

## Unit-I

### Introduction to DBMS give detailed notes

## 1. What is a Database?

A **database** is an organized collection of related data stored in a structured format so that it can be easily accessed, managed, and updated.

### **Example:**

A **student database** may include data about:

- Student ID
  - Name
  - Course
  - Marks
- 

## ◆ 2. What is DBMS?

A **Database Management System (DBMS)** is a **software system** that allows users to **create, maintain, manipulate, and control access** to databases.

### **Functions of DBMS:**

- Data Storage
  - Data Retrieval (Queries)
  - Data Manipulation (Insert, Update, Delete)
  - Data Security and Authorization
  - Backup and Recovery
  - Data Integrity Enforcement
  - Concurrent Data Access
- 

## ◆ 3. Features of DBMS

- **Data Independence:** Changes in storage structure do not affect the application.
  - **Minimal Redundancy:** Prevents duplication of data.
  - **Data Consistency:** Same data across all instances.
  - **Security:** Access controls and permissions.
  - **Backup and Recovery:** Automatic data recovery in case of failure.
  - **Multi-User Access:** Supports multiple users simultaneously.
-

## ◆ 4. Advantages of DBMS

Feature	Description
<b>Reduced Redundancy</b>	Prevents duplicate data
<b>Improved Integrity</b>	Enforces accuracy with constraints
<b>Increased Security</b>	Access control at user and data level
<b>Concurrent Access</b>	Multiple users can use the system simultaneously
<b>Backup &amp; Recovery</b>	Automated tools for data protection

---

## ◆ 5. Disadvantages of DBMS

- High initial cost of hardware/software
  - Complex setup and maintenance
  - Requires trained personnel
  - Performance overhead in large systems
- 

## ◆ 6. Components of DBMS

Component	Description
<b>Hardware</b>	Physical devices where data is stored (e.g., servers)
<b>Software</b>	The DBMS software itself (e.g., MySQL, Oracle)
<b>Data</b>	The actual data and metadata (data about data)
<b>Users</b>	People who interact with the database
<b>Procedures</b>	Instructions and rules for using the DBMS effectively

---

## ◆ 7. Types of Database Users

User Type	Role/Function
<b>Database Administrator (DBA)</b>	Manages and maintains the database system
<b>Application Programmers</b>	Develop programs that interact with the DBMS
<b>End Users</b>	Query the database through applications
<b>System Analysts</b>	Design the database based on requirements

---

## ◆ 8. Database Models

Model Type	Description	Example Use
<b>Hierarchical Model</b>	Data organized in tree structure (parent-child)	File systems

Model Type	Description	Example Use
<b>Network Model</b>	Data as graph with many-to-many relationships	Telecom DBs
<b>Relational Model</b>	Data stored in tables (most widely used)	MySQL, Oracle
<b>Object-Oriented Model</b>	Data as objects (OOP-based)	Multimedia DBs

## ◆ 9. DBMS vs RDBMS

Feature	DBMS	RDBMS
Data Storage	Files	Tables (Relations)
Relationships	Not supported	Supported through keys
Integrity Constraints	Not enforced	Enforced using primary/foreign keys
Examples	XML DB, File System	MySQL, PostgreSQL, Oracle

## ◆ 10. DBMS Languages

Language	Purpose	Example
<b>DDL</b> (Data Definition Language)	Defines structure	<code>CREATE TABLE students (...)</code>
<b>DML</b> (Data Manipulation Language)	Manipulates data	<code>INSERT INTO students VALUES (...)</code>
<b>DCL</b> (Data Control Language)	Grants/revokes access	<code>GRANT SELECT ON students TO user;</code>
<b>TCL</b> (Transaction Control Language)	Controls transactions	<code>COMMIT; , ROLLBACK;</code>

## ◆ 11. Keys in DBMS

Key Type	Description
<b>Primary Key</b>	Uniquely identifies each record
<b>Foreign Key</b>	Refers to primary key in another table
<b>Candidate Key</b>	A field that can serve as a primary key
<b>Composite Key</b>	Combination of two or more fields as a key
<b>Super Key</b>	A set of attributes that uniquely identifies

## ◆ 12. ACID Properties (for Transactions)

Property	Description
<b>Atomicity</b>	All operations in a transaction are completed or none
<b>Consistency</b>	Data remains in a valid state

Property	Description
<b>Isolation</b>	Transactions do not interfere with each other
<b>Durability</b>	Once committed, changes are permanent

---

## ◆ 13. Example: Simple Student Table

```
sql
CopyEdit
CREATE TABLE Students (
    RollNo INT PRIMARY KEY,
    Name VARCHAR(50),
    Course VARCHAR(30),
    Marks INT
);

INSERT INTO Students VALUES (101, 'Alice', 'DBMS', 85);
SELECT * FROM Students;
```

---

## ◆ 14. Popular DBMS Software

- **MySQL** (Open Source, Relational)
- **Oracle** (Enterprise RDBMS)
- **Microsoft SQL Server** (Windows RDBMS)
- **PostgreSQL** (Open Source, Object-relational)
- **MongoDB** (NoSQL, Document-based)

### 2. File system vs DBMS in dbms give detailed notes

#### ✓ 1. What is a File System?

A **file system** is a method of **storing and organizing files** on a storage medium like hard disks. Data is managed through **files created manually** using programming languages (e.g., C, C++, Java).

#### ◆ Characteristics:

- Data is stored in files (.txt, .csv, etc.)
  - Access is done via custom application code
  - No direct support for transactions, concurrency, or data integrity
- 

#### ✓ 2. What is a DBMS?

A **Database Management System (DBMS)** is software that helps in **storing, managing, and retrieving** structured data in databases, using a standard language like **SQL**.

◆ **Characteristics:**

- Data is stored in **tables** (relations)
- Supports querying, data integrity, and multi-user access
- Maintains **ACID** properties for transactions

---

### ✓ 3. Key Differences Between File System and DBMS

Feature	File System	DBMS
<b>Data Storage Format</b>	Data stored in flat files	Data stored in structured tables
<b>Data Redundancy</b>	High redundancy	Controlled redundancy via normalization
<b>Data Consistency</b>	Difficult to ensure	Ensured with integrity constraints
<b>Data Security</b>	Limited access control	Granular user access and encryption
<b>Concurrency Control</b>	Not handled well	Handled efficiently with locking mechanisms
<b>Data Integrity</b>	Must be manually implemented	Built-in support using constraints
<b>Backup &amp; Recovery</b>	Manual and complex	Automated backup and recovery tools
<b>Transaction Support</b>	Not supported	Fully supported (ACID properties)
<b>Querying Language</b>	No standard query language	SQL for data manipulation and retrieval
<b>Scalability</b>	Limited	Highly scalable for large databases
<b>Data Relationships</b>	Hard to manage	Easily managed using keys and constraints
<b>Cost</b>	Low initial cost	Higher setup cost but more powerful
<b>Examples</b>	Text files, CSV files	MySQL, Oracle, PostgreSQL, MongoDB

---

### ✓ 4. Example: Student Records

◆ **In File System:**

Data is stored manually in a `.txt` or `.csv` file:

```
CopyEdit
101, Alice, CS, 85
102, Bob, IT, 90
```

- Updating or querying specific students requires manual coding.

### ◆ In DBMS:

Data is stored in a structured table:

```
sql
CopyEdit
CREATE TABLE Students (
    RollNo INT PRIMARY KEY,
    Name VARCHAR(50),
    Course VARCHAR(30),
    Marks INT
);

INSERT INTO Students VALUES (101, 'Alice', 'CS', 85);
SELECT * FROM Students WHERE Marks > 80;
```

- Easy to retrieve and update using SQL.
- 

## ✓ 5. Limitations of File System

- No **standardized query language**
  - No **centralized control** over data
  - No **multi-user support**
  - Error-prone and difficult to maintain in large applications
- 

## ✓ 6. Advantages of DBMS over File System

Advantage	Explanation
<b>Less Redundancy</b>	Normalization eliminates duplicate data
<b>Improved Integrity</b>	Constraints ensure valid and accurate data
<b>Concurrent Access</b>	Multiple users can work simultaneously
<b>Security &amp; Authorization</b>	Roles and permissions safeguard sensitive data
<b>Data Abstraction</b>	Users interact without knowing physical storage
<b>Efficient Querying</b>	SQL simplifies data access and manipulation

---

## ✓ 7. When to Use What?

Situation	Recommended System
Small application with simple data needs	File System
Multi-user, large-scale enterprise system	DBMS

Situation	Recommended System
Needs data integrity and security	DBMS
Quick scripts or temporary storage	File System

---

## ✓ Conclusion

- **File systems** are suitable for simple data storage and small-scale applications.
- **DBMS** is the preferred choice for complex, secure, multi-user environments requiring efficient data handling.

### 3. Advantages of DBMS

A **DBMS (Database Management System)** is a software suite that helps in efficient **storage, retrieval, and management of data**. It resolves the limitations of traditional file systems and offers several significant benefits.

---

## ✓ 1. Data Redundancy Control

### ◆ Explanation:

In traditional file systems, the same data may be repeated in multiple files. DBMS minimizes this **redundancy** by centralizing data storage and enabling data sharing.

### ◆ Example:

In a file system, student address may be stored in multiple files (admission, examination, fee). In DBMS, it is stored once in a `Students` table and referenced wherever needed.

---

## ✓ 2. Data Consistency

### ◆ Explanation:

Reduced redundancy leads to improved **consistency**. If data is updated in one place, it reflects everywhere.

### ◆ Example:

If a student's name changes, updating it in the DBMS reflects across all modules (results, fee records, etc.).

---

## ✓ 3. Data Integrity

### ◆ Explanation:

DBMS uses **constraints** and **rules** to maintain data accuracy and validity.

### ◆ Common Constraints:

- **Primary Key** – Ensures unique identification
- **Foreign Key** – Maintains referential integrity
- **Check** – Restricts values in a column

---

## ✓ 4. Data Security

### ◆ Explanation:

DBMS allows **access control** through user authentication and authorization. Sensitive data is protected from unauthorized access.

### ◆ Example:

- Admin can access and modify all data
- Students may only view their records

---

## ✓ 5. Concurrent Access

### ◆ Explanation:

Multiple users can access the database **simultaneously** without affecting each other's work. DBMS handles this through **locking and transaction control**.

### ◆ Example:

Two users booking tickets at the same time will not overwrite each other's transactions.

---

## ✓ 6. Data Abstraction



### ◆ Explanation:

DBMS separates **logical data structure** from **physical storage**, enabling users to access data without knowing how it is stored.

### ◆ Levels:

- **Physical Level** – How data is stored
  - **Logical Level** – What data is stored
  - **View Level** – How data is presented
- 

## ✓ 7. Backup and Recovery

### ◆ Explanation:

DBMS provides automatic tools to create **backups** and **restore data** in case of failure (e.g., system crash or power outage).

### ◆ Example:

A scheduled backup can help recover data lost during hardware failure.

---

## ✓ 8. Transaction Management (ACID Properties)

### ◆ Explanation:

DBMS maintains **ACID properties** for reliable transactions:

- **Atomicity** – All or none execution
  - **Consistency** – Valid data at all times
  - **Isolation** – Independent transactions
  - **Durability** – Changes remain after a commit
- 

## ✓ 9. Reduced Application Development Time

### ◆ Explanation:

With **SQL** and **predefined operations**, developers can manage data efficiently without writing large volumes of code.

### ◆ Example:

`SELECT * FROM Employees WHERE Salary > 50000;` retrieves data in one line, unlike complex file handling in traditional systems.

---

## ✓ 10. Data Sharing

### ◆ Explanation:

Centralized databases allow **multiple departments or users** to share data securely and efficiently.

### ◆ Example:

Sales and inventory departments accessing common product data from the same DB.

---

## ✓ 11. Scalability and Flexibility

### ◆ Explanation:

Modern DBMSs are highly **scalable** to handle growing data and **flexible** to adapt to changing requirements.

---

## ✓ 12. Improved Decision-Making

### ◆ Explanation:

Reliable and well-organized data enables businesses to generate **reports, analytics, and insights** for better decision-making.

### ◆ Example:

Management dashboards pulling data from DBMS for monthly sales reports.

---

## Summary Table

Advantage	Description
Data Redundancy Control	Avoids duplication of data
Data Consistency	Same data across all modules
Data Integrity	Enforces valid and accurate data
Security	User roles and permissions
Concurrent Access	Multi-user environment
Abstraction	Hides data storage complexity
Backup & Recovery	Data protection in failures
Transaction Control	Maintains data reliability
Faster Development	SQL and tools simplify programming
Data Sharing	Enables inter-departmental collaboration
Scalability	Handles growing volumes of data
Better Decision Making	Accurate data for business insights

---

## Conclusion

The **advantages of DBMS** make it the backbone of data management in almost every modern organization. It not only ensures the **accuracy, availability, and security** of data but also supports powerful operations and applications.

---

## 4. Database architecture

### ✓ 1. What is Database Architecture?

**Database architecture** defines the **logical and physical structure** of a database system, including how data is stored, accessed, and managed.

It describes the:

- **Components** of the database system
  - **Relationship between users and the system**
  - **Levels of data abstraction**
- 

### ✓ 2. Types of Database Architecture

There are **three major types** of database architectures:

Architecture Type	Description
1-Tier	All operations occur on a single machine

Architecture Type	Description
2-Tier	Client-server architecture
3-Tier	Middleware layer between client and server

---

## ✓ 3. 1-Tier Architecture

### ◆ Description:

- The **user interacts directly with the database**.
- Mainly used for **local applications** or during development.

### ◆ Example:

- Using **MS Access** on a standalone PC.

### ◆ Features:

- Simple and fast
  - Less secure and not scalable
- 

## ✓ 4. 2-Tier Architecture

### ◆ Description:

- Divides system into:
  - **Client:** UI for user interaction
  - **Server:** Hosts the database
- The **application** runs on the client and communicates directly with the database.

### ◆ Example:

- Java or .NET applications connecting to MySQL or Oracle DB.

### ◆ Features:

- Faster communication
  - Tight coupling between client and server
  - Limited scalability
-

## ✓ 5. 3-Tier Architecture

### ◆ Description:

- Consists of **three layers**:
  1. **Presentation Tier (Client)**: User interface
  2. **Application Tier (Middleware)**: Business logic
  3. **Data Tier (Database Server)**: Database storage

### ◆ Features:

- Highly **scalable, secure, and maintainable**
- Supports **distributed applications**

### ◆ Example:

- Web applications using:
    - **Front-end** (HTML/React)
    - **Back-end** (Java/.NET/PHP)
    - **Database** (MySQL/Oracle/PostgreSQL)
- 

## ✓ 6. 3-Level Database Architecture (ANSI-SPARC Model)

This model is defined by **ANSI/SPARC** to support data abstraction and independence.

### ◆ The 3 levels:

Level	Description
<b>External Level</b>	User view (individual user perspective)
<b>Conceptual Level</b>	Community view (overall logical structure of the database)
<b>Internal Level</b>	Physical storage view (how data is stored on hardware)

---

### ◆ a. External Level (View Level)

- Closest to **end-users**
- Multiple user-specific views possible
- Users only see relevant data

*Example:*

A student user may only see personal records, not fee details.

---

### ◆ b. Conceptual Level (Logical Level)

- Defines **logical structure**: entities, relationships, constraints
- Hides physical storage details
- **Single view for entire database**

*Example:*

Defines the schema: `Student(RollNo, Name, Course, Marks)`

---

### ◆ c. Internal Level (Physical Level)

- Describes **how data is stored**
- Includes indexes, data blocks, file structures

*Example:*

Data is stored in binary files on disk with indexing for fast access.

---

## ✓ 7. Data Independence

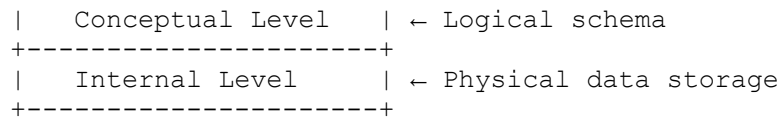
This architecture provides **data independence**, meaning changes in one level don't affect others:

Type	Description
<b>Logical Independence</b>	Changes in conceptual level don't affect external views
<b>Physical Independence</b>	Changes in internal level don't affect conceptual schema

---

## ✓ 8. Diagram: 3-Level Database Architecture

```
pgsql
CopyEdit
+-----+
|   External Level   | ← User Views
+-----+
```



## ✓ 9. Benefits of Layered Architecture

- **Data Abstraction:** Users interact with simplified views
  - **Security:** Different views for different users
  - **Data Independence:** Easy to make changes without affecting the system
  - **Modularity:** Each layer can be developed or maintained independently
  - **Scalability:** Especially in 3-tier architecture
- 

## 📊 Summary Table: Types of Database Architecture

Type	Tiers	Usage	Scalability	Examples
1-Tier	1 (User + DB)	Local apps, prototypes	Low	MS Access, SQLite
2-Tier	Client + DB	Enterprise apps	Moderate	Java + Oracle/MySQL
3-Tier	Client + App + DB	Web apps, cloud systems	High	React + Node.js + PostgreSQL

---

## 🎓 Conclusion

Database architecture plays a critical role in:

- **System performance**
- **Security**
- **Maintainability**
- **Scalability**

## 5.Data model

### ✓ 1. What is a Data Model?

A **data model** in a DBMS defines how data is **logically structured**, **stored**, and **manipulated**. It serves as a **blueprint** for designing a database.

#### ◆ It provides:

- A framework to organize data

- Rules to define relationships
  - Means for querying and updating data
- 

## ✓ 2. Purpose of Data Models

- Describe **data**, **data relationships**, and **constraints**
  - Enable **data abstraction**
  - Facilitate **database design**
  - Help maintain **data consistency** and **integrity**
- 

## ✓ 3. Types of Data Models

Data models are mainly categorized into the following:

Category	Data Model Types
<b>High-Level (Conceptual)</b>	Entity-Relationship (ER) Model
<b>Record-Based (Logical)</b>	Relational, Network, Hierarchical Models
<b>Physical Model</b>	Describes actual data storage
<b>Object-Based Model</b>	Object-Oriented Model

---

## ✓ 4. Entity-Relationship (ER) Model

### ◆ Description:

- A **high-level** conceptual model
- Represents data as **entities** (objects) and **relationships** between them

### ◆ Components:

Element	Description	Example
Entity	Real-world object	Student, Course
Attribute	Property of an entity	Name, Age
Relationship	Association between entities	Enrolled, Teaches

### ◆ Diagram Example:

```
css
CopyEdit
[Student] ---Enrolled---> [Course]
```

---



## ✓ 5. Relational Data Model

### ◆ Description:

- Most widely used model in modern DBMS
- Represents data in **tables (relations)**
- Each table has **rows (tuples)** and **columns (attributes)**

### ◆ Example Table: `student`

RollNo	Name	Age	Course
101	Alice	20	B.Sc
102	Bob	21	BCA

### ◆ Key Features:

- Uses **primary key** to identify records
  - **Foreign keys** represent relationships between tables
  - Manipulated using **SQL**
- 

## ✓ 6. Hierarchical Data Model

### ◆ Description:

- Data is organized in a **tree-like** structure
- Each parent can have **multiple children**, but each child has **one parent**

### ◆ Example:

```
css
CopyEdit
[University]
├── [College of Science]
│   ├── [Dept of Physics]
│   └── [Dept of Chemistry]
```

### ◆ Features:

- Fast for **1-to-many** relationships
  - Difficult to model complex relationships
  - Used in legacy systems (e.g., IBM IMS)
-

## ✓ 7. Network Data Model

### ◆ Description:

- Data is represented as **records and relationships** using a **graph** structure
- A child can have **multiple parents**

### ◆ Example:

- Student enrolled in multiple courses
- Courses taught by multiple instructors

### ◆ Features:

- More flexible than hierarchical
  - Complex to implement and navigate
  - Replaced by relational model in modern DBMS
- 

## ✓ 8. Object-Oriented Data Model

### ◆ Description:

- Combines database capabilities with **object-oriented programming**
- Data and its operations (methods) are stored together

### ◆ Features:

- Supports **inheritance, encapsulation, and polymorphism**
  - Good for applications like **CAD, multimedia databases**
- 

## ✓ 9. Physical Data Model

### ◆ Description:

- Describes **how data is actually stored** in memory (files, indexes, blocks)
- Focuses on **performance and efficiency**

### ◆ Includes:

- File organization (heap, sorted, hash)
- Index structures (B-tree, hash index)
- Disk storage details

---

## ✔ 10. Comparison Table: Common Data Models

Model	Structure	Relationship Type	Usage
ER Model	Diagram-based	Conceptual	Database design
Relational Model	Tables (Relations)	Logical	SQL-based DBMS (MySQL, Oracle)
Hierarchical Model	Tree	1-to-many	Legacy systems
Network Model	Graph	Many-to-many	Complex data with multiple links
Object-Oriented	Objects	Real-world mapping	Multimedia, CAD, OOP databases
Physical Model	Files, Blocks	Hardware-based	Performance optimization

---

## ✔ 11. Advantages of Using Data Models

Benefit	Explanation
<b>Data Abstraction</b>	Hides complex details from users
<b>Better Design</b>	Helps in structured database development
<b>Improved Consistency</b>	Ensures data relationships are maintained
<b>Standardization</b>	Enables use of SQL and design tools
<b>Maintainability</b>	Easier to manage and update schemas

---

## 🎓 Conclusion

A **data model** is a critical component in the design and management of databases. The choice of data model depends on:

- **Type of data**
- **Application domain**
- **System requirements**

**Relational models** dominate today, but **object-oriented and conceptual models** are also important in modern applications.

---

## 6. Schema and instances

### ✔ \*\*1. What is a Database Schema?

A **schema** is the **logical structure** or **blueprint** of a database. It defines how data is organized and how the relations among them are associated.

---

### ◆ Key Points:

- Specifies the **structure** of tables, relationships, constraints, views, indexes, etc.
  - Defined using **Data Definition Language (DDL)** (e.g., `CREATE`, `ALTER`)
  - Schema remains **relatively static** (changes rarely)
- 

### ◆ Example of a Schema:

```
sql
CopyEdit
CREATE TABLE Student (
    RollNo INT PRIMARY KEY,
    Name VARCHAR(50),
    Age INT,
    Course VARCHAR(30)
);
```

Here, `Student` is a table schema with columns `RollNo`, `Name`, `Age`, and `Course`.

---

### ◆ Types of Schema:

Type	Description
<b>Physical Schema</b>	Describes how data is physically stored on disk
<b>Logical Schema</b>	Describes tables, views, relationships, constraints (what users see)
<b>External Schema</b>	Describes user-specific views (used in 3-level architecture)

---

## ✓ 2. What is an Instance in DBMS?

An **instance** of a database is the **actual data stored** in the database **at a particular moment** in time.

---

### ◆ Key Points:

- Represents the **current state** of the database
- **Changes frequently** as data is added, deleted, or modified
- Also called a **snapshot** of the database

---

### ◆ Example:

For the `Student` table, an instance may look like:

RollNo	Name	Age	Course
101	Alice	20	BCA
102	Bob	21	B.Sc

This table content is an **instance** of the `Student` schema.

---

## ✓ 3. Analogy: Schema vs. Instance

Analogy	Schema	Instance
<b>Blueprint vs Building</b>	Blueprint of a building	Actual building at a moment
<b>Class vs Object (OOP)</b>	Class definition	Object created from that class
<b>Form vs Filled Form</b>	Blank form with labels	Filled-in form with data

---

## ✓ 4. Schema vs. Instance – Key Differences

Feature	Schema	Instance
Definition	Logical structure of the database	Actual data in the database
Nature	Static (changes rarely)	Dynamic (changes frequently)
Language Used	Defined using <b>DDL</b>	Modified using <b>DML</b> (e.g., INSERT, UPDATE)
Example	Table definition with columns and types	Actual rows in the table
Lifetime	Exists as long as the database exists	Exists temporarily and changes over time

---

## ✓ 5. Practical Scenario

### □ Schema:

```
sql
CopyEdit
CREATE TABLE Product (
    ProductID INT,
    Name VARCHAR(50),
    Price DECIMAL(8,2)
```

);

## Instance:

ProductID	Name	Price
1	Laptop	55000.00
2	Smartphone	20000.00

The above records are **instances** of the `Product` **schema**.

---

## ✓ 6. Why Are Schema and Instance Important?

### Importance of Schema

Ensures **database design consistency**

Helps in **validating** data entries

Useful for **data modeling** and architecture

### Importance of Instance

Shows **real-time data** in the database

Affects **performance** and **storage**

Crucial for **transactions** and reporting

---

## ✓ 7. Visual Representation

pgsql  
CopyEdit  
SCHEMA (Blueprint)

STUDENT
RollNo
Name
Age
Course

INSTANCE (Snapshot at a time)

RollNo	Name
101	Ram
102	Rita

---

## Conclusion

- A **schema** defines the structure of the database, while an **instance** represents the current data.
- Understanding the distinction is crucial for database **design, maintenance, and operations**.

---

## 7.Data independence

### ✓ 1. What is Data Independence?

**Data Independence** refers to the **capacity to change the schema** at one level of a database system **without altering** the schema at the next higher level.

It allows the **separation of data** from the **applications** that use the data.

---

### ✓ 2. Levels of Database Architecture

Data independence is achieved through the **three-schema architecture** of a DBMS:

Level	Description
<b>External Level</b>	User views (subset of the database)
<b>Logical Level</b>	Logical structure (tables, relationships)
<b>Physical Level</b>	How data is stored (files, indexes, etc.)

---

### ✓ 3. Types of Data Independence

There are **two main types**:

#### ◆ A. Logical Data Independence

- The ability to **change the logical schema** (e.g., adding/removing tables, fields, relationships) **without changing** the external views or application programs.

📖 *Example:*

Adding a new column `PhoneNumber` to the `Student` table should **not require** changes to user applications that don't use this column.

✓ *Benefits:*

- Applications remain unaffected by structural changes
  - Easier to evolve the database design
-

◆ **B. Physical Data Independence**

- The ability to **change the physical storage** or data access methods **without affecting** the logical structure of the database.

📖 *Example:*

Changing the storage from sequential files to indexed files should not affect the logical view of tables and relations.

✓ *Benefits:*

- Improved performance without disrupting applications
- Easier to implement optimizations and storage updates

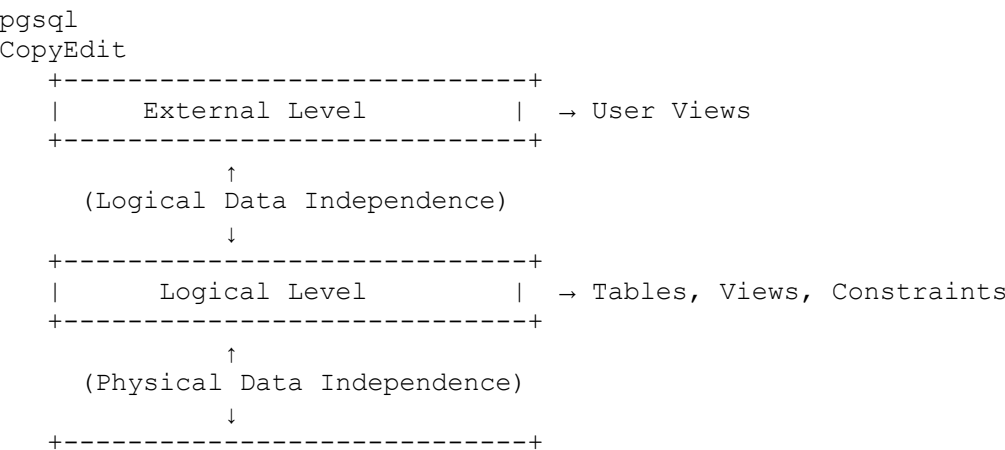
---

✓ **4. Importance of Data Independence**

Benefit	Explanation
Maintainability	Easier to make changes to the database design
Flexibility	Application programs do not depend on data storage structure
Reduced Complexity	Developers can focus on business logic instead of low-level details
Improved Data Security	Data access can be controlled separately from data storage details
Better Evolution Handling	Supports database evolution and changes over time without breaking apps

---

✓ **5. Diagram: Data Independence and Schema Levels**





|      Physical Level      |   → Storage, Indexes, Files  
+-----+  

---

## ✓ 6. Example Scenario

Let's say you have this table:

```
sql
CopyEdit
CREATE TABLE Employee (
    EmpID INT,
    Name VARCHAR(50),
    Salary INT
);
```

### Logical Change:

You add a new column:

```
sql
CopyEdit
ALTER TABLE Employee ADD Department VARCHAR(30);
```

This **should not affect** user applications unless they use the new column → **Logical Data Independence**.

