1. **Active** – Executing its operations.
2. **Partially Committed** – All operations done, but not yet saved to DB.
3. **Committed** – All changes **saved permanently**.
4. **Failed** – If error occurs during execution.
5. **Aborted** – Rolled back to previous consistent state.

 Rollback: Undo all operations.
 Commit: Make all changes permanent.



Transaction States in DBMS

---

4. Schedule

A **schedule** is the **sequence** in which operations of concurrent transactions are executed.

Types:

- **Serial Schedule**:
  - Transactions run **one after another** (no interleaving).
  - Always **correct**, but may be **slow**.
- **Concurrent/Non-Serial Schedule**:
  - Transactions run **in parallel**.

- Need to ensure correctness using **serializability**.

---

5. Conflict Serializability

Used to check if a **non-serial schedule** is **equivalent** to some **serial schedule**.

Two operations conflict if:

- They are from different transactions
- They access the same data
- At least one is a **write**

Steps:

- Build a **precedence graph** (also called conflict graph)
  - Nodes: Transactions
  - Edges: T1 → T2 if T1’s operation conflicts with T2’s and happens before
- **If the graph has no cycle**, schedule is **conflict-serializable**

□ **Acyclic** → Conflict Serializable
□ **Cyclic** → Not Conflict Serializable

---

6. View Serializability

Even if schedules aren’t conflict serializable, they might still be **view serializable**.

View Equivalence Conditions:

Two schedules are view equivalent if:

1. **Same Initial Reads**: For each data item, both schedules must read the same initial value.
2. **Same Final Writes**: For each data item, the final write must be by the same transaction.
3. **Same Reads from Same Writes**: A read must read the value written by the same transaction.

□ **View serializability** is more general than conflict serializability, but harder to test.

---

7. Recoverability

Recoverable schedules ensure that if a transaction **Tj** reads data written by **Ti**, then:

- **Ti must commit before Tj** commits.

This avoids data inconsistency when **Ti fails** but **Tj commits**.

---

8. Cascadelessness

In **cascading schedules**, failure of one transaction leads to **rolling back multiple others** that depended on it.

Cascadeless Schedule:

- Transactions **only read data written by committed transactions**
- Prevents **cascading aborts**

□ Less overhead
□ Easier recovery
□ All cascadeless schedules are recoverable

---

9. Strict Schedule

Stricter than cascadeless.

A schedule is **strict** if:

- **A transaction cannot read or write a data item** until the last transaction that wrote it has **committed or aborted**

Benefits:

- Ensures **recoverability**
- Avoids cascading aborts
- Supports easier recovery using **undo logging**

□ Every strict schedule is also **cascadeless** and **recoverable**

Prathamesh Arvind Jadhav

Chapter-8 : Recovery & Concurrency Control

Recovery & Concurrency Control in DBMS

Recovery Mechanisms

When a **system crash** or **failure** occurs, recovery mechanisms ensure the database is restored to a **consistent state**.

---

1. Log-Based Recovery

- **Log**: A record of all actions (Read, Write, Commit, Abort) performed by transactions.
- Stored on **stable storage**.
- Each log entry has:
    - <Transaction-ID, Data-item, Old-value, New-value>

Two main operations:

- **Undo**: If a transaction is not committed, revert to old values.
- **Redo**: If a transaction is committed, ensure new values are applied.

Types of Logging:

Type	Undo	Redo
Deferred	No	Yes
Immediate	Yes	Yes

---

2. Shadow Paging

- Alternative to logging.
- The **current page table** is never modified directly.
- A **shadow page table** (backup) is maintained.
- **When a transaction commits**, the shadow page is replaced by the current page.

□ Pros:

- No need for undo.

- Simple to implement.

 Cons:

- Costly for large databases.
- Difficult to handle concurrent updates.

---

3. Data Fragmentation

Used in distributed databases. Divides data into **fragments** to improve efficiency and manageability.

Types:

- **Horizontal Fragmentation**: Rows divided into fragments.
- **Vertical Fragmentation**: Columns divided into fragments.
- **Mixed**: Combination of both.

