INDEX

Unit – I: Introduction to DBMS

Unit – II: ER Model & Normalization

Unit – III: SQL & PL/SQL

- *Row level*
- *Statement level*

UNIT-IV: Transaction, Recovery, Security

1.1 Transaction Management
1.2 Transaction States in DBMS
1.3 ACID Properties of Transaction
- *Atomicity*
- *Consistency*
- *Isolation*
- *Durability*

2.Concurrency Control
2.1 Concurrency Control Problems
- *Lost Update Problem*
- *Dirty Read (Uncommitted Dependency) Problem*
- *Non-Repeatable Read Problem*
- *Phantom Read Problem*
- *Inconsistent Analysis Problem*

2.2 Concurrency Control Protocols in DBMS
3.Schedules in DBMS
3.1. Serial Schedules
3.2. Non-Serial Schedules
3.2.1 Serializable Schedules
- *Conflict Serializability*
- *View Serializability*

3.2.2 Non-Serializable Schedule
3.2.2.1. Recoverable Schedules
3.2.2.2 Non-Recoverable Schedule
- *Recoverable Schedule*
- *Cascadeless Schedule*
- *Strict Schedule*

4.Locking Protocols
4.1 ***Locking-Based Protocols***
Shared
Exclusive (Read–Write) Locking
4.2 ***Two-Phase Locking (2PL)***
Phases
- *Growing Phase*
- *Shrinking Phase*

Unit – I
Introduction to DBMS

1. Basic Concepts

1.1 Data, Information, Metadata

Data: -

Data are raw facts and figures that are collected but not yet processed, so they do not convey any clear meaning on their own. Data can be in the form of numbers, text, symbols, images, or sounds.

Example: The numbers **85, 72, 90** or the word **Hyderabad** are data because they do not explain what they represent.

Information: -

Information is data that has been processed, organized, and interpreted to make it meaningful and useful. It helps in understanding situations and making decisions by clearly explaining what the data represents.

Example: "A student scored **85 marks** in Mathematics" or "**Hyderabad is the capital of Telangana**" are examples of information because they give clear meaning to the data.

Metadata: -

Metadata is data about data, which provides descriptive details about the actual data. It helps in identifying, storing, and managing data effectively but does not contain the main content itself.

Example: For a document, details such as **file name, author, date created, and file size** are metadata; for a photograph, details like **date taken, camera type, and resolution** are metadata.

1.2 Database, DBMS

Database: -

A **database** is an organized collection of related data. Data consists of raw facts and figures that can be processed to generate meaningful information. These facts may include numbers, text, or symbols that represent real-world situations. When data is properly processed and analyzed, it becomes useful information for decision-making.

In general, data represents recordable facts. By processing data, information can be derived that helps in understanding patterns and results. For example, if data related to marks obtained by students is available, it can be processed to find information such as class average, highest marks, or top-performing students.

Database Management System (DBMS):

A Database Management System is a software system that helps users store, manage, and retrieve data in an organized way. Instead of keeping data in separate files, a DBMS stores all related data in a single database, which reduces data duplication and improves efficiency. It

allows users to insert, update, delete, and search data easily using simple commands or applications.

A DBMS also acts as a bridge between the user and the database. Users do not need to know how data is physically stored on disk; the DBMS handles all internal operations such as data storage, indexing, and retrieval. It ensures data security by allowing only authorized users to access the data and maintains data consistency and accuracy even when multiple users access the database at the same time. Overall, a DBMS makes data handling simpler, safer, and more reliable.

1.3 Characteristics of DBMS

Earlier, data was managed using traditional file-based systems, where information was stored in separate files. These systems had many limitations such as redundancy, inconsistency, and difficulty in data access. A DBMS was developed to overcome these problems. A modern DBMS has the following important characteristics:

1. Real-World Entity Representation

A DBMS is designed using real-world entities and their attributes. It closely models real-life situations. For example, in a school database, *students* can be treated as entities, and *age, roll number, and class* can be their attributes.

2. Table-Based Structure

DBMS organizes data in the form of tables consisting of rows and columns. Relationships between entities are also represented through tables. By observing table names and structures, users can easily understand the database design.

3. Separation of Data and Applications

In a DBMS, data is independent of application programs. Data is considered passive, while the DBMS actively manages and organizes it. The DBMS also stores **metadata**, which is data about data, such as table definitions and constraints, to manage the database efficiently.

4. Reduced Data Redundancy

DBMS follows the process of normalization, which divides large tables into smaller ones to remove duplicate data. This scientific approach helps in minimizing redundancy and saves storage space.

5. Data Consistency

A DBMS ensures that data remains consistent across the database. If data is updated at one place, the change is reflected everywhere. The system prevents operations that may leave the database in an inconsistent state, which was a common issue in file-based systems.

6. Query Language Support

DBMS provides powerful query languages such as SQL to retrieve and manipulate data easily. Users can apply multiple conditions and filters to extract required data without writing complex programs.

7. ACID Properties

DBMS follows **ACID properties**—Atomicity, Consistency, Isolation, and Durability. These properties ensure that database transactions are processed reliably, especially in environments where multiple transactions occur simultaneously or when system failures happen.

8. Multiuser and Concurrent Access

A DBMS supports multiple users accessing the database at the same time. It manages concurrent operations efficiently and prevents conflicts, even when multiple users try to access the same data.

9. Multiple Views

DBMS allows different users to view the same database in different ways. Each user sees only the data relevant to their role or department, which simplifies usage and enhances security.

10. Security and Authorization

DBMS provides strong security mechanisms to protect data from unauthorized access. It allows administrators to define access rights and constraints. Different users can have different levels of permissions, ensuring that sensitive data is well protected.

2. File-Based System

2.1 File System

A **file system** is a method used by an operating system to store, organize, and manage files on a storage device such as a hard disk, SSD, or USB drive. It provides a structured way of arranging files so that they can be easily created, accessed, modified, and deleted when required. In a file system, files are grouped into **directories (folders)**, and these directories may further contain sub-directories and files, forming a hierarchical structure.

The file system is responsible for performing basic operations such as **file creation and deletion, naming of files, storing file locations, setting access permissions, and controlling how users read or write data**. By managing these activities, the file system ensures efficient use of storage space and quick retrieval of information whenever needed

2.2 Features of File-Based System

- **File-oriented data storage:** Data is stored in individual files, and each file is usually designed for a specific application or task.

- **Simple organization:** Files are organized using directories and sub-directories, making it easy to locate and manage files manually.
- **Program–data dependency:** Application programs are tightly linked with file structures. Any change in the file format requires modification of the related programs.
- **Limited data sharing:** Sharing data between different applications is difficult because each program uses its own files.
- **Minimal security control:** Security is provided only at the file or folder level, offering limited protection against unauthorized access.
- **No centralized control:** Each file is managed separately, and there is no central system to control data storage and access.
- **Low cost and easy implementation:** Does not require specialized software, making it inexpensive and suitable for small-scale applications.
- **Limited concurrency support:** Supports very little or no simultaneous access by multiple users, which can lead to data conflicts.
- **Manual data management:** Backup, recovery, and consistency checks are mostly handled manually by users or programmers.
- **Suitable for small systems:** Best suited for simple applications where data volume and user access are limited.

2.3 Drawbacks of File-Based System

Data redundancy and inconsistency: -

In file systems, the same data are often stored in multiple files and in different file formats. This leads to duplication of information, which wastes storage space and may result in inconsistency when one copy of the data is updated while others are not.

Difficulty in accessing data: -

Accessing data from file systems is not flexible. For every new requirement or query, a new program must be written to retrieve or modify the data. This makes data access time-consuming and inefficient.

Data isolation: -

In a file system, data is stored in separate files using different formats. Because of this isolation, it becomes difficult to combine data from multiple files or perform complex operations across them.

Integrity problems: -

Integrity constraints, such as ensuring that an account balance is always greater than zero, must be enforced through program code in file systems. Adding new constraints or modifying existing ones is difficult and error-prone, which can compromise data accuracy.

Maintenance problems: -

When the structure of a file changes, such as adding a new field, all existing application programs that access the file must be modified. This increases maintenance effort and the chances of errors.

Atomicity of updates: -

File systems do not support atomic transactions. If a failure occurs during an update, the data may be left in an inconsistent state with only partial changes applied. For example, during a fund transfer between two accounts, money may be deducted from one account but not credited to the other.

Concurrent access by multiple users: -

File systems do not handle simultaneous access by multiple users effectively. Although concurrent access is required for better performance, uncontrolled access can lead to data inconsistencies. For instance, if two users read the same account balance and update it at the same time, the final result may be incorrect.

Therefore, due to these limitations, file systems are not preferred for large, multi-user, and complex data applications, and **DBMS** is used instead

2.4 Comparison Between File system and DBMS

Basis	File System	DBMS
Structure	A method of storing and organizing data in the form of files on a storage device.	A software system used to create, store, manage, and retrieve data from databases.
Data Redundancy	Same data may be stored in multiple files, leading to redundancy.	Redundancy is minimized by proper database design and normalization.
Backup and Recovery	No built-in support for backup and recovery of data.	Provides built-in tools for backup and recovery in case of data loss.
Query Processing	Does not support efficient query processing; programs must be written for each task.	Supports efficient and flexible query processing using query languages like SQL.
Consistency	Data consistency is difficult to maintain due to duplication.	High data consistency is maintained through centralized control and normalization.
Complexity	Simple to design and use.	More complex due to advanced features and controls.
Security Constraints	Provides limited security at file or folder level.	Provides strong security mechanisms such as authentication and

		authorization.
Cost	Less expensive as it does not require specialized software.	Comparatively more expensive due to software and maintenance costs.
Data Independence	Does not support data independence.	Supports both **logical** and **physical data independence**.
User Access	Generally supports single-user access at a time.	Allows multiple users to access data simultaneously.
Meaning / Programming	Users are not required to write complex procedures for data handling.	Users need to define procedures and queries for managing data.
Data Sharing	Data is scattered across many files, making sharing difficult.	Centralized storage makes data sharing easy.
Data Abstraction	Exposes details of data storage and structure.	Hides internal details and provides different levels of abstraction.
Integrity Constraints	Difficult to implement and maintain integrity rules.	Easy to define and enforce integrity constraints.
Attributes for Access	Requires file name, file path, and format to access data.	Users access data through queries without knowing storage details.
Examples	COBOL, C++ file handling systems	Oracle, SQL Server, MySQL

3.Database Approach

The **database approach** refers to a method of managing data in which a **single, centralized database** is used and controlled by a **Database Management System (DBMS)**. In this approach, data is stored only once and is shared by multiple users and application programs through the DBMS, instead of being stored separately in different files.

In the database approach, the **DBMS acts as an interface** between the users/applications and the database. All operations such as data storage, retrieval, update, deletion, security, and recovery are handled by the DBMS. This approach overcomes the major drawbacks of the traditional file-based system like data redundancy, inconsistency, and difficulty in data access.

A key feature of the database approach is **data independence**, where application programs are not directly dependent on the physical structure of data. Any change in data structure or storage does not require changes in application programs, as the DBMS manages these changes internally.

The database approach also supports **data sharing and concurrency**, allowing multiple users to access the same data simultaneously without conflicts. It ensures **data integrity, security, consistency, and reliability** by enforcing rules, constraints, and authorization mechanisms.

Example:
In a bank, all customer details, account information, and transaction records are stored in a single centralized database. Different applications such as ATM systems, online banking, and branch operations access the same database through the DBMS, ensuring accurate and up-to-date information at all times.

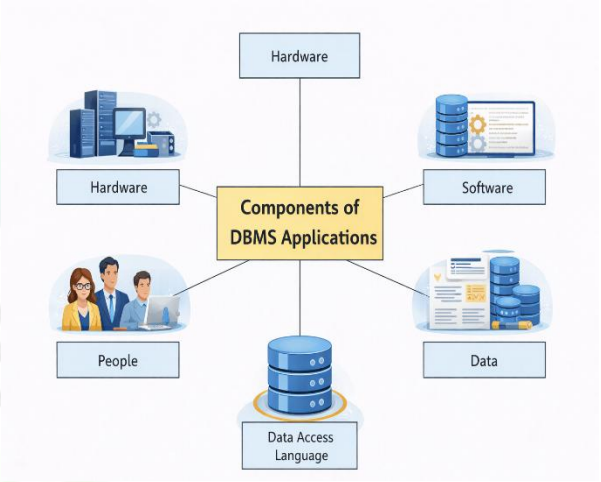
4.Database Environment

4.1 Components of DBMS Applications

Any **DBMS-based application** is built using several components that work together to store, process, and manage data efficiently. The **six key components of DBMS applications** are explained below:

1. Hardware

Hardware refers to the physical devices required to run the DBMS and store data. It provides the platform on which the database system operates and acts as an interface between real-world input and digital data processing.



- Includes servers, hard disks, RAM, keyboards, monitors, printers, and network devices
- Responsible for storing data and executing DBMS operations
- **Examples:** Computer hard disk, RAM, database server, network cables

2. Software

Software includes the actual DBMS and other supporting programs that control and manage the database system.

- DBMS software such as MySQL, Oracle, PostgreSQL
- Operating system, network software, and application programs
- Converts database access commands into machine-level operations

3. Data

Data is the most important component of a DBMS application. It represents raw facts that are stored and processed to produce meaningful information.

- **Operational data:** Actual user data such as names, ages, marks, salaries
- **Metadata:** Data about data, such as data type, size, creation time, and structure
- The primary purpose of DBMS is to store, manage, and retrieve data efficiently

4. Procedures

Procedures are predefined rules and instructions that explain how the DBMS should be used and managed.

- Include database setup, user login/logout, data entry rules
- Define backup and recovery processes
- Ensure proper access control and report generation
- Help maintain security, consistency, and smooth operation

5. Database Access Language

Database access languages are used to interact with the database and perform various operations.

- Allow users to create, retrieve, update, and delete data
- Common languages include **SQL, PL/SQL, MyAccess**
- **DDL (Data Definition Language):** CREATE, ALTER, DROP
- **DML (Data Manipulation Language):** INSERT, UPDATE, DELETE

6. People

People are the users who interact with the database system at different levels.

- **Database Administrator (DBA):** Manages security, backup, performance, and user access
- **Developers:** Design and develop database applications
- **End Users:** Use applications to access data (students, employees, customers)

A **Database Environment = Hardware + Software + Data + People**.

5. Advantages and Disadvantages of DBMS

Advantages of DBMS

1. Control of Data Redundancy

Data redundancy means storing the same data in multiple places. In a DBMS, data is stored in a centralized database and controlled by the Database Administrator (DBA). This centralized control avoids unnecessary duplication of data, reduces storage space requirements, and saves processing time that would otherwise be wasted in handling repeated data.

2. Improved Data Sharing

A DBMS allows data to be shared easily among different users and application programs. Since data is stored centrally, multiple applications can access the same data without creating separate files, which improves efficiency and collaboration.

3. Data Integrity

Data integrity refers to the accuracy and correctness of data stored in the database. A DBMS allows the DBA to define integrity constraints such as domain constraints, key constraints, and business rules. For example, a customer database can restrict entries to customers belonging only to specific cities, ensuring valid data entry.

4. Data Security

DBMS provides strong security mechanisms to protect sensitive data. The DBA can define user permissions and access rights so that only authorized users can access or modify certain data. This ensures that data is accessed only through proper and secure channels.

5. Data Consistency

By eliminating redundancy, DBMS greatly reduces the chances of data inconsistency. Since each data item is stored only once, there is no conflict between different copies of the same data. Updating data becomes easier and more reliable, as changes are made at a single location.

6. Efficient Data Access

DBMS manages data efficiently and provides fast access through query languages such as SQL. All data access is controlled by the DBMS, which optimizes data retrieval and processing, making operations quicker and more reliable.

7. Enforcement of Standards

Centralized data management allows the DBA to define and enforce standards such as naming conventions, data formats, and data quality rules. This ensures uniformity and consistency across the entire database system.

8. Data Independence

DBMS provides data independence by separating application programs from the physical structure of data. Changes in data storage or representation do not affect application programs, as the DBMS handles the necessary transformations internally.

9. Reduced Application Development and Maintenance Time

DBMS provides built-in functions such as data storage, retrieval, security, and concurrency control. Since these common tasks are handled by the DBMS, application development becomes faster and maintenance efforts are significantly reduced.

Disadvantages of DBMS

1. Complexity: DBMS software is complex because it supports many advanced features. Designers, developers, and administrators require specialized knowledge and training to use the system effectively.

2. High Memory and Resource Requirement: Due to its complexity and functionality, a DBMS requires a large amount of memory and processing power. This makes it unsuitable for very small applications with limited resources.

3. Centralized Failure Impact: DBMS works on a centralized system. If the DBMS fails due to hardware or software issues, all users who depend on that database are affected simultaneously.

4. Performance Overhead: Since DBMS is generalized software designed to support many types of applications, some specific applications may experience slower performance compared to simple file-based systems.

6. DBMS Architecture

DBMS architecture defines how data is stored, managed, and accessed by users. The most important model is the **Three-Tier Architecture**, which provides abstraction and independence between physical storage and user applications.

6.1. One-Tier Architecture

In **one-tier architecture**, all the components of a database system—**database, DBMS, and application program**—are present on a **single computer system**. The user directly interacts with the database without any intermediate layer. This architecture is **simple in design and easy to implement** because there is no separation between the user interface, business logic, and data storage.

One-tier architecture is mainly used for **learning purposes, personal applications, and small standalone systems**. Since all operations are performed on the same machine, **no network**

connection is required. However, this architecture is **not suitable for multi-user environments** and provides **limited security, scalability, and performance**.

Example: MS Access or SQLite used on a single personal computer.

6.2. Two-Tier Architecture

In **two-tier architecture**, the database system is divided into **two layers**: the **client layer** and the **database server layer**. The client application communicates **directly** with the database server to retrieve or modify data.

The **client layer** is responsible for providing the user interface and handling a portion of the business logic. It sends requests to the database server and displays the results to the user. The **database server layer** stores the database and executes SQL queries using the DBMS.

This architecture offers **better performance and improved data management** compared to traditional file-based systems. However, its **scalability is limited** because every client connects directly to the database server, which can overload the server when the number of users increases. Therefore, two-tier architecture is commonly used in **small to medium-sized client–server applications**.

Example: A desktop application connected to a **MySQL** or **Oracle** database server.
Layers in Two-Tier Architecture:

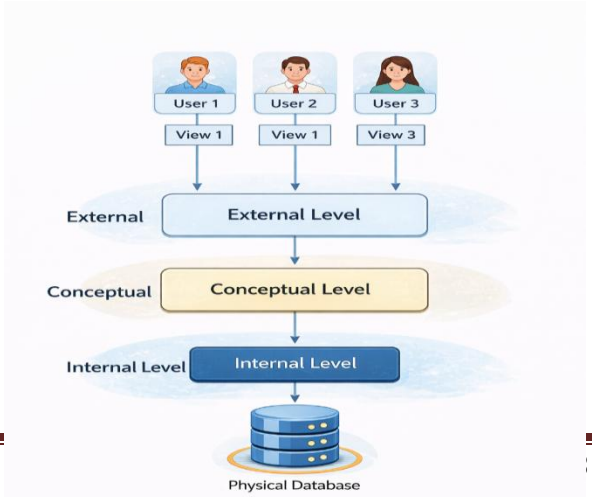
- **Client Layer:** User Interface and Business Logic
- **Database Server Layer:** DBMS and Database

6.3 Three-Tier Architecture

1. External Level (View Level)

The **external level** is the highest level of abstraction in DBMS architecture. It describes how data is viewed by different end users. At this level, each user can see only the data that is relevant to them, while the remaining data is hidden. Even though users have different views, the actual data stored in the database remains the same. This level improves security and makes the database easy to use.

- Highest level of abstraction
- User-specific views of data
- Hides unnecessary data
- Improves security and simplicity



Three-Level Database Architecture

Example:
A student can view only marks and attendance, a teacher can view class performance, and an administrator can view the complete database.

2. Conceptual Level (Logical Level)

The **conceptual level** is the middle level of abstraction and represents the overall logical structure of the database. It defines what data is stored and how different data items are related, without considering how the data is physically stored. This level includes definitions of entities, attributes, relationships, and integrity constraints.

- Middle level of abstraction
- Describes logical structure of the database
- Defines tables, attributes, and relationships
- Independent of physical storage

Example:
A Student table with attributes like Roll_No, Name, Address, and Course represents the conceptual view of student data.

3. Internal Level (Physical Level)

The **internal level** is the lowest level of abstraction and deals with how data is physically stored in the database. It explains the storage structures and access methods used to store and retrieve data efficiently. This level focuses mainly on performance.

- Lowest level of abstraction
- Describes physical storage of data
- Uses indexing, hashing, and access paths
- Focuses on performance and efficiency

Example:
Student records stored in disk blocks with indexing or B-tree structures for faster retrieval.

Advantages of Three-Tier Architecture

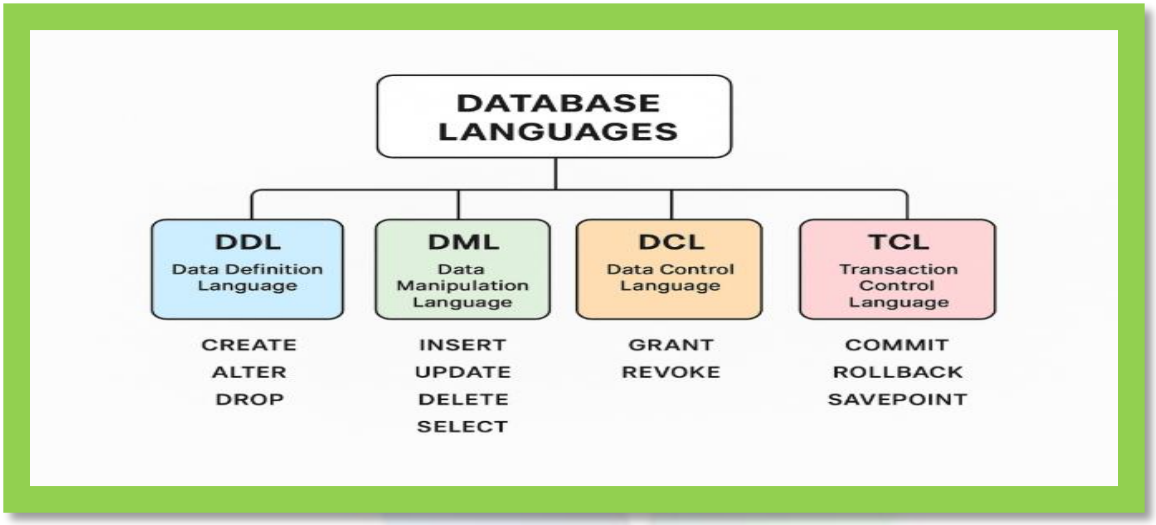
- **High security** – database is not directly accessible
- **Scalability** – layers can be upgraded independently
- **Easy maintenance** – changes in one layer do not affect others
- **Better performance** in multi-user environments
- **Reusability** of business logic

7. Database Languages

Database languages are special languages used to **define, manipulate, control, and access data** in a database. A DBMS provides different types of database languages so that users and programmers can interact with the database efficiently and securely.