### Physical Change:

You move the table to a different storage device or use a new index → The **application remains unchanged** → **Physical Data Independence**.

---

## ✓ 7. Challenges in Achieving Data Independence

- **Logical Data Independence** is **harder to achieve** than physical because:
    - Applications often depend on logical schema (table names, field names)
    - Queries need to be updated if schema changes are major
  - **Physical Data Independence** is more commonly achieved and supported by modern DBMSs.
- 

## ✓ 8. Real-Life Analogy

Analogy	Physical Independence	Logical Independence
---------	-----------------------	----------------------

Reading a book	Changing paper to Kindle	Rewriting chapters but keeping summary unchanged
----------------	--------------------------	--

Using a website Changing backend server Changing data fields shown

---

## Conclusion

- **Data independence** ensures **database flexibility, robustness, and ease of maintenance**.
- It is a key benefit of using a DBMS over traditional file systems.
- While **physical independence** is relatively easy to implement, **logical independence** remains more challenging but essential.

## 8. Database languages

### ✓ 1. What are Database Languages?

Database languages are a set of commands and syntax used to **define, manipulate, and control** data in a Database Management System (DBMS).

They enable users and applications to interact with the database efficiently.

---

### ✓ 2. Types of Database Languages

Language Type	Purpose	Examples of Commands
<b>Data Definition Language (DDL)</b>	Define and modify database structure	CREATE, ALTER, DROP
<b>Data Manipulation Language (DML)</b>	Retrieve and manipulate data	SELECT, INSERT, UPDATE, DELETE
<b>Data Control Language (DCL)</b>	Control access and permissions	GRANT, REVOKE
<b>Transaction Control Language (TCL)</b>	Manage transactions and their execution	COMMIT, ROLLBACK, SAVEPOINT
<b>Query Language</b>	Retrieve data based on conditions	SELECT

---

### ✓ 3. Data Definition Language (DDL)

◆ **Purpose:**

- Used to **create, alter, and remove** database objects like tables, indexes, views, schemas, etc.

#### ◆ Common Commands:

Command	Description	Example
CREATE	Create a new table or database object	<code>CREATE TABLE Student (ID INT, Name VARCHAR(20));</code>
ALTER	Modify existing database object	<code>ALTER TABLE Student ADD Age INT;</code>
DROP	Delete tables or other objects	<code>DROP TABLE Student;</code>
TRUNCATE	Remove all records from a table	<code>TRUNCATE TABLE Student;</code>

---

## ✓ 4. Data Manipulation Language (DML)

#### ◆ Purpose:

- Used to **retrieve, insert, update, and delete** data from the database.

#### ◆ Common Commands:

Command	Description	Example
SELECT	Retrieve data from tables	<code>SELECT * FROM Student WHERE Age &gt; 18;</code>
INSERT	Insert new rows into a table	<code>INSERT INTO Student VALUES (1, 'Alice', 20);</code>
UPDATE	Modify existing data	<code>UPDATE Student SET Age = 21 WHERE ID = 1;</code>
DELETE	Delete rows from a table	<code>DELETE FROM Student WHERE ID = 1;</code>

---

## ✓ 5. Data Control Language (DCL)

#### ◆ Purpose:

- Used to **grant or revoke** user permissions on database objects.

#### ◆ Common Commands:

Command	Description	Example
GRANT	Give user privileges	<code>GRANT SELECT ON Student TO user1;</code>
REVOKE	Remove user privileges	<code>REVOKE SELECT ON Student FROM user1;</code>

---

## ✓ 6. Transaction Control Language (TCL)

### ◆ Purpose:

- Manage **transactions** which are sequences of operations performed as a single unit.

### ◆ Common Commands:

Command	Description	Example
COMMIT	Save all changes made in the transaction	COMMIT;
ROLLBACK	Undo changes made in the current transaction	ROLLBACK;
SAVEPOINT	Set a point within a transaction to rollback to	SAVEPOINT sp1;
SET TRANSACTION	Set transaction properties	SET TRANSACTION READ ONLY;

---

## ✓ 7. Query Language

### ◆ Purpose:

- Primarily used for **data retrieval** using the `SELECT` statement.
- Supports filtering, sorting, grouping, and joining data.

### ◆ Example:

```
sql
CopyEdit
SELECT Name, Age FROM Student WHERE Age > 18 ORDER BY Age DESC;
```

---

## ✓ 8. Summary Table of Database Languages

Language Type	Main Function	Common Commands
DDL	Define database structure	CREATE, ALTER, DROP, TRUNCATE
DML	Manipulate data	SELECT, INSERT, UPDATE, DELETE
DCL	Manage permissions	GRANT, REVOKE
TCL	Manage transactions	COMMIT, ROLLBACK, SAVEPOINT
Query Language	Retrieve data	SELECT

---

## ✓ 9. Importance of Database Languages

- Allow **precise definition and manipulation** of data
  - Facilitate **user and application interaction** with DBMS
  - Help in **maintaining security and data integrity**
  - Manage **concurrent access and transactions**
- 

## Conclusion

Database languages are essential for **defining, querying, updating, controlling access, and managing transactions** in a database system. Understanding each type and its commands is crucial for effective DBMS use.

### 9. Database users and administrators

#### ✓ 1. Introduction

In a Database Management System (DBMS), different people interact with the database system with different roles and responsibilities. These roles are broadly classified as **database users** and **database administrators**.

---

#### ✓ 2. Types of Database Users

Database users are categorized based on their interaction with the database, technical knowledge, and tasks they perform.

##### ◆ A. Types of Users

User Type	Description	Interaction Level
1. Casual Users	Access database occasionally using high-level queries	Use <b>query languages</b> (e.g., SQL) to retrieve data but don't perform frequent transactions.
2. Naive or Parametric Users	Use pre-written programs or interfaces to perform repetitive tasks	Rely on <b>application programs</b> ; do not write queries themselves. Example: bank clerks, reservation clerks.
3. Sophisticated Users	Use advanced tools or database interfaces to develop applications	Write complex queries, generate reports, analyze data. Examples: engineers, scientists.
4. Application Programmers	Write application programs that interact with the database	Develop programs using embedded SQL or API calls.

---

### ✓ 3. Roles of Database Users

User Type	Main Roles and Responsibilities
Casual Users	Query data, generate reports
Naive Users	Use pre-defined interfaces and application software
Sophisticated Users	Develop queries and perform data analysis
Application Programmers	Develop application software integrating DB access

---

### ✓ 4. Database Administrator (DBA)

The **Database Administrator (DBA)** is a key role responsible for managing the DBMS and ensuring smooth operation of the database system.

#### ◆ Responsibilities:

- **Schema definition and modification:** Creating and maintaining database structure.
  - **Granting and revoking access rights:** Manage user permissions for security.
  - **Data security and integrity:** Protect database from unauthorized access and ensure accuracy.
  - **Backup and recovery:** Plan and implement backup procedures; recover data after failures.
  - **Performance monitoring and tuning:** Optimize queries and database performance.
  - **Concurrency control:** Manage simultaneous data access to prevent conflicts.
  - **Database maintenance:** Routine checks, updates, and reorganizations.
  - **User training and support:** Assist users in efficient database usage.
- 

### ✓ 5. Other Roles Related to DBMS

Role	Description
System Analyst	Designs the database system and translates user requirements into specifications
Database Designers	Create the database schema and design the logical and physical structure
End Users	Use the database applications for their daily operations (could be naive, casual, or sophisticated users)
Application Developers	Develop software applications that use the database

---

### ✓ 6. Summary Table

Role	Responsibilities	Example
<b>Database Administrator (DBA)</b>	Overall management, security, backup, performance	Database manager in a company
<b>Casual User</b>	Occasional query and data retrieval	Marketing analyst
<b>Naive User</b>	Use predefined programs to access data	Bank teller using banking software
<b>Sophisticated User</b>	Complex queries, reporting, data analysis	Data scientist
<b>Application Programmer</b>	Develop database applications	Software developer

---

## ✓ 7. Why is Role Differentiation Important?

- **Security:** Limit access based on user role to protect sensitive data.
  - **Efficiency:** Users work at appropriate levels without overloading the system.
  - **Maintenance:** Clear responsibilities for database upkeep.
  - **User convenience:** Tailored interfaces and tools for different user types.
- 

## 🎓 Conclusion

Understanding the **types of database users and their roles** helps in designing better access controls, improving security, and ensuring efficient use of the DBMS. The **Database Administrator (DBA)** plays a crucial role in managing and securing the database system.

---

## 10. Data Dictionary?

### ✓ 1. What is a Data Dictionary?

A **Data Dictionary** (also called a **metadata repository**) is a centralized repository that stores **metadata** — data about data.

It contains information describing the structure, constraints, and usage of the data stored in the database.

---

### ✓ 2. Purpose of Data Dictionary

- Acts as a **reference** for the DBMS and users to understand the database schema and properties.
  - Helps in **database design, implementation, and maintenance**.
  - Ensures **data consistency** and **integrity** by enforcing definitions.
  - Facilitates **query optimization** and **security management**.
  - Enables **automatic validation** of data types, constraints, and access rights.
- 

### ✓ 3. Contents of Data Dictionary

Type of Information	Examples
Database schema information	Table names, column names, data types
Constraints	Primary keys, foreign keys, unique constraints
User information	Username, roles, access permissions
Storage information	File locations, indexing methods
Relationships between tables	Foreign key references
Triggers and stored procedures	Names and definitions
Statistics	Number of tuples, index usage stats

---

### ✓ 4. Types of Data Dictionaries

#### ◆ A. Active Data Dictionary

- Integrated into the DBMS.
- Automatically updated by the DBMS whenever the database structure changes.
- Used directly by the DBMS for query optimization, constraint enforcement, etc.
- Example: Oracle's data dictionary.

#### ◆ B. Passive Data Dictionary

- Maintained manually by the database administrators.
  - Not automatically updated by the DBMS.
  - Used mainly for documentation purposes.
  - Changes to the database require manual updates to this dictionary.
- 

### ✓ 5. Functions of Data Dictionary



Function	Description
<b>Metadata storage</b>	Stores definitions of all database objects
<b>Integrity enforcement</b>	Checks constraints and rules during data manipulation
<b>Access control</b>	Stores permissions for users and roles
<b>Query optimization</b>	Provides statistics and schema info to the query processor
<b>Database design aid</b>	Assists designers and developers in understanding the database structure
<b>Consistency checking</b>	Ensures all users have consistent views of the data

---

## ✓ 6. Example of Data Dictionary Entries

Table Name	Column Name	Data Type	Constraint	Description
Employee	EmpID	INT	PRIMARY KEY	Unique employee identifier
Employee	Name	VARCHAR(50)	NOT NULL	Employee's full name
Employee	DeptID	INT	FOREIGN KEY	References Department table

---

## ✓ 7. Importance of Data Dictionary

- Acts as a **central catalog** that helps DBMS manage and control the database.
  - Facilitates **automated processing** and reduces human errors.
  - Helps maintain **data integrity and consistency**.
  - Enables easier **database administration and troubleshooting**.
  - Supports **database documentation and standardization**.
- 

## ✓ 8. Summary Table

Feature	Description
<b>Location</b>	Stored within the DBMS or externally
<b>Automatic updates</b>	Only in active data dictionaries
<b>Users</b>	DBMS, database administrators, developers, end users
<b>Role in DBMS</b>	Metadata management, integrity, access control, query optimization

---

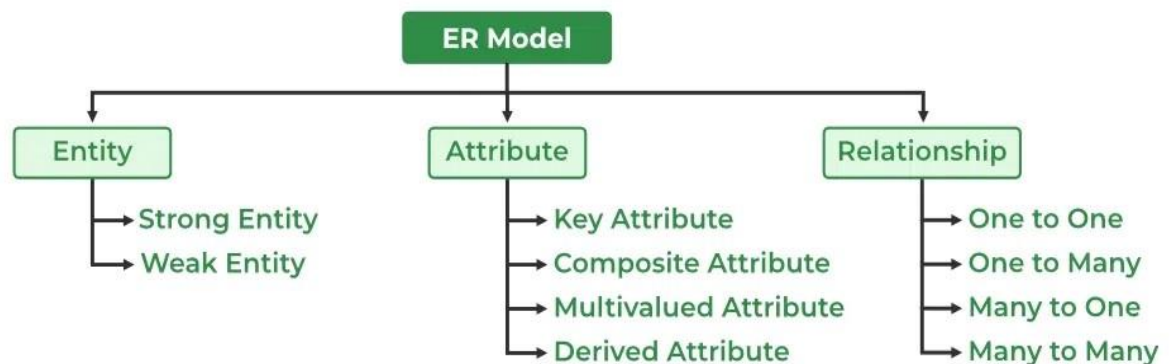
## 🎓 Conclusion

The **data dictionary** is a crucial component of a DBMS, serving as a **metadata repository** that stores all necessary information about the database structure, constraints, users, and usage. It supports the DBMS in managing data efficiently, ensuring integrity, security, and providing a valuable resource for users and developers.

### **11.Entity-Relationship Model: Entities, attributes, relationships, keys, E-R diagram.**

The Entity-Relationship Model (ER Model) is a conceptual model for designing a databases. This model represents the logical structure of a database, including entities, their attributes and relationships between them.

- **Entity:** An objects that is stored as data such as *Student*, *Course* or *Company*.
- **Attribute:** Properties that describes an entity such as *StudentID*, *CourseName*, or *EmployeeEmail*.
- **Relationship:** A connection between entities such as "a *Student* enrolls in a *Course*".



•

Components of ER Diagram

The graphical representation of this model is called an Entity-Relation Diagram (ERD).

## **ER Model in Database Design Process**

We typically follow the below steps for designing a database for an application.

- Gather the requirements (functional and data) by asking questions to the database users.
- Create a logical or conceptual design of the database. This is where ER model plays a role. It is the most used graphical representation of the conceptual design of a database.
- After this, focus on Physical Database Design (like indexing) and external design (like views)






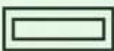
## Why Use ER Diagrams In DBMS?

- ER diagrams represent the E-R model in a database, making them easy to convert into relations (tables).
- These diagrams serve the purpose of real-world modeling of objects which makes them intently useful.
- Unlike technical schemas, ER diagrams require no technical knowledge of the underlying DBMS used.
- They visually model data and its relationships, making complex systems easier to understand.

## Symbols Used in ER Model

ER Model is used to model the logical view of the system from a data perspective which consists of these symbols:

- **Rectangles:** Rectangles represent entities in the ER Model.
- **Ellipses:** Ellipses represent attributes in the ER Model.
- **Diamond:** Diamonds represent relationships among Entities.
- **Lines:** Lines represent attributes to entities and entity sets with other relationship types.
- **Double Ellipse:** Double ellipses represent multi-valued Attributes, such as a student's multiple phone numbers
- **Double Rectangle:** Represents weak entities, which depend on other entities for identification.

Figures	Symbols	Represents
Rectangle		Entities in ER Model
Ellipse		Attributes in ER Model
Diamond		Relationships among Entities
Line		Attributes to Entities and Entity Sets with Other Relationship Types
Double Ellipse		Multi-Valued Attributes
Double Rectangle		Weak Entity

•

## What is an Entity?

An Entity represents a real-world object, concept or thing about which data is stored in a database. It act as a building block of a database. Tables in relational database represent these entities.

Example of entities:

- **Real-World Objects:** Person, Car, Employee etc.
- **Concepts:** Course, Event, Reservation etc.
- **Things:** Product, Document, Device etc.

The entity type defines the structure of an entity, while individual instances of that type represent specific entities.

## What is an Entity Set?

An entity refers to an individual object of an entity type, and the collection of all entities of a particular type is called an entity set. For example, E1 is an entity that belongs to the entity type "Student," and the group of all students forms the entity set.

In the ER diagram below, the entity type is represented as:



Entity Type



Entity Set

Entity Set

We can represent the entity sets in an ER Diagram but we can't represent individual entities because an entity is like a row in a table, and an ER diagram shows the structure and relationships of data, not specific data entries (like rows and columns). An ER diagram is a visual representation of the data model, not the actual data itself.

## Types of Entity

There are two main types of entities:

### 1. Strong Entity

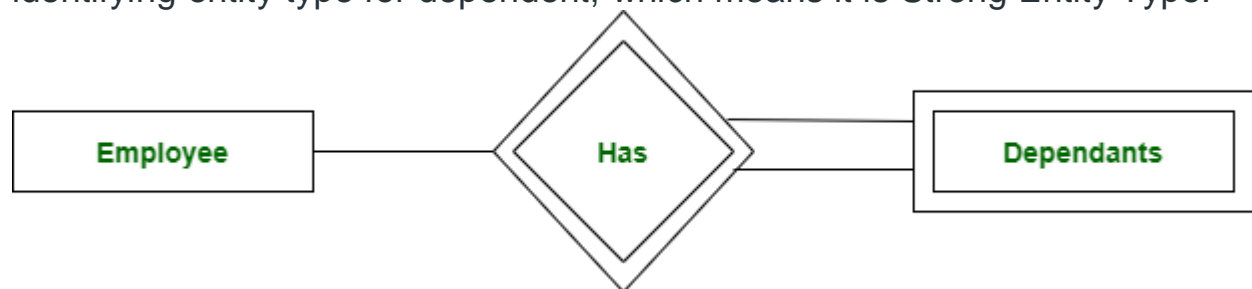
A Strong Entity is a type of entity that has a key Attribute that can uniquely identify each instance of the entity. A Strong Entity does not depend on any other Entity in the Schema for its identification. It has a primary key that ensures its uniqueness and is represented by a rectangle in an ER diagram.

### 2. Weak Entity

A Weak Entity cannot be uniquely identified by its own attributes alone. It depends on a strong entity to be identified. A weak entity is associated with an identifying entity (strong entity), which helps in its identification. A weak entity are represented by a double rectangle. The participation of weak entity types is always total. The relationship between the weak entity type and its identifying strong entity type is called identifying relationship and it is represented by a double diamond.

**Example:**

A company may store the information of dependents (Parents, Children, Spouse) of an Employee. But the dependents can't exist without the employee. So dependent will be a Weak Entity Type and Employee will be identifying entity type for dependent, which means it is Strong Entity Type.



Strong Entity and Weak Entity

## Attributes in ER Model

Attributes are the properties that define the entity type. For example, for a Student entity Roll\_No, Name, DOB, Age, Address, and Mobile\_No are the attributes that define entity type Student. In ER diagram, the attribute is represented by an oval.



Attribute

## Types of Attributes

### 1. Key Attribute

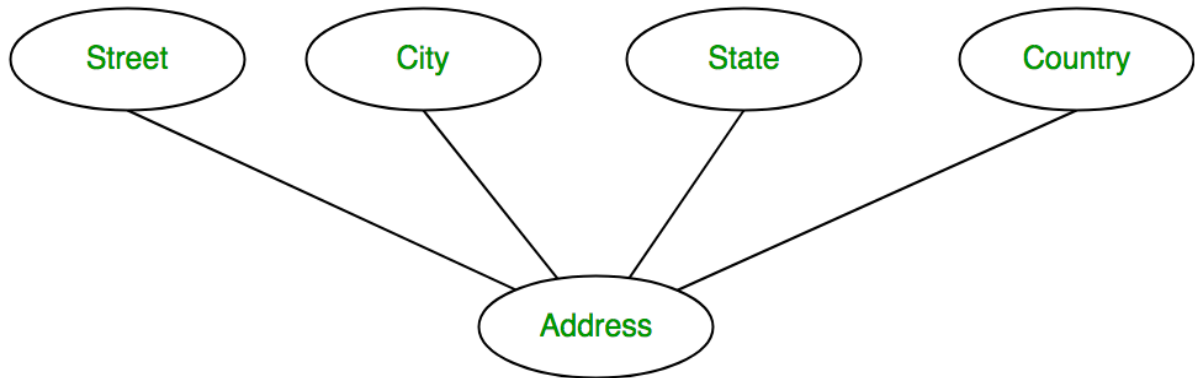
The attribute which uniquely identifies each entity in the entity set is called the key attribute. For example, Roll\_No will be unique for each student. In ER diagram, the key attribute is represented by an oval with an underline.



Key Attribute

## 2. Composite Attribute

An attribute composed of many other attributes is called a composite attribute. For example, the Address attribute of the student Entity type consists of Street, City, State, and Country. In ER diagram, the composite attribute is represented by an oval comprising of ovals.



Composite Attribute

## 3. Multivalued Attribute

An attribute consisting of more than one value for a given entity. For example, Phone\_No (can be more than one for a given student). In ER diagram, a multivalued attribute is represented by a double oval.



Multivalued Attribute

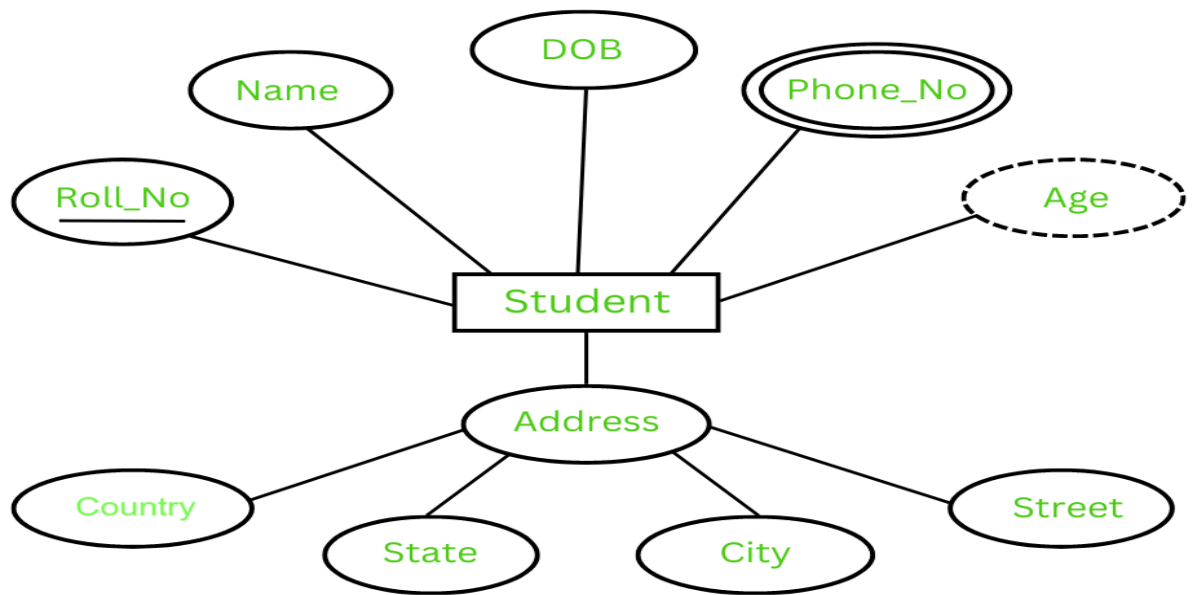
## 4. Derived Attribute

An attribute that can be derived from other attributes of the entity type is known as a derived attribute. e.g.; Age (can be derived from DOB). In ER diagram, the derived attribute is represented by a dashed oval.



Derived Attribute

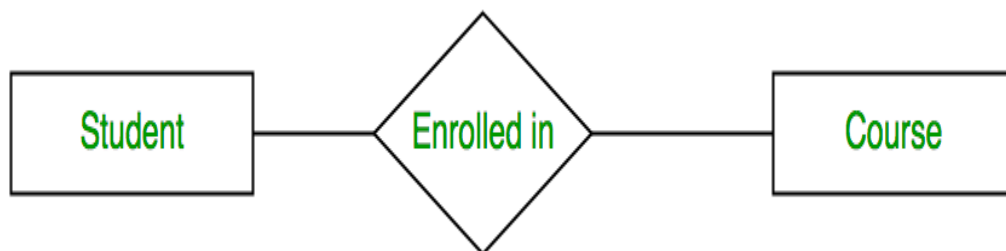
The Complete Entity Type Student with its Attributes can be represented as:



Entity and Attributes

## Relationship Type and Relationship Set

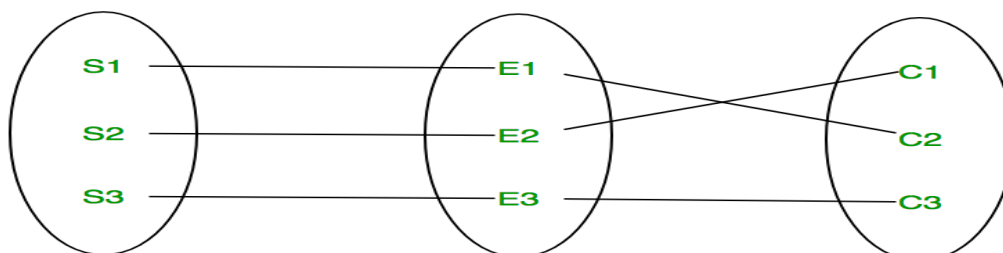
A Relationship Type represents the association between entity types. For example, 'Enrolled in' is a relationship type that exists between entity type Student and Course. In ER diagram, the relationship type is represented by a diamond and connecting the entities with lines.



Entity-

Relationship Set

A set of relationships of the same type is known as a relationship set. The following relationship set depicts S1 as enrolled in C2, S2 as enrolled in C1, and S3 as registered in C3.

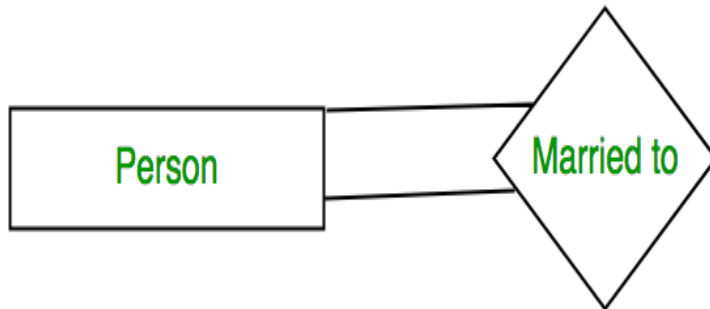


Relationship Set

## Degree of a Relationship Set

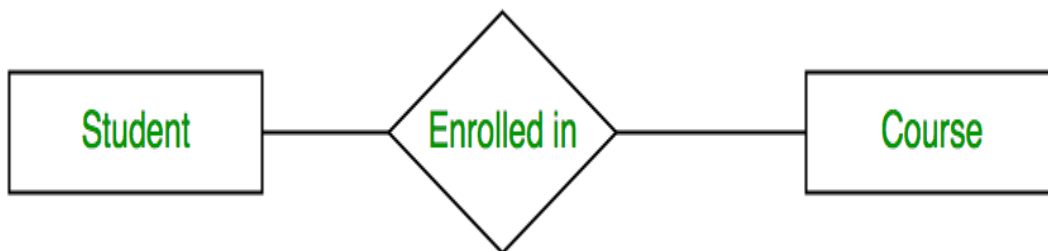
The number of different entity sets participating in a relationship set is called the degree of a relationship set.

**1. Unary Relationship:** When there is only ONE entity set participating in a relation, the relationship is called a unary relationship. For example, one person is married to only one person.



Unary Relationship

**2. Binary Relationship:** When there are TWO entities set participating in a relationship, the relationship is called a binary relationship. For example, a Student is enrolled in a Course.



Binary

Relationship

**3. Ternary Relationship:** When there are three entity sets participating in a relationship, the relationship is called a ternary relationship.

**4. N-ary Relationship:** When there are n entities set participating in a relationship, the relationship is called an n-ary relationship.

## Cardinality in ER Model

The maximum number of times an entity of an entity set participates in a relationship set is known as cardinality.