Used in:

- Distributed databases
- Data replication
- Load balancing

Concurrency Control Protocols

Ensure **correct execution** of concurrent transactions (i.e., without violating data integrity or serializability).

---

4. Timestamp Ordering Protocol

Each transaction is assigned a **timestamp (TS)** when it starts.

- **Read_TS(X)** = largest timestamp of a transaction that successfully read X.
- **Write_TS(X)** = largest timestamp of a transaction that successfully wrote X.

Rules:

- If TS(T) < Write_TS(X): **Abort** T (T wants to read outdated data)
- If TS(T) < Read_TS(X): **Abort** T (T wants to write on outdated data)

□ Ensures **serializability**
□ May cause frequent **aborts**

---

5. Thomas Write Rule

Optimization over timestamp ordering protocol.

Allows **out-of-order writes** if they're obsolete:

- If TS(T) < Write_TS(X): **Ignore** the write (obsolete)
- Else if TS(T) < Read_TS(X): **Abort**
- Else: **Proceed** with write

□ Reduces unnecessary aborts
□ Preserves serializability

Lock-Based Protocols

---

6. Two-Phase Locking (2PL)

Two phases:

1. **Growing phase**: Only acquiring locks
2. **Shrinking phase**: Only releasing locks

Once a lock is released, **no new lock can be acquired**.

□ Guarantees **conflict serializability**
□ May lead to **deadlocks**

---

7. Basic 2PL

- Follows two phases strictly
- Locks are held till all operations are done
- Releases locks in shrinking phase

---

8. Conservative 2PL (Static 2PL)

- All locks are acquired **before transaction begins**
- If not available, **waits** without locking anything

□ **Deadlock-free**
□ May cause **unnecessary waiting**

---

9. Strict 2PL

- A transaction **holds all exclusive (write) locks** until it **commits or aborts**
- Prevents **cascading aborts**

□ Strict + 2PL = **Strict Schedule**

---

10. Rigorous 2PL

- Even **shared (read) locks** are held until commit
- Locks are only released at the **end**

□ Stronger than strict 2PL
□ Ensures **serializability + strictness**

---

11. Validation-Based Protocol (Optimistic Concurrency Control)

Used when **conflicts are rare**.

Phases:

1. **Read Phase**: Transaction reads data into local variables.
2. **Validation Phase**: Check for conflicts with other transactions.
3. **Write Phase**: If valid, apply changes to DB.

□ No locks, so **no deadlocks**
□Transactions may be **aborted during validation**

---

12. Multiple Granularity

- Control locking at various **levels of granularity** (e.g., database, table, page, record).

Locking hierarchy:

- Database → Table → Page → Tuple

Lock types:

- **Shared (S)**, **Exclusive (X)**, **Intention Locks**:
  - IS (Intention Shared)
  - IX (Intention Exclusive)
  - SIX (Shared + Intention Exclusive)

 Improves **efficiency**
 Supports **hierarchical locking**

How to Implement Atomicity and Durability in Transactions

---

1. Recovery Mechanism Component of DBMS

- **Recovery mechanisms** ensure **atomicity** (all-or-nothing execution of transactions) and **durability** (once committed, changes persist even after failures).
- These mechanisms help maintain database integrity during system crashes or failures.

---

2. Shadow-Copy Scheme

This is a simple method to guarantee atomicity and durability using copies of the database.

Key points:

- **Assumption:** Only one transaction (T) is active at a time.
- A **db-pointer** is maintained on disk pointing to the current valid copy of the database.
- When a transaction wants to update the DB, it creates a **complete new copy** of the DB.
- All updates are done on this **new copy**, leaving the **old copy (shadow copy) untouched**.
- If the transaction **aborts or fails**, the new copy is deleted, and the old copy remains intact.
- If the transaction **succeeds and commits**:
  1. The OS ensures all pages of the new copy are fully written to disk.
  2. The **db-pointer** is updated to point to this new copy.
  3. The old copy is deleted.
- The transaction is considered **committed** once the updated db-pointer is safely written to disk.

---

3. Atomicity (Using Shadow Copy)

- If transaction **fails before db-pointer update**, the old DB copy is still valid.
- Aborting the transaction means simply deleting the new copy.
- Therefore, **either all changes of T are applied or none at all**, satisfying atomicity.