Types of Database Languages:

1. ***Data Definition Language (DDL)***
2. ***Data Manipulation Language (DML)***
3. ***Data Control Language (DCL)***
4. ***Transaction Control Language (TCL)***



7.1 Data Definition Language (DDL)

Data Definition Language is used to **define and modify the structure of the database**. It helps in creating, altering, and deleting database objects such as tables, views, and indexes. DDL commands affect the database schema and are usually executed by database designers or administrators.

- Defines database structure
- Creates and modifies tables
- Affects database schema

Common Commands:

a) **CREATE** – Used to create new database objects such as tables, databases, or views.
b) **ALTER** – Used to modify the structure of an existing database object.
c) **DROP** – Used to permanently delete a database object from the database.
d) **TRUNCATE** – Used to remove all records from a table quickly without deleting the table structure
e) **RENAME** – Used to change the name of an existing database object.

a) CREATE Command

CREATE is a **Data Definition Language (DDL)** command used to **create new database objects** such as tables, databases, views, and indexes. It defines the **structure of the database object** by specifying column names, data types, and constraints. Once created, the object becomes part of the database schema and is used to store data.

Syntax:

```
CREATE TABLE table_name (
    column_name datatype,
    column_name datatype
);
```

Example:

```
CREATE TABLE Employee (
    EmpID INT PRIMARY KEY,
    EmpName VARCHAR(50),
    Department VARCHAR(30),
    Salary INT
);
```

After CREATE table structure looks like:

Column	Data Type	Constraint
EmpID	INT	PRIMARY KEY
EmpName	VARCHAR(50)	
Department	VARCHAR(30)	
Salary	INT	

b) ALTER Command

ALTER is a **Data Definition Language (DDL)** command used to **modify or change the existing structure of a database object** such as a table. It changes the **schema of the database** without affecting the existing data. Using the ALTER command, we can **add, modify, or drop columns** and also **add or remove constraints** of a table.

❖ Add Column (ALTER Command)

The **ADD COLUMN** operation is used to **add a new column to an existing table**. It is done using the **ALTER** command and changes the table structure without deleting existing data.

Syntax: ALTER TABLE table_name ADD column_name datatype;

Example *:ALTER TABLE Employee ADD* ***Email*** *VARCHAR(30);*

✓ **Adds new column "Email" in Employee table**

Column	Data Type	Constraint
EmpID	INT	PRIMARY KEY

EmpName	VARCHAR(50)	
Department	VARCHAR(30)	
Salary	INT	
Email	VARCHAR(30)	

❖ Modify Column (ALTER Command)

The **MODIFY COLUMN** operation is used to **change the data type, size, or properties of an existing column** in a table. It is performed using the **ALTER** command and does not delete the table or its data (provided the change is compatible).

Syntax (Modify column): *ALTER TABLE* `table_name MODIFY column_name new_datatype;`

Example: *ALTER TABLE Employee MODIFY Salary DECIMAL (10,2);*

Salary column data type changed

Column	Data Type	Constraint
EmpID	INT	PRIMARY KEY
EmpName	VARCHAR(50)	
Department	VARCHAR(30)	
Salary	**DECIMAL(10,2)**	

❖ Drop Column (ALTER Command)

The **DROP COLUMN** operation is used to **remove an existing column from a table** permanently. It changes the table structure and deletes all data stored in that column.

Syntax (Drop column): *ALTER TABLE* `table_name` *DROP COLUMN* `column_name;`

Example: *ALTER TABLE* `Employee` *DROP COLUMN* `Email;`

✓ Removes column "Email"

Column	Data Type	Constraint
EmpID	INT	PRIMARY KEY
EmpName	VARCHAR(50)	
Department	VARCHAR(30)	
Salary	DECIMAL(10,2)	

c) DROP TABLE Command

EmpID	EmpName	Department	Salary
101	Alice	HR	50000
102	Bob	IT	60000
103	Charlie	Finance	55000
104	David	IT	65000

The DROP TABLE command is a Data Definition Language (DDL) statement used to permanently delete a table from the database. It removes the table structure as well as all the data

Syntax:
DROP TABLE table_name;

Example
DROP TABLE

- ✓ **Deletes Employee table permanently** (both structure & data are gone).

d) TRUNCATE Command

The **TRUNCATE** command is a **Data Definition Language (DDL)** statement used to **remove all records from a table quickly**. It deletes all rows but **keeps the table structure intact**, so the table can be reused.

Syntax
TRUNCATE TABLE table_name;

Example
TRUNCATE TABLE Employee;

- ✓ Removes **all records** from the **Employee** table
- ✓ Table structure, columns, and constraints remain the same
- ✓ operation is fast compared to DELETE
- ✓ Data cannot be rolled back after truncation

e) RENAME Command

The **RENAME** command is a **Data Definition Language (DDL)** statement used to **change the name of an existing database object**, such as a table.

Syntax
RENAME TABLE old_table_name
TO new_table_name;

Example
RENAME TABLE Employee TO Staff;

- ✓ Changes the table name from **Employee** to **Staff**
- ✓ Table structure and data remain unchanged
- ✓ Only the name of the table is modified

7.2 Data Manipulation Language (DML)

Data Manipulation Language (DML) is used to **retrieve, insert, update, and delete data** stored in database tables. These commands work on the **data (records)** inside tables and are commonly used by users and application programs.

EmpID	EmpName	Department	Salary

Common DML Commands

a) **INSERT** – Used to add new records into a table
b) **UPDATE** – Used to modify existing records in a table
c) **DELETE** – Used to remove records from a table
d) **SELECT** – Used to retrieve data from one or more table

a) INSERT Command

The INSERT command is a Data Manipulation Language (DML) statement used to add new records (rows) into a table. It allows users to insert data into one or more columns of a table.

Syntax: *INSERT INTO table name (col1, col2, ...) VALUES (val1, val2, ...);*

Example: *INSERT INTO Employee VALUES (104, 'David', 'IT', 65000);*

✓ **Table after Insert:**

EmpID	EmpName	Department	Salary
101	Alice	HR	50000
102	Bob	IT	60000
103	Charlie	Finance	55000
104	David	IT	65000

b) UPDATE Command

The UPDATE command is a Data Manipulation Language (DML) statement used to modify existing records in a table. It allows changes to be made to one or more columns for selected rows.

Syntax:*UPDATE table_name SET column_name = new_value WHERE condition;*

Example:

UPDATE Employee SET Salary = 70000 WHERE EmpID = 102;

✓ **Table after UPDATE:**

EmpID	EmpName	Department	Salary
101	Alice	HR	50000
102	Bob	IT	70000
103	Charlie	Finance	55000
104	David	IT	65000

c) SELECT Command

The SELECT command is a Data Manipulation Language (DML) statement used to retrieve data from one or more tables in a database. It allows users to view specific columns or all records based on conditions.

Syntax: *SELECT column1, column2, ...FROM table_name WHERE condition;*

Example: *SELECT EmpName, Salary FROM Employee WHERE Department = 'IT';*

✓ **Output:**

EmpName	Salary
Bob	60000

d) DELETE Command

The DELETE command is a Data Manipulation Language (DML) statement used to remove records (rows) from a table. It deletes selected rows based on a condition.

Syntax: *DELETE FROM table_name WHERE condition;*

Example: *DELETE FROM Employee WHERE Department = 'Finance';*

✓ **Table after DELETE:**

EmpID	EmpName	Department	Salary
101	Alice	HR	50000
102	Bob	IT	70000
104	David	IT	65000

7.3 Data Control Language (DCL)

Data Control Language (DCL) is used to control access to data in a database. It helps the database administrator manage permissions and security by allowing or restricting user access to database objects.

DCL Commands

a) GRANT – Used to give specific privileges to users
b) REVOKE – Used to remove privileges from user

a) GRANT Command

The GRANT command is a Data Control Language (DCL) statement used to give permissions or privileges to users on database objects. It allows authorized users to perform specific operations such as SELECT, INSERT, UPDATE, or DELETE on tables.

Syntax: *GRANT privilege(s) ON object TO user;*

Example*:GRANT SELECT, INSERT ON Employee TO user1;;*

- ✓ **Effect:**
  Gives **SELECT** and **INSERT** permissions on the **Employee** table
  The user **user1** can view and insert data into the table

b) REVOKE Command

The REVOKE command is a Data Control Language (DCL) statement used to remove permissions or privileges from users on database objects. It is used when access rights given earlier need to be withdrawn.

Syntax: *REVOKE privilege(s) ON object FROM user;*

Example: *REVOKE INSERT ON Employee FROM user1;*

- ✓ **Effect:**
  The **INSERT** privilege of **user1** on **Employee** table is **removed**, but the **SELECT** privilege remains.

7.4 Transaction Control Language (TCL)

Transaction Control Language (TCL) is used to manage database transactions. It ensures that database operations are performed safely and maintains data consistency and integrity, especially in multi-user environments.

TCL Commands

a) **COMMIT –** Saves all changes made during the transaction permanently
b) **ROLLBACK** – Cancels changes and restores the database to the previous state
c) **SAVEPOINT** – Creates a point within a transaction to which rollback can occur

a) COMMIT

The **COMMIT** command is a **Transaction Control Language (TCL)** statement used to **permanently save all changes** made during a transaction in the database. After executing COMMIT, the changes become permanent and cannot be undone.

Syntax: ***COMMIT;***

Example: ***INSERT INTO Employee VALUES (105, 'Ravi', 'HR', 45000);***

COMMIT;

✓ **Effect:**
The new row is **permanently saved** in the Employee table.
Changes **cannot be rolled back** after commit.

b) ROLLBACK Command

The **ROLLBACK** command is a **Transaction Control Language (TCL)** statement used to **undo changes** made during the current transaction. It restores the database to its previous state before the transaction began.

Syntax:

ROLLBACK;

Example:

```
DELETE FROM Employee
WHERE EmpID = 105;
```

✓ **Effect:**
The delete operation is **undone**.
Row with **EmpID = 105** is restored (since changes were not committed).

c) SAVEPOINT Command

The **SAVEPOINT** command is a **Transaction Control Language (TCL)** statement used to **set a point within a transaction**. It allows partial rollback of a transaction to a specific savepoint without cancelling the entire transaction.

Syntax:

SAVEPOINT savepoint_name;

Example:

```
SAVEPOINT sp1;
DELETE FROM Employee
WHERE Department = 'HR';

ROLLBACK TO sp1;
```

✓ **Effect:**
Creates a **marker** (savepoint) in a transaction.
Allows rollback to that specific point instead of rolling back the entire transaction.

d) SET TRANSACTION Command

The SET TRANSACTION command is a Transaction Control Language (TCL) statement used to define the properties of a transaction. It allows the user to specify characteristics such as isolation level, access mode, or read/write behavior before a transaction begins.

Syntax

SET TRANSACTION [READ WRITE | READ ONLY];

Example:

SET TRANSACTION READ ONLY;
*SELECT * FROM Employee;*

Effect:

✔ Defines a **transaction mode**.

✔ In this case, the transaction is **read-only** (no updates allowed).

Comparison Table

Language Type	Command	Description	Example
DDL (Data Definition Language)	CREATE	Creates a new table or database	CREATE TABLE students (id INT, name VARCHAR(50));
	ALTER	Modifies existing table structure	ALTER TABLE students ADD age INT;
	DROP	Deletes a table or database	DROP TABLE students;
	TRUNCATE	Removes all records but keeps table structure	TRUNCATE TABLE students;
2. DML (Data Manipulation Language)	INSERT	Inserts data into a table	INSERT INTO students VALUES (1, 'John', 20);
	UPDATE	Modifies existing records	UPDATE students SET name = 'Jane' WHERE id = 1;
	DELETE	Removes records from a table	DELETE FROM students WHERE id = 1;
3. DCL (Data Control Language)	GRANT	Gives user access privileges	GRANT SELECT ON students TO user1;
	REVOKE	Withdraws user access privileges	REVOKE SELECT ON students FROM user1;
4. TCL (Transaction Control Language)	COMMIT	Saves all transactions permanently	COMMIT;
	ROLLBACK	Undoes changes before the last COMMIT	ROLLBACK;
	SAVEPOINT	Sets a point for partial rollback	SAVEPOINT save1;
5. DQL (Data Query	SELECT	Retrieves data from a	SELECT * FROM students;

Language)		table	

8. Data Models

A **data model** explains how a database is organized and shows what the database will look like after it is created. It describes the **data items**, the **relationships between data**, and the **rules** used to keep data correct and consistent. In a **DBMS**, data models help us understand how data is **stored, accessed, connected, and updated**. They use simple diagrams, symbols, and text so that users, designers, and developers can easily understand the database. Among all data models, the **Relational Model** is the most widely used.

Types of Data Models in DBMS

The commonly used data models are:

1. ***Hierarchical Model***
2. ***Network Model***
3. ***Entity–Relationship (ER) Model***
4. ***Relational Model***
5. ***Object-Oriented Data Model***
6. ***Document & Other Models (Advanced / NoSQL)***

8.1 Hierarchical Data Model

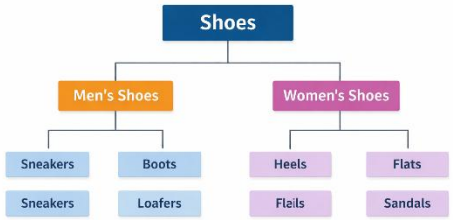
The **Hierarchical Model** is the earliest DBMS model. In this model, data is organized in a **tree-like structure** with a **parent–child relationship**. The hierarchy starts from a **root node**, and each parent can have many children, but each child has **only one parent**. This model represents **one-to-many relationships**.

Example:

A shopping website where Shoes is the parent node and Men's Shoes and Women's Shoes are child nodes.

Features

- One-to-many relationship
- Parent–child structure
- Only one path from parent to child
- Uses pointers to connect records



Advantages of Hierarchical Model

- It is **simple and easy to understand** because data is organized in a tree-like structure.
- Data traversal is **fast and efficient** due to fixed parent–child relationships.

- Any change in the **parent node is automatically reflected in the child nodes**, which helps in maintaining **data integrity**.

Disadvantages of Hierarchical Model

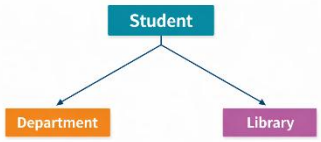
- **Complex relationships are not supported** in this model.
- A child node **cannot have more than one parent**, so many-to-many relationships cannot be represented.
- If a **parent node is deleted**, all its child nodes are **automatically deleted**, which may cause data loss.

8.2 Network Data Model

The **Network Model** is an extension of the hierarchical model. It allows a record to have **multiple parents**, forming a **graph structure**. This model supports **many-to-many relationships** and provides multiple paths to access data.

Example:

A Student record linked to both Department and Library.



Features

- Supports one-to-one and many-to-many relationships
- Multiple paths to reach the same record
- Uses linked lists for navigation

Advantages of Network Data Model

- Supports **many-to-many relationships**, making it more flexible than the hierarchical model.
- **Multiple access paths** are available, so data can be retrieved faster.
- Better representation of **complex real-world relationships**.
- Maintains **data integrity** through parent–child relationships.

Disadvantages of Network Data Model

- **Complex structure**, difficult to design and understand.
- Requires **detailed knowledge** of the database structure to access data.
- Database maintenance becomes difficult as relationships increase.
- Changes in structure may require changes in application programs.

8.3 Entity–Relationship (ER) Model

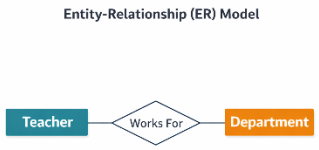
The **Entity–Relationship (ER) Model** is a **high-level conceptual model** used mainly during database design. It represents real-world problems in a **diagrammatic form** called an **ER diagram**, which is easy to understand for users and developers.

Components of ER Model

- **Entity**
  An entity is a real-world object that can be uniquely identified**.**
  Example: Student, Teacher, Department.
- **Attribute**
  Attributes describe the properties of an entity.
  Example: Student_ID, Name, Age, Salary.

- **Relationship**
  A relationship shows how two or more entities are connected.
  Example: Teacher works for Department.

Example:
Teacher works for Department.



Features

- Graphical representation
- Easy to understand
- Widely used in database design

Advantages

- Simple and clear
- Good communication tool
- Easily converted into relational tables

8.4 Relational Data Model

The **Relational Data Model** is the **most widely used data model** in database management systems. In this model, data is organized in the form of **tables**, which are also called **relations**. Each table consists of **rows** and **columns**, where rows represent individual records and columns represent attributes of the data. This model is simple, flexible, and easy to understand, making it very popular for real-world applications.

Key Concepts of Relational Data Model

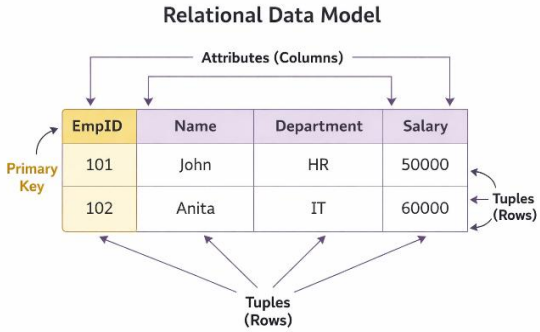
- **Relation (Table):** A collection of related data stored in rows and columns
- **Tuple (Row):** A single record in a table
- **Attribute (Column):** A property or field of the table
- **Primary Key:** A unique identifier for each record
- **Foreign Key:** A key used to link two related tables

Example

EmpID	Name	Department	Salary
101	John	HR	50000
102	Anita	IT	60000

Features of Relational Data Model

- Data stored in **two-dimensional tables**
- Uses **keys** to maintain relationships
- Supports **SQL** for data operations
- Ensures **data integrity and consistency**
- Provides **data independence**



Advantages of Relational Data Model

- Simple and easy to use
- Flexible and scalable
- Easy data retrieval using SQL

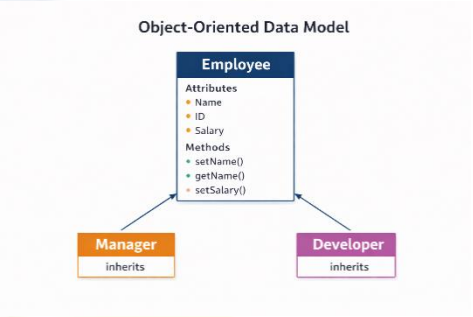
- Reduced data redundancy
- Strong theoretical foundation

Disadvantages of Relational Data Model

- Requires powerful hardware for large databases
- Performance may reduce with very complex queries
- Not ideal for unstructured data

8.5 Object-Oriented Data Model

The **Object-Oriented Data Model** organizes data in the form of **objects**, similar to object-oriented programming concepts. In this model, data and the operations (methods) that work on the data are stored together as a single unit called an **object**. It is mainly used to manage **complex data** such as multimedia, graphics, and engineering data.



Features of Object-Oriented Data Model

- Data is stored as **objects**
- Supports **classes and objects**
- Combines **data and methods** together
- Supports **inheritance**, **encapsulation**, and **polymorphism**
- Each object has a unique **object identity**

Example: Object-Oriented Data Model represents data as objects that contain both attributes and methods. It supports inheritance, where classes like Manager and Developer inherit properties from Employee.

Advantages of Object-Oriented Data Model

- Handles **complex data types** efficiently
- Closely matches real-world objects
- Supports reusability through inheritance
- Better integration with object-oriented programming languages like Java and C++

Disadvantages of Object-Oriented Data Model

- Complex to design and implement
- Requires more memory and processing power
- Not as widely used as the relational model
- Lack of standard query language compared to SQL

8.6 Document & Other Models (Advanced / NoSQL)

Document and other advanced data models, commonly known as **NoSQL data models**, are used to handle **large volumes of data**, **unstructured or semi-structured data**, and **highly scalable applications**. Unlike traditional relational databases, these models do not rely on fixed tables and schemas, making them flexible and suitable for modern web and big data applications.

Types of Document & Other NoSQL Models

- **Document Model** – Stores data in document formats like JSON or XML
- **Key–Value Model** – Stores data as key–value pairs

- **Column-Family Model** – Stores data in column-based format
- **Graph Model** – Stores data as nodes and edges

Features

- Schema-less or flexible schema
- High scalability and performance
- Designed for distributed systems
- Suitable for big data and real-time applications

Advantages

- Handles unstructured and semi-structured data easily
- Scales horizontally across multiple servers
- High performance for large datasets
- Flexible data storage

Disadvantages

- Limited support for complex queries
- Data consistency may be weaker than relational databases
- Lack of standard query language
- Not suitable for all applications

9.Database Users

Database users are the individuals who use a database system in different ways depending on their role. They help in creating, maintaining, and using the database for storing and accessing information.

Database Users
├── End Users
├── Application Programmers
├── Database Designers
├── Database Administrators (DBA)
├── System Analysts
└── Sophisticated User

1. End Users (Naïve Users / Casual Users)

End users are the people who directly use the database through application programs. They do not have knowledge of database structure or SQL commands. Naïve users use predefined applications regularly, while casual users access the database occasionally. **Example:** Students checking exam results, employees viewing payslips.

2. Application Programmers (Developers)

Application programmers write programs that interact with the database. They use programming languages such as Java, C#, or Python along with database interfaces like JDBC or ODBC to

access and manipulate data. Their programs make database usage easy for end users. **Example: A developer creating an online shopping application.**

3. Database Designers

Database designers are responsible for deciding how data should be stored in the database. They design the structure of the database, define relationships between entities, and apply constraints. They prepare ER diagrams, schemas, and perform normalization to reduce data redundancy.

4. Database Administrators (DBA)

The Database Administrator has complete control over the database system. The DBA installs and configures the database, manages user permissions, ensures data security, performs backup and recovery, and monitors performance.
Example: Allowing only authorized staff to access the payroll database.

5. System Analysts

System analysts study the requirements of an organization and plan suitable database solutions. They act as a link between end users and database designers, ensuring that the database meets organizational needs.

6. Sophisticated Users (Specialized Users)

Sophisticated users have deep knowledge of databases and directly write complex SQL queries. They use the database for advanced tasks such as data analysis, reporting, and research. **Example:** Data scientists performing analytical queries.

10. Database Administrator (DBA)

A **Database Administrator (DBA)** is the person who is responsible for the overall management of the database system. The DBA ensures that the database is always available, secure, reliable, and works efficiently for all users.

Key Roles and Responsibilities of DBA

1. Database Installation and Configuration: The DBA installs the DBMS software such as Oracle, MySQL, or SQL Server. They also configure the hardware and software environment so that the database functions properly.

2. Database Design and Implementation: The DBA helps in designing the logical and physical structure of the database. They create database objects such as schemas, tables, views, and indexes according to the system requirements.

3. Security Management: The DBA creates user accounts and assigns roles and permissions. They ensure that only authorized users can access the database and protect data from unauthorized use.

4. Backup and Recovery: The DBA schedules regular backups of the database to prevent data loss. In case of system failure, they use recovery techniques to restore the database to a correct state.

5. Performance Monitoring and Tuning: The DBA continuously monitors database performance. They improve efficiency by using indexing, query optimization, and other tuning techniques.

6. Concurrency Control and Transaction Management: The DBA ensures that multiple users can access the database at the same time without conflicts. They maintain ACID properties to ensure correct and reliable transactions.