Cardinality can be of different types:

### 1. One-to-One

When each entity in each entity set can take part only once in the relationship, the cardinality is one-to-one. Let us assume that a male can marry one female and a female can marry one male. So the relationship will be one-to-one.

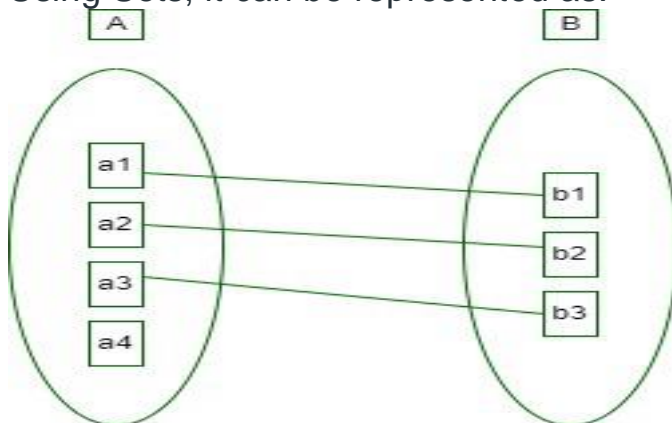




One to

One Cardinality

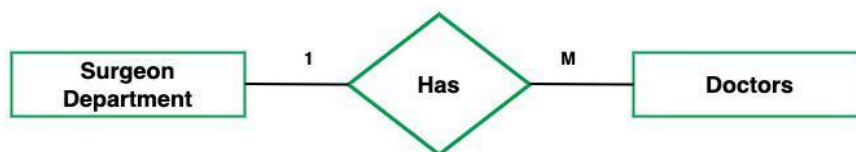
Using Sets, it can be represented as:



Set Representation of One-to-One

## 2. One-to-Many

In one-to-many mapping as well where each entity can be related to more than one entity. Let us assume that one surgeon department can accommodate many doctors. So the Cardinality will be 1 to M. It means one department has many Doctors.



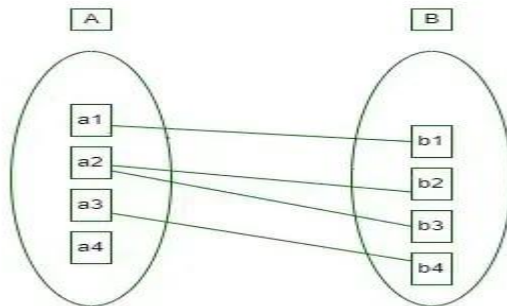
one to

many cardinality

Using sets, one-to-many cardinality can be represented as:

Set Representation of One-to-Many

## 4. Many-to-One



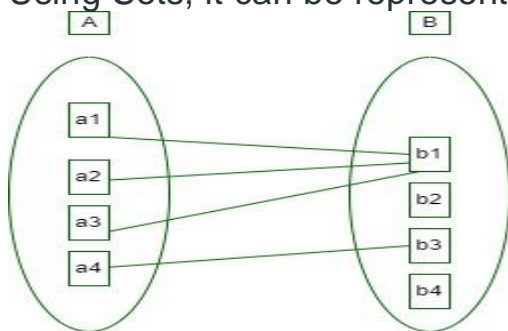
When entities in one entity set can take part only once in the relationship set and entities in other entity sets can take part more than once in the relationship set, cardinality is many to one.

Let us assume that a student can take only one course but one course can be taken by many students. So the cardinality will be n to 1. It means that for one course there can be n students but for one student, there will be only one course.



many to one cardinality

Using Sets, it can be represented as:



Set Representation of Many-to-One

In this case, each student is taking only 1 course but 1 course has been taken by many students.

#### 4. Many-to-Many

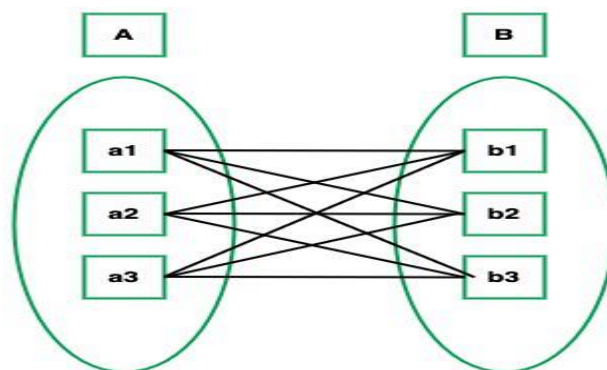
When entities in all entity sets can take part more than once in the relationship cardinality is many to many. Let us assume that a student can

take more than one course and one course can be taken by many students.  
So the relationship will be many to many.



many to many cardinality

Using Sets, it can be represented as:



Many-to-Many Set Representation

In this example, student S1 is enrolled in C1 and C3 and Course C3 is enrolled by S1, S3, and S4. So it is many-to-many relationships.

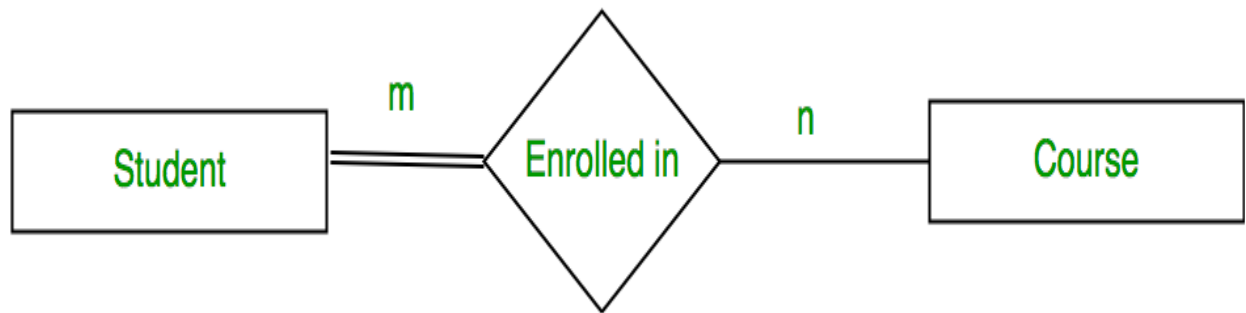
## Participation Constraint

Participation Constraint is applied to the entity participating in the relationship set.

**1. Total Participation:** Each entity in the entity set must participate in the relationship. If each student must enroll in a course, the participation of students will be total. Total participation is shown by a double line in the ER diagram.

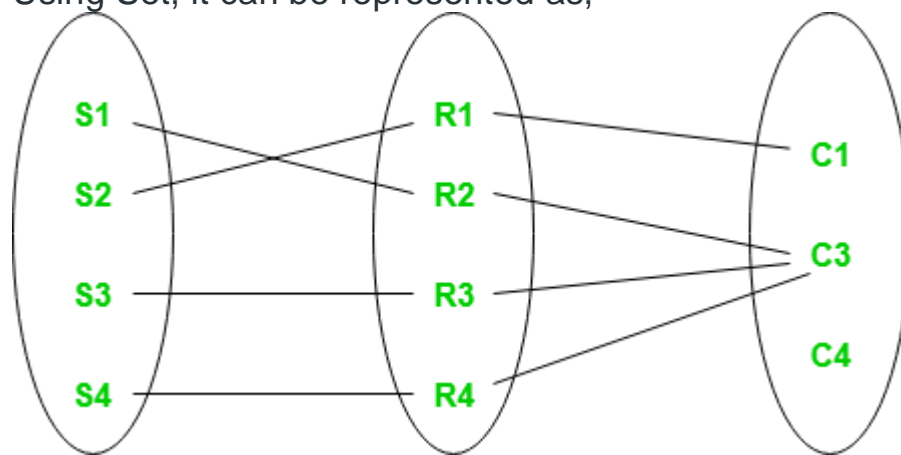
**2. Partial Participation:** The entity in the entity set may or may NOT participate in the relationship. If some courses are not enrolled by any of the students, the participation in the course will be partial.

The diagram depicts the 'Enrolled in' relationship set with Student Entity set having total participation and Course Entity set having partial participation.



Total Participation and Partial Participation

Using Set, it can be represented as,



Set representation of

Total Participation and Partial Participation

Every student in the Student Entity set participates in a relationship but there exists a course C4 that is not taking part in the relationship.

## Unit-II

### Relational Model in DBMS

#### ◆ Definition

The **Relational Model** was proposed by **E.F. Codd** in 1970. It is a method of structuring data using relations, i.e., **tables**. Each relation (or table) is a collection of **tuples (rows)** and **attributes (columns)**.

---

#### ◆ Key Terminologies

Term	Description
<b>Relation</b>	A table with rows and columns.
<b>Tuple</b>	A single row in a table, representing a record.
<b>Attribute</b>	A column in the table, representing a data field.
<b>Domain</b>	The set of allowable values for an attribute.
<b>Relation Schema</b>	Defines the name of the relation and its attributes.
<b>Degree</b>	The number of attributes (columns) in a relation.
<b>Cardinality</b>	The number of tuples (rows) in a relation.
<b>Primary Key</b>	An attribute or set of attributes that uniquely identifies each tuple.
<b>Foreign Key</b>	An attribute that refers to the primary key of another relation.

---

#### ◆ Structure of a Relation

Example:

```
plaintext
CopyEdit
STUDENT (Roll_No, Name, Age, Course)
```

	Roll_No		Name	
	Age		Course	
	101		Alice	
	102		Bob	
	103		Carol	

---

#### ◆ Features of Relational Model

- **Simple structure:** Based on tables (relations).
- **Data Independence:** Logical and physical data are separate.

- **Use of Keys:** Ensures data integrity and avoids duplication.
  - **Declarative Querying:** SQL is used to manipulate and retrieve data.
  - **Normalization:** Eliminates redundancy and maintains consistency.
- 

## ◆ Relational Integrity Constraints

1. **Domain Constraint**
    - Each attribute must hold values from its defined domain.
    - Example: Age must be an integer between 0 and 150.
  2. **Entity Integrity Constraint**
    - The primary key must be unique and not null.
    - Ensures that each record can be uniquely identified.
  3. **Referential Integrity Constraint**
    - A foreign key must match a primary key in another table or be null.
    - Maintains consistency across related tables.
- 

## ◆ Advantages of Relational Model

- **Ease of use:** Intuitive tabular format.
  - **Flexibility:** Easy to add, delete, or modify records.
  - **Powerful query capabilities:** Supports SQL.
  - **Data integrity and accuracy:** Through constraints and keys.
  - **Security:** Can define access privileges.
- 

## ◆ Disadvantages

- **Performance Issues:** For large and complex databases, it may be slower.
  - **Complexity in Relationships:** Handling many-to-many or recursive relationships may be complex.
  - **Requires Normalization:** To avoid redundancy, which may make design more complex.
- 

## ◆ Relational Algebra and SQL

Relational model supports theoretical query languages like:

- **Relational Algebra:** A procedural query language (e.g., select, project, join).
  - **SQL (Structured Query Language):** A standard language to query and manipulate relational databases.
-

## ◆ Normalization in Relational Model

Normalization is the process of organizing data to:

- Minimize redundancy.
- Improve data integrity.

Common normal forms:

- 1NF (First Normal Form)
  - 2NF (Second Normal Form)
  - 3NF (Third Normal Form)
  - BCNF (Boyce-Codd Normal Form)
- 

## ◆ Real-life Applications

- Banking systems
  - Airline reservation systems
  - E-commerce platforms
  - University student databases
- 

## ◆ Example Queries (SQL)

```
sql
CopyEdit
-- Create a table
CREATE TABLE Student (
    Roll_No INT PRIMARY KEY,
    Name VARCHAR(50),
    Age INT,
    Course VARCHAR(30)
);

-- Insert data
INSERT INTO Student VALUES (101, 'Alice', 21, 'CS');

-- Select data
SELECT * FROM Student WHERE Course = 'CS';
```

---

## 2.Relational Algebra

# Relational Algebra in DBMS

## ◆ Definition

**Relational Algebra** is a **procedural query language** in DBMS. It provides a set of operations to manipulate and retrieve data from **relational databases**.

- Introduced by **E.F. Codd**.
  - Takes one or more relations (tables) as input and produces a **new relation** as output.
  - Used as a foundation for **SQL** and **query optimization**.
- 

## ◆ Types of Relational Algebra Operations

Relational Algebra operations are broadly classified into:

1. **Basic Operations (Set-oriented)**
  2. **Special/Advanced Operations (Relational-specific)**
- 

## ◆ 1. Basic Set Operations

These are similar to operations in set theory.

Operation	Symbol	Description
<b>Union</b>	$\cup$	Combines tuples from two relations (no duplicates).
<b>Set Difference</b>	$-$	Returns tuples in one relation but not in the other.
<b>Intersection</b>	$\cap$	Returns tuples present in both relations.
<b>Cartesian Product</b>	$\times$	Pairs each tuple of one relation with every tuple of another.
<b>Rename</b>	$\rho$ (rho)	Renames the relation or attributes.

**Note:** For set operations like union, difference, and intersection to work, the **relations must be union-compatible**:

- Same number of attributes.
  - Corresponding attributes must have the same domain.
- 

## ◆ 2. Relational Operations (Core operations)

Operation	Symbol	Description
<b>Select</b>	$\sigma$ (sigma)	Selects rows (tuples) that satisfy a condition.



Operation	Symbol	Description
Project	$\pi$ (pi)	Selects specific columns (attributes).
Join	$\bowtie$ (bowtie)	Combines tuples from two relations based on a condition.
Division	$\div$	Finds tuples related to all tuples in another relation.

---

## ◆ Detailed Explanation of Each Operation

---

### 1. Select ( $\sigma$ )

- Filters rows based on a **predicate**.
- Notation:  $\sigma<\text{condition}>(\text{Relation})$
- Example:

```
SCSS
CopyEdit
 $\sigma_{\text{Age} > 18}(\text{Student})$ 
```

Returns students older than 18.

---

### 2. Project ( $\pi$ )

- Selects **specific attributes** (columns).
- Removes duplicates.
- Notation:  $\pi<\text{attribute\_list}>(\text{Relation})$
- Example:

```
SCSS
CopyEdit
 $\pi_{\text{Name}, \text{Course}}(\text{Student})$ 
```

---

### 3. Union ( $\cup$ )

- Combines tuples from two relations.
- Removes duplicates.
- Example:

```
nginx
CopyEdit
Student1  $\cup$  Student2
```

---

## 4. Set Difference (−)

- Returns tuples in one relation but not in another.
- Example:

```
nginx
CopyEdit
Student1 − Student2
```

---

## 5. Cartesian Product (×)

- Combines every tuple from the first relation with every tuple from the second.
- Example:

```
nginx
CopyEdit
Student × Course
```

---

## 6. Rename (ρ)

- Renames a relation or its attributes.
- Notation:  $\rho_{\text{NewName}}(\text{Relation})$  or  $\rho(\text{NewName}(A_1, A_2, \dots))(\text{Relation})$
- Example:

```
scss
CopyEdit
ρS(Student)
```

---

## 7. Join (⋈)

Combines related tuples from two relations.

*Types of Join:*

- **Theta Join (θ Join):**  $R \bowtie_{\langle \text{condition} \rangle} S$
- **Equi Join:** Theta join with only equality conditions.
- **Natural Join (⋈):** Joins on all common attributes.
- **Outer Join:** Includes unmatched tuples (left, right, full).

Example:

```
nginx
CopyEdit
Student ⋈ Student.Course = Course.Course_ID Course
```

---

## 8. Division (÷)

- Used when we want tuples from one relation that are associated with **all** tuples of another relation.

Example:

```
css
CopyEdit
A ÷ B
```

Returns tuples in A that are related to **every tuple in B**.

---

## ◆ Relational Algebra Query Example

Given:

```
plaintext
CopyEdit
STUDENT(RollNo, Name, Age, Course)
COURSE(CourseID, CourseName)
Q: Find names of students enrolled in 'CS' course.
```

**Step 1:** Select tuples where Course = 'CS':

```
bash
CopyEdit
σCourse = 'CS' (STUDENT)
```

**Step 2:** Project only the names:

```
scss
CopyEdit
πName(σCourse = 'CS' (STUDENT))
```

---

## ◆ Relational Algebra vs SQL

Relational Algebra	SQL
Procedural query language	Declarative query language
Specifies <b>how</b> to retrieve	Specifies <b>what</b> to retrieve
Foundation for query engines	User-level query language

---

## ◆ Importance of Relational Algebra

- Forms the **theoretical foundation** for relational databases.
  - Helps in **query optimization**.
  - Important in **database engine design**.
  - Used in **compiler design** and **query planning**.
- 

### 3. Relational calculus

## Relational Calculus in DBMS

### ◆ Definition

**Relational Calculus** is a **non-procedural query language** in DBMS. It specifies **what to retrieve** rather than **how to retrieve** it, unlike relational algebra.

- Based on **first-order predicate logic**.
  - Used to express **queries declaratively**.
  - There are **two types**:
    - **Tuple Relational Calculus (TRC)**
    - **Domain Relational Calculus (DRC)**
- 

## Key Characteristics

Feature	Description
<b>Non-procedural</b>	Focuses on what to retrieve, not how.
<b>Declarative logic</b>	Uses logical expressions to define queries.
<b>Foundation for SQL</b>	SQL is influenced more by relational calculus than algebra.

---

## 1. Tuple Relational Calculus (TRC)

### ✓ Syntax

```
less
CopyEdit
{ t | P(t) }
```

- $t$  is a **tuple variable**.
- $P(t)$  is a **predicate** (logical condition).
- The result is a set of all  $t$  for which  $P(t)$  is true.

## ✓ Example

Given:

```
plaintext
CopyEdit
Student(RollNo, Name, Age, Course)
Q: Retrieve names of students older than 18.
pgsql
CopyEdit
{ t.Name | Student(t) ∧ t.Age > 18 }
```

---

## ✓ TRC Operators and Notations

Operator	Meaning
$\wedge$	AND
$\vee$	OR
$\neg$	NOT
$\Rightarrow$	IMPLIES
$\exists t$	There exists a tuple $t$
$\forall t$	For all tuples $t$
$=, <, >, \leq, \geq$ Comparison operators	

---

## ◆ 2. Domain Relational Calculus (DRC)

### ✓ Syntax

```
CopyEdit
{ <x1, x2, ..., xn> | P(x1, x2, ..., xn) }
```

- Each  $x_i$  is a **domain variable**.
- $P(\dots)$  is a predicate using domain variables.
- The result is a set of **domain values** satisfying the condition.

### ✓ Example

```
Q: Get names of students older than 18.
pgsql
CopyEdit
{ <Name> | ∃ RollNo ∃ Age ∃ Course (Student(RollNo, Name, Age, Course) ∧
Age > 18) }
```

---

## ✓ DRC vs TRC

Feature	TRC	DRC
Based on	Tuples (rows)	Domains (columns)
Variable type	Tuple variable (e.g., $t$ )	Domain variables (e.g., $x$ )
Output	Tuple or tuple fields	Set of domain values
Example Syntax	$\{ t \mid \dots \}$	$P(t) \}$

---

## ◆ Safe Expressions in Relational Calculus

Relational calculus expressions must be **safe**, i.e., they must produce **finite and meaningful results**.

### ! Unsafe Query Example:

```
nginx
CopyEdit
{ t | ¬Student(t) }
```

This expression tries to return all tuples *not* in the Student relation — which is undefined (could be infinite).

---

## ◆ Use of Quantifiers

Quantifier Symbol	Meaning
Existential $\exists$	There exists at least one value
Universal $\forall$	For all values

### ✓ Example (with quantifier):

Retrieve students enrolled in **every** course:

```
r
CopyEdit
{ s | Student(s) ∧ ∀ c (Course(c) → Enrolled(s, c)) }
```

---

## ◆ Relational Algebra vs. Relational Calculus

Feature	Relational Algebra	Relational Calculus
Type	Procedural	Non-Procedural
Focus	How to retrieve data	What data to retrieve
Closer to	Programming model	Logic-based model
Foundation for	Query execution engine	SQL query design
Complexity	May be easier to optimize	Harder to optimize automatically

---

## ◆ Advantages of Relational Calculus

- Easier to express **complex queries logically**.
  - More **intuitive** for users familiar with predicate logic.
  - Influential in the development of **SQL**.
- 

## ◆ Limitations

- Not used directly in practice (SQL is used instead).
  - **Harder to optimize** automatically.
  - Queries must be **safe** to avoid infinite results.
- 

## Summary

Topic	Summary
What is it?	A non-procedural way to query relational databases.
Main types	Tuple Relational Calculus (TRC), Domain Relational Calculus (DRC)
Query style	Logical and declarative
Relation to SQL	Forms the theoretical basis of SQL
Common operators	AND, OR, NOT, EXISTS ( $\exists$ ), FOR ALL ( $\forall$ ), comparison operators
Main concern	Only <b>safe</b> expressions are useful in practice

## 4. Keys and Constraints

# 🔑 PART 1: KEYS IN DBMS

### ◆ What is a Key?

A **key** is an attribute or a set of attributes that helps in **uniquely identifying a tuple (row)** in a relation (table). Keys ensure **uniqueness** and help maintain **data integrity**.

---

### ✓ Types of Keys with Examples

#### 1. Super Key

- **Definition:** A set of one or more attributes that **uniquely identifies** each tuple in a table.
- **Example:**

```
plaintext
CopyEdit
Table: Student(RollNo, Name, Email)
Super keys: {RollNo}, {RollNo, Name}, {Email}
```

---

#### 2. Candidate Key

- **Definition:** A minimal super key (no redundant attributes).
- **Example:**

```
plaintext
CopyEdit
Table: Student(RollNo, Email)
Candidate Keys: {RollNo}, {Email}
```

---

#### 3. Primary Key

- **Definition:** A candidate key **chosen** to uniquely identify tuples in a table.
- **Rules:**
  - Cannot be NULL
  - Must be unique
- **Example:**

```
sql
CopyEdit
CREATE TABLE Student (
    RollNo INT PRIMARY KEY,
    Name VARCHAR(50),
    Email VARCHAR(100)
);
```



---

#### 4. Alternate Key

- **Definition:** Candidate keys that **are not selected** as the primary key.
- **Example:**  
If RollNo is the primary key, then Email is the alternate key.

---

#### 5. Composite Key

- **Definition:** A key formed by **combining two or more attributes** to uniquely identify a row.
- **Example:**

```
plaintext
CopyEdit
Table: Enrollment(StudentID, CourseID)
Composite Key: {StudentID, CourseID}
```

---

#### 6. Foreign Key

- **Definition:** An attribute in one table that **references the primary key** in another table.
- **Used to** establish relationships between tables.
- **Example:**

```
sql
CopyEdit
CREATE TABLE Course (
    CourseID INT PRIMARY KEY,
    CourseName VARCHAR(50)
);

CREATE TABLE Enrollment (
    StudentID INT,
    CourseID INT,
    FOREIGN KEY (CourseID) REFERENCES Course(CourseID)
);
```

---

#### 7. Unique Key

- **Definition:** Ensures that all values in a column are **unique**, but **allows one NULL** value.
- **Example:**

```
sql
CopyEdit
CREATE TABLE Users (
    UserID INT PRIMARY KEY,
    Email VARCHAR(100) UNIQUE
);
```

---

## □ Key Comparison Table

Key Type	Unique	Null Allowed	Example
Super Key	✓	✓ / ✗	{RollNo}, {RollNo, Name}
Candidate Key	✓	✗	{RollNo}, {Email}
Primary Key	✓	✗	RollNo
Alternate Key	✓	✗	Email (if not primary)
Composite Key	✓	✗	(StudentID, CourseID)
Foreign Key	✗	✓	CourseID (references Course)
Unique Key	✓	✓ (only one)	Email

---

## 🛡️ PART 2: CONSTRAINTS IN DBMS

### ◆ What is a Constraint?

A **constraint** is a rule enforced on data in a database to ensure **validity, accuracy, and consistency**.

---

### ✓ Types of Constraints with Examples

#### 1. NOT NULL Constraint

- Ensures that a column **cannot have NULL values**.
- Example:**

```
sql
CopyEdit
Name VARCHAR(50) NOT NULL
```

---

#### 2. UNIQUE Constraint

- Ensures that all values in a column are **different**.
- Allows **one NULL**.
- Example:**

```
sql
```

```
CopyEdit
Email VARCHAR(100) UNIQUE
```

---

### 3. PRIMARY KEY Constraint

- Combines NOT NULL and UNIQUE.
- Only **one per table**.
- **Example:**

```
sql
CopyEdit
RollNo INT PRIMARY KEY
```

---

### 4. FOREIGN KEY Constraint

- Maintains **referential integrity** by ensuring values match a primary key in another table.
- **Example:**

```
sql
CopyEdit
FOREIGN KEY (CourseID) REFERENCES Course(CourseID)
```

---

### 5. CHECK Constraint

- Enforces a condition on the values in a column.
- **Example:**

```
sql
CopyEdit
Age INT CHECK (Age >= 18)
```

---

### 6. DEFAULT Constraint

- Assigns a **default value** if no value is provided.
- **Example:**

```
sql
CopyEdit
Status VARCHAR(10) DEFAULT 'Active'
```

---

## Full Table Example with Constraints

```
sql
CopyEdit
CREATE TABLE Student (
    RollNo INT PRIMARY KEY,
    Name VARCHAR(50) NOT NULL,
    Email VARCHAR(100) UNIQUE,
```

```
Age INT CHECK (Age >= 18),
CourseID INT,
Status VARCHAR(10) DEFAULT 'Active',
FOREIGN KEY (CourseID) REFERENCES Course(CourseID)
);
```

---

## ✔ Constraints Summary Table

Constraint	Description	Example
NOT NULL	Value must not be NULL	Name VARCHAR(50) NOT NULL
UNIQUE	All values must be different	Email VARCHAR(100) UNIQUE
PRIMARY KEY	Unique + Not NULL	RollNo INT PRIMARY KEY
FOREIGN KEY	Points to a primary key in another table	FOREIGN KEY (CourseID)
CHECK	Validates values with a condition	Age INT CHECK (Age >= 18)
DEFAULT	Sets default value if none provided	Status VARCHAR(10) DEFAULT 'Active'

---

## □ Why Are Keys and Constraints Important?

Keys	Constraints
Identify records uniquely	Maintain data accuracy
Link related tables	Prevent invalid data entries
Avoid duplication	Enforce business rules
Optimize searching	Keep relationships consistent

---

## Normalization: 1NF, 2NF, 3NF, BCNF, 4NF, and 5NF

### ◆ What is Normalization?

Normalization is the **process of organizing data** in a database to:

- Reduce data redundancy
- Eliminate anomalies (insertion, update, deletion)
- Improve data integrity

It involves **dividing large tables** into smaller, related ones and defining relationships between them.

---

## ☐ Types (Forms) of Normalization

---

### ✓ 1NF – First Normal Form

#### ◆ Rule:

- A relation is in 1NF if:
  - All **attributes contain only atomic (indivisible) values**
  - Each field contains only **single values**, not sets or arrays

#### ◆ Violation Example:

```
plaintext
CopyEdit
Student(RollNo, Name, Subjects)
101, John, {Math, Physics}
```

#### ◆ Conversion to 1NF:

```
plaintext
CopyEdit
Student(RollNo, Name, Subject)
101, John, Math
101, John, Physics
```

---

### ✓ 2NF – Second Normal Form

#### ◆ Rule:

- Must be in **1NF**
- All **non-prime attributes** must be **fully functionally dependent** on the **entire primary key** (no partial dependency)

#### ◆ Violation Example:

```
plaintext
CopyEdit
Table: Enrollment(StudentID, CourseID, StudentName)
Primary Key: (StudentID, CourseID)

Partial dependency: StudentName depends only on StudentID
```

### ◆ Conversion to 2NF:

#### Split into:

```
plaintext
CopyEdit
Student(StudentID, StudentName)
Enrollment(StudentID, CourseID)
```

---

## ✓ 3NF – Third Normal Form

### ◆ Rule:

- Must be in 2NF
- **No transitive dependency** (non-prime attribute depends on another non-prime attribute)

### ◆ Violation Example:

```
plaintext
CopyEdit
Table: Student(RollNo, Name, DeptID, DeptName)
Functional Dependencies:
RollNo → DeptID
DeptID → DeptName (transitive)
```

### ◆ Conversion to 3NF:

```
plaintext
CopyEdit
Student(RollNo, Name, DeptID)
Department(DeptID, DeptName)
```

---

## ✓ BCNF – Boyce-Codd Normal Form

### ◆ Rule:

- Must be in 3NF
- For every functional dependency  $X \rightarrow Y$ , **X should be a super key**

### ◆ Violation Example:

```
plaintext
CopyEdit
Table: Course(FacultyName, CourseCode)
FD: FacultyName → CourseCode
But FacultyName is not a super key (multiple faculties may teach same course)
```

### ◆ Conversion to BCNF:

```
plaintext
CopyEdit
FacultyCourse(FacultyName, CourseCode)
```

Ensure both are uniquely mapped by splitting based on FDs.