---

4. Durability (Using Shadow Copy)

- If the system **fails before the db-pointer is updated**, the system restarts using the old DB copy—so no partial changes are visible.
- If the system **fails after the db-pointer update**, the system restarts with the new DB copy, preserving the committed changes.
- Atomic update of db-pointer is crucial; it must be **written atomically** (all or nothing) to disk.
- Usually, disk systems support atomic updates of a sector/block, so db-pointer is stored in a single disk sector to ensure durability.

---

5. Drawbacks of Shadow Copy Scheme

- Very **inefficient**, because the entire database is copied for every transaction, which is expensive in time and space.

---

6. Log-Based Recovery Methods

- **Log:** A sequential record of all transactions and operations, stored in **stable storage**.
- Every operation on the DB is recorded in the log **before** being applied to the database (Write-Ahead Logging).
- Stable storage guarantees atomicity of writes even under system/power failures.

---

7. Deferred DB Modifications

- **Idea:** Record all modifications in the log but **delay actual database writes** until the transaction completes.
- If the system crashes **before T completes or aborts**, log records are ignored (no changes made).
- If T commits, the logged changes are applied to the DB.

- If failure happens during applying these changes, a **redo** operation is performed using the log.

---

8. Immediate DB Modifications

- Changes from an active transaction are written directly to the DB, even **before the transaction completes**.
- These are called **uncommitted modifications**.
- The log keeps **old values** and **new values** for changes.
- If the system crashes or transaction aborts:
  - Use **old values** from log to **undo** changes.
- If the transaction commits and system crashes:
  - Use **new values** from log to **redo** changes after recovery.

---

9. Failure Handling with Immediate Modifications

- If failure happens **before T completes or T aborts**: undo using old value log records.
- If failure happens **after T commits**: redo using new value log records.

Chapter-9 : NoSQL Databases (Not Only SQL)

1. What are NoSQL Databases?

- **NoSQL databases** are **non-tabular** databases, meaning they do **not** store data in traditional relational tables.
- They store data in flexible formats, not rigid tables with rows and columns.
- They come in **four main types based on their data models**:
  - **Document stores** (e.g., MongoDB, CouchDB)
  - **Key-value stores** (e.g., Redis, DynamoDB)
  - **Wide-column stores** (e.g., Cassandra, HBase)
  - **Graph databases** (e.g., Neo4j, Amazon Neptune)
- NoSQL databases are **schema-free** or **schema-flexible**, which means they don't require a fixed schema before inserting data.
- They can easily **scale horizontally** by distributing data across many servers.
- Designed to **handle large volumes of data** and high user loads (Big Data).
- Most NoSQL systems are **open-source** and support **horizontal scaling** (adding more machines instead of just increasing power on one machine).
- They simply store data in formats other than relational tables, like JSON documents or key-value pairs.

---

2. History Behind NoSQL

- NoSQL emerged in the **late 2000s** due to:
  - Dramatic drop in storage costs.
  - Increasing data becoming **unstructured or semi-structured**.
  - Need for faster development cycles and flexibility — developers wanted to avoid complicated schemas.
- Developers became the **primary cost driver** rather than storage costs.
- NoSQL optimizes **developer productivity** by allowing flexible, schema-less data models.
- It was driven by **cloud computing** and distributed applications requiring:
  - Data distribution across multiple servers and regions.
  - Resilience and scalability.
  - Geo-placement of data for better latency and availability.
- Examples like **MongoDB** offer these cloud-friendly features.

---

3. Advantages of NoSQL Databases

A. Flexible Schema

- Unlike RDBMS with fixed schemas, NoSQL allows changing data structure on the fly.
- Useful when all data or future schema changes aren't known upfront.

B. Horizontal Scaling (Scale-Out)

- Adding nodes to handle increasing load (scale-out) is easier than scaling vertically (more powerful hardware).
- Easy distribution because NoSQL data collections are self-contained and less relational.
- Scaling methods:
  - **Sharding:** Splitting data across multiple machines.
  - **Replica-sets:** Copies of data on multiple servers for fault tolerance.