7. Storage Management: The DBA manages disk space and memory used by the database. They allocate storage for database objects and optimize space usage.

8. Maintaining Data Integrity: The DBA defines and enforces constraints such as Primary Key, Foreign Key, and NOT NULL. This ensures accuracy, consistency, and correctness of data in the database.

11. Types of Databases

Databases are categorized into different types based on **data structure, scalability, and usage requirements**. Modern applications use different database types to efficiently store, process, and manage data according to their needs.

Types of Databases

1. ***Hierarchical Database***
2. ***Network Database***
3. ***Relational Database (RDBMS)***
4. ***NoSQL Database***
5. ***Distributed Database***
6. ***Centralized Database***
7. ***Cloud Database***
8. ***Object-Oriented Database***

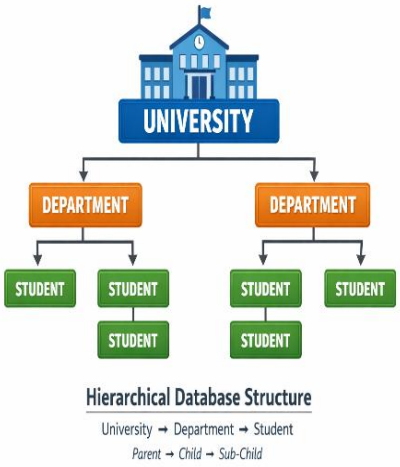
11.1 Hierarchical Database: -

A **Hierarchical Database** organizes data in a **tree-like structure** with a single root. Each record is connected through **parent–child relationships**, where a parent can have many children, but a

child can have only one parent. Data is accessed in a **top-down manner**, which makes retrieval fast but reduces flexibility when changes are required.

Characteristics of Hierarchical Database

- Data is stored in a **tree structure**.
- Supports **one-to-many relationships** only.
- Each child record has **only one parent**.
- Provides **fast data access**.
- **Difficult to modify** the structure once created.



Example

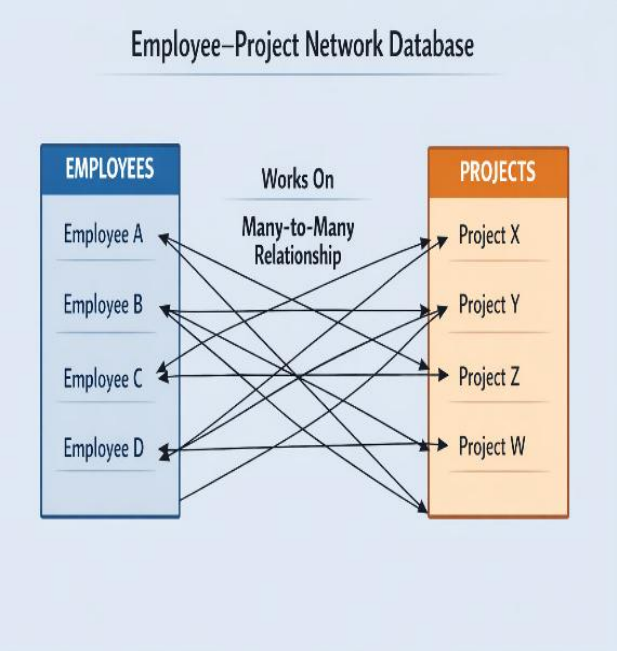
A **University database** is a good example of a hierarchical database.Here, **University** is the parent, **Department** is the child, and **Student** is the sub-child. Each student belongs to only one department, showing a hierarchical relationship.

11.2 Network Database: -

A **Network Database** organizes data in the form of a **graph structure**, where records are connected through links. In this model, a record can have **multiple parent and child records**, which supports **many-to-many relationships**. Compared to the hierarchical model, the network database is more flexible but also more complex to design and manage.

Characteristics of Network Database

- Data is stored using a **graph or network structure**.
- Supports **many-to-many relationships**.
- A record can have **multiple parents and multiple children**.
- Provides **better flexibility** than hierarchical databases.
- **Complex structure**, difficult to design and maintain.

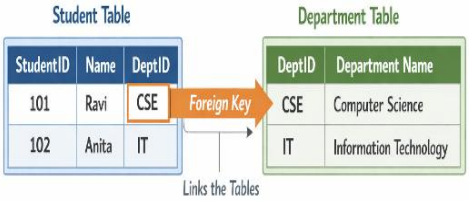


Example

An **Employee–Project database** is a common example of a network database. Here, one employee can work on many projects, and one project can have many employees, which represents a **many-to-many relationship**.

11.3 Relational Database: -

A **Relational Database** stores data in the form of **tables (relations)** made up of rows and columns. Each row represents a record (tuple) and each column represents an attribute. Relationships between tables are created using **primary keys and foreign keys**. This model is easy to understand, highly flexible, and widely used in real-world applications.



Characteristics of Relational Database

- Data is stored in **tables** with rows and columns.
- Uses **primary keys** to uniquely identify records.
- Uses **foreign keys** to establish relationships between tables.
- Supports **SQL** for querying and data manipulation.
- Ensures **data integrity and consistency**.

Example

A **Student–Department database** is a common example of a relational database.

StudentID	Name	DeptID
101	Ravi	CSE
102	Anita	IT

Here, **DeptID** acts as a foreign key linking the student table to the Department table.

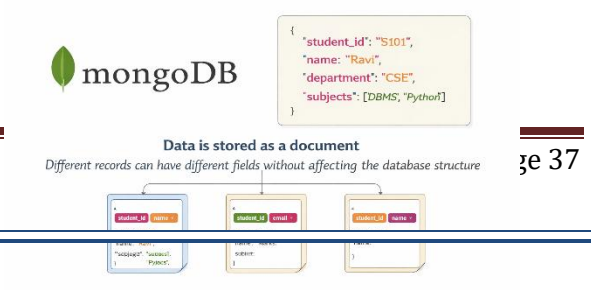
11.4 NoSQL Database: -

A **NoSQL Database** stores data in **non-relational formats** instead of tables. It is designed to handle **large volumes of data**, high-speed operations, and flexible data structures. NoSQL databases do not require a fixed schema, which makes them suitable for big data, web applications, and real-time systems.

Characteristics of NoSQL Database

- Data is stored in **non-tabular formats** (document, key–value, column, graph).
- **Schema-less** or flexible structure.
- Highly **scalable and distributed**.
- Handles **large and unstructured data** efficiently.
- Provides **high performance** for read and write operations.

Example



MongoDB is a popular NoSQL database.

```
{
  "student_id": "S101",
  "name": "Ravi",
  "department": "CSE",
  "subjects": ["DBMS", "Python"]
}
```

Here, data is stored as a **document**, and different records can have different fields without affecting the database structure.

11.5 Distributed Database: -

A **Distributed Database** is a database in which data is stored at **multiple physical locations** but appears as a **single database** to users. The data may be distributed across different computers or network sites. This type of database improves availability, reliability, and performance by sharing the workload among different locations.

Characteristics of Distributed Database

- Data is stored at **multiple sites or locations**.
- Appears as a **single database** to users.
- Provides **high availability and reliability**.
- Improves **performance** through parallel processing
- Complex to manage **data consistency and security**

Example

A **banking system** is a common example of a distributed database.

Each bank branch stores its own customer data locally, but all branches are connected and share information, allowing customers to access their accounts from any branch.

11.6 Centralized Database: -

A **Centralized Database** is a database in which all data is stored and managed at a **single central location**. All users access the database through this central system. This type of database is easy to manage and maintain but depends heavily on one system for its operation.

Characteristics of Centralized Database

- All data is stored at **one central location**.
- Easy to **manage, control, and maintain**.

- Provides **better data consistency**.
- High dependency on the **central server**.
- Failure of the central system can affect all users.

Example

A **college administration system** is an example of a centralized database.

All student, staff, and examination records are stored on a single central server, and users access the data from different departments.

11.7 Cloud Database: -

A **Cloud Database** is a database that is hosted and managed on a **cloud computing platform** and accessed through the internet. It removes the need for local hardware and allows users to scale storage and resources as required. Cloud databases are widely used due to their flexibility and cost efficiency.

Characteristics of Cloud Database

- Hosted on **cloud platforms** and accessed via the internet.
- Provides **scalability** (resources can be increased or decreased easily).
- Reduces **hardware and maintenance costs**.
- Offers **high availability and backup support**.
- Supports **remote access** from anywhere.

Example

Amazon RDS is an example of a cloud database.

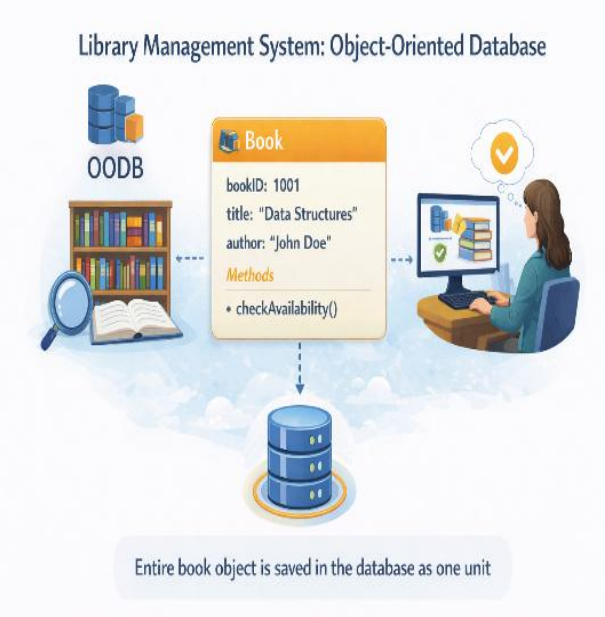
It allows users to create, manage, and scale relational databases on the cloud without handling physical servers.

11.8 Object-Oriented Database (OODB): -

An **Object-Oriented Database (OODB)** is a type of database in which data is stored in the form of **objects**, similar to object-oriented programming. Each object contains both **data (attributes)** and **methods (functions)**. This approach makes it easier to represent real-world entities and complex data in the database.

Characteristics of Object-Oriented Database

- Data is stored as **objects**, not as tables.
- Objects are created using **classes**.
- Supports **encapsulation**, where data and methods are stored together.
- Supports **inheritance**, allowing reuse of existing classes.
- Supports **polymorphism**, where the same method can behave differently.
- Can store **complex data types** such as images, audio, and video.
- Each object has a **unique identity**.
- Objects are **persistent**, meaning they remain stored even after the program ends.



Example

A **library management system** is an example of an object-oriented database. Each Book is stored as an object with attributes like book ID, title, and author, along with methods such as check Availability (). The entire book object is saved in the database as one unit.

11.9 Comparison Table

Type of Database	Description	Examples
1. Centralized Database	All data is stored in a single location, and users access it via network.	Railway reservation system
2. Distributed Database	Data is stored across multiple physical locations, connected via a network.	Google Drive, banking systems
3. Relational Database (RDBMS)	Stores data in tables (relations); supports SQL.	MySQL, PostgreSQL, Oracle
4. NoSQL Database	Non-relational; used for large-scale unstructured data (documents, key-value).	MongoDB, Cassandra, Redis
5. Object-Oriented Database	Stores data as objects, like in object-oriented programming.	db4o, ObjectDB
6. Hierarchical Database	Data is organized in a tree-like structure.	IBM's IMS
7. Network Database	Similar to hierarchical, but child nodes can have multiple parents.	Integrated Data Store (IDS)
8. Cloud Database	Stored on cloud platforms, accessible via the internet.	Amazon RDS, Google Cloud SQL
9. Graph Database	Designed to store relationships using graph	Neo4j, Amazon

	structures.	Neptune
10. Temporal Database	Stores data related to time instances.	Used in stock market, scientific DBs
11. Data Warehouse	Stores historical data for analytics and decision making.	Amazon Redshift, Snowflake
12. Mobile Database	Used in mobile devices with limited storage and processing.	SQLite in Android, iOS apps

Short Questions

1. Define **Data**, **Information**, and **Metadata** with an example.
2. What is a **Database**? Give one real-life example.
3. Write any two drawbacks of the **File-based system**.
4. What is the **Database Approach**?
5. List the **components of a database environment**.
6. Who is a **Database Administrator (DBA)**? Write one role.
7. Define **Schema** and **Instance** in DBMS.
8. What is **Three-tier architecture** in DBMS?
9. Define **DDL** and **DML** with examples.
10. Write any two **advantages of DBMS** over file-based systems.
11. Who are **end users** in DBMS? Give one example.
12. Define **Data Models** and list any two.
13. What is a **Centralized Database**?
14. Mention two differences between **RDBMS and NoSQL DB**.
15. Expand **DBMS** and **RDBMS**.

Long Questions

1. Explain the **drawbacks of file-based systems** and how DBMS overcomes them.
2. Describe the **three-tier architecture of DBMS** with a neat diagram.
3. Discuss the **advantages of DBMS** in detail.
4. Explain the **components of the database environment** with examples.
5. What are **database languages**? Explain **DDL, DML, DCL, TCL** with syntax and examples.
6. Who are the different **types of database users**? Explain each with examples.
7. Write a detailed note on the **roles and responsibilities of a DBA**.
8. Explain the **different types of databases** with examples.
9. What is a **Data Model**? Explain **Hierarchical, Network, and Relational models** with diagrams.
10. Explain the **database approach** and its benefits compared to the file-based approach.

Unit – II: ER Model & Normalization

1. ER (Entity–Relationship) Model

1.1 Introduction to ER Model

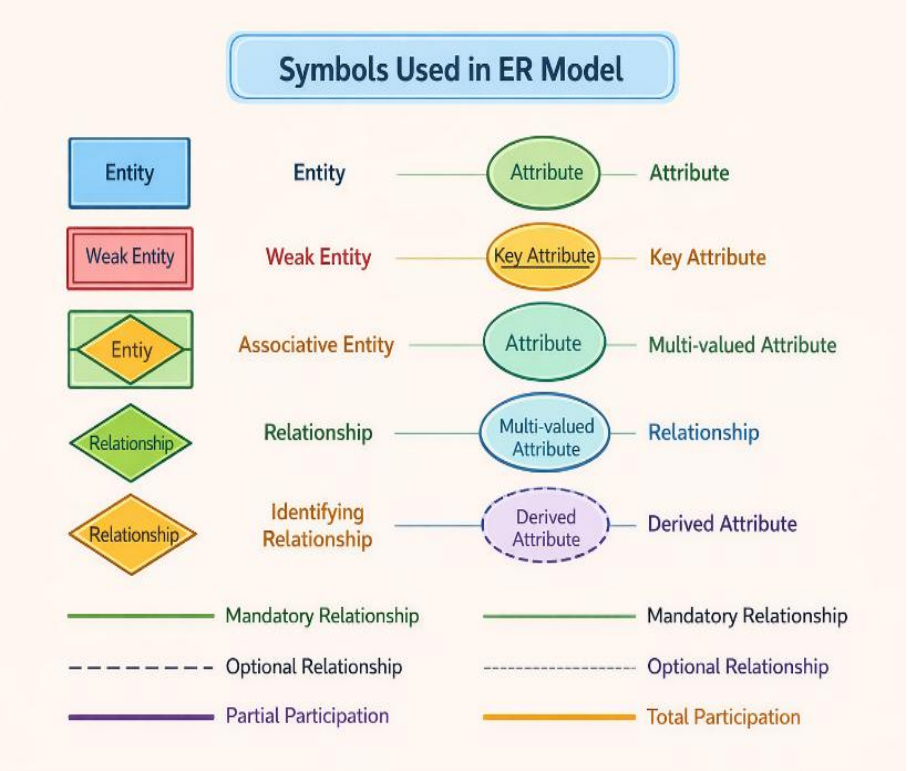
The Entity–Relationship (ER) Model is a high-level conceptual data model used to design databases. It helps in identifying the entities (objects) that need to be stored in a database and the relationships among those entities. The ER model gives a clear picture of the logical structure of a database before it is implemented physically.

An Entity Relationship Diagram (ER Diagram) is a graphical representation of the ER model. It shows entities, their attributes, and the relationships between them using standard symbols. The ER model is mainly used to represent real-world objects such as a person, car, employee, or company and how these objects are related to each other.

In simple words, an ER Diagram describes the structure of a database in a visual and easy-to-understand form.

1.2 Symbols Used in ER Model

The ER model uses different symbols to represent data components clearly. These symbols help in understanding the database design visually.



Symbol	Name	Meaning / Explanation
Rectangle	Entity	Represents a real-world object that exists independently in the database, such as Student or Employee.
Double Rectangle	Weak Entity	Represents an entity that does not have a primary key of its own and depends on a strong entity for identification.
Rectangle with Diamond	Associative Entity	Represents a many-to-many relationship that has its own attributes and behaves like an entity.
Diamond	Relationship	Represents the association between two or more entities in the database.
Double Diamond	Identifying Relationship	Connects a weak entity with its strong entity and helps in uniquely identifying the weak entity.
Oval	Attribute	Represents a property or characteristic of an entity or relationship.
Underlined Oval	Key Attribute	Represents an attribute that uniquely identifies each entity in an entity set.

Double Oval	Multi-Valued Attribute	Represents an attribute that can have more than one value for a single entity.
Dashed Oval	Derived Attribute	Represents an attribute whose value can be calculated from other attributes.
Solid Line	Mandatory Relationship	Shows that participation of an entity in a relationship is compulsory.
Dashed Line	Optional Relationship	Shows that participation of an entity in a relationship is not compulsory.
Single Line	Partial Participation	Indicates that not all entities in an entity set participate in the relationship.
Double Line	Total Participation	Indicates that all entities in an entity set must participate in the relationship.

1.3 Components of ER Diagram

An Entity–Relationship (ER) Diagram is a conceptual model used in database design. It visually represents the structure of a database by showing entities, their attributes, and the relationships among them.
As shown in the ER model diagram, the ER diagram has three main components:

a) Entity
b) Attribute
c) Relationship

Among these, the entity is the most important component because it represents real-world objects for which data is stored.

1.3.1 Entities

a) Entity: An Entity is any real-world object or concept that can be clearly identified and about which information is stored in a database. An entity has an independent existence and is distinguishable from other objects.

Entities may represent:

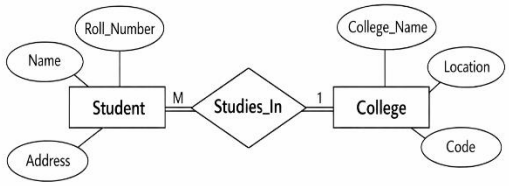
- Physical objects such as Student, Employee, Car, or Book
- Conceptual objects such as Course, Department, Bank, or Transaction

In an ER diagram, an entity is represented by a rectangle.

Example of Entity

Consider a college database system.

In this system:



- Student is an entity because information such as Roll Number, Name, and Address is stored about students.
- College is also an entity because it has its own information like College Name, Location, and Code.

Many students study in one college, so there exists a many-to-one relationship between Student and College. However, at this stage, the focus is only on understanding the entity, not the relationship.

Types of Entities

Entities are broadly classified into:

1. ***Strong Entity***
2. ***Weak Entity***

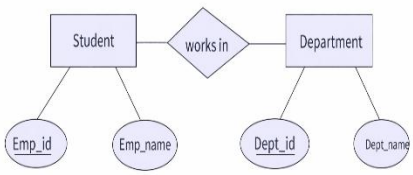
1. Strong Entity: A Strong Entity is an entity that has its own primary key. This primary key uniquely identifies each record in the entity set. A strong entity exists independently and does not depend on any other entity for its identification.

Characteristics of Strong Entity

1. A strong entity always has a primary key.
2. Each entity in the entity set can be uniquely identified.
3. It does not depend on any other entity.
4. It is represented using a single rectangle in an ER diagram.
5. The primary key attribute is shown using an underline.

Example of Strong Entity

Consider an Employee entity:



- Attributes: Employee_ID, Name, Designation, Salary
- Employee_ID is the primary key
- Each employee can be uniquely identified using Employee_ID

The Employee entity can exist independently in the database.

2. Weak Entity: A Weak Entity is an entity that does not have a primary key of its own. It cannot be uniquely identified by its attributes alone and depends on a strong entity for its identification and existence.

Characteristics of Weak Entity

1. A weak entity does not have a primary key.
2. It depends on a strong (owner) entity.
3. It has a partial key (discriminator).
4. It always has total participation in the relationship.
5. It is represented by a double rectangle.
6. The identifying relationship is represented by a double diamond.

Partial Key (Discriminator)

A Partial Key is an attribute of a weak entity that helps distinguish weak entities related to the same strong entity

- Represented using a dashed underline
- Cannot uniquely identify an entity by itself

.**Example of Weak Entity**

Consider a Company Database:

- Employee → Strong Entity
- Dependent → Weak Entity

Attributes of Dependent:

- Dependent_Name (Partial Key)
- Age
- Relationship

A dependent cannot exist without an employee.

Hence, Dependent is a weak entity.

1.5 Strong Entity Vs Weak Entity

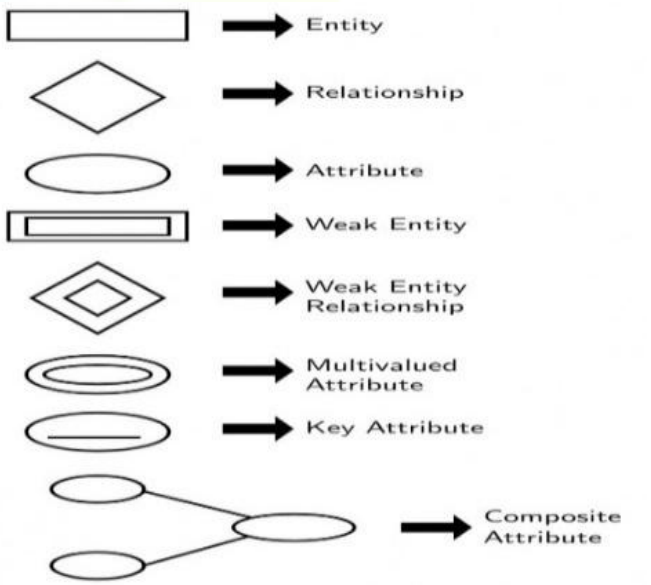
S. No.	Strong Entity	Weak Entity
1	A strong entity has a primary key that uniquely identifies each entity.	A weak entity does not have a primary key of its own.
2	It can be uniquely identified using only its own attributes.	It is identified using the primary key of the strong entity and a partial key.

3	A strong entity exists independently in the database.	A weak entity depends on a strong entity for its existence.
4	It is represented by a single rectangle in the ER diagram.	It is represented by a double rectangle in the ER diagram.
5	The primary key attribute is shown using an underline.	The partial key attribute is shown using a dashed underline.
6	Participation of a strong entity in a relationship may be partial or total.	Participation of a weak entity is always total.
7	Relationships between strong entities are represented using a single diamond.	The identifying relationship is represented using a double diamond.
8	A strong entity does not depend on any other entity for identification.	A weak entity cannot be identified without its strong entity.
9	The member of a strong entity set is called a dominant entity.	The member of a weak entity set is called a subordinate entity.
10	Examples include Employee, Student, Department.	Examples include Dependent, Bank Account, Transaction.

1.3.2 Attributes:

b). **Attributes:** An Attribute is a property or characteristic that describes an entity or a relationship in an Entity–Relationship (ER) model. Attributes store detailed information about entities and help in uniquely identifying and describing them.