---

## ✓ 4NF – Fourth Normal Form

### ◆ Rule:

- Must be in BCNF
- No multivalued dependencies

### ◆ Violation Example:

```
plaintext
CopyEdit
Table: Student(StudentID, Hobby, Language)
Multivalued dependencies:
StudentID →→ Hobby
StudentID →→ Language
```

### ◆ Conversion to 4NF:

```
plaintext
CopyEdit
StudentHobby(StudentID, Hobby)
StudentLanguage(StudentID, Language)
```

---

## ✓ 5NF – Fifth Normal Form / PJ/NF (Project-Join Normal Form)

### ◆ Rule:

- Must be in 4NF
- No join dependency that cannot be enforced via candidate keys

### ◆ Violation Example:

```
plaintext
CopyEdit
Table: Project(ProjectID, SupplierID, PartID)
Project can have many suppliers and parts independently, leading to
redundancy
```

## ◆ Conversion to 5NF:

Break into three relations:

```
plaintext
CopyEdit
ProjectSupplier(ProjectID, SupplierID)
ProjectPart(ProjectID, PartID)
SupplierPart(SupplierID, PartID)
```

---

## □ Summary Table

Normal Form	Rule Summary	Eliminates	Example Fix
1NF	Atomic values only	Repeating groups	Split multi-valued fields into multiple rows
2NF	No partial dependency	Partial dependencies	Separate dependent attributes to another table
3NF	No transitive dependency	Transitive dependencies	Use separate tables for indirectly dependent data
BCNF	Every determinant is a super key	Anomalies from 3NF	Decompose using super key-based FD
4NF	No multivalued dependencies	Redundancy from MVDs	Split into separate one-to-many tables
5NF	No join dependency without candidate key	Redundancy from joins	Decompose to smaller relations

---

## 💡 Why Normalize?

Benefit	Explanation
Data Integrity	Prevents anomalies
Reduced Redundancy	Less repetition, more efficient storage
Easier Maintenance	Modular schema, changes easier to manage
Better Performance	Smaller tables = faster queries (usually)

---

## 🏠 Conclusion

Normalization is a foundational concept in DBMS that ensures:

- Proper data organization
- Efficient querying
- Reduced storage overhead
- Data consistency across operations



## Functional dependencies

### ◆ What is a Functional Dependency?

A **Functional Dependency (FD)** is a constraint between two sets of attributes in a **relation** from a **database**.

- It describes the **relationship** between attributes.
  - If attribute **A** functionally determines attribute **B** (written as  $A \rightarrow B$ ), then **for each value of A, there is exactly one value of B**.
- 

### ✓ Formal Definition

Let **R** be a relation and **X** and **Y** be subsets of attributes in **R**.

We say  $X \rightarrow Y$  (**X** functionally determines **Y**) if:

**For any two tuples t1 and t2 in R, if t1[X] = t2[X], then t1[Y] = t2[Y]**

---

### 🔑 Example of Functional Dependency

Consider a table `Student (RollNo, Name, Dept)`:

- If each `RollNo` is unique, then:

```
nginx
CopyEdit
RollNo → Name
RollNo → Dept
```

Because if two students have the same `RollNo`, they **must** have the same `Name` and `Dept`.

---

## □ Types of Functional Dependencies

### 1. Trivial Functional Dependency

- If  $Y \subseteq X$ , then  $X \rightarrow Y$  is **trivial**.
- Always holds.
- Example:

```
pgsql
CopyEdit
```

$\{\text{RollNo}, \text{Name}\} \rightarrow \text{RollNo}$

---

## 2. Non-Trivial Functional Dependency

- If  $Y \not\subseteq X$ , then  $X \rightarrow Y$  is **non-trivial**.
- Example:

```
nginx
CopyEdit
RollNo → Name
```

---

## 3. Completely Non-Trivial Dependency

- If  $X$  and  $Y$  have no common attributes, then it is **completely non-trivial**.
- Example:

```
scss
CopyEdit
RollNo → Dept (RollNo and Dept are disjoint)
```

---

## 4. Partial Dependency

- When a non-prime attribute is **functionally dependent on part of a composite key**.
- Violates 2NF.
- Example:

```
scss
CopyEdit
(StudentID, CourseID) → StudentName
```

---

## 5. Transitive Dependency

- If  $X \rightarrow Y$  and  $Y \rightarrow Z$ , then  $X \rightarrow Z$ .
- Violates 3NF.
- Example:

```
nginx
CopyEdit
RollNo → DeptID
DeptID → DeptName
⇒ RollNo → DeptName
```

---

## 6. Multivalued Dependency (MVD)

- When one attribute in a table uniquely determines another attribute **independently** of others.
- Related to 4NF.

- Example:

```
sql
CopyEdit
StudentID → Hobby
StudentID → Language
```

---

## ✂ How to Identify Functional Dependencies

Use **real-world logic** and **common sense**:

Attribute(s)	Can Determine	Reason
EmployeeID	Name, Dept	ID is unique per employee
DeptID	DeptName	Each department has one name
CourseCode	CourseName	Code maps to only one name

---

## 📁 Notation and Symbols

- $X \rightarrow Y$ : X functionally determines Y
  - **+** (**Closure**):  $X^+$  is the set of attributes functionally determined by X
  - **FD Set**: A collection of FDs over a relation
- 

## 📁 Example Table and Functional Dependencies

```
plaintext
CopyEdit
Table: Employee(EmpID, Name, DeptID, DeptName, Salary)
```

FDs:  
 $EmpID \rightarrow Name, DeptID, Salary$   
 $DeptID \rightarrow DeptName$

Here,

- EmpID **uniquely identifies** Name, DeptID, Salary
  - DeptID **determines** DeptName
- 

## 📁 Attribute Closure ( $X^+$ )

Used to **find all attributes functionally determined** by a set X.

**Example:**

Given:

```
makefile
CopyEdit
FDs:
A → B
B → C
```

Then:

```
mathematica
CopyEdit
A+ = {A, B, C}
```

---

## ⚙️ Uses of Functional Dependencies

1. **Normalization** (2NF, 3NF, BCNF)
  2. **Decomposition** of tables
  3. **Lossless joins**
  4. **Designing relational schemas**
  5. **Checking redundancy and anomalies**
- 

## 🚀 Armstrong's Axioms (Inference Rules)

Used to infer all functional dependencies logically from a given set.

Axiom	Rule	Example
Reflexivity	If $Y \subseteq X$ , then $X \rightarrow Y$	$\{A, B\} \rightarrow A$
Augmentation	If $X \rightarrow Y$ , then $XZ \rightarrow YZ$	$A \rightarrow B \Rightarrow AC \rightarrow BC$
Transitivity	If $X \rightarrow Y$ and $Y \rightarrow Z$ , then $X \rightarrow Z$	$A \rightarrow B, B \rightarrow C \Rightarrow A \rightarrow C$

### Additional Rules (Derived):

- **Union:**  $X \rightarrow Y$  and  $X \rightarrow Z \Rightarrow X \rightarrow YZ$
  - **Decomposition:**  $X \rightarrow YZ \Rightarrow X \rightarrow Y$  and  $X \rightarrow Z$
  - **Pseudotransitivity:** If  $X \rightarrow Y$  and  $YZ \rightarrow W \Rightarrow XZ \rightarrow W$
- 

## ✓ Example: Checking Normal Forms Using FDs

Table:

```
plaintext
CopyEdit
```

Student(RollNo, Name, DeptID, DeptName)

FDs:

1. RollNo  $\rightarrow$  Name, DeptID
2. DeptID  $\rightarrow$  DeptName

### Analysis:

- Transitive Dependency: RollNo  $\rightarrow$  DeptID and DeptID  $\rightarrow$  DeptName  $\Rightarrow$  RollNo  $\rightarrow$  DeptName
  - Violates **3NF**
  - **Solution:** Split into two tables:
    - Student(RollNo, Name, DeptID)
    - Department(DeptID, DeptName)
- 

## Summary

Term	Meaning
Functional Dependency	Attribute relationship based on uniqueness
$X \rightarrow Y$	X determines Y (one X $\rightarrow$ one Y)
Trivial FD	$Y \subseteq X$
Non-trivial FD	$Y \not\subseteq X$
Transitive Dependency	$X \rightarrow Y, Y \rightarrow Z \Rightarrow X \rightarrow Z$
Multivalued Dependency	$X \twoheadrightarrow Y$
Closure ( $X^+$ )	All attributes functionally determined by X
Armstrong's Axioms	Set of inference rules to derive FDs

---

## Multivalued dependencies

### ◆ What is a Multivalued Dependency?

A **Multivalued Dependency (MVD)** occurs in a relation when one attribute **determines multiple independent values** of another attribute, **separately** from other attributes.

#### ◆ Formal Definition:

A multivalued dependency  $X \twoheadrightarrow Y$  in a relation **R** means:

For each value of **X**, there is a **set of values of Y** that is **independent of other attributes** in R.

---

## ✓ Notation:

- Written as:  $X \twoheadrightarrow Y$
  - Read as: "X multivalued determines Y"
- 

## 🔍 Example:

**Table:** Student (StudentID, Hobby, Language)

Sample Data:

StudentID	Hobby	Language
-----------	-------	----------

1	Reading	English
---	---------	---------

1	Painting	English
---	----------	---------

1	Reading	Hindi
---	---------	-------

1	Painting	Hindi
---	----------	-------

- Here, a student can have **multiple hobbies** and **speak multiple languages**, and both are **independent of each other**.
- Therefore:

```
sql
CopyEdit
StudentID  $\twoheadrightarrow$  Hobby
StudentID  $\twoheadrightarrow$  Language
```

---

## 💡 Key Points:

Concept	Explanation
Independence	The multivalued attributes are <b>not related</b> to each other
MVD vs FD	In <b>FD</b> ( $X \rightarrow Y$ ), Y has only one value per X In <b>MVD</b> ( $X \twoheadrightarrow Y$ ), Y can have <b>multiple values</b> for each X
Functional Dependency as MVD	Every FD is also a MVD, but not every MVD is a FD

---

## ❏ Properties of Multivalued Dependencies

1. **Complementation:**  
If  $X \twoheadrightarrow Y$ , then  $X \twoheadrightarrow (R - X - Y)$   
(If X determines Y, it also determines everything else **independent of Y**)
  2. **Augmentation:**  
If  $X \twoheadrightarrow Y$  and  $Z \subseteq R$ , then  $XZ \twoheadrightarrow Y$
  3. **Transitivity:**  
If  $X \twoheadrightarrow Y$  and  $Y \twoheadrightarrow Z$ , then  $X \twoheadrightarrow Z$  (under certain conditions)
  4. **Replication:**  
If  $X \rightarrow Y$  (a functional dependency), then  $X \twoheadrightarrow Y$
- 

## ⊗ Anomalies Caused by MVDs

### 1. Insertion Anomaly

You can't add a hobby for a student without adding a language (even though they're unrelated).

### 2. Deletion Anomaly

Deleting a language might unintentionally delete hobby info.

### 3. Update Anomaly

If a student has multiple entries, updating one value (e.g., hobby) may require updating all matching rows.

---

## ✓ Eliminating MVDs – Use Fourth Normal Form (4NF)

### ◆ Rule for 4NF:

A relation is in 4NF if:

- It is in **Boyce-Codd Normal Form (BCNF)**
  - It **does not have any non-trivial multivalued dependency**
- 

### 🔧 Conversion to 4NF – Decomposition Example:

From the earlier `Student (StudentID, Hobby, Language)`:

✗ *Not in 4NF:*

Multivalued dependencies exist:

- $\text{StudentID} \twoheadrightarrow \text{Hobby}$
- $\text{StudentID} \twoheadrightarrow \text{Language}$

✓ *4NF Decomposition:*

Split into two tables:

1.  $\text{StudentHobby}(\text{StudentID}, \text{Hobby})$
2.  $\text{StudentLanguage}(\text{StudentID}, \text{Language})$

Now:

- No multivalued dependencies
  - No unnecessary duplication
  - Fewer anomalies
- 

## Another Example:

**Table:**  $\text{Book}(\text{Title}, \text{Author}, \text{Genre})$

Data:

Title	Author	Genre
DB Design	C. J. Date	Education
DB Design	Silberschatz	Education
DB Design	C. J. Date	Reference
DB Design	Silberschatz	Reference

MVDs:

- $\text{Title} \twoheadrightarrow \text{Author}$
- $\text{Title} \twoheadrightarrow \text{Genre}$

This indicates that:

- Each book has multiple authors
- Each book belongs to multiple genres
- Authors and genres are **independent**



## Decompose to 4NF:

1. BookAuthor (Title, Author)
2. BookGenre (Title, Genre)

---

## □ Summary Table

Concept	Explanation
Multivalued Dependency	One attribute determines multiple values of another, <b>independently</b>
Notation	$X \twoheadrightarrow Y$
Normal Form Eliminating MVDs	<b>4NF</b>
Anomalies Removed	Insertion, Update, Deletion anomalies due to unrelated multivalued data
Detection	Observe repeated combinations of values where the attributes are unrelated

---

## ← END Conclusion

Multivalued dependencies are a critical concept in DBMS used to:

- Identify **redundancies** in a relation
- Guide **decomposition** for 4NF
- Prevent **anomalies** caused by unrelated data being stored together

## ◆ What is Decomposition?

**Decomposition** is the process of **splitting a relation (table) into two or more smaller relations** to:

- Eliminate redundancy
- Eliminate anomalies (insertion, deletion, update)
- Improve normalization
- Ensure data integrity

Goal: Break large, complex tables into simpler, more normalized ones **without losing data or meaning**.

---

## ✓ Properties of a Good Decomposition

1. **Lossless Join (or Non-lossy Decomposition)**  
→ No loss of information when decomposed relations are joined back.
  2. **Dependency Preservation**  
→ All functional dependencies in the original relation can still be enforced without joining tables.
  3. **No Redundancy**  
→ Avoid duplicate/redundant data.
- 

## ◆ Why Decompose?

Problem	Solution via Decomposition
Data redundancy	Remove repeated data
Insertion anomaly	Avoid inserting NULLs
Update anomaly	Avoid updating in multiple rows
Deletion anomaly	Avoid losing essential data
Violations of Normal Forms Convert to 1NF, 2NF, 3NF, etc.	

---

## 🔍 Types of Decomposition

### 1. Lossless Join Decomposition

- Ensures **no data is lost** when joining decomposed relations.
- It's **essential** for correctness.

### 2. Lossy Join Decomposition

- Some **data is lost or added** when decomposed relations are joined.
  - **Should be avoided** in practice.
- 

## ✓ Lossless Join – Formal Definition

A decomposition of relation **R** into **R1** and **R2** is **lossless w.r.t. a set of functional dependencies F** if:

```
nginx
CopyEdit
 $R1 \bowtie R2 = R$ 
```

That is, the **natural join** of **R1** and **R2** gives back the **original relation**.

---

## Lossless Join Condition (Using Functional Dependencies)

Let **R** be decomposed into **R1** and **R2**. The decomposition is **lossless** if:

```
nginx
CopyEdit
 $R1 \cap R2 \rightarrow R1 \quad \text{OR} \quad R1 \cap R2 \rightarrow R2$ 
```

That is, the **common attributes** must form a **super key** in at least one of the relations.

---

## Example – Lossless Join

**Given relation:**

```
plaintext
CopyEdit
Student(RollNo, Name, DeptID, DeptName)
```

**Functional Dependencies:**

```
markdown
CopyEdit
1. RollNo  $\rightarrow$  Name, DeptID
2. DeptID  $\rightarrow$  DeptName
```

**Decomposition:**

```
ini
CopyEdit
R1 = StudentInfo(RollNo, Name, DeptID)
R2 = Department(DeptID, DeptName)
```

**Common attribute: DeptID**

Check: Does  $\text{DeptID} \rightarrow R2 \text{ (DeptID, DeptName)}$ ?

✓ Yes ( $\text{DeptID} \rightarrow \text{DeptName}$ )

Hence, **lossless** decomposition.

---

## ✗ Example – Lossy Join

**Given:**

```
plaintext
CopyEdit
Employee(EmpID, EmpName, DeptID, DeptLocation)
```

**Decompose into:**

```
ini
CopyEdit
R1 = (EmpID, EmpName)
R2 = (DeptID, DeptLocation)
```

**Common attributes: None**

Join of R1 and R2 would create a **Cartesian product** → leads to **spurious tuples**

✗ This is a **lossy** decomposition.

---

## □ Dependency Preservation

A decomposition is **dependency preserving** if:

- The **functional dependencies** in the original relation can still be enforced **without joining** the decomposed relations.

Important for **maintaining data integrity** and **simplifying constraint enforcement**.

---

## 🔑 Summary Table

Term	Meaning
Decomposition	Splitting a relation into smaller ones
Lossless Join	No data loss after decomposition and rejoining
Condition for Lossless	$R1 \cap R2 \rightarrow R1$ OR $R1 \cap R2 \rightarrow R2$

Term	Meaning
Lossy Join	Causes information loss or incorrect tuples after join
Dependency Preservation	All FDs still hold after decomposition
Natural Join	Combines rows based on common attributes with equal values

---

## ✂ Real-World Example

**Table: Orders** (OrderID, CustomerID, CustomerName, ProductID, ProductName)

*Functional Dependencies:*

- OrderID  $\rightarrow$  CustomerID, ProductID
- CustomerID  $\rightarrow$  CustomerName
- ProductID  $\rightarrow$  ProductName

*Decompose into:*

1. Orders(OrderID, CustomerID, ProductID)
2. Customer(CustomerID, CustomerName)
3. Product(ProductID, ProductName)

- ✓ All joins are lossless (keys preserved)
  - ✓ All dependencies preserved
- 

## 🏁 Conclusion

Decomposition is a key concept in relational design. For a **successful decomposition**, ensure:

1. **Lossless join** – to preserve original data
  2. **Dependency preservation** – to retain all constraints
  3. **No redundancy** – to avoid anomalies
-

## Unit-III

### SQL: DDL, DML, DCL, TCL

## ◆ What is SQL?

**SQL (Structured Query Language)** is a standard language used to interact with **relational databases**. It includes different categories of commands to **create, manipulate, control, and maintain** database systems.

SQL commands are grouped into the following types:

Type	Full Form	Purpose
DDL	Data Definition Language	Defines database schema
DML	Data Manipulation Language	Manipulates data in tables
DCL	Data Control Language	Controls access to data
TCL	Transaction Control Language	Manages database transactions

---

## ◆ 1. Data Definition Language (DDL)

### ◆ Purpose:

DDL is used to define, modify, and delete **database schema objects** like tables, views, indexes, etc.

### □ Common DDL Commands:

Command	Description
CREATE	Creates a new table or database object
ALTER	Modifies an existing database object
DROP	Deletes a database object
TRUNCATE	Removes all rows from a table quickly
RENAME	Renames a database object

### ✓ Examples:

```
sql
CopyEdit
-- Create a table
CREATE TABLE Students (
    ID INT PRIMARY KEY,
    Name VARCHAR(50),
    Age INT
);

-- Add a column
```

```
ALTER TABLE Students ADD Email VARCHAR(100);

-- Delete a table
DROP TABLE Students;

-- Remove all data (structure remains)
TRUNCATE TABLE Students;

-- Rename table
RENAME TABLE Students TO Learners;
```

---

## ◆ 2. Data Manipulation Language (DML)

### ◆ Purpose:

DML is used to **manipulate and retrieve data** stored in database tables.

### □ Common DML Commands:

Command	Description
SELECT	Retrieves data from one or more tables
INSERT	Adds new data to a table
UPDATE	Modifies existing data
DELETE	Removes data from a table

### ✓ Examples:

```
sql
CopyEdit
-- Insert a record
INSERT INTO Students (ID, Name, Age) VALUES (1, 'John', 20);

-- Retrieve data
SELECT * FROM Students;

-- Update a record
UPDATE Students SET Age = 21 WHERE ID = 1;

-- Delete a record
DELETE FROM Students WHERE ID = 1;
```

---

## ◆ 3. Data Control Language (DCL)

### ◆ Purpose:

DCL is used to **control user access** to the database and its objects.

### □ Common DCL Commands:

Command	Description
GRANT	Gives privileges to users
REVOKE	Removes privileges from users

### ✓ Examples:

```
sql
CopyEdit
-- Grant SELECT privilege
GRANT SELECT ON Students TO user1;

-- Revoke privilege
REVOKE SELECT ON Students FROM user1;
```

---

## ◆ 4. Transaction Control Language (TCL)

### ◆ Purpose:

TCL manages **transactions** to ensure **data integrity** and **consistency**.

### □ Common TCL Commands:

Command	Description
COMMIT	Saves all changes made by the transaction
ROLLBACK	Undoes changes since the last COMMIT
SAVEPOINT	Sets a point within a transaction

### ✓ Examples:

```
sql
CopyEdit
-- Start a transaction (implicit)
INSERT INTO Students VALUES (2, 'Alice', 22);

-- Set a savepoint
SAVEPOINT sp1;

-- Make another change
UPDATE Students SET Age = 23 WHERE ID = 2;

-- Rollback to savepoint
ROLLBACK TO sp1;

-- Commit changes
COMMIT;
```

---

## 📖 Summary Table



SQL Type	Commands	Function
<b>DDL</b>	CREATE, ALTER, DROP, TRUNCATE, RENAME	Defines or changes structure of DB objects
<b>DML</b>	SELECT, INSERT, UPDATE, DELETE	Manipulates data inside tables
<b>DCL</b>	GRANT, REVOKE	Manages user permissions
<b>TCL</b>	COMMIT, ROLLBACK, SAVEPOINT	Controls changes and maintains consistency

---

## □ Key Points:

- **DDL commands** auto-commit changes (they are permanent immediately).
  - **DML commands** can be rolled back if enclosed within a transaction.
  - **DCL and TCL** help manage access control and consistency in multi-user environments.
- 

## □ Practice Tip

Try combining commands:

```
sql
CopyEdit
-- Create table
CREATE TABLE Account(ID INT, Name VARCHAR(50), Balance DECIMAL);

-- Insert data and use transaction control
BEGIN;
INSERT INTO Account VALUES (101, 'Raj', 1000);
SAVEPOINT sp1;
UPDATE Account SET Balance = Balance - 200 WHERE ID = 101;
ROLLBACK TO sp1;
COMMIT;
```

---

## ← END Conclusion

Understanding SQL command categories is essential for database administration, development, and query writing. Each type serves a unique purpose, and together they offer a **full-featured toolkit** for managing relational databases.

---

## Table creation, modification, and deletion

### 1 Table Creation

#### What is it?

Creating a table means defining a new table structure in the database to store data. This includes specifying the table name, columns, data types, and constraints.

#### Syntax (Basic):

```
sql
CopyEdit
CREATE TABLE table_name (
    column1 datatype constraints,
    column2 datatype constraints,
    ...
);
```

#### Common Data Types:

- INT – Integer numbers
- VARCHAR(size) – Variable-length string
- CHAR(size) – Fixed-length string
- DATE – Date values
- FLOAT, DECIMAL – Floating-point numbers

#### Common Constraints:

- PRIMARY KEY – Uniquely identifies each row
- NOT NULL – Column cannot be NULL
- UNIQUE – Values must be unique
- CHECK – Ensures data meets a condition
- DEFAULT – Default value if none provided
- FOREIGN KEY – References another table's key

#### Example:

```
sql
CopyEdit
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(50) NOT NULL,
    Age INT CHECK (Age >= 18),
    Email VARCHAR(100) UNIQUE,
    EnrollmentDate DATE DEFAULT CURRENT_DATE
);
```

---

### 2 Table Modification

## What is it?

Modifying a table means changing its structure after creation — adding, deleting, or changing columns or constraints.

## Using the `ALTER TABLE` command.

---

### Common Modifications:

Operation	Syntax Example	Description
Add a column	<code>ALTER TABLE table_name ADD column_name datatype;</code>	Adds a new column to the table
Drop a column	<code>ALTER TABLE table_name DROP COLUMN column_name;</code>	Removes a column from the table
Modify a column type	<code>ALTER TABLE table_name MODIFY column_name datatype;</code>	Changes the data type of a column
Rename a column	<code>ALTER TABLE table_name RENAME COLUMN old_name TO new_name;</code>	Renames a column (DBMS dependent)
Add a constraint	<code>ALTER TABLE table_name ADD CONSTRAINT constraint_name UNIQUE(column_name);</code>	Adds constraints like UNIQUE, FOREIGN KEY, etc.

---

### Examples:

- **Add a new column**

```
sql
CopyEdit
ALTER TABLE Students ADD Phone VARCHAR(15);
```

- **Drop a column**

```
sql
CopyEdit
ALTER TABLE Students DROP COLUMN Phone;
```

- **Modify a column data type**

```
sql
CopyEdit
ALTER TABLE Students MODIFY Age SMALLINT;
```

- **Rename a column (syntax may vary)**

```
sql
CopyEdit
ALTER TABLE Students RENAME COLUMN Email TO StudentEmail;
```

- **Add a foreign key constraint**

```
sql
CopyEdit
ALTER TABLE Students ADD CONSTRAINT fk_dept FOREIGN KEY (DeptID) REFERENCES
Departments (DeptID);
```

---

## 3 Table Deletion

### What is it?

Deleting a table means permanently removing the entire table and all its data from the database.

### Syntax:

```
sql
CopyEdit
DROP TABLE table_name;
```

- This operation **cannot be undone**.
- It removes both the data and the table structure.

### Example:

```
sql
CopyEdit
DROP TABLE Students;
```

---

## 4 Additional: Truncating a Table

### What is it?

TRUNCATE removes **all rows** from a table **without deleting the table structure**.

### Syntax:

```
sql
CopyEdit
TRUNCATE TABLE table_name;
```

- Faster than DELETE without a WHERE clause.
- Cannot be rolled back in most DBMS.
- Resets any auto-increment counters in some DBMS.

### Example:

```
sql
CopyEdit
```

```
TRUNCATE TABLE Students;
```

---

## ⚠ Summary Table of Commands

Operation	Command Syntax	Description
Create Table	<code>CREATE TABLE table_name (...);</code>	Defines a new table
Add Column	<code>ALTER TABLE table_name ADD column datatype;</code>	Adds new column
Drop Column	<code>ALTER TABLE table_name DROP COLUMN column;</code>	Deletes a column
Modify Column	<code>ALTER TABLE table_name MODIFY column datatype;</code>	Changes column datatype
Rename Column	<code>ALTER TABLE table_name RENAME COLUMN old TO new;</code>	Renames a column (DBMS specific)
Drop Table	<code>DROP TABLE table_name;</code>	Deletes entire table
Truncate Table	<code>TRUNCATE TABLE table_name;</code>	Deletes all rows, keeps structure

---

### □ Key Points:

- Use `CREATE TABLE` to define structure before inserting data.
  - Use `ALTER TABLE` carefully; some changes (like dropping columns) may cause data loss.
  - `DROP TABLE` permanently deletes table and data.
  - `TRUNCATE TABLE` is fast but deletes all data without logging individual row deletes.
  - Always back up important data before dropping or truncating.
- 

## 1 Indexes in DBMS

---

### ◆ What is an Index?

An **Index** is a **database object** that improves the speed of data retrieval operations on a table at the cost of additional writes and storage space.