C. High Availability

- Data is replicated automatically across servers.
- If one server fails, data remains accessible from other servers.

D. Fast Insert and Read Operations

- Data often stored in denormalized or pre-joined formats.
- Fewer joins lead to faster queries.
- Example: MongoDB stores related data in the same document for fast retrieval.

- However, **delete and update operations are often more complex**.

E. Caching Mechanism

- Many NoSQL systems have built-in caching for quicker reads.

F. Best suited for Cloud Applications due to scalability and distribution.

---

4. When to Use NoSQL?

- Agile and rapid development environments.
- Handling large volumes of **structured and semi-structured data**.
- When you need **scale-out architecture**.
- Use cases involving **microservices** and **real-time streaming**.
- Applications that require flexible, evolving data models.

---

5. Common Misconceptions About NoSQL

Misconception 1: NoSQL can't handle relationships

- Actually, NoSQL can store relationships, just differently (e.g., nesting related data inside documents).
- Modeling relationships can be easier because data is stored together rather than split across tables.

Misconception 2: NoSQL does not support ACID transactions

- Some NoSQL databases, like **MongoDB**, support ACID transactions.
- It depends on the NoSQL system; many are moving towards better consistency guarantees.

---

6. Types of NoSQL Data Models

6.1 Key-Value Stores

- Simplest form: data stored as key-value pairs (like a dictionary).
- Examples: Redis, Amazon DynamoDB, Oracle NoSQL.
- Use cases: session storage, user preferences, caching.
- Optimized for fast lookups by key.
- Suitable for simple query patterns without complex joins.

6.2 Wide-Column Stores (Columnar Stores)

- Data stored column-wise instead of row-wise.
- Efficient for analytical queries on a subset of columns.
- Examples: Cassandra, HBase, Amazon Redshift.
- Use cases: large-scale analytics, time series data.

6.3 Document Stores

- Store data in documents similar to JSON objects.
- Each document contains key-value pairs where values can be complex types (arrays, nested objects).
- Examples: MongoDB, CouchDB.
- Supports ACID transactions in some implementations.
- Use cases: content management, e-commerce platforms.

6.4 Graph Databases

- Focus on relationships (edges) between entities (nodes).
- Efficient for traversing and querying relationships.
- Examples: Neo4j, Amazon Neptune.
- Use cases: social networks, fraud detection, recommendation engines.

---

7. Disadvantages of NoSQL

- **Data redundancy:** denormalized data leads to storage duplication.
- **Costly updates/deletes:** denormalized data can make these operations complex.
- **No one-size-fits-all:** different NoSQL models serve different use cases, may need multiple DBs.
- **ACID properties:** not fully supported by all NoSQL DBs (though improving).

---

8. SQL vs NoSQL Comparison

Feature	SQL Databases	NoSQL Databases
Data Model	Tables with fixed rows & columns	Documents, key-value, wide-column, graph
Schema	Fixed, predefined	Flexible, schema-free
Scaling	Vertical (scale-up)	Horizontal (scale-out)
ACID Compliance	Strong ACID support	Varies; some support ACID (MongoDB)

Feature	SQL Databases	NoSQL Databases
Joins	Supports complex joins	Typically no joins; denormalized data
Use Cases	Structured data, transactional apps	Big Data, real-time apps, flexible schemas
Examples	Oracle, MySQL, PostgreSQL	MongoDB, Cassandra, Redis, Neo4j
Development History	Since 1970s, focus on reducing duplication	Since late 2000s, focus on scalability & flexibility

Chapter-9 : Types of DataBase

1. Relational Databases

- **Based on:** Relational Model (introduced in the 1970s by E.F. Codd).
- **Concept:** Data is stored in **tables (relations)** made up of rows and columns.
- **Key features:**
    - Tables have rows (records) and columns (fields).
    - Tables can be linked using **foreign keys** (columns that refer to primary keys of other tables).
    - Use **SQL (Structured Query Language)** for data operations like Create, Read, Update, Delete (CRUD).
- **Example:** You might have a Users table with user info, and a Purchases table with purchase records, linked by user IDs.
- **Popular RDBMS:** MySQL, Microsoft SQL Server, Oracle.
- **Advantages:**
    - Highly optimized for **structured data**.
    - Support **data normalization** (reduces redundancy).
    - Well-established querying language (SQL).
    - Ubiquitous with a large user base.
- **Disadvantages:**
    - Difficulty scaling horizontally (across multiple servers).
    - As data grows huge and relationships complex, performance and management can become problematic.