For example, for the entity Student, attributes may include Roll_Number, Name, Date_of_Birth, and Address. In an ER diagram, attributes are represented using ovals (ellipses).



Types of Attributes

Attribute Type	Description	Example
Simple Attribute	An attribute that cannot be divided	Age, Gender

	into smaller parts and stores a single atomic value.	
Composite Attribute	An attribute that can be divided into sub-attributes, each having its own meaning.	Address → Street, City, State
Key Attribute	An attribute that uniquely identifies each entity in an entity set.	Roll_Number, Employee_ID
Single-Valued Attribute	An attribute that has only one value for each entity.	Date_of_Birth
Multi-Valued Attribute	An attribute that can have more than one value for a single entity.	Phone_Number, Email_ID
Derived Attribute	An attribute whose value is calculated from other attributes.	Age (derived from DOB)
Stored Attribute	An attribute whose value is stored in the database and used to derive other attributes.	Date_of_Birth
Partial Key (Discriminator)	An attribute of a weak entity used to distinguish entities related to the same strong entity.	Dependent_Name

1.3.3 Relationships

c) Relationships

A relationship represents an association among two or more entity sets. It explains how entities are connected to each other in a database system. Relationships help us understand how data stored in one entity is related to data stored in another entity. Without relationships, entities would exist independently and meaningful connections between data could not be represented.
In an ER (Entity–Relationship) diagram, a relationship is represented using a diamond symbol, which connects the related entity sets.

1.3.3.1 Relationship Based on Degree:-

The degree of a relationship refers to the number of entity sets participating in that relationship. Based on degree, relationships are classified into the following types:

1. ***Unary Relationship (1 entity set)***
2. ***Binary Relationship (2 entity sets)***
3. ***Ternary Relationship (3 entity sets)***

1. Unary Relationship (Recursive Relationship)

A Unary Relationship, also called a Recursive Relationship, is a relationship in which a single entity set is related to itself. In this type of relationship, the same entity participates more than

once but performs different roles. Unary relationships are used to represent real-world situations where objects of the same type are associated with one another.

Example

Consider the Course entity in a college database system.A course may require another course as a prerequisite. Here, both the course and its prerequisite belong to the same Course entity set, but they play different roles such as Main Course and Prerequisite Course.

2. Binary Relationship (2 Entity Sets)

A Binary Relationship is a relationship in which two different entity sets participate. It is the most common type of relationship used in database design. Binary relationships are used to show how entities from one entity set are related to entities from another entity set in a real-world system.

In an ER diagram, a binary relationship is represented by a diamond symbol placed between the two entity sets, which are shown using rectangles. This relationship helps in clearly understanding how data in one entity is connected to data in another entity.

Example

Consider a Student–Course relationship in a college database system.A student enrolls in a course, and a course can have many students. Here, Student and Course are two different entity sets participating in the relationship Enrolls. This forms a binary relationship because exactly two entity sets are involved.

3. Ternary Relationship (3 Entity Sets)

A Ternary Relationship is a relationship in which three different entity sets participate at the same time. It is used when the relationship among the entities cannot be correctly represented using only binary relationships. In a ternary relationship, all three entities are dependent on each other to give complete meaning.

In an ER diagram, a ternary relationship is represented by a diamond symbol connected to three entity sets, each shown using a rectangle.

Example

Consider a Supplier–Product–Warehouse relationship in a database system.A supplier supplies a particular product to a specific warehouse. Here, the meaning of the relationship depends on all three entity sets together:

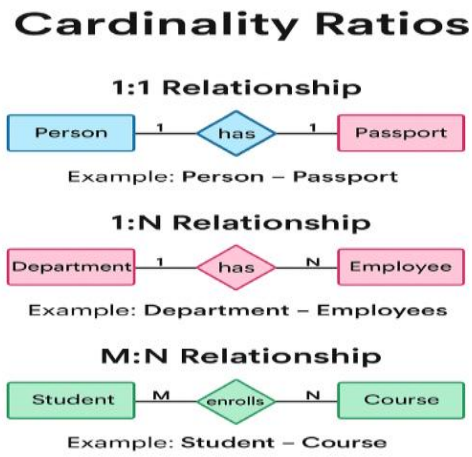
- Supplier
- Product
- Warehouse

If any one entity is removed, the relationship becomes incomplete. Hence, this is a ternary relationship.

1.3.3.2 Relationship Based on Cardinality:-

Cardinality describes the **number of entities** of one entity set that can be associated with the entities of another entity set through a relationship. In simple terms, it explains **how many times one entity can be related to another entity**. Cardinality is very important in database design because it helps define the **rules and limitations** of relationships between entities.

In an **ER diagram**, cardinality is usually shown near the relationship line using symbols or numbers such as **1, N, or M**. It ensures that the database correctly represents real-world constraints and prevents invalid data relationships.



Based on cardinality, relationships are classified into the following types:

1. ***One-to-One (1 : 1)***
2. ***One-to-Many (1 : N)***
3. ***Many-to-One (N : 1)***
4. ***Many-to-Many (M : N)***

1. One-to-One Relationship (1 : 1)

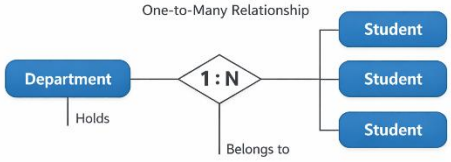
A **one-to-one relationship** is a relationship in which **one entity from the first entity set is associated with only one entity from the second entity set**, and **each entity of the second set is also associated with only one entity of the first set**. This type of relationship is used when there is a **unique and exclusive connection** between two entities, and no entity can participate in more than one relationship.

Example:

In a database system, each **Student** is assigned **only one Roll Number**, and each roll number uniquely identifies **only one student**. A student cannot have more than one roll number, and a roll number cannot be assigned to multiple students. Hence, the relationship between **Student** and **Roll Number** is a **one-to-one (1 : 1) relationship**.

2. One-to-Many Relationship (1 : N)

A **one-to-many relationship** is a relationship in which **one entity from the first entity set can be related to multiple entities of the second**

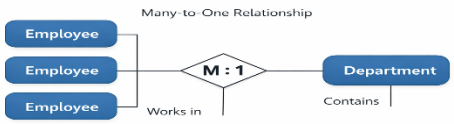


entity set, but **each entity of the second set is related to only one entity of the first set**. This is one of the **most commonly used relationships** in database systems.

Example:
In a college database, one **Department** can have many **Students**. However, each student can belong to only one department at a time. Here, the department is related to many students, but each student is linked to a single department, forming a one-to-many relationship.

3. Many-to-One Relationship (N : 1)

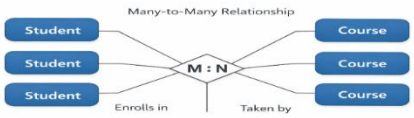
A **many-to-one relationship** is the **reverse of a one-to-many relationship**. In this case, **many entities from the first entity set are associated with only one entity of the second entity set**. This relationship shows that multiple records can depend on a single common entity.



Example:
In an organization, many **Employees** work in one **Department**. Each employee is assigned to only one department, but a department can have many employees working under it. Therefore, the relationship between Employee and Department is many-to-one.

4. Many-to-Many Relationship (M : N)

A **many-to-many relationship** is a relationship in which **many entities from one entity set are related to many entities from another entity set**.
In this type of relationship, both entity sets can participate multiple times. To implement this relationship in a relational database, an **associative (junction) table** is usually required.



Example:
In a university system, many **Students** can enroll in many **Courses**, and each course can have many students. A student may register for multiple courses, and a course may be taken by multiple students. Hence, the Student–Course relationship is a many-to-many relationship.

1.4 Keys in DBMS

In a **Database Management System (DBMS) key** is an attribute or a group of attributes that is used to **identify records (tuples) uniquely** in a relation (table).Keys ensure that **no two rows are identical**, help in **maintaining data consistency**, and are essential for **linking tables** in a database.

Without keys, a database may contain **duplicate records**, leading to incorrect results and data inconsistency.

Importance of Keys in DBMS

- **Prevent duplicate data:** By restricting duplicate values, especially in primary keys, keys prevent repeated and redundant data entries.
- **Maintain entity integrity:** Keys enforce entity integrity by ensuring that primary key values are unique and never NULL for any record.
- **Maintain referential integrity:** Keys maintain referential integrity by ensuring that foreign key values correctly reference existing primary key values in related tables.
- **Help in efficient data retrieval:** Keys enable faster searching and retrieval of records by allowing the database to access data using indexed key values.
- **Ensure uniqueness of records**: Keys uniquely identify each record in a table, ensuring that no two records represent the same entity.
- **Establish relationships between tables:** Keys connect different tables through primary key–foreign key relationships, enabling structured and meaningful data associations.

1.4.1 Classification of Keys in DBMS

1. ***Super Key***
2. ***Candidate Key***
3. ***Primary Key***
4. ***Alternate Key***
5. ***Foreign Key***
6. ***Composite Key***

1. Super Key

A super key is a set of one or more attributes that can uniquely identify each record (tuple) in a table. It ensures that no two records have identical values for the attributes included in the super key. A super key may contain extra attributes that are not required for uniqueness, but the combination still uniquely identifies records.

Every table has at least one super key, and all candidate keys and primary keys are also super keys

Syntax

Super Key = {Attribute1 [, Attribute2, Attribute3, ...] }

Each student is issued an ID card that contains:

- Student ID number

Example

College Identity Card System.

```
CREATE TABLE STUDENT
(
   STUDENT_ID   INT NOT NULL,
   STU_NAME     VARCHAR (35),
   DEPARTMENT   VARCHAR
(30),DOB          DATE
);
```

- Student name
- Department

The Student ID number alone can uniquely identify a student.
So, Student ID acts like a key.
Now:

- Student ID → uniquely identifies a student
- Student ID + Name → also identifies the same student
- Student ID + Department → still identifies the same student

All these combinations can identify the student without confusion
Therefore:

- Student ID
- Student ID + Name
- Student ID + Department

are like super keys.
This analogy shows that a super key may include extra information, but as long as it can uniquely identify a person or record, it is considered a super key.

2. Candidate Key

A candidate key is an attribute or a set of attributes that can uniquely identify each record (tuple) in a table without containing any extra attributes. It is a minimal super key, meaning that no attribute can be removed from it without losing uniqueness .A table can have more than one candidate key, and each candidate key is equally capable of identifying records. From all candidate keys, one is chosen as the primary key, and the remaining candidate keys are called alternate keys.

Syntax
Candidate Key = {Attribute1 [, Attribute2, ...]}

Each student has the following details:

- Student ID number
- Email ID
- Student name
- Department

In **above example**:

- Student ID number is unique for every student.
- Email ID is also unique for every student.

So, the candidate keys are:

Example
College Identity Card System

```
CREATE TABLE STUDENT
(
   STUDENT_ID   INT NOT
NULL,
   EMAIL_ID     VARCHAR(40)
NOT NULL UNIQUE,
   STU_NAME     VARCHAR(35),
   DEPARTMENT
VARCHAR(30)
);
```

- Student ID
- Email ID

Both can uniquely identify a student by themselves and do not contain extra information.
If Student ID is chosen as the primary key, then Email ID becomes an alternate key.
This example shows that a candidate key is the smallest and most efficient identifier that uniquely identifies a record.

3. Primary Key

A **primary key** is **one of the candidate keys** selected to **uniquely identify each record (tuple)** in a table. It is the **main identifier** of the table and is used to distinguish one record from all others. A primary key **cannot contain NULL values** and **cannot have duplicate values**. Each table can have **only one primary key**.

Example
College Identity Card System

```
CREATE TABLE STUDENT
(
   STUDENT_ID   INT NOT NULL,
   EMAIL_ID     VARCHAR(40) NOT NULL
UNIQUE,
   STU_NAME     VARCHAR(35),
   DEPARTMENT   VARCHAR(30),
   PRIMARY KEY (STUDENT_ID)
                        );
```

Syntax:PRIMARY KEY (Attribute)

Each student has the following details:

- Student ID number
- Email ID
- Student name
- Department

In the above example:

- **Student ID number** is selected as the **primary key**.
- It uniquely identifies each student.
- It does not allow **NULL or duplicate values**.

Although **Email ID** is also unique, only **one attribute can be chosen** as the primary key.
If **Student ID** is selected as the primary key, then **Email ID becomes an alternate key**.
This example shows that a **primary key is the official and main identifier** used by the database to uniquely identify each record.

4. Alternate Key

An **alternate key** is a **candidate key that is not selected as the primary key**. Even though it is not chosen as the primary key, it can still **uniquely identify each record** in a table. A table can have **more than one alternate key**, depending on the number of candidate keys available.Alternate keys help in **maintaining uniqueness** of data other than the primary key.

Syntax
UNIQUE (Attribute)

Example
College Identity Card System

```
CREATE TABLE STUDENT
(
   STUDENT_ID   INT NOT NULL,
   EMAIL_ID     VARCHAR(40) NOT NULL UNIQUE,
   STU_NAME     VARCHAR(35),
   DEPARTMENT   VARCHAR(30),
   PRIMARY KEY (STUDENT_ID)
);
```

Each student has the following details:

- Student ID number
- Email ID
- Student name
- Department

In the above example:

- **Student ID** is chosen as the **primary key**.
- **Email ID** is also unique but **not selected** as the primary key.

Therefore, **Email ID becomes the alternate key**.

This example shows that an **alternate key is a candidate key kept as an alternative identifier** when it is not chosen as the primary key.

5. Foreign Key

A **foreign key** is an attribute or a **set of attributes** in one table that **refers to the primary key of another table**. It is used to **establish and maintain a relationship** between two tables in a database. A foreign key helps maintain **referential integrity**, ensuring that the data in related tables remains consistent. A foreign key **can contain duplicate values** and **may allow NULL values** (if permitted).

Syntax:

FOREIGN KEY (Attribute) REFERENCES Table_Name(Primary_Key)

Example
College Identity Card System

```
CREATE TABLE COURSE
(
   COURSE_ID   INT NOT NULL,
   COURSE_NAME VARCHAR(30),
   PRIMARY KEY (COURSE_ID)
);
CREATE TABLE STUDENT
(
   STUDENT_ID   INT NOT NULL,
   STU_NAME     VARCHAR(35),
   COURSE_ID    INT,
   PRIMARY KEY (STUDENT_ID),
   FOREIGN KEY (COURSE_ID) REFERENCES COURSE(COURSE_ID)
);
```

Each student has the following details:

- Student ID number
- Student name
- Course ID

In the above example:

- **COURSE_ID** is the **primary key** in the COURSE table.
- **COURSE_ID** in the STUDENT table is a **foreign key**.
- It links each student to a valid course.
- A student cannot be assigned a course that does not exist in the COURSE table.

This example shows that a **foreign key is used to connect two tables and**

ensure valid relationships between records.

6. Composite Key

A **composite key** is a **key formed by combining two or more attributes** to uniquely identify each record (tuple) in a table. It is used when **a single attribute is not sufficient** to uniquely identify records. Composite keys are commonly used in **relationship tables** and **many-to-many relationships**.

A composite key can also act as a **primary key**.

Syntax

PRIMARY KEY (Attribute1, Attribute2)

Example
College Enrollment System

```
CREATE TABLE ENROLLMENT
(
   STUDENT_ID  INT NOT NULL,
   COURSE_ID   INT NOT NULL,
   SEMESTER    INT,
   PRIMARY KEY (STUDENT_ID, COURSE_ID)
);
```

Each enrollment record contains:

- Student ID
- Course ID
- Semester

In the above example:

- **STUDENT_ID alone** cannot uniquely identify a record.
- **COURSE_ID alone** cannot uniquely identify a record.
- The **combination of STUDENT_ID and COURSE_ID** uniquely identifies each enrollment.

This example shows that a **composite key uses multiple attributes together** to uniquely identify a record when a single attribute is not sufficient.

2.Special Concepts

Specialization/generalization

2.1 Introduction:

In **DBMS**, **specialization** and **generalization** are abstraction techniques used in the **Enhanced Entity–Relationship (EER) model** to represent relationships between a higher-level entity and lower-level entities. **Generalization** is a bottom-up approach in which two or more entity sets with common attributes are combined to form a higher-level generalized entity set. It helps in reducing redundancy and improving data consistency by storing common attributes only once. **Specialization** is a top-down approach in which an entity set is divided into one or more lower-level entity sets based on distinguishing characteristics. Each specialized entity inherits the attributes of the parent entity and may have its own specific attributes. These concepts help in modeling real-world constraints clearly and support better database design and semantic clarity.

Superclass: A **superclass** is a class that provides the fundamental attributes and operations that can be shared by other classes. It serves as the base for inheritance, allowing derived classes to obtain its data members and member functions. By defining common features in a superclass, object-oriented programs achieve better reusability, logical organization of classes, and simplified program maintenance.

Example:
Consider a class Vehicle that defines common data members such as speed and common member functions such as start(). If another class Car is derived from Vehicle, the Car class automatically acquires the properties and behaviors of the Vehicle class and may also include additional functions like openDoor(). In this case, Vehicle is the superclass and Car is the subclass, demonstrating how shared features are defined once in the superclass and reused in derived classes.

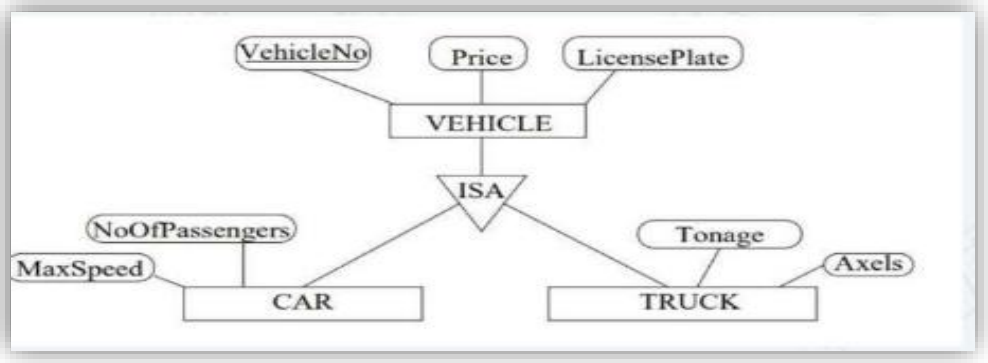
Subclass: A **subclass** (also known as a **derived class** or **child class**) is a class that is created from another class called a superclass. The subclass inherits the data members and member functions of the superclass and may extend or modify them by adding new features. Subclasses help in specialization of classes, promote code reuse, and support hierarchical organization in object-oriented programming.

Example:
Consider a class Vehicle that contains common data members such as speed and common member functions such as start(). When a class Car is created by inheriting from Vehicle, the Car class automatically gains access to the data members and member functions of Vehicle and can also define additional functions like openDoor(). In this relationship, Car is the subclass derived from the superclass Vehicle.

2.2 Generalization

Generalization is a concept of the Enhanced Entity–Relationship (EER) Model in which two or more lower-level entity sets having common attributes are combined into a single higher-level entity set. The purpose of generalization is to identify shared characteristics among entities and represent them in a common entity. This process follows a bottom-up approach, where individual entities are examined and their common attributes are grouped together. The higher-level entity formed through this process is called the super entity, while the lower-level entities are known as sub entities**.**

ER Diagram Example (Generalization)

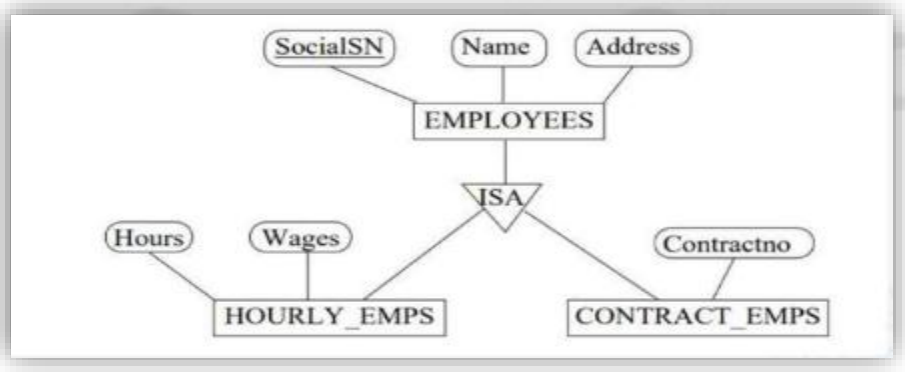


- **CAR** and **TRUCK** are two entity sets.
- They share common attributes like **VehicleNo, Price, LicensePlate**.
- These common attributes are grouped into a superclass called **VEHICLE**.
- **CAR** and **TRUCK** become subclasses of VEHICLE.
- This process is called **Generalization**.

2.3 Specialization

Specialization is a concept in the Enhanced Entity–Relationship (EER) Model in which a single entity is divided into two or more sub-entities based on their distinct features. It follows a top-down approach, where a general entity is split into more specific entities. The general entity is called the superclass, and the resulting entities are called subclasses

Example ER Diagram:



EMPLOYEES is the superclass with attributes: SocialSN, Name, Address.
It is specialized into two subclasses:

HOURLY_EMPS (with additional attributes Hours, Wages)
CONTRACT_EMPS (with additional attribute Contractno)

- EMPLOYEES is the superclass with attributes SocialSN, Name, Address.

- It is specialized into HOURLY_EMPS and CONTRACT_EMPS.
- HOURLY_EMPS inherits all attributes of EMPLOYEES and has Hours, Wages.
- CONTRACT_EMPS inherits all attributes of EMPLOYEES and has ContractNo.

2.3 Aggregation

Aggregation is a special type of relationship in the ER / EER model that represents a whole–part relationship between entities. In aggregation, one entity (called the whole) is made up of other entities (called the parts). An important feature of aggregation is that the part entities can exist independently, even if the whole entity does not exist. Hence, aggregation does not imply total dependency.

Aggregation is also known as a Has–A relationship.

Example

Consider a college database system.

- Department is an entity.
- Professor is another entity.
- A Department has Professors.

In this example, Department is the whole entity and Professor is the part entity. Even if a department is closed, a professor can still exist and may move to another department. Therefore, this relationship is an example of Aggregation and not a weak entity relationship.

2.4 Composition

Composition is a stronger form of aggregation that represents a whole–part relationship where the part entity cannot exist without the whole entity. In composition, the part is completely dependent on the whole for its existence. If the whole entity is removed, the part entities are also removed automatically.

Composition is also known as a "Contains-A" relationship.

Example

Consider a House and Room.

- House is the whole entity.
- Room is the part entity.
- A House contains Rooms.

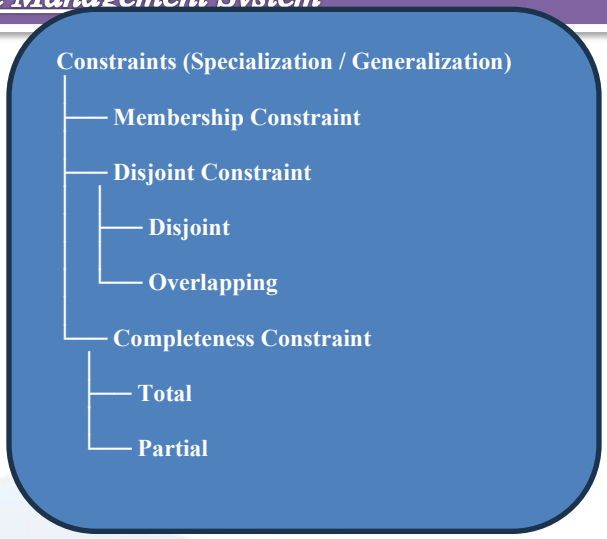
If the house is deleted, all the rooms are automatically deleted. A room cannot exist without a house. Therefore, this relationship is an example of Composition, not aggregation.