- Think of an index like a **book's index** — it helps you quickly locate information without scanning the whole book.
  - Without an index, searching for a record might require scanning every row (full table scan).
  - With an index, the DBMS can quickly find the row(s) matching the search criteria.
-

## ◆ Purpose of Indexes

- **Speed up SELECT queries** by quickly locating data.
  - Help enforce **uniqueness** with UNIQUE indexes.
  - Improve performance for **JOINS**, **WHERE**, and **ORDER BY** clauses.
- 

## ◆ Types of Indexes

Type	Description
<b>Single-level Index</b>	Simple index on one or more columns
<b>Unique Index</b>	Ensures indexed column(s) have unique values
<b>Composite Index</b>	Index on multiple columns
<b>Clustered Index</b>	Data stored physically in order of the index (only one per table)
<b>Non-clustered Index</b>	Separate structure from data; table stored unordered
<b>Bitmap Index</b>	Efficient for columns with low cardinality (few distinct values)

---

## ◆ Creating an Index

```
sql
CopyEdit
CREATE INDEX index_name ON table_name(column_name);
```

## ◆ Creating a Unique Index

```
sql
CopyEdit
CREATE UNIQUE INDEX unique_index_name ON table_name(column_name);
```

---

## ◆ Examples

```
sql
CopyEdit
-- Create an index on the "Name" column of Students table
CREATE INDEX idx_name ON Students(Name);

-- Create a unique index on Email column
CREATE UNIQUE INDEX idx_email_unique ON Students(Email);

-- Composite index on (LastName, FirstName)
CREATE INDEX idx_name_composite ON Students(LastName, FirstName);
```

---

## ◆ Dropping an Index

```
sql
CopyEdit
```

```
DROP INDEX index_name;
```

---

### ◆ Important Notes:

- Indexes **speed up read** but **slow down write** operations (INSERT, UPDATE, DELETE).
  - Use indexes wisely; too many indexes can degrade performance.
  - Clustered index determines the **physical order** of data.
- 

## 2 Views in DBMS

---

### ◆ What is a View?

A **View** is a **virtual table** based on the result-set of an SQL query.

- It looks and behaves like a table but **does not store data physically**.
  - It is defined by a **SELECT query**.
  - When you query a view, the DBMS runs the underlying query to produce the data.
- 

### ◆ Purpose of Views

- Simplify complex queries by encapsulating them.
  - Enhance **security** by restricting access to certain columns or rows.
  - Present a **consistent, customized interface** to the data.
  - Can be used to aggregate or join data from multiple tables.
- 

### ◆ Creating a View

```
sql
CopyEdit
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

---

### ◆ Examples

```
sql
CopyEdit
-- View to show only active students
```

```
CREATE VIEW ActiveStudents AS
SELECT StudentID, Name, Age
FROM Students
WHERE Status = 'Active';

-- View to show student names with their department names (joining two
tables)
CREATE VIEW StudentDept AS
SELECT s.StudentID, s.Name, d.DeptName
FROM Students s
JOIN Departments d ON s.DeptID = d.DeptID;
```

---

## ◆ Querying a View

```
sql
CopyEdit
SELECT * FROM ActiveStudents;
```

---

## ◆ Updating a View

- Views are usually **read-only**.
  - Some DBMS allow **updatable views** if the view is simple (single table, no aggregates).
  - To update data through a view, it must map directly to the underlying table columns.
- 

## ◆ Dropping a View

```
sql
CopyEdit
DROP VIEW view_name;
```

---

## ⚠ Summary Table

Feature	Index	View
Definition	Data structure to speed up data retrieval	Virtual table based on a query
Stores Data?	Yes	No (virtual, stores query definition only)
Purpose	Improve query performance	Simplify queries, enhance security
Can be Updated?	No (except some advanced indexes like bitmap)	Sometimes (if simple view)
Example Use	Index on columns used in WHERE or JOIN	Provide limited data access or abstraction

---



## □ Key Points

- Use **indexes** to improve SELECT query performance but beware of overhead during INSERT/UPDATE/DELETE.
  - Use **views** to simplify complex SQL, enforce security, or present data in a user-friendly manner.
  - Views do not contain data; they run the underlying query every time you access them.
- 

## Nested Query

### ① What is a Nested Query?

A **Nested Query** or **Subquery** is an SQL query embedded inside another query (called the **outer query**).

- The subquery is executed **first** and its result is passed to the outer query.
  - Used to perform complex operations by breaking them into simpler queries.
  - Can be placed in SELECT, FROM, WHERE, or HAVING clauses.
- 

### ② Purpose of Nested Queries

- Filter data based on the results of another query.
  - Retrieve data that depends on other data.
  - Simplify complex SQL statements.
  - Avoid multiple round-trips between application and database.
- 

### ③ Types of Subqueries

Type	Description	Example Location
<b>Single-row</b>	Returns a single row/value	Used with =, <, >, <= etc. in WHERE
<b>Multiple-row</b>	Returns multiple rows/values	Used with IN, ANY, ALL
<b>Multiple-column</b>	Returns multiple columns	Used with tuple comparisons
<b>Correlated</b>	Subquery depends on outer query for values	Re-executed for each outer row
<b>Non-correlated</b>	Independent subquery	Executed once

---

## 4 Syntax Examples

### Basic Subquery in WHERE clause:

```
sql
CopyEdit
SELECT column1, column2
FROM table1
WHERE column1 = (SELECT column1 FROM table2 WHERE condition);
```

---

## 5 Detailed Examples

### Example 1: Single-row Subquery

Find employees with salary equal to the highest salary.

```
sql
CopyEdit
SELECT EmpName, Salary
FROM Employees
WHERE Salary = (SELECT MAX(Salary) FROM Employees);
```

- The subquery (SELECT MAX(Salary) FROM Employees) returns the maximum salary.
  - The outer query fetches employee(s) with that salary.
- 

### Example 2: Multiple-row Subquery with IN

Find employees who work in departments located in 'New York'.

```
sql
CopyEdit
SELECT EmpName, DeptID
FROM Employees
WHERE DeptID IN (SELECT DeptID FROM Departments WHERE Location = 'New York');
```

- Subquery returns all department IDs in New York.
  - Outer query finds employees in those departments.
- 

### Example 3: Multiple-column Subquery

Find employees who have the same salary and department as employee 'John'.

```
sql
CopyEdit
SELECT EmpName
```

```
FROM Employees
WHERE (Salary, DeptID) = (SELECT Salary, DeptID FROM Employees WHERE
EmpName = 'John');
```

---

### Example 4: Correlated Subquery

Find employees whose salary is greater than the average salary in their department.

```
sql
CopyEdit
SELECT EmpName, Salary, DeptID
FROM Employees e1
WHERE Salary > (
    SELECT AVG(Salary)
    FROM Employees e2
    WHERE e1.DeptID = e2.DeptID
);
```

- Subquery refers to `e1.DeptID` from the outer query.
  - Executed for each employee.
- 

### Example 5: Subquery in FROM clause (Derived Table)

Find average salary per department and then select departments with average salary > 50000.

```
sql
CopyEdit
SELECT DeptID, AvgSalary
FROM (
    SELECT DeptID, AVG(Salary) AS AvgSalary
    FROM Employees
    GROUP BY DeptID
) AS DeptAvg
WHERE AvgSalary > 50000;
```

---

## 6 Key Points About Nested Queries

- **Subqueries can be nested multiple levels deep.**
- Subqueries can return:
  - A single value (scalar)
  - A row (tuple)
  - A table (multiple rows and columns)
- Use **EXISTS** for checking existence of rows:

```
sql
CopyEdit
SELECT EmpName
FROM Employees e
WHERE EXISTS (
    SELECT 1 FROM Projects p WHERE p.EmpID = e.EmpID
```

);

- `EXISTS` returns `TRUE` if subquery returns at least one row.

---

## 7 Advantages of Nested Queries

- Improves query readability.
- Breaks complex logic into manageable parts.
- Allows writing powerful queries in a structured way.

---

## 8 Disadvantages

- Sometimes less efficient than joins.
- Correlated subqueries can be slow due to repeated execution.
- Not always intuitive for beginners.

---

## Summary Table

Subquery Type	Description	Example Use Case
Single-row	Returns single value	Compare column with one value
Multiple-row	Returns list of values	Use <code>IN</code> to filter multiple matches
Multiple-column	Returns multiple columns	Compare tuples
Correlated	Depends on outer query column	Filter rows conditionally
Non-correlated	Independent subquery	Used for aggregation or filtering

### Joins

## 1 What is a Join?

A **Join** is an SQL operation used to **combine rows from two or more tables** based on a related column between them.

- Joins help retrieve related data stored across multiple tables in a relational database.
- The result of a join is a new table containing columns from the joined tables.

---

## 2 Why Use Joins?

- To query data distributed across multiple tables.
  - To maintain normalization and avoid redundant data.
  - To analyze relationships between entities (e.g., employees & departments).
- 

## 3 Types of Joins

Join Type	Description
<b>INNER JOIN</b>	Returns rows with matching values in both tables
<b>LEFT JOIN (LEFT OUTER JOIN)</b>	Returns all rows from left table + matching from right; NULL if no match
<b>RIGHT JOIN (RIGHT OUTER JOIN)</b>	Returns all rows from right table + matching from left; NULL if no match
<b>FULL JOIN (FULL OUTER JOIN)</b>	Returns all rows when there is a match in either left or right table
<b>CROSS JOIN</b>	Returns Cartesian product (all possible combinations)
<b>SELF JOIN</b>	Joining a table with itself
<b>NATURAL JOIN</b>	Join automatically on all columns with the same name

---

## 4 Join Syntax and Examples

---

### 4.1 INNER JOIN

Returns only the rows where there is a match in both tables.

```
sql
CopyEdit
SELECT e.EmpID, e.Name, d.DeptName
FROM Employees e
INNER JOIN Departments d ON e.DeptID = d.DeptID;
```

---

### 4.2 LEFT JOIN (LEFT OUTER JOIN)

Returns all rows from the **left** table, and matched rows from the right table. If no match, NULL appears in right table columns.

```
sql
CopyEdit
SELECT e.EmpID, e.Name, d.DeptName
FROM Employees e
LEFT JOIN Departments d ON e.DeptID = d.DeptID;
```

---

### 4.3 RIGHT JOIN (RIGHT OUTER JOIN)

Returns all rows from the **right** table, and matched rows from the left table. NULL if no match on left.

```
sql
CopyEdit
SELECT e.EmpID, e.Name, d.DeptName
FROM Employees e
RIGHT JOIN Departments d ON e.DeptID = d.DeptID;
```

---

### 4.4 FULL JOIN (FULL OUTER JOIN)

Returns all rows where there is a match in either left or right table. NULLs for missing matches.

```
sql
CopyEdit
SELECT e.EmpID, e.Name, d.DeptName
FROM Employees e
FULL OUTER JOIN Departments d ON e.DeptID = d.DeptID;
```

*Note:* Some DBMS like MySQL don't support FULL OUTER JOIN directly.

---

### 4.5 CROSS JOIN

Returns the Cartesian product of rows from tables (every row from left joined with every row from right).

```
sql
CopyEdit
SELECT e.Name, d.DeptName
FROM Employees e
CROSS JOIN Departments d;
```

---

### 4.6 SELF JOIN

Join a table with itself, useful to find hierarchical data.

Example: Find employees and their managers.

```
sql
CopyEdit
SELECT e.EmpName AS Employee, m.EmpName AS Manager
FROM Employees e
LEFT JOIN Employees m ON e.ManagerID = m.EmpID;
```

---

### 4.7 NATURAL JOIN

Joins tables based on columns with the **same name and compatible data types** automatically.

```
sql
CopyEdit
SELECT *
FROM Employees
NATURAL JOIN Departments;
```

*Note:* Use with caution because it depends on column names.

---

## 5 Visual Example of INNER JOIN

**Employees Table**   **Departments Table**

EmpID	Name
-----	-----
1	John
2	Mary
3	Alex

**INNER JOIN on DeptID** returns:

EmpID	Name	DeptID	DeptName
1	John	10	Sales
2	Mary	20	Marketing
3	Alex	30	IT

---

## 6 Important Notes

- Join condition is specified by **ON** clause: which columns relate tables.
  - Without a join condition (in INNER JOIN), you get a **Cartesian product**.
  - Outer joins (LEFT, RIGHT, FULL) include rows with **no matching counterpart** with NULLs.
  - Performance depends on indexes on join columns.
- 

## 7 Summary Table of Joins

Join Type	Returns	Use Case
INNER JOIN	Only matching rows	Find matches between tables
LEFT OUTER JOIN	All rows left + matched right	Show all from left, match if any on right

Join Type	Returns	Use Case
RIGHT OUTER JOIN	All rows right + matched left	Show all from right, match if any on left
FULL OUTER JOIN	All rows matched or unmatched from both	Combine all data, keep unmatched too
CROSS JOIN	Cartesian product (all combinations)	Generate combinations or test
SELF JOIN	Table joined to itself	Hierarchies, comparisons within table
NATURAL JOIN	Automatically on common columns	Simple join when columns are named same

---

## Aggregate functions

### 1 What are Aggregate Functions?

**Aggregate Functions** are built-in SQL functions that perform a calculation on a set of values and return a single value.

- Used to summarize or analyze data.
- Operate on a **group of rows** rather than on individual rows.
- Commonly used with the `GROUP BY` clause but can also be used without it.

---

### 2 Common Aggregate Functions

Function	Description
<b>COUNT()</b>	Counts the number of rows or non-null values
<b>SUM()</b>	Calculates the sum of numeric values
<b>AVG()</b>	Calculates the average (mean) of numeric values
<b>MIN()</b>	Finds the minimum value
<b>MAX()</b>	Finds the maximum value

---

### 3 Syntax of Aggregate Functions

```
sql
CopyEdit
SELECT AGG_FUNC(column_name)
FROM table_name
WHERE condition;
```

Example:

```
sql
```



```
CopyEdit
SELECT COUNT(*) FROM Employees;
```

---

## 4 Examples of Aggregate Functions

### Example 1: COUNT()

Count the total number of employees.

```
sql
CopyEdit
SELECT COUNT(*) AS TotalEmployees
FROM Employees;
```

---

### Example 2: SUM()

Find the total salary paid to all employees.

```
sql
CopyEdit
SELECT SUM(Salary) AS TotalSalary
FROM Employees;
```

---

### Example 3: AVG()

Find the average salary of employees.

```
sql
CopyEdit
SELECT AVG(Salary) AS AverageSalary
FROM Employees;
```

---

### Example 4: MIN() and MAX()

Find the lowest and highest salary in the company.

```
sql
CopyEdit
SELECT MIN(Salary) AS MinSalary, MAX(Salary) AS MaxSalary
FROM Employees;
```

---

## 5 Using Aggregate Functions with GROUP BY

The `GROUP BY` clause groups rows that have the same values in specified columns into summary rows.

---

### Example: Total salary per department

```
sql
CopyEdit
SELECT DeptID, SUM(Salary) AS DeptSalaryTotal
FROM Employees
GROUP BY DeptID;
```

---

### Example: Count employees per department

```
sql
CopyEdit
SELECT DeptID, COUNT(*) AS NumberOfEmployees
FROM Employees
GROUP BY DeptID;
```

---

## 6 Using Aggregate Functions with HAVING Clause

- HAVING is used to filter groups after aggregation (like WHERE but for groups).
- 

### Example: Departments with more than 5 employees

```
sql
CopyEdit
SELECT DeptID, COUNT(*) AS NumberOfEmployees
FROM Employees
GROUP BY DeptID
HAVING COUNT(*) > 5;
```

---

## 7 Important Notes

- Aggregate functions **ignore NULL values** except COUNT (\*) which counts all rows.
  - You cannot mix aggregated and non-aggregated columns in SELECT without GROUP BY.
  - Can be nested inside other queries or combined with joins.
- 

## 8 Summary Table of Aggregate Functions

Function	Purpose	Example
COUNT	Number of rows or non-null values	SELECT COUNT(*) FROM Employees;
SUM	Sum of numeric column	SELECT SUM(Salary) FROM Employees;

Function	Purpose	Example
AVG	Average of numeric column	SELECT AVG(Salary) FROM Employees;
MIN	Minimum value	SELECT MIN(Salary) FROM Employees;
MAX	Maximum value	SELECT MAX(Salary) FROM Employees;

---

## Grouping and Ordering

### ◆ What is Grouping?

- **Grouping** is a way to arrange identical data into groups.
  - The **GROUP BY** clause groups rows that have the same values in specified columns.
  - Used primarily with **aggregate functions** (like `SUM()`, `COUNT()`, `AVG()`) to perform calculations on each group.
- 

### ◆ Syntax of GROUP BY

```
sql
CopyEdit
SELECT column1, AGG_FUNC(column2)
FROM table_name
WHERE condition
GROUP BY column1;
```

---

### ◆ How it works

- The table is split into groups based on unique values of the grouping column(s).
  - Aggregate functions calculate summary statistics per group.
  - You can group by one or multiple columns.
- 

### ◆ Example: Grouping by single column

Find total salary paid in each department.

```
sql
CopyEdit
SELECT DeptID, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY DeptID;
```

---

### ◆ Example: Grouping by multiple columns

Count employees by department and job role.

```
sql
CopyEdit
SELECT DeptID, JobRole, COUNT(*) AS EmployeeCount
FROM Employees
GROUP BY DeptID, JobRole;
```

---

### ◆ Important Points on Grouping

- Columns in `SELECT` that are not aggregated must be listed in `GROUP BY`.
  - `GROUP BY` is applied **after** filtering rows with `WHERE`.
  - To filter groups (not rows), use `HAVING` clause.
- 

### ◆ Filtering Groups: `HAVING`

Filter groups based on aggregate conditions.

Example: Departments with total salary > 100,000.

```
sql
CopyEdit
SELECT DeptID, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY DeptID
HAVING SUM(Salary) > 100000;
```

---

## 2 Ordering in DBMS (`ORDER BY`)

---

### ◆ What is Ordering?

- **Ordering** sorts the result rows of a query by one or more columns.
  - The `ORDER BY` clause controls the output sequence.
- 

### ◆ Syntax of `ORDER BY`

```
sql
CopyEdit
SELECT column1, column2
FROM table_name
WHERE condition
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC];
```

- ASC = ascending order (default).
- DESC = descending order.

---

## ◆ Examples of Ordering

*Example 1: Order employees by salary (ascending)*

```
sql
CopyEdit
SELECT EmpName, Salary
FROM Employees
ORDER BY Salary ASC;
```

*Example 2: Order employees by department descending and salary ascending*

```
sql
CopyEdit
SELECT EmpName, DeptID, Salary
FROM Employees
ORDER BY DeptID DESC, Salary ASC;
```

## 3 Combining Grouping and Ordering

You can group data and then order the groups based on aggregate values.

Example: List departments by total salary paid, highest first.

```
sql
CopyEdit
SELECT DeptID, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY DeptID
ORDER BY TotalSalary DESC;
```

## 4 Summary Table

Clause	Purpose	Example
GROUP BY	Group rows for aggregate calculation	GROUP BY DeptID
HAVING	Filter groups based on aggregate condition	HAVING COUNT(*) > 5
ORDER BY	Sort rows in ascending or descending order	ORDER BY Salary DESC

## 5 Important Notes

- WHERE filters rows **before** grouping.
- HAVING filters groups **after** grouping.

- ORDER BY sorts the final result set.
  - Can use column names or column positions in ORDER BY.
- 

## PL/SQL: Procedures,

### ✦ What is a PL/SQL Procedure?

A **Procedure** in PL/SQL is a **named block of code** that performs a specific task. It can accept input parameters, process them, and optionally return results via **OUT parameters**. Procedures are stored in the **database** and can be executed as needed.

---

### 🔧 Syntax of a PL/SQL Procedure

```
sql
CopyEdit
CREATE [OR REPLACE] PROCEDURE procedure_name
    (parameter1 [mode] datatype, parameter2 [mode] datatype, ...)
IS
    -- Declarations
BEGIN
    -- Executable statements
EXCEPTION
    -- Exception handling
END procedure_name;
/
```

#### Parameter Modes:

- IN – (Default) Used to pass a value into the procedure.
  - OUT – Used to return a value to the caller.
  - IN OUT – Used to pass an initial value and return an updated value.
- 

### ✓ Example 1: Simple Procedure (No Parameters)

This procedure displays a message.

```
sql
CopyEdit
CREATE OR REPLACE PROCEDURE say_hello IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hello, PL/SQL!');
END say_hello;
/

-- Execute it:
```

```
BEGIN
    say_hello;
END;
/
```

---

## ✓ Example 2: Procedure with IN Parameter

```
sql
CopyEdit
CREATE OR REPLACE PROCEDURE greet_user(p_name IN VARCHAR2) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hello, ' || p_name || '!');
END greet_user;
/

-- Call the procedure:
BEGIN
    greet_user('Alice');
END;
/
```

---

## ✓ Example 3: Procedure with OUT Parameter

```
sql
CopyEdit
CREATE OR REPLACE PROCEDURE get_square(
    p_number IN NUMBER,
    p_result OUT NUMBER
) IS
BEGIN
    p_result := p_number * p_number;
END get_square;
/

-- Call the procedure:
DECLARE
    result NUMBER;
BEGIN
    get_square(5, result);
    DBMS_OUTPUT.PUT_LINE('Square: ' || result);
END;
/
```

---

## ✓ Example 4: Procedure to Insert into Table

Assume a table `students(name VARCHAR2(50), marks NUMBER)`.

```
sql
CopyEdit
CREATE OR REPLACE PROCEDURE add_student(
    p_name IN VARCHAR2,
    p_marks IN NUMBER
```

```
) IS
BEGIN
    INSERT INTO students(name, marks)
    VALUES (p_name, p_marks);
    DBMS_OUTPUT.PUT_LINE('Student added successfully.');
```

---

```
END add_student;
/

-- Call:
BEGIN
    add_student('John', 85);
END;
/
```

---

## Modifying a Procedure

To modify a procedure, use `CREATE OR REPLACE` so you don't need to drop it first.

---

## ! Dropping a Procedure

```
sql
CopyEdit
DROP PROCEDURE procedure_name;
```

---

## Viewing Procedures in the Database

To list procedures:

```
sql
CopyEdit
SELECT object_name
FROM user_objects
WHERE object_type = 'PROCEDURE';
```

---

## Advantages of Using Procedures

1. **Modularization:** Code can be reused easily.
  2. **Maintainability:** Easier to manage logic.
  3. **Performance:** Stored procedures run on the server side, reducing network traffic.
  4. **Security:** Permissions can be granted to procedures without exposing underlying tables.
-



## Functions

### ✦ What is a PL/SQL Function?

A **function** in PL/SQL is a **named subprogram** that **performs a computation** and **returns a single value**. Like procedures, functions can accept parameters but **must return a value using the RETURN clause**.

---

### 🔧 Syntax of a PL/SQL Function

```
sql
CopyEdit
CREATE [OR REPLACE] FUNCTION function_name
    (parameter1 [mode] datatype, ...)
RETURN return_datatype
IS
    -- Variable declarations
BEGIN
    -- Function logic
    RETURN value;
EXCEPTION
    -- Exception handling
END function_name;
/
```

#### Note:

- Functions must return a **single value**.
  - The parameter mode in functions can be **IN** only (no **OUT** or **IN OUT** allowed directly).
- 

### ✓ Example 1: Simple Function (Returns Square of a Number)

```
sql
CopyEdit
CREATE OR REPLACE FUNCTION get_square(p_number IN NUMBER)
RETURN NUMBER
IS
    v_result NUMBER;
BEGIN
    v_result := p_number * p_number;
    RETURN v_result;
END;
/

-- Call the function in an anonymous block:
DECLARE
    res NUMBER;
```

```

BEGIN
    res := get_square(6);
    DBMS_OUTPUT.PUT_LINE('Square is: ' || res);
END;
/

```

---

## ✓ Example 2: Function to Calculate Factorial

```

sql
CopyEdit
CREATE OR REPLACE FUNCTION factorial(n IN NUMBER)
RETURN NUMBER
IS
    result NUMBER := 1;
BEGIN
    FOR i IN 1..n LOOP
        result := result * i;
    END LOOP;
    RETURN result;
END;
/

-- Call:
DECLARE
    res NUMBER;
BEGIN
    res := factorial(5);
    DBMS_OUTPUT.PUT_LINE('Factorial: ' || res);
END;
/

```

---

## ✓ Example 3: Function in SQL Statement

Assume a table `students(name VARCHAR2(50), marks NUMBER)`.

Create a function that gives grade based on marks:

```

sql
CopyEdit
CREATE OR REPLACE FUNCTION get_grade(m NUMBER)
RETURN VARCHAR2
IS
BEGIN
    IF m >= 90 THEN
        RETURN 'A';
    ELSIF m >= 75 THEN
        RETURN 'B';
    ELSIF m >= 60 THEN
        RETURN 'C';
    ELSE
        RETURN 'F';
    END IF;
END;
/

```

```
-- Use the function in a SELECT query:
SELECT name, marks, get_grade(marks) AS grade
FROM students;
```

---

## ❑ Differences Between Function and Procedure

Feature	Function	Procedure
Returns value	Must return a value	Does not need to return a value
Call in SQL	Can be called from SQL queries	Cannot be called from SQL directly
Use	Computation, returning results	Performing actions, like DML
Parameters	Only IN parameters	IN, OUT, and IN OUT supported

---

## ❑ Benefits of Functions in DBMS

1. **Encapsulation:** Logic in one place.
  2. **Reusability:** Use the same logic in multiple queries.
  3. **Modularity:** Easier to read and maintain code.
  4. **Performance:** Reduces query complexity when reused.
- 

## ! Dropping a Function

```
sql
CopyEdit
DROP FUNCTION function_name;
```

---

## 🔍 Viewing All Functions

```
sql
CopyEdit
SELECT object_name
FROM user_objects
WHERE object_type = 'FUNCTION';
```

---

## Triggers,

## 🚀 What is a Trigger in DBMS?

A **trigger** is a stored PL/SQL block that **automatically executes** (fires) **in response to specific events** on a table or view.

## 🔗 Common Triggering Events:

- INSERT
- UPDATE
- DELETE

## 📦 Use Case:

Triggers are useful for:

- Auditing (logging changes),
  - Enforcing business rules,
  - Preventing invalid transactions,
  - Automatically updating related tables.
- 

## 🔧 Syntax of a Trigger

```
sql
CopyEdit
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF}
{INSERT | UPDATE | DELETE} ON table_name
[FOR EACH ROW]
[WHEN (condition)]
BEGIN
    -- Trigger logic
END;
/
```

## Key Clauses:

- BEFORE: Trigger fires before the event.
  - AFTER: Trigger fires after the event.
  - FOR EACH ROW: Trigger fires once per affected row.
  - WHEN: Optional condition.
- 

## ✓ Example 1: BEFORE INSERT Trigger (Audit Logging)

Assume you have a table for user activity logs:

```
sql
CopyEdit
CREATE TABLE audit_log (
    username VARCHAR2(50),
    action_time DATE,
```

```
        action VARCHAR2(100)
    );
```

Create a trigger to log inserts on a users table:

```
sql
CopyEdit
CREATE OR REPLACE TRIGGER trg_user_insert
BEFORE INSERT ON users
FOR EACH ROW
BEGIN
    INSERT INTO audit_log (username, action_time, action)
    VALUES (:NEW.username, SYSDATE, 'New user inserted');
END;
/
```

### Explanation:

- :NEW.username accesses the new row being inserted.
  - This will log each insert into audit\_log.
- 

## ✓ Example 2: AFTER DELETE Trigger

Log deletion of records:

```
sql
CopyEdit
CREATE OR REPLACE TRIGGER trg_user_delete
AFTER DELETE ON users
FOR EACH ROW
BEGIN
    INSERT INTO audit_log (username, action_time, action)
    VALUES (:OLD.username, SYSDATE, 'User deleted');
END;
/
```

---

## ✓ Example 3: BEFORE UPDATE Trigger (Enforce Business Rule)

Prevent salary from being reduced:

```
sql
CopyEdit
CREATE OR REPLACE TRIGGER trg_prevent_salary_cut
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
    IF :NEW.salary < :OLD.salary THEN
        RAISE_APPLICATION_ERROR(-20001, 'Salary reduction not allowed');
    END IF;
END;
```

/

---

## ✓ Example 4: Trigger on Multiple Events