---

2. Object-Oriented Databases (OODB)

- **Based on:** Object-Oriented Programming (OOP) paradigm.
- **Concept:** Data is stored as **objects**, just like in OOP languages (Java, C++).
    - Objects encapsulate data and behavior (methods).
    - Supports **inheritance**, **encapsulation**, **object identity**.
- **Difference from relational databases:** Instead of splitting data into tables, objects contain all relevant data and functions together.

- **Advantages:**
    - Easier and faster to store and retrieve complex data.
    - Can handle complex relationships and more varied data types.
    - Models real-world problems more naturally using OOP concepts.
- **Disadvantages:**
    - More complex to manage, which can lead to slower operations (read/write/update).
    - Less community and industry adoption compared to relational databases.
    - Does not support **views** like relational databases.
- **Examples:** ObjectDB, GemStone.

---

3. NoSQL Databases

- **Meaning:** "Not Only SQL" – they don't rely solely on the relational table model.
- **Types:**
    - Document databases (store JSON-like documents)
    - Key-value stores (simple key-value pairs)
    - Wide-column stores (tables with dynamic columns)
    - Graph databases (nodes and edges to represent relationships)
- **Features:**
    - Schema-free (flexible data structures).
    - Designed for **big data** and **high scalability**.
    - Can scale horizontally across many servers.
    - Usually open-source.
- **Use cases:** Handling large volumes of unstructured or semi-structured data, real-time web apps, big data analytics.
- **Advantages:**
    - Flexible schemas.
    - Easily handle large amounts of data.
    - Good for applications needing fast, scalable storage.
- **Disadvantages:** Lack of standard query language like SQL; consistency models vary.
- **Note:** See LEC-15 notes for more on NoSQL.

---

4. Hierarchical Databases

- **Structure:** Tree-like, hierarchical structure.
- **Concept:** Data organized in a **parent-child** relationship.
    - Each parent can have multiple children.
    - Each child has only one parent.
- **Use case example:** Company departments and employees where employees belong to a single department.
- **Characteristics:**

- Root node at the top, branches represent child nodes.
- Data stored in records and fields; each field stores a single value.
- Data retrieval requires traversal from root to leaves.

- **Advantages:**
  - Simple and fast for one-to-many relationships.
  - Good for fixed, hierarchical data like file systems or organizational charts.
  - Easy to add/delete records without affecting the whole database.
- **Disadvantages:**
  - Inflexible for complex many-to-many relationships.
  - Traversal can be slow due to sequential top-down searching.
  - Data redundancy because relationships cannot be expressed flexibly.
- **Example:** IBM IMS.

---

5. Network Databases

- **Extension of:** Hierarchical databases.
- **Structure:** Organized as a **graph**, allowing many-to-many relationships.
- **Features:**
  - Child records can have multiple parent records.
  - Can represent more complex relationships than hierarchical model.
- **Advantages:**
  - Better at modeling complex relationships.
- **Disadvantages:**
  - Maintenance and management are complex and tedious.
  - Retrieval can be slower due to complex network links.
  - Limited modern community and web support.
- **Examples:** Integrated Data Store (IDS), Integrated Database Management System (IDMS), Raima Database Manager, TurboIMAGE.

Chapter-10: Clustering in DBMS

What is Database Clustering?

- **Database Clustering** is the process of connecting multiple servers or instances together so they operate as a single database system.
- When **one server is not enough** to handle large amounts of data or many user requests, clustering helps distribute the workload.
- Clustering is closely linked to **SQL Server Clustering** because SQL is the language commonly used to manage databases.
- In clustering, the **same dataset is replicated** (copied) on different servers, creating **replica sets**.

Why use Clustering?