2.5 Constraints on specialization/generalization

There are three constraints that may apply to a specialization/generalization:

1. Membership constraints
2. Disjoint constraints

3. Completeness constraints



1.Membership Constraints

Membership constraints specify **how entities of a superclass become members of its subclasses** in a specialization or generalization relationship. These constraints define the rules used to decide subclass membership.

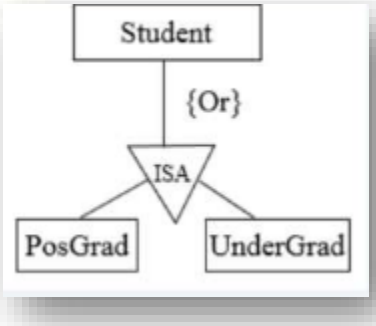
- **Condition Defined Membership:**
  In this case, membership in a subclass is determined by a **specific condition** given in the requirements.
  Example: A **Tanker** is a type of **Ship** where the cargo = oil.
  Here, the condition (cargo = oil) decides whether a Ship belongs to the Tanker subclass.
- **User Defined Membership:**
  Sometimes, subclass membership is **defined by the database designer** rather than by a strict condition. This approach is used to **simplify the design** or to represent **complex real-world relationships**.
  In such cases, the designer manually decides which entities belong to which subclasses.

2. Disjoint Constraint

a) Disjoint Constraints (OR Relationship)

Disjoint Constraint is a rule used in specialization/generalization when a superclass is divided into two or more subclasses. This constraint states that each entity of the superclass can belong to only one subclass at a time. In other words, subclass membership is mutually exclusive.

In ER/EER diagrams, the disjoint constraint is represented using the {OR} notation near the ISA (is-a) relationship. The meaning of this constraint is "A OR B, but not both." This rule helps in clearly separating subclasses and avoids confusion or overlap in classification.



Example

Consider a Student entity as the superclass. It is specialized into two subclasses: UnderGrad and PostGrad. A student can be either an Undergraduate or a

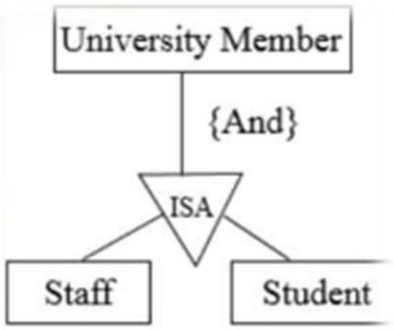
Postgraduate, but cannot be both at the same time. Therefore, this specialization uses a Disjoint Constraint (OR relationship).

b) Overlapping Constraint (AND Relationship)

Overlapping constraint is used in specialization/generalization when an entity of a superclass is allowed to belong to more than one subclass at the same time. This constraint is applied when subclass memberships are not mutually exclusive and an entity can play multiple roles.

In ER/EER diagrams, the overlapping constraint is represented using the {AND} notation near the ISA relationship. The meaning of this constraint is that an entity can be A and B simultaneously.

For example, consider University Member as a superclass with two subclasses: Student and Staff. A person in a university can be both a student and a staff member at the same time, such as a research scholar who studies and also teaches. Therefore, this specialization follows an overlapping constraint. This constraint helps in modeling real-world situations where roles naturally overlap and ensures that the database design accurately reflects such cases.



3. Completeness Constraints

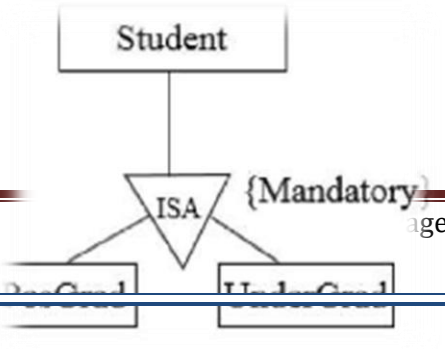
Completeness constraints are used in specialization/generalization to specify whether every entity of a superclass must belong to at least one subclass or not. In other words, they define whether subclass membership is mandatory or optional for superclass entities. Completeness constraints help in deciding how fully the superclass is covered by its subclasses.

There are two types of completeness constraints:

Total Specialization (Mandatory)

Total specialization, also called mandatory specialization, is a type of completeness constraint in specialization/generalization. In total specialization, every entity of the superclass must belong to at least one subclass. This means that it is compulsory for each superclass entity to be classified into a subclass.

In ER/EER diagrams, total specialization is represented using a double line between the superclass and the ISA relationship. The double line indicates that no entity of the superclass can exist without being a member of a subclass.

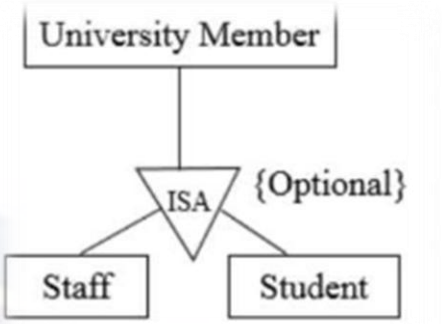


Example:

If **Student** is a superclass and **Undergraduate** and **PostGraduate** are subclasses, then total specialization means **every student must be either an undergraduate or a postgraduate**. No student can exist without belonging to at least one of these subclasses.

Partial Specialization (Optional)

Partial specialization, also called optional specialization, is a type of completeness constraint in specialization/generalization. In partial specialization, some entities of the superclass may not belong to any subclass. This means that subclass membership is not compulsory, and an entity can exist only in the superclass without being classified into a subclass.

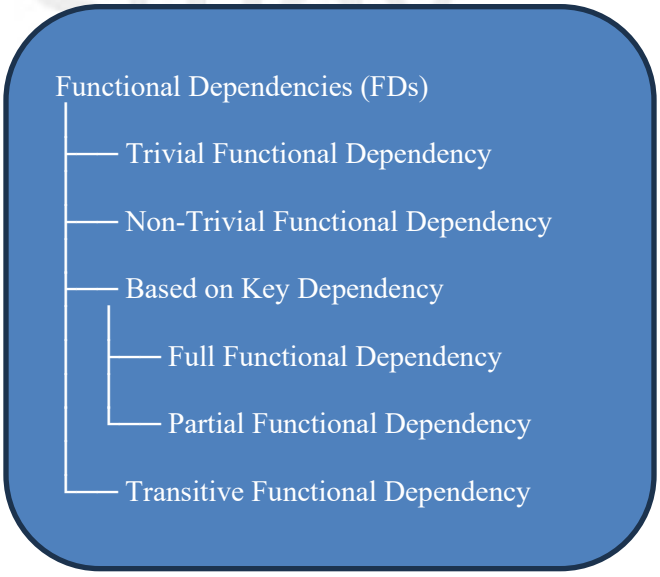


Example:

Consider University Member as a superclass with Staff and Student as subclasses. Some university members may be neither staff nor students (for example, visiting members or alumni). Such members can exist only as University Members without belonging to any subclass. Therefore, this specialization is partial.

3.Functional Dependencies (FDs)

Functional Dependency (FD) determines the relation of one attribute to **another** attribute in a database management system (DBMS) system. Functional dependency helps you to maintain the **quality of data** in the database.



A Functional Dependency (FD) is a relationship between two attributes in a relation. A functional dependency A → B holds in a relation if for any two tuples, whenever the values of attribute A are the same, the values of attribute B are also the same. In simple words, attribute A uniquely determines attribute B.

If a column A of a table can uniquely identify the value of another column B in the same table, then the functional dependency is written as A → B.

Meaning of A → B

- **B is functionally dependent on A**
  This means the value of B depends on the value of A.
- **A is the determinant**
  A is the attribute (or set of attributes) that determines the value of another attribute.
- B is the dependent attribute

- B is the attribute whose value is determined by A.

Types of Functional Dependency

Functional Dependency is broadly classified into:

1. ***Trivial functional dependency***
2. ***Non-trivial functional dependency***

1. Trivial Functional Dependency

A functional dependency **A → B** is called a **trivial functional dependency** if **B is a subset of A**. This means that the dependent attribute is already included in the determinant, so the dependency is **always true**.

For example, the dependencies **A → A** and **B → B** are trivial functional dependencies because an attribute can always determine itself. Such dependencies do not give any new information about the relation and are mainly used for theoretical understanding in database design.

A→B has trivial functional dependency if B is a subset of A.
The following dependencies are also trivial like: A→A, B→B
Example: School identity card.

- Consider a table with two columns. **Student_ID** ,**Student_Name**

The card contains:

- **Student_ID**
- **Student_Name**

Now consider this relationship:
{Student_ID, Student_Name} → Student_ID

This is a **trivial functional dependency** because **Student_ID is already included** in the given information. You don't need any extra data to know the Student_ID—it is already there.

2. Non-trivial Functional Dependency

A→B has a non-trivial functional dependency if B is **not** a subset of A. When A∩B is **NULL**, then A→B is called as **complete non-trivial**.

Example: Consider the following functional dependencies:

- ID → Name
- Name → DOB

Both of these are non-trivial functional dependencies because the attribute on the right-hand side is not a subset of the left-hand side.

Explanation:

- In ID → Name, the value of ID uniquely determines the Name. Since Name is not part of ID, this dependency is non-trivial.

- In Name → DOB, the value of Name determines the Date of Birth (DOB). Again, DOB is not part of Name, so this is also non-trivial..

3.1 Transitive Dependency

A Transitive Dependency is a type of functional dependency in which one attribute depends on another attribute through a third attribute. In simple words, if A determines B and B determines C, then A indirectly determines C. This indirect dependency is called a transitive dependency.
Formally, a transitive dependency exists when:

- **A → B**
- **B → C**
- and **B is not a key attribute**

Then,

- **A → C** is a **transitive dependency**

Note: A transitive dependency can only occur in a relation of three or more attributes.

Example

Consider a **company database** with the following details:

- Employee_ID
- Project_ID
- Project_Name

Now look at the dependencies:

- Employee_ID → Project_ID
  (Each employee is assigned to one project)

- Project_ID → Project_Name
  (Each project ID has a fixed project name)

From these, we get:

- Employee_ID → Project_Name

Explanation:
Here, Employee_ID does not directly determine Project_Name.
It determines Project_ID first, and Project_ID determines Project_Name.
So, Project_Name depends on Employee_ID through Project_ID.

4. Data Redundancy

Data redundancy is the condition in which the same data is stored repeatedly in different locations within a database system. This repetition of data usually occurs due to poor database design or the absence of proper normalization. When identical data items appear in multiple files or tables, it results in unnecessary duplication.

Example Table (Unnormalized)

StudentID	StudentName	CourseID	CourseName	Faculty	FacultyPhone
101	Ravi	CSE101	DBMS	Sharma	9876543210
102	Priya	CSE101	DBMS	Sharma	9876543210
103	Amit	CSE102	OS	Reddy	9876549876

Explanation of Data Redundancy

- Data redundancy is clearly visible because the course **CSE101**, course name **DBMS**, faculty **Sharma**, and faculty phone number **9876543210** are repeated for both **Ravi** and **Priya**.
- This shows that the same course and faculty information is stored **multiple times** in the table.
- If the faculty phone number of **Sharma** changes, it must be updated in **all rows** where CSE101 appears.
- If one row is not updated, it will lead to **data inconsistency**.
- Adding a new course without enrolling any student is difficult, which causes an **insertion anomaly**.
- If all students of **CSE101** are deleted, the course and faculty details will also be removed, causing a **deletion anomaly**.
- This example clearly shows that **data redundancy causes errors and maintenance problems**.
- Therefore, **normalization is required** to store data efficiently and avoid these problems.

Problems Caused by Data Redundancy

- **Data inconsistency** occurs because multiple copies of the same data may not be updated correctly.
- **Update anomaly** arises when a change in one place is not reflected in other places.
- **Insertion anomaly** happens when new data cannot be inserted without adding duplicate information.
- **Deletion anomaly** occurs when deleting data causes accidental loss of important information.
- **Increased storage space** is required due to repeated storage of the same data.
- **Reduced efficiency** results because redundant data slows down database operations.
- **Maintenance becomes difficult** as multiple copies of data must be managed and updated.

How to Reduce Data Redundancy

- **Use normalization techniques** to divide tables into smaller, well-structured tables so that each piece of data is stored only once.
- **Apply generalization and specialization in EER models** to place common attributes in a superclass and avoid repeating them in multiple entities.
- **Store common attributes in separate tables** and link them using keys, instead of repeating the same data in many tables.

Design databases using proper ER modeling to clearly identify entities, attributes, and relationships before creating tables, which helps prevent duplication of data

4.1 Normalization

Normalization is a method used in **database design** to arrange data in a clear and systematic way. The main aim of normalization is to **reduce repeated data** and store each piece of information **only once**. This is done by breaking a large table into **smaller, meaningful tables** and connecting them using proper relationships.

Normalization helps in avoiding common database problems such as **data redundancy**, **data inconsistency**, and **update, insertion, and deletion anomalies**. By using normalization, the database becomes **more reliable**, **easier to manage**, and **more accurate**. It also improves the overall structure of the database and makes future changes simpler.

Why Normalization is Needed

- To **avoid storing the same data again and again**
- To **keep data consistent and correct**
- To **eliminate update, insertion, and deletion anomalies**
- To **use storage space efficiently**
- To **improve the performance and maintenance of the database**

Un Normalized Form (UNF)

StudentID	Name	Courses (with Instructor)	Hobbies
1	Ravi	{DBMS–Sharma, OS–Rao}	{Cricket, Music}
2	Kavya	{DBMS–Sharma, OS–Rao}	{Dance}
3	Arjun	{DBMS–Verma}	{Cricket}

A relation is in **Un-Normalized Form (UNF)** when it contains:

- **Multi-valued attributes**
- **Repeating groups**
- **Non-atomic values**

In this table:

- The attribute **Courses (with Instructor)** contains multiple values in a single cell.
- The attribute **Hobbies** also contains more than one value.
- Data is not stored in a simple, structured manner.

Because of this:

- Data redundancy exists
- Update, insertion, and deletion anomalies may occur

Hence, the table is in **UNF**.

Example: Student, Course, Instructor

We start with this **single table**:

4.1.1Normal Forms

The different Normal Forms used in database normalization are:

1. ***First Normal Form (1NF)***
2. ***Second Normal Form (2NF)***
3. ***Third Normal Form (3NF)***
4. ***Boyce–Codd Normal Form (BCNF)***
5. ***Fourth Normal Form (4NF)***
6. ***Fifth Normal Form (5NF)***

(a) First Normal Form (1NF)

A relation (table) is said to be in **First Normal Form (1NF)** if all its attributes contain only atomic (single) values and there are no repeating groups or multi-valued attributes. In 1NF, each field of a table must store exactly one value, and each record must be uniquely identifiable.

In simple terms, **1NF removes multi-valued attributes**. If a column contains more than one value (such as multiple courses or hobbies in a single cell), the table is not in 1NF. To convert such a table into 1NF, the multi-valued attributes are split so that each value appears in a separate row.

1. Rule of 1NF

• Each attribute must contain only one value in each cell.
• Multi-valued attributes are not allowed.
• Repeating groups must be removed to satisfy 1NF.

2. Example – Conversion from UNF to 1NF

• The original table is in **Un-Normalized Form (UNF)** because it contains multi-valued attributes.
• To convert it into **First Normal Form**, the following actions are taken.

3. Action Taken

• The attribute **Courses (with Instructor)** is split into two separate attributes:
o Course
o Instructor

• The **Hobbies** attribute is split so that each row contains only one hobby.
• Multiple rows are created where necessary to ensure atomic values.

✓ After these steps, the table satisfies all the rules of **First Normal Form (1NF)**.

Table in 1NF

StudentID	Name	Course	Instructor	Hobby
1	Ravi	DBMS	Sharma	Cricket
1	Ravi	DBMS	Sharma	Music
1	Ravi	OS	Rao	Cricket
1	Ravi	OS	Rao	Music
2	Kavya	DBMS	Sharma	Dance
2	Kavya	OS	Rao	Dance
3	Arjun	DBMS	Verma	Cricket

✓ Now all values are atomic
✓ Table is in **First Normal Form**

(b) Second Normal Form (2NF)

A relation (table) is said to be in **Second Normal Form (2NF)** if it is already in **First Normal Form (1NF)** and **there is no partial dependency**. This means that every non-key attribute must be **fully dependent on the entire primary key**.

In simple terms, **2NF removes partial dependency**. Partial dependency occurs when a table has a **composite primary key** and a non-key attribute depends on **only a part of that key**, not on the whole key.

1. Rule of 2NF

• The table must be in **First Normal Form (1NF)**.
• There should be **no partial dependency**.
• Every non-key attribute must depend on the **entire primary key**.

2. Example – Checking 2NF

• The table obtained after 1NF has the attributes:
StudentID, Name, Course, Instructor, Hobby
• The **primary key is a composite key**:
(StudentID, Course)

3. Identify Partial Dependency

• **StudentID → Name**
• **Course → Instructor**
• Here, **Name** depends only on StudentID, not on the full key.
• **Instructor** depends only on Course, not on the full key.
• These are **partial dependencies**.
➞ Hence, the table **violates the rule of Second Normal Form (2NF)**.

4. Action Taken (Conversion to 2NF)

• Separate the attributes involved in partial dependency into new relations.
• Create **STUDENT(StudentID, Name)**.
• Create **COURSE(Course, Instructor)**.
• Keep the relation that links students and courses separately.
Tables in 2NF
STUDENT

StudentID	Name
1	Ravi
2	Kavya
3	Arjun

STUDENT_COURSE

StudentID	Course	Hobby
1	DBMS	Cricket
1	DBMS	Music
1	OS	Cricket
1	OS	Music
2	DBMS	Dance
2	OS	Dance
3	DBMS	Cricket

COURSE

Course	Instructor
DBMS	Sharma
OS	Rao
DBMS	Verma

✓ Partial dependency removed

✓ All non-key attributes depend on the **entire primary key**

✓ Tables are in **Second Normal Form (2NF)**

(c) Third Normal Form (3NF)

A relation is said to be in **Third Normal Form (3NF)** if it satisfies the following conditions:

1. The relation is in **Second Normal Form (2NF)**
2. **No transitive dependency exists**, that is, non-key attributes should not depend on other non-key attributes

In simple words, **3NF removes transitive dependency**. A transitive dependency occurs when one non-key attribute depends on another non-key attribute instead of depending directly on the primary key.

1. Rule of Third Normal Form (3NF)

• A table is in Third Normal Form (3NF) if it is already in Second Normal Form (2NF).
• There should be no transitive dependency in the table.
• All non-key attributes must depend only on the primary key.

2. Transitive Dependency (Given Example)

• Instructor → InstructorPhone
• Course → Instructor
• From the above dependencies, we get:
o Course → InstructorPhone
• This is a transitive dependency because **InstructorPhone depends on Course through Instructor**.

3. Action Taken (Removing Transitive Dependency)

• Identify the attributes involved in the transitive dependency.
• Separate the instructor details into a new relation.
• Create **INSTRUCTOR(Instructor, InstructorPhone)**.
• Retain **COURSE(Course, Instructor)** as a separate relation.
• After decomposition, all non-key attributes depend directly on the primary key.

✓ **The table now satisfies the conditions of Third Normal Form (3NF).**

INSTRUCTOR Table

Instructor	Phone
Sharma	xxxx
Rao	yyyy
Verma	zzzz

COURSE Table

Course	Instructor
DBMS	Sharma
OS	Rao

STUDENT Table

StudentID	Name

STUDENT_COURSE_HOBBY Table

StudentID	Course	Hobby

✓ Transitive dependency removed

✓ Tables are now in **Third Normal Form**

(d) Boyce-Codd Normal Form (BCNF)

Boyce–Codd Normal Form (BCNF) is an advanced form of normalization and is considered a stronger version of Third Normal Form (3NF). It was introduced to remove certain types of anomalies that may still exist even after a relation is converted into 3NF.

A relation is said to be in Boyce–Codd Normal Form if for every functional dependency A → B, A is a super key. This means that every determinant must be a candidate key or a part of a candidate key. In BCNF, no non-key attribute is allowed to determine another attribute.

BCNF ensures that the database structure is free from redundancy and inconsistency caused by functional dependencies. When a relation violates BCNF, it is decomposed into smaller relations so that each relation satisfies the BCNF condition. Thus, BCNF provides a more strict and reliable database design compared to 3NF.

Rule of BCNF

A relation is in **Boyce–Codd Normal Form (BCNF)** if **for every functional dependency A → B, A is a super key**.

In simple words, **only keys are allowed to determine other attributes**.

Checking BCNF for Our Example

STUDENT

- StudentID → Name
- StudentID is a **primary key**

✓ Satisfies BCNF

INSTRUCTOR

- Instructor → Phone
- Instructor is a **primary key**

✓ Satisfies BCNF

COURSE

STUDENT_COURSE_HOBBY

Result
All relations satisfy the BCNF condition.
✅ **Our database is already in BCNF**

(e)Fourth Normal Form (4NF)

Fourth Normal Form (4NF) is a higher level of normalization that deals with multi-valued dependencies in a relation. A relation is said to be in 4NF if it is in Boyce–Codd Normal Form (BCNF) and does not contain any non-trivial multi-valued dependency.

In simple terms, 4NF removes problems caused by multiple independent multi-valued attributes in a single table. When two or more attributes are independent of each other but depend on the same key, storing them together causes redundancy.

1: Why STUDENT_COURSE_HOBBY is NOT in 4NF

- In the relation **STUDENT_COURSE_HOBBY**, a student can enroll in **many courses**.
- A student can also have **many hobbies**.
- **Courses and hobbies are independent** of each other.
- This results in the multi-valued dependencies:
  - **StudentID →→ Course**
  - **StudentID →→ Hobby**
- Since two independent multi-valued dependencies exist in the same relation, it **violates the rule of Fourth Normal Form (4NF)**.

2: Conversion into Fourth Normal Form (4NF)

- To remove the multi-valued dependency, the relation is **decomposed**.
- The original table is split into **two separate relations**.
- The first relation is **STUDENT_COURSE(StudentID, Course)**.
- The second relation is **STUDENT_HOBBY(StudentID, Hobby)**.

- Each table now represents **one independent fact**.
- This decomposition **eliminates redundancy** and the resulting relations **satisfy the conditions of 4NF**

STUDENT_COURSE

StudentID	Course

STUDENT_HOBBY

StudentID →→ Hobby

StudentID	Hobby

Result
Multi-valued dependency removed
No redundancy due to independent attributes
✅ Tables are now in Fourth Normal Form (4NF)
.

Fifth Normal Form (5NF)

Fifth Normal Form (5NF) is also known as Project–Join Normal Form (PJNF). A relation is said to be in 5NF if it is already in Fourth Normal Form (4NF) and cannot be further decomposed into smaller relations without losing information. In other words, every join dependency in the relation is implied by the candidate keys.