```
sql
CopyEdit
CREATE OR REPLACE TRIGGER trg_multi_event
AFTER INSERT OR DELETE OR UPDATE ON students
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('Students table was modified');
END;
/
```

---

## 📁 OLD and NEW Pseudorecords

- :OLD.column\_name – Value before the change (used in DELETE and UPDATE).
  - :NEW.column\_name – Value after the change (used in INSERT and UPDATE).
- 

## ☐ Types of Triggers

Type	Description
<b>BEFORE Trigger</b>	Executes before the triggering event
<b>AFTER Trigger</b>	Executes after the triggering event
<b>INSTEAD OF Trigger</b>	Used on views to simulate changes
<b>Row-Level Trigger</b>	Executes for each affected row
<b>Statement-Level Trigger</b>	Executes once per triggering statement

---

## ! Dropping a Trigger

```
sql
CopyEdit
DROP TRIGGER trigger_name;
```

---

## 🔍 Viewing Existing Triggers

```
sql
CopyEdit
SELECT trigger_name, table_name, triggering_event
FROM user_triggers;
```

---

## ⚠ Notes on Triggers

- Triggers should not perform **complex logic** or **DML operations** that affect the same table (can lead to **mutating table errors**).
  - Avoid overuse — triggers can make debugging and maintenance harder.
- 

## Cursor

### ✦ What is a Cursor in DBMS?

A **cursor** is a **pointer to a context area** in memory where SQL statements are processed. It allows **row-by-row processing** of the result set returned by a SQL query.

---

### 🔗 Types of Cursors in PL/SQL

Type	Description
<b>Implicit Cursor</b>	Automatically created by Oracle for single SQL statements like INSERT, UPDATE, DELETE, SELECT INTO.
<b>Explicit Cursor</b>	Manually declared and controlled by the programmer for queries that return more than one row.
<b>Cursor FOR Loop</b>	Simplifies looping through explicit cursors.
<b>Parameterized Cursor</b>	Accepts parameters to filter rows dynamically.

---

### ✓ Implicit Cursor Example

```
sql
CopyEdit
BEGIN
    UPDATE employees SET salary = salary + 500 WHERE department_id = 10;
    DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' rows updated.');
```

```
END;
/
```

- `SQL%ROWCOUNT` returns the number of affected rows.
  - Other attributes:
    - `SQL%FOUND`, `SQL%NOTFOUND`, `SQL%ISOPEN`
- 

### ✓ Explicit Cursor – Syntax

```

sql
CopyEdit
DECLARE
    CURSOR cursor_name IS
        SELECT_statement;

    variable1 table.column%TYPE;
    ...
BEGIN
    OPEN cursor_name;
    LOOP
        FETCH cursor_name INTO variable1, variable2, ...;
        EXIT WHEN cursor_name%NOTFOUND;
        -- Process data
    END LOOP;
    CLOSE cursor_name;
END;
/

```

---

## ✓ Example: Explicit Cursor

Print names and salaries of employees:

```

sql
CopyEdit
DECLARE
    CURSOR emp_cursor IS
        SELECT first_name, salary FROM employees;

    v_name employees.first_name%TYPE;
    v_salary employees.salary%TYPE;
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO v_name, v_salary;
        EXIT WHEN emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Name: ' || v_name || ', Salary: ' || v_salary);
    END LOOP;
    CLOSE emp_cursor;
END;
/

```

---

## ✓ Cursor FOR Loop – Simplified Syntax

```

sql
CopyEdit
BEGIN
    FOR emp_rec IN (SELECT first_name, salary FROM employees) LOOP
        DBMS_OUTPUT.PUT_LINE('Name: ' || emp_rec.first_name || ', Salary: '
|| emp_rec.salary);
    END LOOP;
END;
/

```

- Automatically **opens**, **fetches**, and **closes** the cursor.



---

## ✓ Parameterized Cursor Example

```
sql
CopyEdit
DECLARE
    CURSOR emp_cursor(dept_id NUMBER) IS
        SELECT first_name, salary FROM employees WHERE department_id =
dept_id;

    v_name employees.first_name%TYPE;
    v_salary employees.salary%TYPE;
BEGIN
    OPEN emp_cursor(10);
    LOOP
        FETCH emp_cursor INTO v_name, v_salary;
        EXIT WHEN emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Name: ' || v_name || ', Salary: ' || v_salary);
    END LOOP;
    CLOSE emp_cursor;
END;
/
```

---

## 🔍 Cursor Attributes

Attribute	Description
%FOUND	TRUE if fetch returns a row
%NOTFOUND	TRUE if fetch does <b>not</b> return a row
%ISOPEN	TRUE if cursor is open
%ROWCOUNT	Number of rows fetched so far

---

## ❑ Why Use Cursors?

- When processing **each row individually** is necessary.
  - To perform **row-level operations** (e.g., logging, transformation).
  - When a SQL query returns **multiple rows** and you want to iterate over them.
- 

## ⚠ Cursor Performance Tips

- Use **bulk collect** for better performance when processing large datasets.
  - Avoid unnecessary cursor usage; use **set-based operations** where possible.
-

## Closing a Cursor

Always `CLOSE` an explicit cursor when done to free up memory:

```
sql
CopyEdit
CLOSE cursor_name;
```

---

## Transactions: ACID properties

### What is a Transaction in DBMS?

A **transaction** is a **sequence of one or more SQL operations** (such as `INSERT`, `UPDATE`, `DELETE`) **treated as a single unit of work**.

- A transaction **must be either fully completed or fully failed** (rolled back).
  - Common SQL transaction control statements:
    - `COMMIT` – Saves the changes.
    - `ROLLBACK` – Undoes changes.
    - `SAVEPOINT` – Sets a point to roll back to.
    - `SET TRANSACTION` – Specifies characteristics of a transaction.
- 

## ACID Properties of Transactions

ACID is an acronym for the **four key properties** that ensure reliable processing of database transactions.

---

### 1. Atomicity (All or Nothing)

Ensures that all operations in a transaction **either complete successfully or fail as a whole**. If any part fails, the whole transaction is rolled back.

- ☐ Think of it like a light switch: it's either fully ON or fully OFF—never halfway.

#### **Example:**

```
sql
CopyEdit
BEGIN
    UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
    UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
    COMMIT; -- Both operations succeed
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK; -- If one fails, none take effect
```

```
END;  
/  

```

---

## 2. ✔ Consistency (Preserve Database Rules)

Ensures that a transaction brings the database from one **valid state to another**, maintaining all **data integrity constraints**.

- Constraints like primary key, foreign key, NOT NULL, etc., must remain valid.

### Example:

```
sql  
CopyEdit  
-- Suppose the constraint is balance >= 0  
BEGIN  
    UPDATE accounts SET balance = balance - 500 WHERE account_id = 1; --  
balance was 400  
    COMMIT; -- This violates the constraint → Transaction fails  
EXCEPTION  
    WHEN OTHERS THEN  
        ROLLBACK; -- Keeps the database in a consistent state  
END;  
/  

```

---

## 3. ✔ Isolation (No Interference)

Ensures that concurrent transactions do **not affect each other's execution**. Intermediate states of a transaction are **invisible** to others.

- Maintains **data accuracy in concurrent environments**.
- Isolation levels: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE

### Example:

Two users transferring money at the same time:

- Without isolation: they could both read the same balance before the other's update.
  - With isolation: one waits for the other to finish, preventing dirty reads.
- 

## 4. ✔ Durability (Permanent Changes)

Once a transaction is committed, its changes are **permanent**, even in case of system failure.

- Committed data is written to **non-volatile storage**.
- Ensures **data survivability** after crashes or shutdowns.

### Example:

```
sql
CopyEdit
UPDATE products SET quantity = quantity - 1 WHERE product_id = 10;
COMMIT;
-- Even if system crashes now, quantity change persists
```

---

## ✓ Example of Complete Transaction

```
sql
CopyEdit
BEGIN
    -- Step 1: Withdraw from Account A
    UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;

    -- Step 2: Deposit into Account B
    UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;

    -- If both succeed
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        -- Rollback on any failure
        ROLLBACK;
END;
/
```

---

## 💡 Summary Table

Property	Description	Ensures
Atomicity	All operations succeed or none at all	No partial updates
Consistency	Data remains valid according to constraints	Integrity of data
Isolation	Transactions don't interfere with each other	Correctness in concurrency
Durability	Committed changes are permanent	Survive crashes and failures

---

### Concurrency control

## ✦ What is Concurrency Control in DBMS?

**Concurrency control** in DBMS refers to the **management of simultaneous operations** (transactions) without conflicting with each other, ensuring **data integrity and consistency** in a **multi-user environment**.

### □ Why is it needed?

- Multiple users may **read/write the same data** at the same time.
- To **avoid conflicts, data loss, and inconsistency**.
- To **ensure ACID properties** (especially **Isolation** and **Consistency**).

---

## 🔄 Problems in Concurrency (Anomalies)

### 1. ☐ Lost Update Problem

Two transactions read the same data and update it, but one update is **overwritten** by the other.

**Example:**

```
text
CopyEdit
T1: Read salary = 5000
T2: Read salary = 5000
T1: Update salary = 5000 + 500 → 5500
T2: Update salary = 5000 + 1000 → 6000 (overwrites T1)
```

### 2. ☐ Dirty Read (Uncommitted Dependency)

One transaction reads data **modified by another transaction** that is **not yet committed**.

**Example:**

```
text
CopyEdit
T1: Update salary = 6000
T2: Read salary = 6000
T1: Rollback (salary back to 5000)
T2: Used a value that never existed
```

### 3. ☐ Non-repeatable Read

A transaction reads the **same data twice**, and gets **different values** because another transaction modified it between reads.

### 4. ☐ Phantom Read

A transaction reads a **set of rows**, and on re-execution sees **new rows** inserted by another transaction.

---

## ✓ Concurrency Control Techniques

### 1. 🗑️ Lock-Based Protocols

Locks restrict access to data to ensure consistency.

- **Shared Lock (S):** For reading.

- **Exclusive Lock (X):** For writing.

### Two-Phase Locking (2PL)

#### Two phases:

1. **Growing phase:** Transaction acquires all required locks.
2. **Shrinking phase:** Releases locks and cannot acquire new ones.

✓ Ensures **serializability** but may cause **deadlocks**.

---

## 2. □ Timestamp Ordering Protocol

Each transaction gets a **unique timestamp**. The DBMS uses these timestamps to order the execution of operations to avoid conflicts.

- Older transactions get priority.
  - Avoids deadlocks.
  - May lead to **starvation** of newer transactions.
- 

## 3. Optimistic Concurrency Control

Assumes conflict is rare:

- Transactions **execute without locks**.
  - Validation is done **before commit** to ensure no conflict occurred.
  - Best for **read-heavy systems** with low contention.
- 

## 4. □ Multiversion Concurrency Control (MVCC)

Maintains **multiple versions of data** to allow concurrent **reads and writes**.

- Readers don't block writers and vice versa.
  - Used in PostgreSQL, Oracle, and MySQL (InnoDB).
- 

## 5. □ Serialization Graph Checking

Constructs a **graph** of transactions and their dependencies.

- If the graph has **no cycles**, the schedule is serializable.
- Complex to implement.

---

## ❑ Example: Lock-Based Concurrency Control

```
sql
CopyEdit
-- Transaction T1
BEGIN;
SELECT balance FROM accounts WHERE id = 101 FOR UPDATE;
UPDATE accounts SET balance = balance - 100 WHERE id = 101;
COMMIT;

-- Transaction T2 (waits if T1 holds lock)
BEGIN;
SELECT balance FROM accounts WHERE id = 101 FOR UPDATE;
UPDATE accounts SET balance = balance + 100 WHERE id = 101;
COMMIT;
```

- FOR UPDATE locks the row.
- T2 will wait until T1 completes.

---

## 🛡 Isolation Levels (SQL Standard)

Isolation Level	Description	Problems Prevented
READ UNCOMMITTED	Lowest, allows dirty reads	None
READ COMMITTED	No dirty reads	✓ Dirty read
REPEATABLE READ	No dirty or non-repeatable reads	✓ Dirty + ✓ Non-repeatable read
SERIALIZABLE	Highest, no anomalies	✓ All anomalies

---

## ⚠ Deadlocks in Concurrency

A **deadlock** occurs when two or more transactions are **waiting on each other's resources indefinitely**.

### Example:

```
text
CopyEdit
T1 locks A → waits for B
T2 locks B → waits for A
```

✂ DBMS uses **deadlock detection** and **timeout mechanisms** to resolve.

---

## ✓ Best Practices

- Use proper isolation levels based on the use case.
- Use locks wisely (too many = low performance, too few = risk of conflicts).
- For large-scale applications, MVCC is a good choice.
- Handle exceptions and rollbacks properly.

---

## □ Summary

Problem	Technique/Tool
Lost Update	Locking, MVCC
Dirty Read	READ COMMITTED or higher isolation
Non-repeatable Read	REPEATABLE READ or SERIALIZABLE
Phantom Read	SERIALIZABLE, predicate locking
Deadlock	Timeout, wait-die/wound-wait strategies

---

## Locking techniques

## ✦ What is Locking in DBMS?

**Locking** is a concurrency control mechanism used in databases to **restrict access to data items** when multiple transactions are executing simultaneously. The goal is to **prevent data anomalies** (like dirty reads, lost updates) and **ensure data consistency**.

---

## 🔒 Why Use Locks?

To control problems caused by concurrent access such as:

- Lost updates
  - Dirty reads
  - Non-repeatable reads
  - Phantom reads
-



## 🔑 Types of Locks in DBMS

Lock Type	Description
<b>Binary Lock</b>	Each data item has two states: locked or unlocked.
<b>Shared Lock (S)</b>	Allows multiple transactions to <b>read</b> a data item, but not write to it.
<b>Exclusive Lock (X)</b>	Allows a transaction to <b>read and write</b> a data item exclusively.
<b>Read Lock</b>	Similar to shared lock — allows only reading.
<b>Write Lock</b>	Similar to exclusive lock — allows reading and writing.
<b>Intention Locks</b>	Used in <b>hierarchical locking</b> , such as table + row level locking.

---

## 🔄 Lock Granularity

- **Fine-grained locks:** row-level locking (more concurrency, more overhead).
  - **Coarse-grained locks:** table-level locking (less concurrency, less overhead).
- 

## ✓ Example 1: Shared vs Exclusive Lock

Let's say we have a table `accounts` with a row:

account_id	balance
101	5000

### Shared Lock

Multiple transactions can read it:

```
sql
CopyEdit
-- T1:
SELECT balance FROM accounts WHERE account_id = 101; -- Shared lock

-- T2:
SELECT balance FROM accounts WHERE account_id = 101; -- Shared lock allowed
```

### Exclusive Lock

Only one transaction can write:

```
sql
CopyEdit
-- T1:
UPDATE accounts SET balance = balance + 100 WHERE account_id = 101; --
Exclusive lock

-- T2:
```

```
UPDATE accounts SET balance = balance - 50 WHERE account_id = 101; -- Must wait until T1 finishes
```

---

## 🔄 Lock-Based Protocols

### 1. Two-Phase Locking (2PL)

Ensures **serializability** by dividing the execution of a transaction into two phases:

- **Growing Phase:** A transaction may acquire locks but cannot release any.
- **Shrinking Phase:** A transaction may release locks but cannot acquire any.

#### 🔒 Strict 2PL

- All **exclusive (write) locks are held until commit/rollback**.
  - Prevents **cascading rollbacks**.
- 

## 🔄 Example: Two Transactions

```
sql
CopyEdit
-- Transaction T1
BEGIN;
SELECT balance FROM accounts WHERE account_id = 101 FOR UPDATE;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 101;
COMMIT;

-- Transaction T2
BEGIN;
SELECT balance FROM accounts WHERE account_id = 101 FOR UPDATE; -- Waits for T1 to finish
UPDATE accounts SET balance = balance + 100 WHERE account_id = 101;
COMMIT;
```

Here, FOR UPDATE acquires a **write/exclusive lock**.

---

## ☐ Lock Compatibility Matrix

	Shared (S)	Exclusive (X)
Shared	✓ Yes	✗ No
Exclusive	✗ No	✗ No

---

## 🔄 Intention Locking (Advanced)

Used in multi-level locking (e.g., table and rows) to avoid conflicts between **row-level** and **table-level** locks.

Types:

- IS (Intention Shared)
  - IX (Intention Exclusive)
  - SIX (Shared and Intention Exclusive)
- 

## ⚡ Deadlocks and Locks

When two or more transactions **wait for each other to release locks**, causing a **cycle of waiting**:

**Example:**

```
text
CopyEdit
T1: Locks A → waits for B
T2: Locks B → waits for A
```

**Deadlock Resolution:**

- **Timeouts** (abort transaction after waiting too long)
  - **Deadlock detection algorithms** (wait-for graph)
- 

## ✓ Locking in SQL (Example – Oracle or MySQL)

```
sql
CopyEdit
-- Lock a row for update
SELECT * FROM employees WHERE emp_id = 100 FOR UPDATE;

-- Lock a table
LOCK TABLE employees IN EXCLUSIVE MODE;
```

---

## 💡 Best Practices for Locking

- Keep transactions **short** and **fast** to reduce lock time.
  - Use **row-level locking** for high concurrency needs.
  - Avoid **manual locking** unless necessary.
  - Choose the appropriate **isolation level**.
-

## □ Summary Table

Concept	Description
Shared Lock (S)	Multiple transactions can read
Exclusive Lock (X)	Only one transaction can write
Two-Phase Locking	Ensures serializability via locking phases
Strict 2PL	Prevents cascading rollbacks
Intention Locks	Used in multi-level locking scenarios
Lock Granularity	Row-level vs Table-level
Deadlock	Cycle of waiting — resolved by detection or timeout

### Deadlock handling

## ↻ What is a Deadlock in DBMS?

A **deadlock** is a situation in which **two or more transactions are waiting indefinitely** for resources (like locks) that are held by each other.

### □ Real-world analogy:

Two people holding a key the other needs, refusing to give it up — both are stuck!

---

## ⚠ Deadlock Condition (Coffman's Conditions)

A deadlock occurs when all the following four conditions hold simultaneously:

1. **Mutual Exclusion:** At least one resource is held in a non-shareable mode.
  2. **Hold and Wait:** A transaction holds one resource and waits for another.
  3. **No Preemption:** Resources cannot be forcibly taken away.
  4. **Circular Wait:** A circular chain of transactions exists, where each waits for a resource held by the next.
- 

## □ Example of a Deadlock

Consider two transactions:

```
text
CopyEdit
T1: Locks Resource A      → then waits for B
T2: Locks Resource B      → then waits for A
```

They **wait on each other** forever — this is a **deadlock**.

## SQL Example:

```
sql
CopyEdit
-- T1
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE id = 1; -- Locks row 1
-- waits for T2's lock on row 2

-- T2
BEGIN;
UPDATE accounts SET balance = balance + 100 WHERE id = 2; -- Locks row 2
-- waits for T1's lock on row 1
```

---

## 🛡️ Deadlock Handling Techniques

### 1. ✓ Deadlock Prevention

**Prevent at least one Coffman condition** to ensure deadlock cannot occur.

*Techniques:*

- **Resource ordering:** Assign a fixed order and always acquire in that order.
  - **Preemptive locking:** Force a transaction to release its locks.
  - **Wait-die and wound-wait protocols:**
    - **Wait-Die:** Older transaction waits, younger one is aborted.
    - **Wound-Wait:** Older transaction forces younger to release resource.
- 

### 2. 🔄 Deadlock Avoidance

**Check for potential deadlocks before allowing a transaction to proceed.**

*Algorithm: Wait-For Graph*

- Represent transactions and dependencies as a graph.
- If adding an edge creates a cycle, **do not grant the lock**.

*Example:*

- If  $T1 \rightarrow T2$  and  $T2 \rightarrow T3$  and  $T3 \rightarrow T1 \rightarrow \text{cycle} \rightarrow \text{deadlock}$
- 

### 3. 🚧♂ Deadlock Detection and Recovery

**Let deadlocks occur**, but detect and resolve them.

*Detection:*

- Construct a **Wait-For Graph (WFG)**.
- Check for cycles periodically.

*Recovery:*

- **Abort one or more transactions** to break the cycle.
  - Prefer to abort the **youngest, least resource-consuming**, or **non-critical** transaction.
- 

## 🔄 Example of Wait-For Graph

```
text
CopyEdit
T1 waiting for T2 → edge T1 → T2
T2 waiting for T3 → edge T2 → T3
T3 waiting for T1 → edge T3 → T1 → Cycle → Deadlock
```

---

## ❑ Deadlock Detection in SQL (MySQL Example)

In MySQL InnoDB engine:

```
sql
CopyEdit
SHOW ENGINE INNODB STATUS;
```

Output shows information about the most recent deadlock and the transaction that was rolled back.

---

## ⚙️ Deadlock Resolution Strategy (DBMS Behavior)

DBMS	Behavior
Oracle	Automatically detects and kills one transaction
MySQL	InnoDB detects and rolls back
SQL Server	Detects and chooses a victim

---

## 📁 Summary Table

Technique	Description	Example Tool/Method
Prevention	Disallow some conditions	Wait-die / Wound-wait
Avoidance	Predict deadlock using resource allocation info	Wait-for graph, ordering
Detection	Allow, then detect via cycle detection	Wait-for graph + rollback
Recovery	Abort transactions involved in a deadlock	Youngest transaction aborted

---

## ✓ Best Practices to Minimize Deadlocks

- Access tables and rows in the **same order** in all transactions.
  - Keep transactions **short** and **fast**.
  - **Avoid user interaction** within a transaction.
  - Use **row-level locking** instead of table-level when possible.
  - Set proper **isolation levels**.
- 

## Unit-IV

### File Organization

## 📁 What is File Organization in DBMS?

**File Organization** refers to the **way data is stored in files** on disk. It defines the method of arranging **records within a file** to enable efficient access, update, and retrieval operations.

---

## □ Why File Organization is Important?

- Determines **speed of access** to records.
  - Affects the **efficiency of storage** and **query performance**.
  - Influences **insertion, deletion, and updating** speed.
- 

## 📁 Types of File Organization

Type	Description	Use Case
1. Heap (Unordered)	Records stored in the order they arrive	Simple inserts

Type	Description	Use Case
2. Sequential	Records stored sorted by a key field	Batch processing, reports
3. Hashed	Uses a hash function to map key to location	Fast direct access
4. Clustered	Records of related tables stored together physically	Optimized for joins
5. Indexed Sequential	Combines sequential storage with indexing	Fast lookup + range access

---

## 1 Heap File Organization (Unordered)

- Records are **inserted at the end** of the file.
- No specific order.
- Searching requires **scanning the whole file** (linear search).

### ✓ Advantages:

- Fast for inserts.
- Easy to maintain.

### ✗ Disadvantages:

- Slow for search/update/delete.

### 📌 Example:

Insert Records:

→ R1 → R2 → R3 → R4

Stored as:

[R1] [R2] [R3] [R4]

---

## 2 Sequential File Organization

- Records stored in **sorted order** of a key (e.g., Employee ID).
- **Binary search** can be used.
- Efficient for **range queries** and **batch updates**.

### ✓ Advantages:

- Fast for reading sorted data.
- Efficient range-based searches.



### ✗ Disadvantages:

- Inserting a new record requires shifting.
- Costly updates and deletions.

### ✚ Example:

Sorted by ID:

[101] [104] [106] [110]

---

## 3 Hash File Organization

- Uses a **hash function** on key field to determine the record's address.
- Provides **direct access**.

### ✓ Advantages:

- Very fast for exact match lookups.
- Best for **equality searches**.

### ✗ Disadvantages:

- Doesn't support range queries well.
- May have **collisions** → handled by chaining or open addressing.

### ✚ Example:

Hash function:  $\text{key} \% 10$

Keys: 23, 34, 12 → stored at 3, 4, 2

---

## 4 Clustered File Organization

- Records from **multiple related tables** stored **physically close together**.
- Enhances performance of **frequent joins**.

### ✓ Advantages:

- Optimized for complex queries and joins.

### ✗ Disadvantages:

- Complicated to maintain.
- Slower insert/update/delete if clustering key changes.

### ✚ Example:

Orders and Customers frequently joined on Customer\_ID → stored together on disk.

---

## 5 Indexed Sequential File Organization

- Combines **sequential** storage with **an index** for fast access.
- Index stores **pointers to records**.

### ✓ Advantages:

- Efficient for both direct and range access.
- Faster than pure sequential files.

### ✗ Disadvantages:

- Requires extra space for the index.
- Index needs to be maintained.

### ✈ Example:

Index:

[101] → Block 1

[106] → Block 2

Data:

Block 1: [101, 102, 103]

Block 2: [106, 107, 109]

---

## □ Comparison Table

File Type	Search Time	Insert Time	Best For
Heap	$O(n)$	$O(1)$	Simple inserts
Sequential	$O(\log n)$	$O(n)$	Range queries, sorted access
Hash	$O(1)$	$O(1)$	Direct access
Clustered	$O(\log n)$	$O(\log n)$	Joins, grouped data
Indexed Seq.	$O(\log n)$	$O(n)$	Mixed search and batch work

---

## ✓ Best File Type Per Use Case

Use Case	File Organization
Equality search (e.g., by ID)	Hashed

Use Case	File Organization
Sorted reports or range query	Sequential or Indexed
High insert frequency	Heap
Frequent joins (multi-table)	Clustered
Mixed queries (fast access + sort)	Indexed Sequential

---

## ← END Conclusion

File organization is crucial for:

- **Database performance**
- **Efficient storage management**
- **Optimized query processing**

## Indexing

### 🔑 What is Indexing in DBMS?

**Indexing** is a data structure technique used to **quickly retrieve records** from a database file. Just like an index in a book helps you find a topic quickly, a database index allows the DBMS to find rows **without scanning the entire table**.

---

### □ Why Indexing?

Without an index, a DBMS would need to **scan every row** in a table to find matching records (called **full table scan**), which is inefficient for large tables.

Indexing improves:

- **Search performance**
  - **Sorting**
  - **Join performance**
- 

### 🔑 How Indexing Works

An index creates a **mapping between a key column and its data row location**. When a query involves that column, the DBMS uses the index to jump directly to the data.

## Analogy:

Like using an index in a textbook:

**Topic → Page Number**

In DBMS:

**Column value → Row address (pointer)**

---

## □ Types of Indexing

Type	Description
1. <b>Primary Index</b>	Created automatically on primary key
2. <b>Secondary Index</b>	Created on non-primary columns (for search optimization)
3. <b>Clustered Index</b>	Sorts the actual table data based on key
4. <b>Non-Clustered</b>	Index and data stored separately
5. <b>Unique Index</b>	Ensures uniqueness of values in a column
6. <b>Composite Index</b>	Created on multiple columns
7. <b>B-Tree Index</b>	Balanced tree structure (common default)
8. <b>Bitmap Index</b>	Uses bitmap arrays, suitable for low-cardinality columns

---

### 1 Primary Index

- Automatically created on **primary key**.
- Records are **physically ordered** based on this index.
- Only **one primary index** per table.

📌 *Example:*

```
sql
CopyEdit
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(50)
);
```

---

### 2 Secondary Index

- Created on columns **other than the primary key**.
- Useful for **frequent searches** on non-key columns.

🚀 *Example:*

```
sql
CopyEdit
CREATE INDEX idx_name ON employees(name);
```

---

### 3 Clustered Index

- The data is **physically sorted** based on the indexed column.
- Only **one clustered index** per table.
- Often used **by default** on the primary key.

🚀 *Example:*

```
sql
CopyEdit
CREATE CLUSTERED INDEX idx_salary ON employees(salary);
```

---

### 4 Non-Clustered Index

- Index is stored separately from table data.
- Can create **multiple non-clustered indexes** on a table.

🚀 *Example:*

```
sql
CopyEdit
CREATE NONCLUSTERED INDEX idx_dept ON employees(department_id);
```

---

### 5 Unique Index

- Enforces **unique values** in a column.
- Similar to a UNIQUE constraint.

🚀 *Example:*

```
sql
CopyEdit
CREATE UNIQUE INDEX idx_email ON users(email);
```

---

### 6 Composite Index

- Indexes **two or more columns** together.
- Useful for queries using **WHERE col1 AND col2**.

🚀 *Example:*