Clustering addresses challenges like data volume, request load, and fault tolerance by spreading the load across several servers working together.

Advantages of Database Clustering

1. **Data Redundancy**
    - Clustering creates multiple copies of the same data across different servers.
    - This redundancy is **intentional and synchronized** to avoid anomalies.
    - If one server fails, the same data is accessible from another server, preventing data loss or downtime.
2. **Load Balancing (Scalability)**
    - Clustering distributes incoming requests evenly among all servers.
    - This ensures **no single server is overwhelmed** by traffic.
    - More users and higher spikes in traffic can be supported seamlessly.
    - Load balancing helps maintain **high availability** because the system can handle large or fluctuating workloads.
3. **High Availability**
    - High availability means the database is accessible and operational most of the time.
    - Clustering increases availability by having multiple servers ready to serve data.
    - If one server goes down, another takes over, so the database remains available with minimal or no downtime.
    - The level of availability needed depends on the workload and transaction volume.

How Does Clustering Work?

- In a **cluster architecture**, user requests are split across multiple computers (nodes).
- Each node handles part of the workload; requests are processed by several machines rather than just one.
- This approach enables **load balancing** (distributing work) and **high availability** (reducing downtime).
- If one node fails, the cluster automatically reroutes the request to another functioning node.
- This redundancy **minimizes system failure risks** and provides continuous service.

Chapter-11: Partitioning & Sharding in DBMS (DB Optimization)

---

1. What is Partitioning?

- **Partitioning** is a technique where a large database or large database tables are divided into smaller, manageable pieces called **partitions**.
- Instead of handling the entire huge database table at once, the system works on these smaller partitions.
- These partitions can be handled directly by SQL queries without changing the queries.
- The main goal is to **break down big problems into smaller sub-problems** to improve manageability and performance.
- Partitioning is especially useful for databases with large data volumes or high request rates.

---

2. Why Partition?

- When data becomes too large, managing and processing it becomes slow and inefficient.
- Partitioning allows the system to operate on smaller datasets, improving **query performance** and **system responsiveness**.
- It also makes it easier to scale the database horizontally (adding more servers).
- Partitioning distributes data among servers, which helps in **optimizing resource usage** and improving control over the data.

---

3. Types of Partitioning

a) Vertical Partitioning

- **Vertical partitioning** slices the table **column-wise**.
- Different columns of a table are stored separately, possibly on different servers.
- To retrieve a full record (tuple), you might need to access multiple servers.
- Example: Storing user profile info on one server and login credentials on another.

b) Horizontal Partitioning

- **Horizontal partitioning** slices the table **row-wise**.
- Different rows (tuples) are stored as independent chunks across different servers.
- Each partition contains a subset of the rows.
- Example: Customers from different regions stored on different servers.

---

4. When to Apply Partitioning?

- When the **dataset grows very large**, making it difficult and slow to manage and query.
- When the **number of database requests becomes very high**, causing a single database server to be a bottleneck and leading to high response times.

---

5. Advantages of Partitioning

1. **Parallelism**
   - Different partitions can be processed in parallel, speeding up queries.
2. **Availability**
   - If one partition/server is down, others can still function, improving fault tolerance.
3. **Performance**
   - Queries scan smaller partitions instead of the entire table, making them faster.
4. **Manageability**
   - Easier to maintain and backup smaller partitions than one giant table.
5. **Reduce Cost**
   - Scaling horizontally with partitioning is usually cheaper than vertical scaling (upgrading to more powerful machines).

---

6. Distributed Database

- A **distributed database** is a single logical database spread across multiple physical locations (servers).
- These servers are interconnected via a network and appear as one database to users.
- Distributed databases result from combining optimization techniques such as **clustering, partitioning, and sharding**.
- This is necessary to handle large datasets and high request loads efficiently (see point 4).

---

7. What is Sharding?

- **Sharding** is a way to implement **horizontal partitioning** in a database system.
- Instead of keeping all data on one database instance, data is split across multiple database instances called **shards**.
- A **routing layer** directs queries to the correct shard that holds the requested data.
- This routing layer is essential for the system to know where each piece of data lives.