In simple language, 5NF deals with complex join dependencies. Sometimes, even after removing multi-valued dependencies (4NF), a table may still contain redundancy that appears only when tables are joined. 5NF ensures that the relation is broken down into the smallest possible tables, and when these tables are joined back, they reconstruct the original table without any loss of data.

A relation is in Fifth Normal Form (5NF) (also called Project–Join Normal Form) if:

- It is in 4NF
- It cannot be decomposed further without losing information
- All join dependencies are implied by candidate keys

Checking 5NF for Our Example
Current tables:

- STUDENT
- COURSE
- INSTRUCTOR
- STUDENT_COURSE
- STUDENT_HOBBY

Each table:

- Represents a **single fact**
- Cannot be split further meaningfully

- Can be **joined back without data loss**

There are **no complex join dependencies** remaining.

4.2 De-normalization

De-normalization is a database design technique in which data redundancy is intentionally introduced into a database. It is the reverse process of normalization. While normalization divides data into multiple related tables to reduce redundancy, de-normalization combines tables or repeats data to improve the performance of data retrieval operations.

De-normalization is mainly used in situations where fast query performance is more important than minimizing storage space. By reducing the number of table joins required to retrieve data, de-normalization helps in speeding up read operations. However, since data is duplicated, it may lead to data inconsistency if updates are not handled carefully.

De-normalization is commonly applied in data warehouses, reporting systems, and large-scale applications, where complex queries are executed frequently. Although it increases redundancy, controlled de-normalization can significantly enhance system performance**.**

Example

After 4NF (Fully Normalized):

We had 4 separate tables:

1. Student–Course

StudentID	Course
1	DBMS
1	OS
2	DBMS
2	OS
3	DBMS

2. Student–Hobby

StudentID	Hobby
1	Cricket
1	Music
2	Dance
3	Cricke

3. Course–Instructor

Course	Instructor
DBMS	Sharma
DBMS	Verma
OS	Rao

4. Instructor

Instructor	Dept
Sharma	CS
Verma	CS
Rao	IT

✅ Data is clean, no redundancy — but if we want to **show all Student details with Hobbies and Instructor**, we need **multiple joins** (slow for big databases).

De-Normalized Table: We **merge tables** for performance:

StudentID	Name	Course	Instructor	Dept	Hobby
1	Ravi	DBMS	Sharma	CS	Cricket
1	Ravi	DBMS	Sharma	CS	Music
1	Ravi	OS	Rao	IT	Cricket
1	Ravi	OS	Rao	IT	Music
2	Kavya	DBMS	Sharma	CS	Dance
2	Kavya	OS	Rao	IT	Dance
3	Arjun	DBMS	Verma	CS	Cricket

Short Questions

1. Define an **Entity** in ER Model.
2. What is a **Weak Entity**? Give an example.
3. What are **Composite Attributes**?
4. Define **Derived Attribute** with an example.
5. What is a **Candidate Key**?
6. Differentiate between **1:1 and 1:N relationship**.
7. What is **Degree of a relationship**?
8. Define **Cardinality** in ER diagram.
9. What is **Generalization** in ER diagrams?
10. What is **Specialization**?
11. Define **Aggregation**.
12. Differentiate between **Aggregation and Composition**.
13. What is **Data Redundancy**?
14. Define **Functional Dependency** with an example.
15. What is **1NF**?
16. What is **BCNF**?
17. Define **De-Normalization**.

Long Questions

1. Explain the **components of an ER Diagram** with neat examples.
2. Discuss **Strong Entity vs Weak Entity** with examples.

3. Explain **types of attributes** with examples (Simple, Composite, Multivalued, Derived).
4. Describe the different **types of relationships (1:1, 1:N, M:N)** with examples and diagrams.
5. Explain the concepts of **Degree and Cardinality** in relationships with examples.
6. Explain **Generalization and Specialization** with suitable examples.
7. What is **Aggregation**? How is it different from **Composition**? Give examples with diagrams.
8. What is **Data Redundancy**? Explain with an example how normalization reduces it.
9. Explain the concept of **Functional Dependency** with examples.
10. Normalize the following table up to **3NF** with clear steps:
11. Discuss **1NF, 2NF, 3NF, and BCNF** with suitable examples
12. What is **De-Normalization**? Explain with an example.

Unit – III: SQL & PL/SQL

1.SQL Basics

1.1 SQL Data Types

SQL Data Types specify the **type of data** that can be stored in a column of a database table. When a table is created, each column is assigned a data type, which tells the database **what kind of values are allowed**, **how much space to allocate**, and **how the data should be processed**.

Using proper SQL data types is very important because they help in:

- Maintaining **data accuracy**
- Preventing **invalid data entry**
- Saving **storage space**
- Improving **database performance**

Classification of SQL Data Types

SQL data types are classified based on the **type of data** they store. The main classifications are:

1. ***Numeric Data Types***
2. ***Character / String Data Types***
3. ***Date and Time Data Types***
4. ***Boolean Data Type***
5. ***Binary Data Types***

These classifications help in **organizing data efficiently** and choosing the **appropriate data type** for each column in a database table.

1. Numeric Data Types

Numeric data types are used to store **numbers** in a database table. These numbers can be **whole numbers** or **decimal values**, depending on the type used. Choosing the correct numeric data type helps in **saving storage space**, **maintaining accuracy**, and **improving performance**.

Data Type	What it Stores	Size	Example Declaration	CREATE TABLE Syntax
INT / INTEGER	Whole numbers	4 bytes	Age INT	CREATE TABLE Sample_Data (Age INT);

SMALLINT	Small range whole numbers	2 bytes	Marks SMALLINT	CREATE TABLE Sample_Data (Marks SMALLINT);
BIGINT	Very large whole numbers	8 bytes	Population BIGINT	CREATE TABLE Sample_Data (Population BIGINT);
DECIMAL(p,s)	Exact decimal values	Depends on p	Salary DECIMAL(10,2)	CREATE TABLE Sample_Data (Salary DECIMAL(10,2));
NUMERIC(p,s)	Exact decimal values	Depends on p	Price NUMERIC(8,2)	CREATE TABLE Sample_Data (Price NUMERIC(8,2));
FLOAT	Approximate decimal values	4 bytes	Temperature FLOAT	CREATE TABLE Sample_Data (Temperature FLOAT);
DOUBLE	High-precision decimal values	8 bytes	Distance DOUBLE	CREATE TABLE Sample_Data (Distance DOUBLE);

2. Character/String Data Types

Character (String) data types are used to store textual data such as names, addresses, descriptions, and codes. These data types allow storing letters, numbers, and special characters. The choice of a character data type depends on whether the text length is fixed or variable.

Data Type	What it Stores	Size	Example Declaration	CREATE TABLE Syntax
CHAR(n)	Fixed-length character strings	n bytes	Gender CHAR(1)	CREATE TABLE Person (Gender CHAR(1));
VARCHAR(n)	Variable-length character strings	Up to n bytes	Name VARCHAR (50)	CREATE TABLE Student (Name VARCHAR(50));

TEXT	Long text or paragraphs	Large	Description TEXT	CREATE TABLE Article (Description TEXT);

3. Date and Time Data Types

Date and Time data types are used to store **date values, time values, or both date and time together** in a database. These data types help in recording information such as **date of birth, order date, login time, and timestamps** accurately.

Data Type	What it Stores	Format	Example Declaration	CREATE TABLE Syntax
DATE	Only date	YYYY-MM-DD	DOB DATE	CREATE TABLE Student (DOB DATE);
TIME	Only time	HH:MM:SS	LoginTime TIME	CREATE TABLE Login (LoginTime TIME);
DATETIME	Date and time	YYYY-MM-DD HH:MM:SS	OrderDate DATETIME	CREATE TABLE Orders (OrderDate DATETIME);
TIMESTAMP	Date and time with system tracking	YYYY-MM-DD HH:MM:SS	CreatedAt TIMESTAMP	CREATE TABLE Logs (CreatedAt TIMESTAMP);

4. Boolean Data Type

The **Boolean data type** is used to store **logical values** in a database. It can store only **two possible values**: **TRUE** or **FALSE**. Boolean values are mainly used to represent **conditions or status information**, such as whether an account is active or not.

Boolean Data Type Details

Data Type	What it Stores	Possible Values	Example Declaration	CREATE TABLE Syntax
BOOLEAN / BOOL	Logical values	TRUE or FALSE	IsActive BOOLEAN	CREATE TABLE Account (IsActive BOOLEAN);

5. Binary Data Types

Binary data types are used to store **binary (non-text) data** such as **images, audio files, videos, documents, and raw byte data**. Unlike character data types, binary data is stored in **byte format** and is not meant to be read directly as text.

Common Binary Data Types

Data Type	What it Stores	Size	Example Declaration	CREATE TABLE Syntax
BINARY(n)	Fixed-length binary data	n bytes	Code BINARY(4)	CREATE TABLE Sample (Code BINARY(4));
VARBINARY(n)	Variable-length binary data	Up to n bytes	FileData VARBINARY(100)	CREATE TABLE Files (FileData VARBINARY(100));
BLOB	Large binary objects (images, videos, files)	Very large	Photo BLOB	CREATE TABLE Employee (Photo BLOB);

1.2 Integrity Constraints in SQL

Integrity constraints in SQL are a set of rules that are used to maintain the correctness, consistency, and validity of data stored in a database. These constraints ensure that the data entered into a table follows certain conditions and does not violate the logical rules of the database. Integrity constraints restrict the values that can be inserted, updated, or deleted from a table, thereby protecting the database from invalid data.

Integrity constraints are defined at the time of table creation or can be added later. They help the database system enforce data integrity automatically, without relying only on application programs. By using integrity constraints, a database becomes more reliable, accurate, and easy to maintain.

1. NOT NULL Constraint

The NOT NULL constraint is an integrity constraint used in SQL to ensure that a column cannot store NULL values. This means that a value must be provided for the column whenever a new record is inserted into the table. The NOT NULL constraint is commonly applied to attributes that are mandatory and must always contain meaningful data.

Syntax:

CREATE TABLE Student (RollNo INT NOT NULL, Name VARCHAR(50) NOT NULL, Age INT);

Here, RollNo and Name must always contain values.

2. UNIQUE Constraint

The UNIQUE constraint is an integrity constraint in SQL that ensures all values in a column or a group of columns are distinct. This means that no two rows in a table can have the same value

for the column(s) defined with the UNIQUE constraint. It is used to prevent duplicate entries in attributes that must be unique but are not primary keys.

Unlike the PRIMARY KEY constraint, the UNIQUE constraint allows NULL values (depending on the DBMS, usually one NULL value is permitted). The UNIQUE constraint helps maintain data consistency by enforcing uniqueness where required by business rules.

Syntax:

CREATE TABLE Employee (EmpID INT UNIQUE, Email VARCHAR(100) UNIQUE, Name VARCHAR(50));

Both EmpID and Email must have unique values.

3. *PRIMARY KEY Constraint*

The **PRIMARY KEY constraint** is used in SQL to make sure that each record in a table can be **uniquely identified**. It does not allow **duplicate values** and also **does not allow NULL values** in the specified column. This constraint is commonly used for attributes like student ID, employee ID, or account number, where each value must be unique and always present.

The PRIMARY KEY constraint helps the database clearly identify each record and keep the data accurate and reliable. If someone tries to insert a duplicate value or a NULL value into a primary key column, the database will not allow it. Unlike the UNIQUE constraint, a table can have **only one primary key**, and it is mandatory for identifying records in a table.

- Combination of **NOT NULL + UNIQUE**.

Syntax:

CREATE TABLE Department (DeptID INT PRIMARY KEY, DeptName VARCHAR(50) NOT NULL);

DeptID uniquely identifies each department.

Composite Primary Key Example:

CREATE TABLE StudentCourse (StudentID INT, CourseID INT, PRIMARY KEY (StudentID, CourseID);

4. *FOREIGN KEY Constraint*

The **FOREIGN KEY constraint** is used in SQL to **create a relationship between two tables**. It ensures that the value stored in one table **must already exist in another table**. In simple words, a foreign key helps maintain a **link between related data** and keeps the database consistent.

The FOREIGN KEY constraint is usually defined in a **child table**, and it refers to the **PRIMARY KEY** of a **parent table**. This means that the database will not allow a value in the

foreign key column if that value does not exist in the referenced primary key column. By doing this, the foreign key prevents invalid or unrelated data from being entered.

- Establishes a **relationship between two tables**.
- The foreign key in one table refers to the **primary key in another**.

Syntax:

CREATE TABLE Student (RollNo INT PRIMARY KEY, Name VARCHAR(50));

Example:

CREATE TABLE Exam (ExamID INT PRIMARY KEY, RollNo INT, FOREIGN KEY (RollNo) REFERENCES Student(RollNo));

Ensures only valid RollNo values from Student are allowed in Exam.

5. *CHECK Constraint*

The **CHECK constraint** is used in SQL to **restrict the values** that can be stored in a column based on a specific condition. It ensures that the data entered into a table **follows certain rules** defined by the user. If the condition is not satisfied, the database does not allow the value to be inserted or updated.

In simple words, the CHECK constraint acts like a **filter** for data. It checks whether a value is valid before storing it in the table. This is useful for attributes such as age, salary, or marks, where values must fall within a certain range or meet specific criteria.

- Ensures that values in a column satisfy a **condition**.

Syntax:

CREATE TABLE Product (ProductID INT PRIMARY KEY, Price DECIMAL(8,2) CHECK (Price > 0), Quantity INT CHECK (Quantity >= 1));

Prevents negative or zero price and quantity.

6. *DEFAULT Constraint*

The **DEFAULT constraint** is used in SQL to **automatically assign a value** to a column when no value is provided during data insertion. It helps ensure that a column always has a meaningful value, even if the user does not explicitly enter one.

In simple words, the DEFAULT constraint acts like a **pre-set value**. If a value is not given for a column, the database uses the default value instead. This is useful for attributes such as status, country, or joining date, where a common value is often used.

- Provides a **default value** if no value is supplied for a column.

Syntax:

CREATE TABLE Customer (CustID INT PRIMARY KEY,Name VARCHAR(50),City VARCHAR(50) DEFAULT 'Hyderabad');

1. 3 NULL Values

In SQL, **NULL** represents **missing, unknown, or unavailable data**. It indicates that **no value exists** for a particular attribute. A NULL value is different from both **zero (0)** and an **empty string ('')**, as each has a distinct meaning in a database.

The value **0** is a valid numeric value, and **'' (empty string)** is a valid string with no characters. However, **NULL means that the value is not known or not provided at all**. Because NULL does not represent an actual value, it cannot be compared using normal comparison operators such as = or !=.

To check for NULL values in SQL, special operators **IS NULL** and **IS NOT NULL** are used.

Key Points about NULL Values

- NULL represents **missing or unknown data**
- NULL is **not equal to 0**
- NULL is **not equal to an empty string ('')**
- NULL means **"no value exists"**
- The = operator **cannot be used** to compare NULL values
- Use **IS NULL** or **IS NOT NULL** to check for NULL

Example:

-- Create table

CREATE TABLE Employee (emp_id INT,emp_name VARCHAR(50),salary INT);

-- Insert values

INSERT INTO Employee VALUES (1, 'Ravi', 25000);
INSERT INTO Employee VALUES (2, 'Sita', NULL);
INSERT INTO Employee VALUES (3, 'Kiran', 0);

-- Query to find NULL salaries

SELECT emp_name, salary FROM EmployeeWHERE salary IS NULL;

1.4 Clauses

In DBMS, clauses are the building blocks of an SQL query that tell the database what operation to perform on the data. Each clause has a specific role, such as choosing the required columns, selecting data from tables, applying conditions, grouping records, or sorting the output. When these clauses are used together in a proper order, they form a complete SQL statement. Clauses help users to retrieve accurate and meaningful information from the database in a clear and systematic manner, making data handling easier and more efficient.

❖ ***ORDER BY***

- ***GROUP BY***
- ***HAVING***

❖ *ORDER BY Clause:*

The **ORDER BY** clause in SQL is used to **sort the result set** of a query in a specific order. It arranges the rows based on the values of one or more columns. By default, ORDER BY sorts data in **ascending order (ASC)**. If required, the sorting order can be changed to **descending order (DESC)**.

ORDER BY is commonly used when we want data to appear in a **meaningful sequence**, such as sorting names alphabetically, salaries from highest to lowest, or dates in chronological order. It does not change the data stored in the table; it only affects the **display of query results**.

- By default, sorting is **ascending (ASC)**.
- To sort in **descending order**, use DESC.
- Multiple columns can be specified; sorting is applied sequentially from left to right.

Example:

Suppose a table Employee(emp_id, emp_name, salary) exists:

emp_id	emp_name	salary
1	Ravi	25000
2	Sita	30000
3	Kiran	20000

SELECT emp_id, emp_name, salary FROM Employee ORDER BY salary DESC;

Result:

emp_id	emp_name	salary
2	Sita	30000
1	Ravi	25000
3	Kiran	20000

- Employees are sorted from **highest to lowest salary**.

  `ORDER BY` → Sorts query results in ascending or descending order based on one or more columns.

❖ *GROUP BY Clause:*

The **GROUP BY** clause in SQL is used to **group rows that have the same values** in one or more specified columns. It is mainly used along with **aggregate functions** such as **COUNT, SUM, AVG, MAX, and MIN** to perform calculations on each group rather than on individual rows.

The GROUP BY clause **collects similar records into groups** and then applies an aggregate function to each group. It is commonly used to obtain **summary information**, such as the total salary of employees in each department or the number of employees working in each department.

- GROUP BY groups rows that have **identical values** in the specified column(s).
- It is usually combined with **aggregate functions** like SUM(), COUNT(), and AVG().
- Each group produces **one result row** for each unique value in the grouped column(s).

Example:

Suppose a table Employee(dept_id, emp_name, salary):

dept_id	emp_name	salary
1	Ravi	25000
1	Sita	30000
2	Kiran	20000

SELECT dept_id, SUM(salary) AS total_salary FROM Employee GROUP BY dept_id;

Result:

dept_id	total_salary
1	55000
2	20000

- Employees are grouped by `dept_id`.
- `SUM(salary)` calculates **total salary per department**.

`GROUP BY` → Groups rows to perform aggregate calculations on each group.

❖ HAVING Clause

The HAVING clause in SQL is used to filter groups of rows after they have been grouped using the GROUP BY clause. It is mainly used with aggregate functions such as COUNT, SUM, AVG, MAX, and MIN. While the WHERE clause filters individual rows, the HAVING clause filters the results of grouped data.

In simple terms, the HAVING clause is applied after grouping to decide which groups should be included in the final output. It is commonly used when we want conditions on aggregated values, such as displaying only those departments where the total salary is greater than a certain amount or showing only groups with more than a specific number of records.

- **WHERE** filters rows **before aggregation**.
- **HAVING** filters groups **after aggregation**.
- Used with `GROUP BY` to filter **grouped results** based on conditions.

Example:
SELECT dept_id, SUM(salary) AS total_salary FROM Employee GROUP BY dept_id HAVING SUM(salary) > 50000;Result (using previous table):

dept_id	total_salary
1	55000

Note: HAVING → Filters aggregated groups; WHERE → Filters rows before aggregation.

1.5 Aggregate Functions

Aggregate functions in SQL are used to **perform calculations on a set of rows** and return **a single summarized result**. These functions are commonly used along with the **GROUP BY** clause to obtain summary information such as totals, averages, counts, maximum values, and minimum values from a database table.

Aggregate functions are useful for **data analysis and reporting**, as they allow meaningful information to be derived from large volumes of data. For instance, they can be used to calculate the total salary of employees, determine the average marks obtained by students, or count the number of records present in a table.

1. **SUM()** – Adds up numeric values.
2. **AVG()** – Calculates average.
3. **COUNT()** – Counts rows.
4. **MAX() / MIN()** – Finds maximum/minimum value.

Example Table for Aggregate Functions

Consider the following **EMPLOYEE** table:

EmpID	EmpName	Department	Salary
101	Ravi	IT	30000
102	Sita	HR	25000
103	Kiran	IT	40000
104	Meena	HR	28000
105	Arjun	IT	35000

Common Aggregate Functions

Aggregate Function	Description	General Syntax	Example Query	Output
COUNT()	Returns the total	SELECT	SELECT	5

	number of rows	COUNT(column) FROM table_name;	COUNT(*) FROM Employee;	
SUM()	Returns the total of a numeric column	SELECT SUM(column) FROM table_name;	SELECT SUM(salary) FROM Employee;	158000
AVG()	Returns the average value of a numeric column	SELECT AVG(column) FROM table_name;	SELECT AVG(marks) FROM Student;	72.5
MAX()	Returns the maximum value in a column	SELECT MAX(column) FROM table_name;	SELECT MAX(salary) FROM Employee;	40000
MIN()	Returns the minimum value in a column	SELECT MIN(column) FROM table_name;	SELECT MIN(marks) FROM Student;	45

1.6 Logical Operators.

Logical operators in SQL are used to **combine multiple conditions** in a WHERE clause. They help in filtering records based on **more than one condition**. Logical operators return a result based on whether the given conditions are **true or false**.

Consider the following **EMPLOYEE** table:

EmpID	EmpName	Department	Salary
101	Ravi	IT	30000
102	Sita	HR	25000
103	Kiran	IT	40000
104	Meena	HR	28000
105	Arjun	IT	35000

Logical Operators

Logical Operator	Condition Used	Example Query	Output
AND	Both conditions must be true	SELECT EmpName FROM Employee WHERE Department='IT' AND Salary > 30000;	Kiran, Arjun
OR	Any one condition must be true	SELECT EmpName FROM Employee WHERE Department='HR' OR Salary > 35000;	Sita, Meena, Kiran
NOT	Negates the condition	SELECT EmpName FROM Employee WHERE NOT Department='IT';	Sita, Meena

Special Operators

Special operators in SQL are used to perform specific types of comparisons in queries. They are mainly used in the WHERE clause to filter data based on patterns, ranges, lists, NULL values, or subquery results. These operators make SQL queries more flexible and powerful.

Types of Special Operators in SQL

Special Operator	Purpose	Example Query	Output
IN	Checks whether a value matches **any value in a list**	SELECT EmpName FROM Employee WHERE Department IN ('IT','HR');	Ravi, Sita, Kiran, Meena, Arjun
NOT IN	Checks whether a value does **not match any value in a list**	SELECT EmpName FROM Employee WHERE Department NOT IN ('IT');	Sita, Meena
BETWEEN	Checks whether a value lies **within a range**	SELECT EmpName FROM Employee WHERE Salary BETWEEN 25000 AND 35000;	Ravi, Sita, Meena, Arjun
NOT BETWEEN	Checks whether a value lies **outside a range**	SELECT EmpName FROM Employee WHERE Salary NOT BETWEEN 25000 AND 35000;	Kiran
LIKE	Searches for a **pattern** in a column	SELECT EmpName FROM Employee WHERE EmpName LIKE 'A%';	Arjun
IS NULL	Checks for **NULL values**	SELECT EmpName FROM Employee WHERE Salary IS NULL;	No rows
IS NOT NULL	Checks for **non-NULL values**	SELECT EmpName FROM Employee WHERE Salary IS NOT NULL;	Ravi, Sita, Kiran, Meena, Arjun
EXISTS	Checks whether a **subquery returns any rows**	SELECT EmpName FROM Employee E WHERE EXISTS (SELECT * FROM Employee WHERE Salary > 38000);	Ravi, Sita, Kiran, Meena, Arjun

1.7 SQL Set Operators

SQL Set Operators are used to **combine the results of two or more SELECT queries** into a single result set. These operators work in a similar way to **set operations in mathematics**. The combined SELECT statements must have the **same number of columns**, **compatible data types**, and the **same column order**.