```
sql
CopyEdit
CREATE INDEX idx_name_dept ON employees(name, department_id);
```

---

## 7 B-Tree Index (Balanced Tree)

- Most common index type in DBMS.
  - Keeps index balanced for  $O(\log n)$  performance.
  - Efficient for range and exact match queries.
- 

## 8 Bitmap Index

- Uses bits instead of tree structure.
- Efficient for **low-cardinality columns** (e.g., gender, status).

🔗 *Example:*

A column “gender” with values M or F:

```
text
CopyEdit
M → 1 0 1 0 0
F → 0 1 0 1 1
```

---

## VS Clustered vs Non-Clustered Index

Feature	Clustered Index	Non-Clustered Index
Data Order	Data is sorted with index	Index is separate from data
Number per Table	Only one	Many allowed
Speed	Faster for range queries	Faster for direct lookup

---

## ✓ Advantages of Indexing

- **Faster data retrieval**
  - Improves **performance of WHERE, JOIN, ORDER BY, and GROUP BY**
  - **Reduces I/O operations**
- 

## ✗ Disadvantages of Indexing

- **Consumes extra storage**
- Slower **INSERT, DELETE, UPDATE** due to index maintenance
- Too many indexes → **degraded performance**

---

## □ Practical Example

### Table:

```
sql
CopyEdit
CREATE TABLE products (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    price DECIMAL(10, 2),
    category VARCHAR(20)
);
```

### Create an index on price:

```
sql
CopyEdit
CREATE INDEX idx_price ON products(price);
```

### Now a query:

```
sql
CopyEdit
SELECT * FROM products WHERE price > 500;
```

→ Will use `idx_price` for faster retrieval.

---

## □ When to Use Indexing?

Use indexing on:

- Columns used frequently in **WHERE**, **JOIN**, or **ORDER BY**
- Large tables with many rows
- Columns with high selectivity (many unique values)

Avoid indexing:

- Small tables
- Columns with low cardinality (few unique values)
- Frequently updated columns

---

## □ Summary Table

Index Type	Key Feature	Best Use Case
Primary Index	Auto-created on primary key	Uniquely identifying rows
Secondary Index	On non-primary key columns	Frequent queries on non-key fields
Clustered	Data sorted with index	Range queries
Non-Clustered	Separate index	General fast lookup
Unique Index	Enforces uniqueness	Email, ID, etc.
Composite Index	Multiple columns	Combined WHERE conditions
Bitmap Index	Bit representation	Low-cardinality fields

---

## Hashing techniques

### Hashing Techniques in DBMS

#### What is Hashing?

Hashing is a technique to **directly access data** in a database by computing the **address** of the data from its key using a **hash function**. It converts a search key into a **hash value** (an address or bucket number), enabling **fast data retrieval**.

---

#### Why Use Hashing?

- To avoid scanning entire files.
- To achieve **constant time,  $O(1)$ , average-case data access**.
- Efficient for **equality searches** (e.g., find record with key = 123).

---

#### How Hashing Works?

1. A **hash function**  $h(k)$  takes a search key  $k$  and returns an address (bucket number).
2. The record with key  $k$  is stored at the address  $h(k)$ .
3. When searching, apply the hash function and directly access the bucket.

---

#### Characteristics of Hashing



Feature	Description
Direct Access	Hash function maps key → bucket address
Efficient Search	Average $O(1)$ time for search
Works Best for	Equality search ( <code>WHERE key = value</code> )
Poor for	Range queries ( <code>WHERE key BETWEEN a AND b</code> )

---

## Types of Hashing Techniques

### 1. Static Hashing

- The number of buckets is fixed.
  - Hash function maps keys to these fixed buckets.
  - Problem: **Overflow buckets** if many keys map to the same bucket.
- 

### 2. Dynamic Hashing

- Number of buckets can **grow or shrink dynamically**.
  - Good for databases with **frequent insertions/deletions**.
  - Examples: **Extendible hashing**, **Linear hashing**.
- 

## 1 Static Hashing Details

- Use hash function like:  

$$h(k) = k \bmod N$$
 where  $N$  is the fixed number of buckets.
- If multiple keys hash to same bucket → **Collision** occurs.
- Collisions handled by:
  - **Chaining** (linked list of records per bucket)
  - **Open addressing** (probing)

### Example:

```

text
CopyEdit
Keys: 12, 22, 32, 42
Buckets: 5


$$h(k) = k \bmod 5$$



$$12 \bmod 5 = 2 \rightarrow \text{Bucket } 2$$


$$22 \bmod 5 = 2 \rightarrow \text{Bucket } 2 \text{ (collision with 12)}$$


```

$32 \bmod 5 = 2 \rightarrow \text{Bucket } 2 \text{ (collision again)}$   
 $42 \bmod 5 = 2 \rightarrow \text{Bucket } 2 \text{ (collision again)}$

So, bucket 2 contains 4 records  $\rightarrow$  overflow!

---

## 2 Dynamic Hashing Techniques

### a. Extendible Hashing

- Uses a **directory** with pointers to buckets.
- Directory size grows with data.
- Hash function uses first  $i$  bits of the key.
- When bucket overflows, split bucket and possibly double directory size.

*Example:*

- Initial directory size:  $2^1 = 2$  entries.
  - If bucket overflows, directory doubles  $\rightarrow 2^2 = 4$  entries.
- 

### b. Linear Hashing

- Buckets are split **one at a time**.
  - Hash function changes gradually.
  - No need for directory doubling.
- 

## Collision Handling Techniques

### 1. Chaining (Separate Chaining)

- Each bucket contains a linked list of records that hash to it.
- Easy to implement, handles overflow gracefully.

### 2. Open Addressing

- All records stored in the bucket array.
  - If collision, find next available bucket by probing:
    - **Linear probing:** next bucket sequentially
    - **Quadratic probing:** jumps quadratically
    - **Double hashing:** uses second hash function to find next bucket
-

# Example of Chaining

Bucket      Records

0	
1	11 → 21 → 31
2	12 → 22
3	13
4	

## Advantages of Hashing

- **Fast data retrieval** for equality searches.
- Efficient for large datasets.
- Simple collision resolution methods.

## Disadvantages of Hashing

- Poor performance for **range queries**.
- Handling collisions can add complexity.
- Static hashing suffers from overflow.
- Dynamic hashing requires extra overhead to manage directory or bucket splits.

## Summary Table

Hashing Type	Description	Pros	Cons
Static Hashing	Fixed buckets, fixed hash fn	Simple, fast for small data	Overflow, inflexible
Extendible Hashing	Directory-based dynamic size	Grows dynamically, less overflow	Directory overhead
Linear Hashing	Incremental bucket splitting	Smooth expansion, simple	Slightly complex logic
Chaining	Linked list per bucket	Handles overflow well	Extra memory for pointers

Hashing Type	Description	Pros	Cons
Open Addressing	Probe for next free slot	Space efficient	Clustering, more probes needed

---

## Sample SQL Example (conceptual)

Assume you want to simulate hashing on a column `student_id`:

```
sql
CopyEdit
-- Create hash index (some DBMS support it)
CREATE INDEX idx_student_hash ON students(student_id) USING HASH;
```

Queries like:

```
sql
CopyEdit
SELECT * FROM students WHERE student_id = 1234;
```

Use the hash index for fast lookup.

---

## B+ Trees

### B+ Trees in DBMS

---

#### What is a B+ Tree?

A **B+ Tree** is a **balanced tree data structure** widely used in DBMS for indexing large amounts of data. It maintains sorted data and allows **efficient insertion, deletion, and search operations**.

---

#### Why B+ Tree?

- Ideal for **disk-based storage** (minimizes disk reads).
  - Supports **range queries** efficiently.
  - Guarantees **logarithmic height**, ensuring fast access.
- 

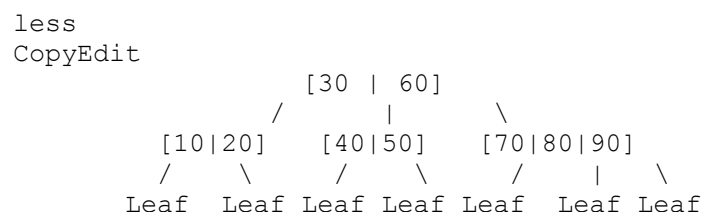
#### Structure of B+ Tree

- It is a **multi-level index**.
- Consists of:
  - **Internal nodes**: store keys and pointers to child nodes.
  - **Leaf nodes**: store actual data pointers or records.
- **All data records are stored only in leaf nodes.**
- Leaf nodes are linked together as a **linked list** for efficient range queries.
- Internal nodes store keys to guide searches, but **no actual data**.

## Key Properties

- A B+ Tree of order  $m$  satisfies:
  - Each internal node can have **at most  $m$  children**.
  - Each internal node (except root) has at least  **$\lceil m/2 \rceil$  children**.
  - All leaf nodes are at the **same level** (balanced).
  - Leaf nodes store between  **$\lceil (m-1)/2 \rceil$  and  $(m-1)$  keys**.
  - Internal nodes store  **$\lceil m/2 \rceil - 1$  to  $m - 1$  keys**.

## Visual Representation (Example: B+ Tree of order 4)



- Internal nodes have up to 3 keys (order 4 means max 4 children).
- Leaves contain actual data pointers.
- Leaves linked for range queries.

## Operations on B+ Tree

### 1. Search

- Start at root.
- Compare search key with keys in node.
- Follow pointer to correct child.
- Repeat until reaching leaf.
- Search leaf node for key.

**Cost:**  $O(\log_m n)$ , where  $n$  is number of keys,  $m$  is order.

## 2. Insertion

- Insert key in correct leaf node.
  - If leaf node overflows (exceeds max keys), **split** the leaf.
  - Middle key moves up to parent.
  - If parent overflows, split recursively.
  - May cause root to split → tree height increases by 1.
- 

## 3. Deletion

- Remove key from leaf node.
  - If underflow occurs (fewer than minimum keys), try to **borrow** from sibling.
  - If borrowing not possible, **merge** with sibling.
  - Update parent keys accordingly.
  - May cause recursive merging up to root.
- 

## Advantages of B+ Tree

- All data at leaf level → internal nodes smaller → more keys per node → **shallower tree**.
  - Efficient **range queries** because leaves are linked.
  - Balanced tree → guaranteed performance.
  - Disk-friendly: nodes sized to disk blocks.
- 

## Comparison with B-Tree

Feature	B-Tree	B+ Tree
Data storage	Keys and data in all nodes	Data stored only in leaves
Leaf nodes linked	No	Yes
Range queries	Less efficient	More efficient
Tree height	Usually taller	Usually shorter

---

## Example Scenario

Suppose a B+ tree index is created on a `Student_ID` column in a students table. When a query searches for `Student_ID = 105`, the B+ tree is traversed from root to leaf to find the pointer to the record.

Range queries like `Student_ID BETWEEN 100 AND 200` can efficiently scan linked leaf nodes without traversing back up.

---

## Summary Table

Term	Description
Order (m)	Max children per internal node
Internal Node	Stores keys and pointers, no data
Leaf Node	Stores actual data pointers or records
Balanced	All leaves at same depth
Height	$O(\log_m n)$ , small for large datasets
Linked Leaves	Enables efficient range queries

---

## Pseudocode: Search in B+ Tree

```
text
CopyEdit
function BPlusTreeSearch(node, key):
    if node is leaf:
        return search key in leaf records
    else:
        for i in keys of node:
            if key < keys[i]:
                return BPlusTreeSearch(child[i], key)
        return BPlusTreeSearch(last_child, key)
```

---

## Query Processing

### Query Processing in DBMS

Query processing is the series of steps that a Database Management System (DBMS) follows to execute a user query and retrieve the desired results efficiently.

---

### Why Query Processing?

- Users write queries in high-level languages like SQL.
  - DBMS needs to **interpret, optimize, and execute** queries.
  - Efficient query processing improves system performance.
- 

### Main Goals of Query Processing

- Translate the query into an efficient internal form.

- Optimize the query execution plan.
  - Minimize resource usage (CPU, memory, disk I/O).
  - Return correct results quickly.
- 

## Stages of Query Processing

### 1. Parsing and Translation

- The DBMS checks query **syntax** and **semantics**.
- Translates SQL query into an **internal representation** (e.g., relational algebra or query tree).

#### Example:

```
sql
CopyEdit
SELECT name FROM students WHERE age > 20;
```

- Translated to relational algebra:  
 $\pi_{\text{name}}(\sigma_{\text{age} > 20}(\text{students}))$
- 

### 2. Query Optimization

- Generate different **query execution plans** (strategies).
- Estimate cost of each plan (based on I/O, CPU, memory).
- Choose the **least costly plan**.

**Example:** Join order, choice of indexes, access methods.

---

### 3. Query Evaluation (Execution)

- Execute the chosen query plan.
  - Access the database files, perform joins, selections, projections.
  - Return the result to the user.
- 

## Important Concepts in Query Processing

### a) Query Tree

- A tree representing relational algebra operations.
- Leaves: relations (tables)



- Internal nodes: operations (selection, projection, join)
- 

## b) Query Optimization Techniques

- **Heuristic Optimization:** Use rules like push selections down, combine projections.
  - **Cost-Based Optimization:** Use statistics (table size, indexes) to estimate cost.
  - **Join Order Optimization:** Reorder joins for efficiency.
- 

## Query Execution Example

Given the query:

```
sql
CopyEdit
SELECT S.name, C.course_name
FROM Students S, Enrollments E, Courses C
WHERE S.student_id = E.student_id AND E.course_id = C.course_id AND S.age > 20;
```

### Execution Steps:

1. **Join** Students and Enrollments on `student_id`.
  2. **Join** result with Courses on `course_id`.
  3. **Select** students with `age > 20`.
  4. **Project** `name` and `course_name`.
- 

## Query Processing Example: Using Index

If `Students` table has an index on `age`:

- Use index to find students with `age > 20` quickly.
  - Then join filtered students with Enrollments.
  - Then join with Courses.
- 

## Query Execution Plans

- **Nested Loop Join:** For each row in outer table, scan inner table.
  - **Sort-Merge Join:** Sort both tables on join key, then merge.
  - **Hash Join:** Build hash table on smaller relation, probe with larger.
-

## Summary Table

Stage	Description
Parsing & Translation	Syntax check, convert to relational algebra
Optimization	Choose best plan via heuristics or cost
Execution	Perform operations, return result

---

## Key Takeaways

- Query processing transforms SQL into efficient low-level operations.
  - Query optimization is critical for performance.
  - Different join algorithms exist for different scenarios.
  - Indexes can drastically improve query speed.
- 

### Query Optimization

## Query Optimization in DBMS

Query optimization is the process of selecting the most efficient way to execute a given query by considering possible query plans.

---

### Why Query Optimization?

- Multiple ways to execute the same query.
  - Different execution strategies have different costs.
  - Goal: minimize resource usage (CPU, memory, I/O) and response time.
- 

## Key Components of Query Optimization

### 1. Query Execution Plan (QEP)

- A roadmap of operations to execute the query.
- Includes order of operations, join methods, access paths.

### 2. Cost Estimation

- Cost depends on CPU, disk I/O, network usage.
- DBMS uses statistics like table size, indexes, data distribution.
- Goal: estimate cost of each plan to choose the cheapest.

---

# Types of Query Optimization

## a) Heuristic Optimization

- Uses rules or heuristics (experience-based rules).
- Example heuristics:
  - Push selections (WHERE) as close to base tables as possible.
  - Perform projections early to reduce tuple size.
  - Join smaller tables first.

## b) Cost-Based Optimization

- Considers cost of each possible execution plan.
  - Uses statistics to estimate cost.
  - Explores different join orders, access methods.
  - Picks plan with least estimated cost.
- 

# Query Optimization Steps

1. **Parsing & translation** → Generate query tree (relational algebra).
  2. **Apply heuristics** → Rewrite query tree to reduce cost.
  3. **Generate alternative plans** → Different join orders, methods.
  4. **Estimate cost** of each plan.
  5. **Select best plan** for execution.
- 

# Important Concepts

## a) Selection Pushdown

- Move selection operations down the tree to reduce intermediate results.

## b) Join Ordering

- Reorder joins to reduce size of intermediate results.
  - For example, join smaller relations first.
- 

# Example Query

sql  
CopyEdit

```
SELECT *
FROM Employees E, Departments D, Projects P
WHERE E.dept_id = D.dept_id AND D.project_id = P.project_id AND E.salary > 50000;
```

---

## Possible Join Orders:

- (E JOIN D) JOIN P
- (D JOIN P) JOIN E
- (E JOIN P) JOIN D

Each order might have different costs based on table sizes and indexes.

---

## Using Selection Pushdown

Apply `E.salary > 50000` before join to filter employees first, reducing tuples early.

---

## Join Algorithms and Cost Impact

Join Algorithm	Description	When to Use
Nested Loop Join	For each row in outer, scan inner	Small tables or indexes exist
Sort-Merge Join	Sort both inputs, then merge	Large sorted data
Hash Join	Build hash on smaller table, probe larger	Large unsorted tables

---

## Cost Estimation Factors

- Number of tuples in tables.
  - Number of tuples after selection.
  - Presence of indexes.
  - I/O cost to read pages.
  - CPU cost for processing tuples.
- 

## Summary Table

Optimization Technique	Description
Selection Pushdown	Apply WHERE early to reduce data
Projection Pushdown	Reduce columns early
Join Reordering	Change join order to reduce intermediate sizes
Use of Indexes	Use indexes for faster data access

---

## Example: Heuristic Optimization

Given:

```
sql
CopyEdit
SELECT name FROM Employees WHERE salary > 50000;
```

- Instead of scanning entire Employees table, apply selection first using an index on salary.
- If index exists, use index scan → faster retrieval.

---

### – Cost estimation

## Cost Estimation in DBMS

### What is Cost Estimation?

Cost estimation is the process by which the DBMS query optimizer estimates the **resource usage** (cost) of different query execution plans. The optimizer uses these estimates to choose the most efficient plan.

---

### Why Cost Estimation?

- Many query execution plans exist for the same SQL query.
- Cost estimation helps predict which plan will run fastest and use fewer resources.
- Resources considered: **disk I/O, CPU time, memory usage**, network costs.

---

## Components of Cost

Component	Description
<b>I/O Cost</b>	Cost to read/write data blocks from/to disk (usually dominant cost).
<b>CPU Cost</b>	Cost of processing tuples, comparisons, joins, etc.
<b>Memory Cost</b>	Cost related to memory usage for sorting, hashing, buffering.

---

## Cost Model Basics

- DBMS uses a **cost model** to estimate costs.

- Usually **disk I/O** cost dominates, so it's the main factor.
- Costs are **estimated, not exact**.
- Statistics like number of tuples, tuple size, index selectivity used.

---

## Important Statistics for Cost Estimation

Statistic	Meaning
<b>Cardinality (N)</b>	Number of tuples in a relation
<b>Tuple size (T)</b>	Average size of each tuple (in bytes)
<b>Selectivity (S)</b>	Fraction of tuples qualifying a predicate ( $0 \leq S \leq 1$ )
<b>Number of pages (P)</b>	Number of disk pages relation occupies

---

## Example: Cost Estimation for Selection

Query:

```
sql
CopyEdit
SELECT * FROM Employees WHERE salary > 50000;
```

- Employees table has 10,000 tuples.
- Average tuple size = 100 bytes.
- Disk page size = 4 KB.
- Number of pages =  $(10,000 * 100) / 4096 \approx 245$  pages.
- Selectivity of salary > 50000 is 0.1 (10%).

### Cost Estimate:

- **Full table scan cost:** Reading all 245 pages  $\rightarrow$  245 I/O.
  - **Using index:** Assume index is a B+ tree with height 3.
    - Traverse index: 3 I/O.
    - Retrieve matching tuples: 10% of 10,000 = 1,000 tuples.
    - Suppose these 1,000 tuples spread over 500 pages.
    - Total cost  $\approx 3 + 500 = 503$  I/O (more than full scan, so may choose full scan).
- 

## Cost Estimation for Joins

### Nested Loop Join

Cost =  
 $\text{Cost}(\text{outer}) + (\text{Number of tuples in outer}) * \text{Cost}(\text{inner})$

If outer relation has 1000 tuples and inner requires scanning 2000 pages:

$\text{Cost} = \text{Cost}(\text{outer}) + 1000 * \text{Cost}(\text{inner})$   
If  $\text{Cost}(\text{outer}) = 100$  pages,  $\text{Cost}(\text{inner}) = 2000$  pages,  
Total cost =  $100 + 1000 * 2000 = 2,000,100$  pages (very high).

---

## Sort-Merge Join

Cost =  
 $\text{Cost}(\text{sort outer}) + \text{Cost}(\text{sort inner}) + \text{Cost}(\text{merge})$

- Sorting cost depends on number of pages and available memory.
  - Merge cost is linear in total pages.
- 

## Hash Join

Cost =  
 $\text{Cost}(\text{build hash table on smaller relation}) + \text{Cost}(\text{probe hash with larger relation})$

---

## Summary Table of Cost Components

Operation	Cost Formula	Notes
Full Table Scan	Number of pages in table	Simple, but expensive on large tables
Index Scan	Height of index + number of pages for tuples	Effective if selective
Nested Loop Join	$\text{Cost}(\text{outer}) + (\text{tuples in outer}) * \text{Cost}(\text{inner})$	Expensive if both large
Sort-Merge Join	$\text{Cost}(\text{sort outer}) + \text{Cost}(\text{sort inner}) + \text{Cost}(\text{merge})$	Efficient for large sorted data
Hash Join	$\text{Cost}(\text{build hash}) + \text{Cost}(\text{probe})$	Efficient for large unsorted data

---

## Practical Example

Suppose:

- Relation R: 1000 tuples, 100 pages.
- Relation S: 500 tuples, 50 pages.

Estimate Nested Loop Join (R as outer):

- $\text{Cost}(\text{R}) = 100$  pages.

- $\text{Cost}(S) = 50$  pages.
- Total cost =  $100 + 1000 * 50 = 50,100$  pages.

Estimate Hash Join:

- Build hash on smaller S: 50 pages.
  - Probe R: 100 pages.
  - Total cost =  $50 + 100 = 150$  pages → Much cheaper.
- 

## Conclusion

- Cost estimation relies heavily on statistics and heuristics.
  - Accurate statistics improve plan choice.
  - Disk I/O generally dominates cost.
  - Cost estimation enables the optimizer to pick efficient query plans.
- 

## Query execution plan in dbms



## Query Execution Plan (QEP) in DBMS

---

### What is a Query Execution Plan?

A **Query Execution Plan (QEP)** is a detailed roadmap or blueprint that the Database Management System (DBMS) creates to execute a SQL query efficiently. It shows the sequence of operations, access methods, join algorithms, and data retrieval strategies that the DBMS will follow to produce the result.

---

### Why is Query Execution Plan Important?

- The same SQL query can be executed in multiple ways.
  - Different plans may have vastly different performance.
  - The DBMS optimizer generates multiple possible plans and selects the most cost-effective plan.
  - Understanding QEP helps database developers optimize queries and improve performance.
- 

### Components of a Query Execution Plan



Component	Description
Access Methods	How data is accessed — full table scan, index scan, etc.
Join Methods	Algorithms for joining tables — nested loops, hash join, merge join
Operation Order	The sequence of operations like filtering, joining, sorting
Cost Estimation	Estimated resource cost for each operation and total plan
Output Rows	Estimated number of rows at each step

---

## How is QEP Represented?

- Usually represented as a **tree** or **hierarchical plan**.
  - **Leaves** are base table access (scans, index reads).
  - **Internal nodes** are relational operations (joins, filters).
  - The plan flows bottom-up, with the final output at the root.
- 

## Example 1: Simple Query Execution Plan

### SQL Query:

```
sql
CopyEdit
SELECT name FROM Employees WHERE age > 30;
```

### Possible QEP steps:

1. Use an **index scan** on the `age` column (if an index exists).
2. Retrieve the `name` column for the qualifying rows.
3. Return the result set.

If no index exists, a **full table scan** may be done instead.

---

## Example 2: Join Query Execution Plan

### SQL Query:

```
sql
CopyEdit
SELECT E.name, D.dept_name
FROM Employees E
JOIN Departments D ON E.dept_id = D.dept_id
WHERE E.salary > 50000;
```

### Possible QEP:

- Step 1: **Filter** Employees where `salary > 50000` using an index or scan.

- Step 2: Access Departments table (full scan or index scan).
- Step 3: **Join** Employees and Departments on dept\_id using:
  - **Nested Loop Join** (for small tables or indexed joins),
  - **Hash Join** (for large unsorted tables),
  - **Sort-Merge Join** (if tables are sorted).
- Step 4: **Project** columns name and dept\_name.
- Step 5: Return results.

---

## Common Operations in QEP

Operation	Description
Table Scan	Sequentially reads all rows in the table
Index Scan	Uses index to access only qualifying rows
Filter	Applies WHERE clause conditions
Projection	Selects specific columns
Join	Combines rows from multiple tables
Sort	Sorts rows for ORDER BY or merge joins
Aggregation	Computes aggregates like SUM, COUNT

---

## Join Algorithms in QEP

Join Type	Description	Best For
Nested Loop Join	For each row in outer table, scan inner table	Small datasets, indexed joins
Sort-Merge Join	Sort both tables, then merge based on join keys	Large sorted tables
Hash Join	Build hash table on smaller table, probe with larger	Large unsorted tables

---

## Tools to View QEP

- **Oracle:** EXPLAIN PLAN FOR <SQL> + SELECT \* FROM TABLE (DBMS\_XPLAN.DISPLAY ( ) ) ;
  - **MySQL:** EXPLAIN <SQL>
  - **PostgreSQL:** EXPLAIN ANALYZE <SQL>
  - **SQL Server:** Graphical execution plan or SET SHOWPLAN\_XML ON
- 

## Sample Output of EXPLAIN (MySQL)

sql

CopyEdit

```
EXPLAIN SELECT * FROM Employees WHERE age > 30;
```

id	select_type	table	type	possible_keys	key	rows	Extra
1	SIMPLE	Employees	range	age_index	age_index	100	Using where

- **type:** Access method (range means index range scan).
- **key:** Index used.
- **rows:** Estimated rows scanned.
- **Extra:** Additional info.

---

## Summary Table

Aspect	Details
Purpose	Plan how DBMS executes the query
Representation	Tree or step-by-step operations
Key Components	Access methods, join methods, filters, projections
Importance	Performance tuning and optimization
Tools	EXPLAIN commands in various DBMS

---

### Performance tuning

## ⚡ Performance Tuning in DBMS

Performance tuning in DBMS refers to the process of improving the efficiency and speed of database operations such as queries, transactions, and overall system throughput. The goal is to reduce response time, optimize resource usage, and handle larger loads effectively.

---

### Why Performance Tuning is Important?

- Databases often handle large volumes of data and complex queries.
- Poorly tuned databases can lead to slow response times.
- Efficient performance enhances user experience and system scalability.
- Reduces hardware costs by making better use of existing resources.

---

## Key Areas of Performance Tuning

### 1. Query Tuning

- Optimize SQL queries to reduce execution time.
- Use proper indexing.

- Avoid unnecessary columns and rows in SELECT.
- Rewrite queries for efficiency (e.g., using joins instead of subqueries).
- Analyze query execution plans and optimize bottlenecks.

**Example:**

Avoid `SELECT *` when only specific columns are needed:

```
sql
CopyEdit
SELECT name, salary FROM Employees WHERE dept_id = 10;
```

## 2. Index Tuning

- Create indexes on columns used in WHERE clauses, JOINS, ORDER BY.
- Drop unused indexes to reduce overhead.
- Use composite indexes when multiple columns are often used together.

**Example:**

Index on salary for quick filtering:

```
sql
CopyEdit
CREATE INDEX idx_salary ON Employees(salary);
```

## 3. Database Design Tuning

- Normalize to reduce redundancy, or denormalize for performance if needed.
- Partition large tables for better I/O performance.
- Use appropriate data types and constraints.

## 4. Memory and Cache Tuning

- Allocate adequate memory to buffer pools and caches.
- Tune cache sizes to reduce disk I/O.
- Use database features like query result caching.

## 5. Configuration and Resource Tuning

- Configure DBMS parameters for connection pooling, parallelism.
- Optimize transaction log size and disk placement.
- Balance workload across CPUs and disks.

## 6. Locking and Concurrency Control

- Minimize locking contention by using appropriate isolation levels.
  - Use row-level locking instead of table-level when possible.
  - Avoid long transactions.
-

# Performance Tuning Tools in DBMS

- **Explain Plan / Query Plan Analyzers:** Visualize execution plans.
  - **Profiler and Trace Tools:** Monitor query execution time and resource usage.
  - **Statistics Collector:** Provides data about table size, index usage.
  - **Automated Tuning Advisors:** Some DBMS have advisors recommending tuning actions.
- 

## Example: Query Performance Tuning

### Original query:

```
sql
CopyEdit
SELECT * FROM Orders WHERE customer_id IN (SELECT customer_id FROM
Customers WHERE city = 'New York');
```

- This uses a subquery, which can be inefficient.

### Tuned query:

```
sql
CopyEdit
SELECT O.*
FROM Orders O
JOIN Customers C ON O.customer_id = C.customer_id
WHERE C.city = 'New York';
```

- Using a JOIN often improves performance.
- 

## Example: Index Usage

Suppose you often query:

```
sql
CopyEdit
SELECT * FROM Employees WHERE department = 'Sales' AND salary > 50000;
```

### Indexing:

```
sql
CopyEdit
CREATE INDEX idx_dept_salary ON Employees(department, salary);
```

- Composite index speeds up queries filtering on both columns.
-

## Summary Table

Tuning Area	Techniques
Query Tuning	Optimize SQL, rewrite queries, use EXPLAIN
Index Tuning	Create/drop indexes, use composite indexes
Database Design	Normalize/denormalize, partitioning
Memory & Cache	Allocate buffer pools, cache tuning
Configuration	Tune DB parameters, parallelism, logging
Locking & Concurrency	Use appropriate isolation levels, minimize locks

## Distributed databases

### What is a Distributed Database?

A **Distributed Database** is a collection of multiple, logically interrelated databases distributed over a computer network. Each site (node) stores part or whole of the database and is capable of processing queries independently or cooperatively.

---

### Key Characteristics

- **Data Distribution:** Data is stored across multiple physical locations (sites).
  - **Autonomy:** Each site can operate independently.
  - **Transparency:** The system hides the complexity of distribution from users (location transparency, replication transparency).
  - **Reliability & Availability:** Failure at one site doesn't affect the entire system.
  - **Scalability:** Easily add new sites to the system.
- 

### Types of Distributed Databases

Type	Description
<b>Homogeneous</b>	All sites use the same DBMS software.
<b>Heterogeneous</b>	Different DBMS software at different sites.

---

### Data Distribution Strategies

Strategy	Description	Example
<b>Fragmentation</b>	Break database into smaller pieces (fragments).	Horizontal or vertical fragmentation
<b>Replication</b>	Copy entire or part of database at multiple sites.	Full or partial replication

Strategy	Description	Example
<b>Allocation</b>	Combination of fragmentation and replication.	Different fragments replicated at different sites

---

## Types of Fragmentation

- **Horizontal Fragmentation:** Partition table rows based on some condition.  
Example: Employee table split by department.
  - **Vertical Fragmentation:** Partition table columns.  
Example: Employee table split into (Employee\_ID, Name) and (Employee\_ID, Salary).
  - **Hybrid Fragmentation:** Combination of horizontal and vertical.
- 

## Advantages of Distributed Databases

- Improved reliability and availability.
  - Localized data access improves performance.
  - Parallel processing capabilities.
  - Better scalability and flexibility.
- 

## Challenges in Distributed Databases

- Complexity in design and management.
  - Ensuring data consistency across sites.
  - Distributed transaction management.
  - Query processing and optimization over network.
  - Handling network failures and synchronization.
- 

## Example Scenario

Consider a multinational company with branches in **New York** and **London**.

- Employees data in New York branch is stored in the New York site.
  - Employees data in London branch is stored in the London site.
  - Queries accessing local data are fast.
  - Some queries need data from both sites and involve distributed joins.
- 

## Example: Horizontal Fragmentation

Employee table

**EmpID Name Dept Location**

101	Alice	Sales	New York
102	Bob	HR	London
103	Carol	Sales	New York
104	Dave	IT	London

**Fragmentation:**

- Fragment 1 (New York): Rows where Location = 'New York'
- Fragment 2 (London): Rows where Location = 'London'

Each fragment stored in respective site.

---

**Distributed Query Processing**

When a query accesses distributed data, the DBMS:

- Decomposes the query into sub-queries.
- Executes sub-queries locally at relevant sites.
- Transfers and combines results.

---

**Distributed Transaction Management**

- Ensures **ACID** properties across multiple sites.
- Uses protocols like **Two-Phase Commit (2PC)** to maintain atomicity and consistency.

---

**Summary Table**

Aspect	Description
Definition	Database spread over multiple sites
Types	Homogeneous, Heterogeneous
Data Distribution	Fragmentation, Replication, Allocation
Advantages	Reliability, Availability, Performance
Challenges	Complexity, Consistency, Transaction Mgmt
Query Processing	Distributed execution and result merging

---

**NoSQL databases****What is NoSQL?**



**NoSQL** (Not Only SQL) databases are a category of database management systems designed to handle large volumes of diverse and rapidly changing data that do not fit well into traditional relational database schemas.

- They provide flexible schema designs.
- Often used for big data, real-time web applications, and unstructured data.
- Designed for scalability, high availability, and distributed architectures.

---

## Why NoSQL?

- Traditional RDBMS are limited by rigid schemas and scalability issues.
- NoSQL offers:
  - Schema flexibility (can handle semi-structured or unstructured data).
  - Horizontal scaling across commodity servers.
  - High throughput and low latency.
  - Support for diverse data types.

---

## Types of NoSQL Databases

Type	Description	Examples
<b>Document Store</b>	Store data as JSON-like documents (key-value pairs with nested data).	MongoDB, CouchDB
<b>Key-Value Store</b>	Data stored as simple key-value pairs.	Redis, DynamoDB
<b>Column Family</b>	Store data in columns rather than rows; good for analytical queries.	Cassandra, HBase
<b>Graph Database</b>	Model data as nodes and edges; ideal for relationships.	Neo4j, Amazon Neptune

---

## Characteristics of NoSQL

Feature	Description
Schema-less	Data can be stored without a fixed schema.
Scalability	Built to scale horizontally across many servers.
High Availability	Designed for distributed environments and fault tolerance.
Flexible Data Models	Support for diverse data formats (documents, graphs, etc.).
Eventually Consistent	Some NoSQL systems relax strict ACID guarantees for scalability.

---

## Example: Document Store (MongoDB)

json

CopyEdit

```
{
  "_id": "1001",
  "name": "Alice",
  "age": 30,
  "skills": ["Python", "MongoDB", "Node.js"],
  "address": {
    "street": "123 Maple St",
    "city": "New York",
    "zip": "10001"
  }
}
```

- Data stored as a document (JSON-like).
- Nested fields and arrays are supported.
- Query language is flexible and powerful.

---

## Example: Key-Value Store (Redis)

- Stores data as pairs: key -> value.
- Example: Store user session data.

plaintext

CopyEdit

```
SET user:1001 "session_data_here"
GET user:1001
```

---

## Advantages of NoSQL

- Handles large volumes of structured, semi-structured, and unstructured data.
- Flexible data models adapt to application needs.
- Designed for distributed and cloud environments.
- High performance for specific use cases (e.g., caching, real-time analytics).
- Easier to evolve application data over time without migrations.

---

## Disadvantages of NoSQL

- Lack of standardization compared to SQL.
- Limited support for complex transactions (some provide eventual consistency).
- Learning curve and tooling ecosystem can be immature.
- Not always suitable for applications requiring strong ACID compliance.

---

## Use Cases for NoSQL

- Social networks (storing user profiles, connections).

- Real-time analytics.
- Content management systems.
- IoT data storage.
- Caching layers.

---

### Summary Table

Aspect	Details
Definition	Non-relational, schema-flexible databases
Types	Document, Key-Value, Column, Graph
Strengths	Scalability, flexibility, high performance
Weaknesses	Limited ACID, no standard query language
Use Cases	Big data, real-time apps, flexible schema apps

---

### Data Warehousing –

#### What is Data Warehousing?

A **Data Warehouse (DW)** is a centralized repository that stores large volumes of historical data collected from multiple heterogeneous sources for the purpose of query, analysis, and reporting.

- It supports **decision making** by providing a consolidated view of organizational data.
- Data is integrated, cleaned, and structured specifically for analysis.
- Typically used in Business Intelligence (BI) and analytics.

---

#### Key Features of Data Warehouse

Feature	Description
Subject-Oriented	Organized around key subjects like sales, customers, products.
Integrated	Data from various sources is combined into a consistent format.
Non-volatile	Data is stable and not frequently updated or deleted.
Time-variant	Maintains historical data with time stamps for trend analysis.
Supports Complex Queries	Optimized for read-heavy operations and complex analytical queries.

---

#### Architecture of Data Warehouse

1. **Data Sources:** Operational databases, external sources.
2. **ETL Process:** Extraction, Transformation, and Loading of data into the warehouse.
3. **Data Warehouse Storage:** Central repository for integrated data.

4. **Metadata:** Data about data — schema, definitions.
5. **Query Tools & OLAP:** Tools for reporting, querying, and multidimensional analysis.
6. **Users:** Decision-makers, analysts, business users.

---

## Types of Data Warehouse

Type	Description
<b>Enterprise Data Warehouse (EDW)</b>	Centralized data warehouse for entire organization.
<b>Data Mart</b>	Subset of data warehouse focused on a specific business line or department.

---

## Data Modeling in Data Warehouse

- Uses **Star Schema** or **Snowflake Schema** for organizing data.

### Star Schema:

- Central **Fact Table** containing measures (e.g., sales amount).
- Multiple **Dimension Tables** with descriptive attributes (e.g., date, product, customer).

### Example:

#### Fact Table: Sales

sale\_id, date\_id, product\_id, customer\_id, amount

| Dimension Tables: Date, Product, Customer |

---

## OLAP vs OLTP

Aspect	OLAP (Data Warehouse)	OLTP (Operational DB)
Purpose	Analysis, reporting, decision making	Transaction processing
Data Volume	Large, historical	Smaller, current
Query Type	Complex, read-intensive	Simple, write-intensive
Schema	Denormalized (star/snowflake)	Highly normalized

---

## Example Scenario

A retail company wants to analyze sales trends over several years.

- Data from stores, online sales, and suppliers are collected.

- ETL process consolidates and cleans the data.
  - Data warehouse stores sales facts and dimensions.
  - Analysts query data warehouse for sales by region, product, time period.
- 

## Benefits of Data Warehousing

- Improved data quality and consistency.
  - Faster query performance for complex analyses.
  - Historical intelligence enables better forecasting.
  - Supports strategic decision making.
- 

## Example: Simple Star Schema Query

```
sql
CopyEdit
SELECT p.product_name, SUM(f.amount) AS total_sales
FROM Sales_Fact f
JOIN Product_Dim p ON f.product_id = p.product_id
WHERE f.date_id BETWEEN '2023-01-01' AND '2023-12-31'
GROUP BY p.product_name;
```

- This query summarizes total sales per product for a year.
- 

## Summary Table

Aspect	Details
Definition	Central repository for integrated historical data
Key Features	Subject-oriented, integrated, time-variant, non-volatile
Architecture	Data sources, ETL, warehouse, metadata, OLAP tools
Data Models	Star Schema, Snowflake Schema
Use Cases	Business Intelligence, Reporting, Analytics

---

## OLAP and OLTP

### What is OLTP?

**Online Transaction Processing (OLTP)** systems are designed to manage day-to-day transactional data and support routine business operations.

- Handles large number of short online transactions (INSERT, UPDATE, DELETE).
- Focuses on fast query processing and maintaining data integrity in multi-access environments.
- Uses highly normalized database design to reduce redundancy.

**Examples:** Banking systems, retail sales, airline booking systems.

---

## What is OLAP?

**Online Analytical Processing (OLAP)** systems are designed for complex queries, data analysis, and decision making.

- Focuses on read-intensive queries with complex aggregations and summaries.
- Works with historical and consolidated data.
- Uses denormalized schemas (like star or snowflake schema) for faster query response.

**Examples:** Business intelligence, sales forecasting, market research.

---

## Key Differences Between OLTP and OLAP

Feature	OLTP	OLAP
Purpose	Manage daily transactions	Support complex analytical queries
Data Volume	Small transactions, current data	Large volumes, historical data
Query Complexity	Simple queries (e.g., single record)	Complex queries with aggregations
Database Design	Highly normalized schema	Denormalized schema (star, snowflake)
Transaction Type	Insert, update, delete	Select (read-only)
Response Time	Fast for individual transactions	Longer, but optimized for complex queries
Examples	ATM transactions, order entry systems	Sales analysis, budgeting, trend analysis

---

## OLTP System Example

A banking application processes transactions like:

- Withdrawals
- Deposits
- Transfers

Each transaction updates the customer's account balance instantly.

```
sql
CopyEdit
UPDATE accounts
SET balance = balance - 500
WHERE account_id = 12345;
```

- This is a quick, atomic transaction.

---

## OLAP System Example

A retail company wants to analyze sales over several years:

```
sql
CopyEdit
SELECT product_category, SUM(sales_amount) AS total_sales, YEAR(sales_date)
AS year
FROM sales_fact
GROUP BY product_category, YEAR(sales_date);
```

- This aggregates large amounts of historical data.
- Enables managers to identify sales trends by category and year.

---

## OLTP vs OLAP: Data Modeling

- **OLTP:** ER model with normalized tables to reduce redundancy.
- **OLAP:** Dimensional modeling (star or snowflake schema) with fact and dimension tables.

---

## Summary Table

Aspect	OLTP	OLAP
Data	Current, detailed	Historical, summarized
Users	Clerks, front-line workers	Executives, analysts
Updates	Frequent, many	Rare
Data Size	Small	Very large
Typical Operations	Insert, Update, Delete	Complex Select (aggregate queries)
Schema Design	Normalized	Denormalized

---

## Big Data and Cloud Databases

### 1. Big Data in DBMS

---

## What is Big Data?

Big Data refers to extremely large and complex datasets that traditional database management tools cannot handle efficiently. These datasets come from various sources such as social media, sensors, transactions, and logs.

---

## Characteristics of Big Data (The 5 Vs)

V	Description
<b>Volume</b>	Massive amount of data (terabytes to petabytes)
<b>Velocity</b>	High speed of data generation and processing
<b>Variety</b>	Diverse data types: structured, semi-structured, unstructured
<b>Veracity</b>	Uncertainty or reliability of data
<b>Value</b>	Useful insights extracted from the data

---

## Challenges of Big Data

- Storing huge volumes efficiently.
  - Processing and analyzing in real-time or near-real-time.
  - Integrating heterogeneous data sources.
  - Ensuring data quality and security.
- 

## Big Data Technologies

- **Hadoop Ecosystem:** Distributed storage (HDFS) + processing (MapReduce, YARN).
  - **NoSQL Databases:** Handle unstructured or semi-structured data.
  - **Spark:** Fast in-memory data processing framework.
- 

## Example Scenario

- A social media platform generates terabytes of user interaction data daily.
  - Using Big Data tools, they analyze user behavior to target advertisements.
- 

## 2. Cloud Databases in DBMS

---



*What is a Cloud Database?*

A **Cloud Database** is a database service built, deployed, and accessed via cloud computing platforms. It provides database capabilities without the need to manage physical hardware or infrastructure.

---

*Types of Cloud Databases*

Type	Description	Examples
Relational Cloud DB	Traditional SQL databases hosted on cloud	Amazon RDS, Google Cloud SQL
NoSQL Cloud DB	NoSQL databases designed for scalability	Amazon DynamoDB, Azure Cosmos DB
Data Warehouse as a Service	Cloud data warehouses optimized for analytics	Snowflake, Google BigQuery

---

*Advantages of Cloud Databases*

- **Scalability:** Scale resources up/down easily.
  - **Cost Efficiency:** Pay-as-you-go pricing models.
  - **High Availability:** Built-in redundancy and failover.
  - **Managed Services:** Automated backups, patching, and maintenance.
  - **Global Accessibility:** Accessible from anywhere.
- 

*Challenges of Cloud Databases*

- Data security and compliance.
  - Network latency.
  - Vendor lock-in concerns.
  - Data migration complexities.
- 

**Big Data and Cloud Database Integration**

- Big Data workloads increasingly run on cloud platforms.
  - Cloud storage services (like AWS S3) integrate with big data processing frameworks.
  - Cloud databases offer scalable backends for big data applications.
- 

**Example: Using Amazon Web Services (AWS)**

- **Amazon S3:** Stores massive amounts of unstructured data.
  - **Amazon EMR:** Runs Hadoop and Spark clusters for big data processing.
  - **Amazon Redshift:** Cloud data warehouse for fast analytics.
  - **Amazon DynamoDB:** NoSQL cloud database for low-latency applications.
- 

## Summary Table

Aspect	Big Data	Cloud Databases
Data Type	Massive, varied (structured/unstructured)	Managed database service on cloud
Storage	Distributed file systems (HDFS, S3)	Cloud storage with managed DB instances
Processing	Batch & real-time (MapReduce, Spark)	Query & transaction processing in cloud DB
Scalability	Horizontal scaling over commodity hardware	Elastic scaling via cloud infrastructure
Examples	Hadoop, Spark, NoSQL DB	Amazon RDS, DynamoDB, Google BigQuery
Use Cases	Data analytics, machine learning, IoT	Web apps, mobile backends, BI reporting

---

## Database Security

### What is Database Security?

Database Security refers to the range of measures, controls, and practices designed to protect the database from unauthorized access, misuse, damage, or loss. It ensures the confidentiality, integrity, and availability of data stored in the database.

---

### Objectives of Database Security

Objective	Description
<b>Confidentiality</b>	Prevent unauthorized users from accessing sensitive data.
<b>Integrity</b>	Protect data from unauthorized modifications or corruption.
<b>Availability</b>	Ensure database services are available to authorized users when needed.
<b>Authentication</b>	Verify the identity of users accessing the database.
<b>Authorization</b>	Control what authenticated users can do within the database.

Objective	Description
Auditing	Track and record database activities for monitoring and compliance.

---

## Threats to Database Security

Threat	Description
Unauthorized Access	Users gaining access without permission.
SQL Injection	Malicious SQL code inserted via user input.
Privilege Abuse	Authorized users exceeding their privileges.
Data Leakage	Sensitive data being exposed or stolen.
Data Tampering	Unauthorized modification of data.
Denial of Service (DoS)	Attacks that disrupt database availability.

---

## Database Security Measures

### 1. Authentication

- Methods: Passwords, biometrics, multi-factor authentication.
- Example: User logs in with username and password before accessing the DB.

### 2. Authorization

- Access Control mechanisms define what users can access or modify.
- Types:
  - **Discretionary Access Control (DAC):** Users control access to their owned objects.
  - **Mandatory Access Control (MAC):** Access based on fixed policies (used in high-security environments).
  - **Role-Based Access Control (RBAC):** Permissions assigned to roles, and users assigned roles.

### Example:

```
sql
CopyEdit
GRANT SELECT, INSERT ON employees TO hr_role;
GRANT hr_role TO user_john;
```

---

### 3. Encryption

- Encrypt data at rest and in transit.
- Protects data from being readable if intercepted or stolen.

#### Example:

- Transparent Data Encryption (TDE) in Oracle encrypts data files.
  - SSL/TLS encrypts data communication between client and database.
- 

### 4. Auditing and Monitoring

- Logs user activities such as login attempts, queries run, changes made.
- Helps detect suspicious activities.

#### Example:

- Enable auditing in SQL Server to track who accessed or modified data.
- 

### 5. Views and Stored Procedures

- Use views to restrict access to sensitive columns or rows.
  - Use stored procedures to control data operations, avoiding direct table access.
- 

### 6. Backup and Recovery

- Regular backups prevent data loss.
  - Secure backups to avoid unauthorized access.
- 

#### Example: Preventing SQL Injection

- Use parameterized queries or prepared statements.

```
sql
CopyEdit
-- Vulnerable:
EXEC('SELECT * FROM users WHERE username = ''' + @user_input + ''');

-- Safe:
PREPARE stmt FROM 'SELECT * FROM users WHERE username = ?';
EXECUTE stmt USING @user_input;
```

---

#### Example Scenario

A company uses RBAC:

- **Roles:** admin, sales, hr.
  - **Permissions:** admin can read/write all data; sales can only read sales data; hr can access employee data.
  - Users assigned roles accordingly to restrict their access.
- 

## Summary Table

Security Aspect	Description	Example
Authentication	Verify user identity	Password, MFA
Authorization	Define user permissions	GRANT/REVOKE commands
Encryption	Data confidentiality	TDE, SSL/TLS
Auditing	Track user actions	Database audit logs
Views/Stored Procedures	Control data access	Creating restricted views
Backup and Recovery	Protect against data loss	Regular, secured backups

## – Role-based access control –

### What is RBAC?

**Role-Based Access Control (RBAC)** is a security mechanism in database management systems where permissions are assigned to roles, and users are assigned to these roles. This simplifies management of user privileges by grouping privileges under roles instead of assigning privileges to individual users.

---

### Why RBAC?

- Simplifies **privilege management** in large organizations.
  - Ensures **principle of least privilege** — users get only the permissions needed for their role.
  - Makes it easier to **audit and modify** permissions.
  - Supports separation of duties and compliance requirements.
- 

### Key Components of RBAC

Component	Description
<b>Users</b>	Individuals who need access to the database.
<b>Roles</b>	Named collections of privileges.
<b>Permissions</b>	Rights to perform operations (SELECT, INSERT, DELETE, etc.) on database objects.
<b>Sessions</b>	Mapping between users and activated roles during a login session.

---

## How RBAC Works

- Database administrators create **roles** based on job functions.
  - Privileges are assigned to roles.
  - Users are assigned one or more roles.
  - When users connect, they acquire privileges of their assigned roles.
- 

## Advantages of RBAC

- Easier to assign/revoke permissions.
  - Improves security by limiting privileges.
  - Reduces errors from assigning excessive permissions.
  - Supports organizational policies and compliance.
- 

## RBAC Example in SQL

Suppose an organization has these roles:

- **admin\_role**: full access to all tables.
  - **hr\_role**: read/write access to employee-related tables.
  - **sales\_role**: read access to sales tables.
- 

### *Step 1: Create Roles*

```
sql
CopyEdit
CREATE ROLE admin_role;
CREATE ROLE hr_role;
CREATE ROLE sales_role;
```

---

### *Step 2: Grant Privileges to Roles*

```
sql
CopyEdit
-- Admin role has all privileges on employees and sales tables
```

```
GRANT ALL PRIVILEGES ON employees TO admin_role;
GRANT ALL PRIVILEGES ON sales TO admin_role;

-- HR role can select and update employees table
GRANT SELECT, UPDATE ON employees TO hr_role;

-- Sales role can only select from sales table
GRANT SELECT ON sales TO sales_role;
```

---

### Step 3: Assign Roles to Users

```
sql
CopyEdit
GRANT admin_role TO user_admin;
GRANT hr_role TO user_hr;
GRANT sales_role TO user_sales;
```

---

### Step 4: Using Roles

- When **user\_hr** logs in, they can query and update employees but cannot access sales.
  - When **user\_sales** logs in, they can only view sales data.
- 

## Example Scenario

A hospital database:

- **doctor\_role**: Can view patient records and add diagnosis.
- **nurse\_role**: Can view patient records only.
- **billing\_role**: Can update billing information.

RBAC enforces access according to these roles rather than managing each user individually.

---

## Summary Table

Aspect	Description
Roles	Group of privileges based on job function
Users	Assigned one or more roles
Privileges	Permissions to perform actions
Benefits	Simplifies privilege management, enhances security

---

## Threats and countermeasures

# What are Threats in DBMS?

Threats are potential risks that can compromise the confidentiality, integrity, or availability of data stored in a database. They can come from external attackers, insiders, or accidental failures.

## Common Threats to DBMS

Threat	Description	Example
Unauthorized Access	Access by users without permission.	A hacker gains access to sensitive records.
SQL Injection	Attacker inserts malicious SQL code via input.	' OR '1'='1 ' injected into login forms to bypass authentication.
Privilege Abuse	Legitimate users misuse their privileges.	A database admin deletes critical data intentionally or by mistake.
Data Tampering	Unauthorized modification of data.	Modifying financial records to commit fraud.
Data Theft / Leakage	Sensitive data is stolen or leaked.	Exporting confidential customer data without authorization.
Denial of Service (DoS)	Overloading the database to make it unavailable.	Flooding DB with queries to crash it.
Backup Failures	Failure to properly back up data causing loss.	Backup media corruption or loss.
Malware / Viruses	Malicious software infects DBMS or server.	Ransomware encrypting database files.
Software Bugs	Errors in DBMS software causing crashes or data loss.	Crash during transaction commit leading to data inconsistency.

## Countermeasures for DBMS Threats

Threat	Countermeasure	Description & Example
Unauthorized Access	Authentication & Authorization	Use strong passwords, multi-factor authentication (MFA), and role-based access control (RBAC). Example: Grant limited access with GRANT statements.



Threat	Countermeasure	Description & Example
SQL Injection	<b>Input Validation &amp; Parameterized Queries</b>	Use prepared statements and validate all inputs. Example: Use <code>PreparedStatement</code> in Java or bind variables in PL/SQL.
Privilege Abuse	<b>Principle of Least Privilege &amp; Auditing</b>	Grant minimum necessary permissions; log all user actions. Example: Assign roles with limited rights and review audit logs regularly.
Data Tampering	<b>Integrity Constraints &amp; Encryption</b>	Use constraints like primary keys, foreign keys; encrypt sensitive data. Example: Use checksums or hashes to detect unauthorized changes.
Data Theft / Leakage	<b>Data Encryption &amp; Access Control</b>	Encrypt data at rest and in transit; restrict access. Example: Enable Transparent Data Encryption (TDE).
Denial of Service (DoS)	<b>Resource Limits &amp; Monitoring</b>	Limit connections per user and monitor unusual traffic. Example: Configure connection throttling.
Backup Failures	<b>Regular Backups &amp; Verification</b>	Schedule backups and test restore procedures. Example: Use automated backup tools and verify backup integrity.
Malware / Viruses	<b>Antivirus &amp; Patch Management</b>	Install antivirus software and keep DBMS updated. Example: Regularly apply security patches from vendors.
Software Bugs	<b>Testing &amp; Updates</b>	Test DBMS updates in staging environments before production. Example: Follow vendor's update cycle and best practices.

---

## Example: Preventing SQL Injection

### Vulnerable query:

```
sql
CopyEdit
String query = "SELECT * FROM users WHERE username = '" + userInput + "';";
```

If `userInput` is: `' OR '1'='1`, the attacker gains access.

### Safe query using parameterized statements:

```
sql
CopyEdit
PreparedStatement stmt = conn.prepareStatement("SELECT * FROM users WHERE username = ?");
stmt.setString(1, userInput);
ResultSet rs = stmt.executeQuery();
```

---

## Example: Role-Based Access Control (Countermeasure to Privilege Abuse)

```
sql
CopyEdit
CREATE ROLE read_only;
GRANT SELECT ON employees TO read_only;
GRANT read_only TO user_jane;
```

User `user_jane` can only read employee data, preventing unauthorized changes.

---

### Summary Table

Threat	Countermeasure	Example
Unauthorized Access	Strong authentication & RBAC	Multi-factor authentication
SQL Injection	Parameterized queries	Prepared statements
Privilege Abuse	Least privilege & auditing	Role-based access control
Data Tampering	Constraints & encryption	Checksums, TDE
Data Theft	Encryption & access control	SSL/TLS, encrypted backups
DoS	Resource limits & monitoring	Connection throttling
Backup Failures	Regular backup & restore tests	Automated backup scheduling
Malware	Antivirus & patches	Regular patch updates
Software Bugs	Testing & updates	Staging environment testing

---