---

8. Pros and Cons of Sharding

Pros	Cons
Scalability: Supports scaling out by	**Complexity**: Designing partition mapping and routing

Pros	Cons
adding more shards.	logic is complicated.
Availability: Failure of one shard doesn't take down entire system.	**Re-sharding**: Non-uniform data distribution requires periodic re-sharding (redistributing data).
	Not suited for analytical queries: Since data is split, querying across shards (scatter-gather) can be inefficient.

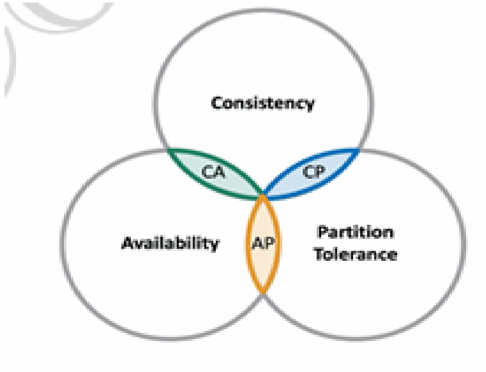
Chapter-12: CAP Theorem

What is CAP Theorem?

- **CAP Theorem** is a fundamental concept in **Distributed Databases**.
- It helps design efficient distributed systems by understanding the trade-offs between three critical properties.

Breakdown of CAP

1. **Consistency (C)**
    - All nodes see the same data at the same time.
    - After a write operation, any read operation from any node returns the most recent write.
    - Ensures that all users, regardless of which node they connect to, get identical and up-to-date data.
    - Data is replicated across nodes after writing to one node.
2. **Availability (A)**
    - The system always responds to every request, even if some nodes fail.
    - Every request gets a response (not necessarily the latest data).
    - The system continues to operate despite node failures or downtime.
3. **Partition Tolerance (P)**
    - The system continues to function despite communication breakdowns between nodes (network partitions).
    - The system can handle message loss or delays between nodes without failing.
    - Requires data replication across nodes and networks.



What Does the CAP Theorem State?

- **A distributed system can guarantee only two out of the three properties (Consistency, Availability, Partition Tolerance) at the same time.**
- It forces trade-offs when a partition (network failure) happens: you can have either consistency or availability, but not both.

CAP Theorem and NoSQL Databases

- NoSQL databases are designed for distributed environments, allowing horizontal scaling across nodes.
- When choosing a NoSQL database, CAP considerations are critical.

Types of Databases in Terms of CAP

Type	Guarantees	Description	Examples
CA (Consistency + Availability)	Consistency and Availability, no Partition Tolerance	Works well without network failures. Not fault tolerant for partitions.	Some RDBMS like MySQL, PostgreSQL (with replication)
CP (Consistency + Partition Tolerance)	Consistency and Partition Tolerance, no Availability	During partition, some nodes might become unavailable to maintain consistency.	MongoDB (NoSQL)
AP (Availability + Partition Tolerance)	Availability and Partition Tolerance, no Consistency	System remains available during partitions but may return stale data. Eventually consistent.	Apache Cassandra, Amazon DynamoDB

Practical Examples

- **CP databases** (like MongoDB) are suitable when **consistency is critical**, such as banking systems, where accurate and up-to-date data is vital. The system may sacrifice availability temporarily during network issues to maintain data correctness.
- **AP databases** (like Cassandra) prioritize **availability**, used in social networks like Facebook, where data can be eventually consistent, but users expect the system to be always accessible.

The Master-Slave Database Concept (Brief Overview)

1. **Purpose:**
    - To optimize database IO under heavy loads when a single DB server can't handle all requests efficiently.
2. **Pattern:**
    - Part of Database Scaling Pattern known as **Command Query Responsibility Segregation (CQRS)**.
3. **How it works:**
    - The **Master** DB holds the true, latest data and handles **write operations**.
    - **Slave** DBs handle **read operations** only.
    - This separation improves availability, reduces latency, and distributes load.
4. **Replication:**
    - Data is replicated from Master to Slaves.
    - Replication can be **synchronous** (waits for slaves to confirm writes) or **asynchronous** (writes immediately return without waiting).