Set operators are mainly used when we want to **compare or merge data** from multiple queries.

- All queries must select the **same number of columns**.
- Data types of corresponding columns must be **compatible**.

Table 1: PROJECT_A_EMP

EmpID	EmpName	Department	Project
101	Ravi	IT	Project_A
102	Sita	HR	Project_A
103	Kiran	IT	Project_A
104	Meena	HR	Project_A

Table 2: PROJECT_B_EMP

EmpID	EmpName	Department	Project
103	Kiran	IT	Project_B
104	Meena	HR	Project_B
105	Arjun	IT	Project_B
106	Rahul	Finance	Project_B

SQL Set Operators –

Set Operator	Definition	Syntax	Example Query	Output
UNION	Combines results of two SELECT queries and removes duplicate rows	SELECT cols FROM T1 UNION SELECT cols FROM T2;	SELECT EmpName FROM PROJECT_A_EMP UNION SELECT EmpName FROM PROJECT_B_EMP;	Ravi, Sita, Kiran, Meena, Arjun, Rahul
UNION ALL	Combines results of two SELECT queries and includes duplicate rows	SELECT cols FROM T1 UNION ALL SELECT cols FROM T2;	SELECT EmpName FROM PROJECT_A_EMP UNION ALL SELECT EmpName FROM PROJECT_B_EMP;	Ravi, Sita, Kiran, Meena, Kiran, Meena, Arjun, Rahul
INTERSECT	Returns only the rows that are common in both SELECT queries	SELECT cols FROM T1 INTERSECT SELECT cols FROM T2;	SELECT EmpName FROM PROJECT_A_EMP INTERSECT SELECT EmpName FROM PROJECT_B_EMP;	Kiran, Meena
EXCEPT / MINUS	Returns rows from the first SELECT query that are not in	SELECT cols FROM T1 EXCEPT SELECT cols	SELECT EmpName FROM PROJECT_A_EMP EXCEPT SELECT EmpName FROM	Ravi, Sita

	the second	FROM T2;	PROJECT_B_EMP;	

Difference between UNION and UNION ALL

Aspect	UNION	UNION ALL
Definition	Combines the result of two SELECT queries and **removes duplicate rows**	Combines the result of two SELECT queries and **keeps duplicate rows**

1.8 SQL Joins

SQL Joins are used to **combine data from two or more tables** based on a **related column** between them. Joins help retrieve meaningful information that is spread across multiple tables in a database.

Why Joins Are Needed

- Data is stored in **multiple related tables**
- Joins allow us to **view combined data** in a single result
- They help avoid data redundancy and improve database design

Given Tables

STUDENT

RollNo	Name	CourseID	CourseName	Department	Year	City
1	Aditi	C101	DBMS	CSE	2	Hyderabad
2	Rohan	C102	OS	CSE	3	Pune
3	Neha	C103	CN	ECE	2	Chennai

COURSE

CourseID	CourseName	Credits	Duration	Department	Instructor
C101	DBMS	4	1 Semester	CSE	Dr. Sharma
C102	OS	3	1 Semester	CSE	Dr. Rao
C104	Networks	4	1 Semester	ECE	Dr. Verma

1. INNER JOIN

An INNER JOIN in SQL is used to retrieve only those records that have matching values in both tables. If a row in one table does not have a corresponding matching row in the other table, it is not included in the result.

INNER JOIN Example

SELECT RollNo, Name, CourseName, Instructor FROM Student INNER JOIN Course ON Student. CourseID = Course. CourseID;

Output

RollNo	Name	CourseName	Instructor
1	Aditi	DBMS	Dr. Sharma
2	Rohan	OS	Dr. Rao

- *The join condition is **Student.CourseID = Course.CourseID***
- *Only **C101 and C102** are common in both tables*
- ***Neha (C103)** is excluded because there is no matching course*
- ***Networks (C104)** is excluded because no student is enrolled*

2. LEFT OUTER JOIN

A LEFT OUTER JOIN (or LEFT JOIN) in SQL is used to retrieve all records from the left table and the matching records from the right table. If there is no matching record in the right table, the result will contain NULL values for the columns of the right table.

- Returns **all rows from left table** (Student), and matching rows from right (Course).
- *Non-matching rows → NULL.*

LEFT OUTER JOIN Syntax

```
SELECT column_list FROM table1 LEFT OUTER JOIN table2
ON table1.common_column = table2.common_column;
```

LEFT OUTER JOIN Example

```
SELECT RollNo, Name, CourseName, Instructor FROM Student LEFT OUTER JOIN Course
ON Student.CourseID = Course.CourseID;
```

Output

RollNo	Name	CourseName	Instructor
1	Aditi	DBMS	Dr. Sharma
2	Rohan	OS	Dr. Rao
3	Neha	CN	NULL

- *All records from the **STUDENT table** are included*
- *Matching course details are shown where available*
- ***Neha (C103)** has no matching course in COURSE table*
- *Hence, **Instructor value is NULL** for Neha*

3. RIGHT OUTER JOIN

A **RIGHT OUTER JOIN** (or **RIGHT JOIN**) in SQL is used to **retrieve all records from the right table** and the **matching records from the left table**. If there is **no matching record** in the left table, the result will contain **NULL values** for the columns of the left table

- Returns **all rows from right table** (Course), and matching rows from left (Student).

RIGHT OUTER JOIN Syntax

```
SELECT column_list
FROM table1
RIGHT OUTER JOIN table2
ON table1.common_column = table2.common_column;
```

RIGHT OUTER JOIN Example

SELECT RollNo, Name, CourseName, Instructor FROM Student RIGHT OUTER JOIN Course ON Student.CourseID = Course.CourseID;

Output

RollNo	Name	CourseName	Instructor
1	Aditi	DBMS	Dr. Sharma
2	Rohan	OS	Dr. Rao
NULL	NULL	Networks	Dr. Verma

- *All records from the COURSE table are included*
- *Matching student details are shown where available*
- *Networks (C104) has no student enrolled*
- *Hence, student details are shown as NULL*

4. FULL OUTER JOIN

A **FULL OUTER JOIN** in SQL is used to **retrieve all records from both tables**. If a record in one table does not have a matching record in the other table, the result will contain **NULL values** for the missing fields.

In simple words, a FULL OUTER JOIN is a **combination of LEFT OUTER JOIN and RIGHT OUTER JOIN**.

FULL OUTER JOIN Syntax

```
SELECT column_list FROM table1 FULL OUTER JOIN table2
ON table1.common_column = table2.common_column;
```

FULL OUTER JOIN Example

SELECT RollNo, Name, CourseName, Instructor FROM Student FULL OUTER JOIN Course ON Student.CourseID = Course.CourseID;

Output

RollNo	Name	CourseName	Instructor
1	Aditi	DBMS	Dr. Sharma
2	Rohan	OS	Dr. Rao
3	Neha	CN	NULL
NULL	NULL	Networks	Dr. Verma

- *All records from both STUDENT and COURSE tables are displayed*
- *Matching records show complete details*
- *Neha (C103) has no matching course → Instructor is NULL*
- *Networks (C104) has no enrolled student → Student details are NULL*

5. CROSS JOIN

A **CROSS JOIN** in SQL returns the **Cartesian product** of two tables.

This means **each row of the first table is combined with every row of the second table**.

CROSS JOIN does **not use a join condition**.

- If table A has *m* rows and table B has *n* rows, the result will have **m × n rows**.

Given Tables

STUDENT

RollNo	Name	CourseID

COURSE

CourseID	CourseName
C101	DBMS
C102	OS
C104	Networks

CROSS JOIN Syntax

```
SELECT column_list
FROM table1
CROSS JOIN table2;
```

CROSS JOIN Example

```
SELECT Name, CourseName
FROM Student
```

OU

Adi	
Adi	
Aditi	Networks
Rohan	DBMS
Rohan	OS
Rohan	Networks
Neha	DBMS
Neha	OS
Neha	Networks

1.9 Subqueries

Nested Sub query: A **Nested Subquery** is a **query written inside another SQL query**.The **inner query** (subquery) is executed first, and its result is then used by the **outer query** to produce the final output.

Nested subqueries are mainly used when the result of one query **depends on the result of another query**.

- The **inner query executes first**.
- The **outer query executes after** the result of the inner query is returned.
- Inner query result is **independent** of outer query.

General Syntax

```
SELECT column_nameFROM table_nameWHERE column_name operator
 (
   SELECT column_name
   FROM table_name
   WHERE condition
);
```

Example EMPLOYEE Table

EmpID	EmpName	Department	Salary
101	Ravi	IT	30000
102	Sita	HR	25000
103	Kiran	IT	40000
104	Meena	HR	28000
105	Arjun	IT	35000

Nested Subquery Using Comparison Operator

Find the names of employees whose salary is greater than the average salary.

Query

```
SELECT EmpName
FROM EmployeeWHERE Salary >
 (SELECT AVG(Salary)
   FROM Employee
);
```

Output

EmpName
Kiran
Arjun

The inner query calculates the average salary

The outer query selects employees whose salary is greater than that average

Correlated Subquery: A Correlated Subquery is a type of subquery that depends on the outer query for its values.Unlike a simple nested subquery, a correlated subquery cannot run independently because it uses columns from the outer query.
The correlated subquery is executed once for each row processed by the outer query.

General Syntax

```
SELECT column_name FROM table1 outer WHERE condition operator (
   SELECT column_name
   FROM table2 inner
   WHERE inner.column = outer.column
);
```

Example: Correlated Subquery

Find employees who earn **more than the average salary of their own department**.

Query

```
SELECT EmpName, Department, Salary FROM Employee E WHERE Salary > (
   SELECT AVG(Salary)
   FROM Employee
   WHERE Department = E.Department
);
```

- *The **outer query** selects employees one by one*
- *For each employee, the **subquery calculates the average salary of that employee's department***
- *The employee is selected only if their salary is **greater than their department's average***

Output

EmpName	Department	Salary
Kiran	IT	40000
Arjun	IT	35000

✔ Inner query runs **row-by-row**.

2. Views in SQL

A View in SQL is a logical or virtual table that is created based on the result of a SELECT query. Unlike a regular table, a view does not store data physically in the database. Instead, it stores only the query definition. Whenever a view is accessed, the database executes the stored query and displays the current data from the underlying base table(s).

Views act as a window to the data, allowing users to see only the required information. They are mainly used to simplify complex queries, improve data security, and provide customized representations of data without changing the actual database structure.

Characteristics of a View

- A view is created using one or more tables.
- It behaves like a table when queried.
- It does not occupy separate storage space for data.
- Data shown in a view always reflects the latest changes in base tables.
- A view can be created from another view.

Why Views Are Required

In real-world databases, tables often contain a large number of columns and complex relationships. Writing the same long queries repeatedly is difficult and error-prone. Views solve this problem by allowing frequently used queries to be stored and reused easily.

Views are also important for security purposes. Instead of giving users direct access to base tables, a view can be created to expose only selected columns or rows. This ensures that sensitive information remains hidden.

2.1 Creating a View in DBMS : is the process of defining a virtual table based on the result of an SQL query. A view does not store data physically; instead, it stores the query definition and displays data from one or more tables whenever it is accessed. Views are created to simplify complex queries, improve security by restricting access to specific columns or rows, and provide a customized representation of data for different users. Once created, a view can be used like a regular table for data retrieval. Creating views helps in better database organization, easier data access, and controlled data exposure.

Syntax for Creating a View

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

Example: Consider an EMPLOYEE table containing employee details such as ID, name, department, and salary. If we want to display only the details of employees working in the IT department, a view can be created.

Source Table: EMPLOYEE

EmpID	EmpName	Department	Salary
101	Ravi	IT	30000
102	Sita	HR	25000
103	Kiran	IT	40000
104	Meena	HR	28000

105	Arjun	IT	35000

*This **EMPLOYEE table** is the **source table** because it physically stores employee data.*

CREATE VIEW IT_Employees ASSELECT EmpID, EmpName, Salary FROM Employee WHERE Department = 'IT';

View: IT_Employees (Result Data)

EmpID	EmpName	Salary
101	Ravi	30000
103	Kiran	40000
105	Arjun	35000

- *The view IT_Employees is created using the EMPLOYEE source table*
- *It displays only IT department employees*
- *The view does not store data*
- Any change in the EMPLOYEE table will be reflected in the view automatically

2.2 Dropping a View

A **view** can be removed from the database using the **DROP VIEW** command.Dropping a view means **deleting only the view definition**, not the data stored in the source (base) table.When a view is dropped, it no longer exists in the database and cannot be used for queries. However, the **underlying tables remain unchanged**, and all data in those tables is safe.

Syntax

DROP VIEW view_name;

Example

DROP VIEW IT_Employees;

- *The view **IT_Employees** is permanently removed.*
- *The **EMPLOYEE** source table is **not affected**.*
- *No data is deleted from the base table.*
- *Only the virtual structure (view) is removed.*

Limitations of Views

- Views do not store data
- Views do not improve performance for complex queries.
- Updating data through views is restricted.
- Views with joins, group functions, or subqueries are usually read-only.

- Excessive use of views may make debugging difficult.

2.3 Updatable Views

An **Updatable View** is a view through which data modification operations can be performed. Any change made through an updatable view is directly reflected in the **source (base) table**.

Characteristics of Updatable Views

- Created from **a single base table**
- Does **not use aggregate functions**
- Does **not contain GROUP BY, HAVING, DISTINCT, or JOIN**
- Allows **INSERT, UPDATE, and DELETE** operations

Example

CREATE VIEW IT_Employees AS SELECT EmpID, EmpName, Salary FROM Employee WHERE Department = 'IT';

This view is updatable because it is created from one table and contains no complex clauses.

2.4 Non-Updatable Views

A **Non-Updatable View** is a view through which data modification operations are **not allowed**. Such views are generally used only for **data display or reporting purposes**.

Characteristics of Non-Updatable Views

- Created using **multiple tables**
- Contains **JOINs, GROUP BY, HAVING, DISTINCT, or aggregate functions**
- Does **not allow INSERT, UPDATE, or DELETE**
- Used mainly for **summary and analytical queries**

Example

CREATE VIEW Dept_Avg_Salary AS SELECT Department, AVG(Salary) AS AvgSalary FROM Employee GROUP BY Department;

This view is non-updatable because it uses an aggregate function and GROUP BY clause.

3.PL/SQL

3.1 Introduction

PL/SQL (Procedural Language / Structured Query Language) is a block-structured programming language developed by Oracle. It extends the capabilities of SQL by adding procedural

programming features such as variables, conditions, loops, and exception handling. PL/SQL is mainly used to write programs that interact efficiently with the Oracle database.

PL/SQL allows multiple SQL statements to be grouped and executed as a single program unit. This improves performance, security, and manageability of database applications. Unlike SQL, which executes one statement at a time, PL/SQL supports logical control structures, making it suitable for writing complex database logic.

Need for PL/SQL

PL/SQL was developed to overcome the limitations of SQL, such as:

- SQL does not support procedural control structures like IF and LOOP.
- SQL does not provide exception handling.
- Writing modular and reusable programs is difficult using SQL alone.

By combining SQL with procedural features, PL/SQL provides a complete programming environment within the database.

Key Characteristics of PL/SQL

- PL/SQL is Oracle's procedural extension of SQL.
- It combines:
  - SQL for data manipulation
  - Procedural constructs for program control
- It is tightly integrated with the Oracle database engine.

Features of PL/SQL

- PL/SQL is a block-structured language that organizes programs into logical sections, making them easy to read and maintain.
- It supports procedural constructs such as IF, CASE, LOOP, FOR, and WHILE for decision making and iteration.
- PL/SQL provides exception handling to manage runtime errors using predefined and user-defined exceptions.
- SQL statements like SELECT, INSERT, UPDATE, DELETE, and DDL commands can be used directly inside PL/SQL blocks.
- It supports modular programming through procedures, functions, and packages, which helps in code reuse.
- PL/SQL improves performance by executing multiple SQL statements together as a single block.
- It enhances security by storing business logic inside the database rather than in client applications.
- PL/SQL programs run inside the Oracle database engine, making them portable across Oracle platforms.

3.2 Structure of a PL/SQL Block

A PL/SQL block is the basic unit of a PL/SQL program. It defines how a PL/SQL program is organized and executed. A PL/SQL block is divided into three main parts, out of which only the BEGIN–END part is mandatory.

Structure of a PL/SQL Block

```
DECLARE
  -- Declaration section
BEGIN
  -- Executable section
EXCEPTION
  -- Exception handling section
END;
```

DECLARE Section

The **DECLARE section** is the first part of a PL/SQL block and is used to define all the objects required for program execution. In this section, **variables, constants, cursors, and user-defined exceptions** are declared. These declarations allow the program to store data temporarily, control execution, and handle special conditions. The DECLARE section is **optional** and is included only when such definitions are needed. If a PL/SQL block does not require any variables or declarations, this section can be omitted.

Example:

```
DECLARE
  v_salary NUMBER;
  v_name   VARCHAR2(20);
BEGIN
  NULL;
END;
```

BEGIN Section

The **BEGIN section** is the **core of the PL/SQL block** where the actual program logic is written. It contains **executable statements**, including SQL commands (SELECT, INSERT, UPDATE, DELETE) and PL/SQL statements such as assignments, conditional statements (IF, CASE), and

loops (FOR, WHILE). This section controls the flow of execution and performs the required operations. The BEGIN section is **mandatory** in every PL/SQL block because, without it, no execution can take place.

Example:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Welcome to PL/SQL');
END;
```

EXCEPTION Section

The **EXCEPTION section** is used to handle **runtime errors** that may occur during the execution of the BEGIN section. It allows the program to respond gracefully to errors instead of terminating abruptly. This section can handle both **predefined exceptions** (such as NO_DATA_FOUND, TOO_MANY_ROWS) and **user-defined exceptions**. Proper exception handling improves program reliability and makes debugging easier. The EXCEPTION section is **optional** and is written only when error handling is required.

Example:

```
DECLARE
  v_result NUMBER;
BEGIN
  v_result := 10 / 0;
  DBMS_OUTPUT.PUT_LINE(v_result);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Division by zero error');
END;
```

END Statement

The **END statement** marks the **termination of the PL/SQL block**. It signals the Oracle engine that the block definition is complete. Every PL/SQL block must end with the END keyword followed by a **semicolon (;)**. Without the END statement, the PL/SQL block is considered incomplete and will result in a compilation error.

Example:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('PL/SQL block completed');
END;
```

3.3 Elements of Pl/SQL

1. Variables

Variables are used in PL/SQL to store temporary data during the execution of a program. The value of a variable can change as the program runs. Variables must be declared before they are used, and each variable must have a data type that defines what kind of data it can store. Variables help in calculations, decision making, and storing intermediate results.

Types of Variables:

- Scalar variables (store single values)
- Composite variables (store multiple related values)
- Reference variables (store references to data)

Example:

```
DECLARE
  v_salary NUMBER;
BEGIN
  v_salary := 30000;
  DBMS_OUTPUT.PUT_LINE(v_salary);
END;
```

2. Constants

Constants are similar to variables, but their values cannot be changed once they are assigned. They are used to store fixed values such as tax rates, limits, or mathematical constants. Declaring constants improves program safety and readability by preventing accidental changes to important values.

- Declared using the CONSTANT keyword
- Value must be assigned at declaration time
- Cannot be modified later

Example:

```
DECLARE
  c_tax CONSTANT NUMBER := 10;
BEGIN
  DBMS_OUTPUT.PUT_LINE(c_tax);
END;
```

3. Data Types

Data types define the type of data that a variable can store, such as numbers, characters, or dates. Choosing the correct data type is important for efficient memory usage and accurate data handling. PL/SQL supports a wide range of data types to handle different kinds of data.
Categories of Data Types:

- ***Scalar data types***

- ***Composite data types***
- ***Reference data types***
- ***LOB data types***

Example:

```
DECLARE
  v_name VARCHAR2(20);
  v_dob  DATE;
BEGIN
  v_name := 'Ravi';
  v_dob := SYSDATE;
END;
```

4. Operators

Operators are symbols used to perform operations on variables and values. They are used for arithmetic calculations, comparisons, and logical conditions. Operators help in decision making and data manipulation within PL/SQL programs.

Types of Operators:

- Arithmetic operators (+, -, *, /)
- Comparison operators (=, <, >, <=, >=)
- Logical operators (AND, OR, NOT)

Example:

```
DECLARE
  v_total NUMBER;
BEGIN
  v_total := 20000 + 5000;
  DBMS_OUTPUT.PUT_LINE(v_total);
END;
```

5. Control Statements

Control statements manage the flow of execution in a PL/SQL program. They allow the program to make decisions and repeat actions based on conditions. Control statements make PL/SQL a true procedural language.

Types of Control Statements:

- Conditional statements (IF, CASE)
- Looping statements (FOR, WHILE, LOOP)

Example:

```
DECLARE
```

```
  v_marks NUMBER := 70;
BEGIN
  IF v_marks >= 50 THEN
    DBMS_OUTPUT.PUT_LINE('Pass');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Fail');
  END IF;
END;
```

6. Cursors

Cursors are used to handle query results that return more than one row. They allow PL/SQL programs to process rows one at a time. Cursors are especially useful when performing row-by-row operations.

Types of Cursors:

- Implicit cursors
- Explicit cursors
- REF cursors

Example:

```
DECLARE
  CURSOR emp_cur IS SELECT EmpName FROM Employee;
  v_name Employee.EmpName%TYPE;
BEGIN
  OPEN emp_cur;
  FETCH emp_cur INTO v_name;
  CLOSE emp_cur;
END;
```

7. Exceptions

Exceptions are used to handle runtime errors that occur during the execution of a PL/SQL program. Instead of stopping the program abruptly, exception handling allows the program to respond gracefully to errors and continue execution if required.

***Types of Exceptions**:*

- ***Predefined exceptions***
- ***User-defined exceptions***
- ***OTHERS exception***

Example:

```
DECLARE
  v_num NUMBER;
```

```
BEGIN
  v_num := 10 / 0;
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Division by zero error');
END;
```

3.4 PL/SQL Data Types

PL/SQL data types specify the **kind of values** that variables, constants, parameters, and database objects can hold. Every variable in PL/SQL must be declared with a data type before it is used. The data type determines **how much memory is allocated**, **what operations are allowed**, and **how the value is processed** by the Oracle database engine.

Using appropriate data types improves **data accuracy, performance, and program reliability**. PL/SQL supports both SQL data types and its own additional data types to handle complex programming requirements.

PL/SQL data types are broadly classified into **four major categories**.

1. Scalar Data Types

Scalar data types store **only one value at a time**. These are the most commonly used data types in PL/SQL and are mainly used for simple data storage such as numbers, characters, and dates.

- Hold a single atomic value
- Easy to use and efficient
- Commonly used in calculations and conditions

Common Scalar Data Types

- ***NUMBER*** *– stores numeric values (integers and decimals)*
- ***VARCHAR2*** *– stores variable-length character strings*
- ***CHAR*** *– stores fixed-length character strings*
- ***DATE*** *– stores date and time values*
- ***BOOLEAN*** *– stores TRUE, FALSE, or NULL (PL/SQL only)*

Example

```
DECLARE
  v_id     NUMBER := 101;
  v_name   VARCHAR2(30) := 'Ravi';
  v_dob    DATE := SYSDATE;
  v_status BOOLEAN := TRUE;
```

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(v_name);
END;
```

2. Composite Data Types

Composite data types store **multiple related values together** as a single unit. They are useful when dealing with structured data such as rows of a table or a group of related variables.

- Can store multiple values of different data types
- Improve code organization
- Useful for handling complex data structures

Types of Composite Data Types

- ***RECORD*** *– represents a single row with multiple fields*
- ***COLLECTIONS***
  - Associative Arrays
  - Nested Tables
  - VARRAYs

Example (RECORD)

```
DECLARE
  TYPE emp_rec IS RECORD (
    emp_id   NUMBER,
    emp_name VARCHAR2(20),
    salary   NUMBER
  );
  emp emp_rec;
BEGIN
  emp.emp_id := 101;
  emp.emp_name := 'Ravi';
  emp.salary := 30000;
END;
```

3. Reference Data Types

Reference data types do not store actual values. Instead, they **refer to the data type of a table column or a database row**. These data types help maintain consistency between PL/SQL variables and database table structures.

- Automatically adapt to table structure changes
- Reduce maintenance effort
- Increase program reliability

Types of Reference Data Types

- ***%TYPE*** *– refers to the data type of a table column*
- ***%ROWTYPE*** *– refers to an entire row of a table*
- ***REF CURSOR*** *– reference to a cursor result set*

Example (%TYPE and %ROWTYPE)

```
DECLARE
  v_salary Employee.salary%TYPE;
  emp_row  Employee%ROWTYPE;
BEGIN
  v_salary := 35000;
END;
```

4. LOB (Large Object) Data Types

LOB data types are used to store **large volumes of data** that cannot be efficiently handled by normal scalar data types. These include text documents, images, audio, and video files.

- Can store very large data
- Used for multimedia and document storage
- Some LOBs may be stored outside the database

Types of LOB Data Types

- ***CLOB*** *– stores large character data*
- ***NCLOB*** *– stores large Unicode character data*
- ***BLOB*** *– stores large binary data*
- ***BFILE*** *– stores binary files outside the database*

Example

```
DECLARE
  v_text CLOB;
BEGIN
  v_text := 'This is a large character data example...';
  DBMS_OUTPUT.PUT_LINE(SUBSTR(v_text, 1, 50));
END;
```

3.4 Cursor in PL/SQL

In PL/SQL, a **cursor** is a pointer or reference to a **memory area** where the result of a SQL query is stored. When a SQL statement such as a SELECT query returns **more than one row**, PL/SQL needs a way to process those rows **one at a time**. This is done using a cursor.

In simple terms, a cursor allows PL/SQL to **fetch and process each row individually** from the result set of a query. Without cursors, PL/SQL can handle only single-row queries. Therefore, cursors play a very important role in row-by-row processing.

Why Cursors Are Needed

- SQL works on a **set of rows**, but PL/SQL works procedurally.
- When a SELECT statement returns multiple rows, PL/SQL cannot handle them directly.
- A cursor bridges the gap between **set-based SQL** and **procedural PL/SQL**.

Types of Cursors

There are **two main types**:

1. Implicit Cursor

An **implicit cursor** is a cursor that is **automatically created and managed by the Oracle database** whenever a SQL statement is executed in PL/SQL. The programmer does **not need to declare, open, fetch, or close** this cursor explicitly. Oracle internally uses the implicit cursor to process the SQL statement.

Implicit cursors are mainly used when a SQL statement affects **only one row** or when the result of a query is expected to return a **single row**. They are commonly associated with SQL statements such as **SELECT INTO, INSERT, UPDATE, and DELETE**.

When Implicit Cursors Are Used

- Implicit cursors are used in the following situations:
- When a **SELECT INTO** statement returns exactly one row
- When **INSERT, UPDATE, or DELETE** statements are executed
- When no user-defined (explicit) cursor is declared

If a SELECT statement returns **no rows or more than one row**, Oracle raises an exception.

Example (Implicit Cursor):

```
BEGIN
  UPDATE employees
  SET salary = salary + 1000
  WHERE department_id = 10;
  IF SQL%ROWCOUNT > 0 THEN
    DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' rows updated.');
```

```
  ELSE
    DBMS_OUTPUT.PUT_LINE('No rows updated.');
  END IF;
END;
```

Case 1: **If 3 employees belong to department_id = 10**
Output:
3 rows updated.
Case 2: **If no employees belong to department_id = 10**
Output:
No rows updated.

1. Explicit Cursor

An explicit cursor is a cursor that is defined and controlled by the programmer. It is used when a SQL query returns more than one row, and each row needs to be processed one at a time. Unlike implicit cursors, explicit cursors must be declared, opened, fetched, and closed explicitly in the PL/SQL program.

Explicit cursors give the programmer full control over the result set and are especially useful for implementing complex business logic that requires row-by-row processing.

When Explicit Cursors Are Used
Explicit cursors are used in the following situations:

- When a SELECT statement returns multiple rows
- When row-by-row processing is required
- When cursor attributes need to be checked manually
- When greater control over query execution is needed

Declared explicitly by the programmer when the query **returns multiple rows**.
Steps:

1. ***DECLARE*** *the cursor*
2. ***OPEN*** *the cursor*
3. ***FETCH*** *rows from cursor*
4. ***CLOSE*** *the cursor*

```
DECLARE
  -- Declare the explicit cursor
  CURSOR it_emp_cur IS
    SELECT EmpName, Salary
    FROM Employee
    WHERE Department = 'IT';
```

```
  -- Variables to store fetched values
  v_name   Employee.EmpName%TYPE;
  v_salary Employee.Salary%TYPE;

BEGIN
  -- Open the cursor
  OPEN it_emp_cur;
  -- Fetch rows one by one
  LOOP
    FETCH it_emp_cur INTO v_name, v_salary;
    EXIT WHEN it_emp_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_name || ' earns ' || v_salary);
  END LOOP;
  -- Close the cursor
  CLOSE it_emp_cur;

END;
```

EMPLOYEE Table

EmpID	EmpName	Department	Salary
101	Ravi	IT	30000
102	Sita	HR	25000
103	Kiran	IT	40000
104	Meena	HR	28000

Output:

Ravi earns 30000
Kiran earns 40000

3.6 PL/SQL Procedure

A **PL/SQL procedure** is a **named block of PL/SQL code** that is created to perform a specific task in the database. Once a procedure is created, it is **stored permanently in the Oracle database** and can be executed whenever required. Procedures help in organizing database programs in a structured and modular way.

A procedure may contain SQL statements, PL/SQL statements, control structures, and exception handling code. It can take input values through parameters, process them, and optionally return results using output parameters. However, unlike functions, a procedure **does not return a value directly** using the RETURN statement.

Need for PL/SQL Procedures

In large database applications, the same logic is often required at multiple places. Writing the same SQL and PL/SQL code repeatedly makes programs lengthy and difficult to maintain. Procedures solve this problem by allowing programmers to write the logic once and reuse it many times.

Procedures also improve database security by hiding the internal logic from users. Users can be given permission to execute a procedure without giving direct access to the underlying tables.

Characteristics of a PL/SQL Procedure

- It is a stored program unit in the database
- It performs a specific task
- It can accept parameters
- It does not return a value directly
- It can be executed multiple times
- It supports exception handling
-

Structure of a PL/SQL Procedure

A procedure consists of two main parts:

1. ***Procedure specification*** *– contains the procedure name and parameters*
2. ***Procedure body*** *– contains the executable statements*

Syntax:

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

Example:

```
CREATE OR REPLACE PROCEDURE welcome_message
AS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Welcome to PL/SQL Procedures');
END;
```

Execution:

EXEC welcome_message;

Example:

1.Creating a Procedure: The following procedure increases the salary of an employee based on the employee ID and the increment amount passed as parameters.

```
CREATE OR REPLACE PROCEDURE increase_salary (  p_emp_id     IN
employees.employee_id%TYPE,

  p_increment  IN NUMBER

)

IS

BEGIN

  UPDATE employees

  SET salary = salary + p_increment

  WHERE employee_id = p_emp_id;

DBMS_OUTPUT.PUT_LINE('Salary updated for Employee ID: ' || p_emp_id);

END increase_salary;
```

- *p_emp_id and p_increment are input parameters.*
- *The UPDATE statement modifies the salary.*
- *DBMS_OUTPUT.PUT_LINE displays a confirmation message.*
- *The procedure is stored in the database after creation.*

2. Executing a PL/SQL Procedure

Once a procedure is created, it can be executed whenever needed. A procedure can be executed using the EXECUTE (EXEC) command or from inside an anonymous PL/SQL block.

```
EXECUTE increase_salary(101, 2000);

Execution Using Anonymous Block

BEGIN
```

increase_salary(101, 2000);

END;

- *The procedure increase_salary is called with employee ID 101.*
- *The salary of employee 101 is increased by 2000.*
- *The procedure performs the update operation in the database.*

3. Dropping a PL/SQL Procedure

Dropping a procedure means removing it permanently from the database. Once dropped, the procedure cannot be executed unless it is created again.

Syntax

DROP PROCEDURE procedure_name;

Example

DROP PROCEDURE increase_salary;

Types of Parameters

- ***IN parameter**: Used to pass input values to the procedure. This is the default mode.*
- ***OUT parameter** : Used to return values from the procedure.*
- ***IN OUT parameter** : Used to pass values to the procedure and return modified values back.*

Procedure with Input Parameter

This procedure displays the name of an employee based on employee ID.

```
CREATE OR REPLACE PROCEDURE display_employee
(
  p_empid IN Employee.EmpID%TYPE
)
AS
  v_name Employee.EmpName%TYPE;
BEGIN
  SELECT EmpName INTO v_name
  FROM Employee
  WHERE EmpID = p_empid;

  DBMS_OUTPUT.PUT_LINE('Employee Name: ' || v_name);
```

END;

Execution:
EXEC display_employee(101);

3.7 Function in PL/SQL

A function in PL/SQL is a named, stored program unit that is designed to perform a computation and return a single value to the calling program. Functions are mainly used when some logic or calculation needs to be performed and the result of that logic is required for further use.

A function is created and stored in the Oracle database and can be executed whenever needed. The most important feature of a function is that it must return a value using the RETURN statement. Because of this, functions can be used not only in PL/SQL blocks but also directly inside SQL statements such as SELECT, WHERE, and HAVING.

Syntax

```
CREATE [OR REPLACE] FUNCTION function_name
  (parameter1 datatype, parameter2 datatype, ...)
RETURN return_datatype
IS
  -- Declarations
BEGIN
  -- Executable statements
  RETURN value;
EXCEPTION
  -- Optional error handling
END  function_name;
```

Example : PL/SQL Program Using Function
***"Problem** Calculate the **bonus amount (10% of salary)** using a PL/SQL function and display the result."*

Step 1: Create the Function

```
CREATE OR REPLACE FUNCTION calculate_bonus
(
  p_salary IN NUMBER
)
```

```
RETURN NUMBER
IS
  v_bonus NUMBER;
BEGIN
  v_bonus := p_salary * 0.10;
  RETURN v_bonus;
END;
```

Step 2: Call the Function from a PL/SQL Block

```
DECLARE
 v_salary NUMBER := 30000;
v_bonus  NUMBER;
BEGIN
  v_bonus := calculate_bonus(v_salary);
  DBMS_OUTPUT.PUT_LINE('Salary : ' || v_salary);
  DBMS_OUTPUT.PUT_LINE('Bonus  : ' || v_bonus);
END;
```

Output

```
Salary : 30000
Bonus  : 3000
```

Advantages

- Functions support code reusability, as the same function can be called multiple times.
- They improve modularity by dividing large programs into smaller logical units.
- Functions can be used directly in SQL statements such as SELECT, WHERE, and HAVING.
- They help in performing calculations and validations efficiently.
- Functions make programs easy to understand and maintain.
- They reduce duplicate code, improving consistency.
- Functions return a single value, making result handling simple.
- They improve database performance by executing logic inside the database.

Disadvantages

- Functions can return only one value.
- They cannot perform transaction control operations like COMMIT or ROLLBACK.
- Functions are not suitable for complex data manipulation tasks.
- Error handling in functions can make code complex.
- Overusing functions inside SQL queries may affect performance.
- Functions cannot have OUT parameters.
- Debugging functions can be difficult in large applications.

Advantages of Functions over Procedures

Feature	Procedure	Function
Return value	May or may not return value	**Must return exactly one value**
Use in SQL	Cannot be used directly in SQL statements	**Can be called inside SQL** (SELECT, WHERE, etc.)
Purpose	Perform operations (DML, business logic)	Perform calculations and return result
Integration	Called from PL/SQL blocks only	Called from PL/SQL and SQL statements
Best Use	For tasks like updating rows, inserting logs	For computations like tax, bonus, discount

3.8 Package in PL/SQL

A **package in PL/SQL** is a **collection of related program units** such as procedures, functions, variables, cursors, and exceptions that are grouped together under a single name. Packages are stored permanently in the Oracle database and are used to organize large PL/SQL programs in a structured and efficient manner.

In simple words, a package acts like a **container** that holds logically related PL/SQL components. By grouping related items together, packages make database programs **modular, reusable, secure, and easy to maintain**.

Why Packages Are Needed

As database applications grow, having many independent procedures and functions becomes difficult to manage. Packages solve this problem by:

- Organizing related code into a single unit
- Reducing complexity
- Improving performance
- Providing better security

Components of a Package

A package has **two main parts**:

1. ***Package Specification (Spec)***
   - Declares the **interface**: procedures, functions, variables, cursors that are accessible to users.
   - Acts like a **header file**.
2. ***Package Body***
   - Contains the **actual implementation** of the procedures, functions, and cursors declared in the specification.
   - Can also include private elements (not visible in spec).

Steps to Create a Package:-

Step 1: Package Specification

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE increase_salary (p_emp_id IN NUMBER, p_increment IN NUMBER);
  FUNCTION get_salary (p_emp_id IN NUMBER) RETURN NUMBER;
END emp_pkg;
```

Explanation

- *This part is called the package specification.*
- *It defines the interface of the package.*
- *It tells users what procedures and functions are available, but not how they work.*

Here:

- increase_salary is a procedure to update salary.
- get_salary is a function to return salary.
- Other programs can call these because they are declared in the specification.

Step 2: Package Body

CREATE OR REPLACE PACKAGE BODY emp_pkg IS

The package body contains the actual implementation of the procedure and function declared in the specification.

Procedure Implementation

```
PROCEDURE increase_salary (p_emp_id IN NUMBER, p_increment IN NUMBER) IS
BEGIN
  UPDATE employees
  SET salary = salary + p_increment
  WHERE employee_id = p_emp_id;
  DBMS_OUTPUT.PUT_LINE('Salary updated for Employee ID: ' || p_emp_id);
END increase_salary;
```

Explanation

- *This procedure increases the salary of an employee.*
- *p_emp_id identifies the employee.*
- *p_increment specifies how much salary to increase.*
- *The UPDATE statement modifies the employees table.*
- *A message is displayed to confirm the update.*

Function Implementation

```
FUNCTION get_salary (p_emp_id IN NUMBER) RETURN NUMBER IS
  v_sal employees.salary%TYPE;
BEGIN
  SELECT salary INTO v_sal
  FROM employees
  WHERE employee_id = p_emp_id;

  RETURN v_sal;
END get_salary;
```

Explanation

- *This function retrieves and returns the salary of an employee.*
- *It uses SELECT INTO to fetch salary.*
- *The fetched salary is returned using the RETURN statement.*
- *Since it returns a value, it is a function.*

END emp_pkg;

Marks the end of the package body.

Now the package is fully created and stored in the database.

Step 3: Executing the Package

```
BEGIN
  emp_pkg.increase_salary(101, 2000);
  DBMS_OUTPUT.PUT_LINE('Current Salary: ' || emp_pkg.get_salary(101));
END;
```

Explanation

- *This is an anonymous PL/SQL block used to execute the package.*
- *emp_pkg.increase_salary(101, 2000)*
- *Calls the procedure inside the package.*
- *Increases salary of employee 101 by 2000.*
- *emp_pkg.get_salary(101)*
- *Calls the function inside the package.*
- *Returns the updated salary.*

- *Dot notation (package_name.member_name) is used to access package components.*

Output

- Salary updated for Employee ID: 101
- Current Salary: 52000

3.8 Exception Handling in PL/SQL

Exception handling in PL/SQL is a mechanism used to detect, handle, and respond to runtime errors that occur during the execution of a PL/SQL program. Instead of abruptly stopping the program when an error occurs, exception handling allows the program to take corrective actions or display meaningful error messages.

An exception is an error condition that arises at runtime, such as division by zero, invalid data, or no data found. PL/SQL provides a structured way to handle these errors using the EXCEPTION section of a PL/SQL block.

General Syntax

```
BEGIN
  -- executable statements
EXCEPTION
  WHEN exception_name THEN
    -- handling statements
  WHEN OTHERS THEN
    -- handling for all other exceptions
END;
```

Types of Exceptions in PL/SQL

In PL/SQL, **exceptions** are runtime errors that occur during the execution of a program. PL/SQL provides a structured mechanism to handle these errors using the **EXCEPTION** block. Based on how they are defined and handled, exceptions in PL/SQL are classified into **three main types**.

1. Predefined Exceptions

Predefined exceptions are **automatically defined and raised by Oracle** whenever a runtime error occurs. These exceptions already have **predefined names**, so the programmer can handle them directly without declaring them.

Common situations where predefined exceptions occur include:

- When a SELECT statement returns no rows
- When a SELECT statement returns more than one row
- When division by zero occurs

Example: ***ZERO_DIVIDE***

```
DECLARE
  v_result NUMBER;
BEGIN
  v_result := 100 / 0;
  DBMS_OUTPUT.PUT_LINE('Result = ' || v_result);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Error: Division by zero!');
END;
```

Output:

Error: Division by zero!

Examples of predefined exceptions:

- NO_DATA_FOUND
- TOO_MANY_ROWS
- ZERO_DIVIDE
- INVALID_NUMBER
- DUP_VAL_ON_INDEX

These exceptions improve program reliability by allowing the developer to handle common runtime errors easily.

2. User-Defined Exceptions

User-defined exceptions are **explicitly declared by the programmer** to handle specific business rules or application-specific error conditions. These exceptions are not raised automatically; instead, they are raised using the **RAISE** statement when a particular condition is met.

User-defined exceptions are useful when:

- Business logic violations need to be handled
- Custom error messages are required
- Application-specific conditions must be checked

They provide flexibility to handle errors that are not covered by predefined exceptions.

Example:

```
DECLARE
  v_salary employees.salary%TYPE;
BEGIN
  SELECT salary INTO v_salary
  FROM employees
  WHERE employee_id = 999;   -- employee does not exist

  DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary);
```

```
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No employee found with given ID');
END;
```

3. Non-Predefined Exceptions

Non-predefined exceptions are **Oracle errors that do not have predefined names** in PL/SQL. These errors are identified by **error numbers**, and to handle them, the programmer must associate the error number with an exception name using the **PRAGMA EXCEPTION_INIT** directive.

These exceptions are useful when:

- Handling specific Oracle error codes
- Managing database constraint violations not covered by predefined exceptions

Non-predefined exceptions bridge the gap between Oracle error codes and PL/SQL exception handling.

Example:

```
DECLARE
  duplicate_value EXCEPTION;
  PRAGMA EXCEPTION_INIT(duplicate_value, -1);
BEGIN
  INSERT INTO employees(employee_id, employee_name)
  VALUES (101, 'Ravi');   -- assuming 101 already exists

EXCEPTION
  WHEN duplicate_value THEN
    DBMS_OUTPUT.PUT_LINE('Duplicate employee ID not allowed');
END;
```

3.9 Trigger in PL/SQL

A **trigger in PL/SQL** is a **special type of stored program unit** that is **automatically executed by the Oracle database** when a specific event occurs. These events are usually related to changes in data, such as inserting, updating, or deleting rows in a table. Unlike procedures and functions, triggers **do not need to be explicitly called** by the user or application.

Conceptually, a trigger acts as an **automatic reaction mechanism**. When a predefined event takes place on a database object, the trigger fires and executes the associated PL/SQL code. This makes triggers very useful for enforcing rules and maintaining consistency in the database without relying on user actions.

Triggers are needed in database systems because some rules must be enforced **automatically** whenever data changes. It is not always possible or safe to depend on users or applications to apply these rules manually.

Key Characteristics of Triggers

- Triggers are **event-driven**, not user-driven
- They are associated with a **table or view**
- They execute **automatically** when an event occurs
- They cannot accept parameters
- They cannot be executed using EXEC or CALL
- They execute either **before or after** a data modification

Syntax of a Trigger

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE | AFTER
INSERT | UPDATE | DELETE
ON table_name
FOR EACH ROW
BEGIN
  -- trigger body
END;
```

Types of Triggers in PL/SQL

Triggers are commonly classified based on different criteria:

1. Based on Timing

- ***BEFORE Trigger:***

A BEFORE trigger is executed before a DML operation such as INSERT, UPDATE, or DELETE is performed on a table. It is mainly used for data validation and to prevent invalid changes before they occur.

- ***AFTER Trigger:***

An AFTER trigger is executed after a DML operation has been completed on a table. It is commonly used for auditing, logging, or performing follow-up actions after data changes.

BEFORE UPDATE Trigger Program

```
CREATE OR REPLACE TRIGGER check_salary_before_update
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
  IF :NEW.salary < :OLD.salary THEN
```

```
    RAISE_APPLICATION_ERROR(-20001, 'Salary reduction is not allowed');
  END IF;
END;
```

- ✓ *This is a **BEFORE UPDATE trigger**, so it executes **before** the update operation.*
- ✓ *:OLD.salary refers to the existing salary.*
- ✓ *:NEW.salary refers to the new salary value.*
- ✓ *If the new salary is less than the old salary, the update is stopped.*
- ✓ *The error message is displayed to the user.*

AFTER INSERT Trigger Program

```
CREATE OR REPLACE TRIGGER after_insert_employee AFTER INSERT
  ON employees FOR EACH ROW
  BEGIN
    DBMS_OUTPUT.PUT_LINE(
      'Employee added successfully with ID: ' || :NEW.employee_id
    );
  END;
```

- ✓ *This is an **AFTER INSERT trigger**, so it executes **after** the INSERT operation is completed.*
- ✓ *:NEW.employee_id refers to the newly inserted employee ID.*
- ✓ *The trigger automatically displays a confirmation message.*
- ✓ *The trigger does not stop or modify the insert; it only performs a follow-up action.*

2. Based on Event

- **INSERT Trigger:** An **INSERT trigger** is automatically fired when a new row is inserted into a table. It is commonly used to validate inserted data, enforce business rules, or record audit information related to new entries.
- **UPDATE Trigger :** An UPDATE trigger is automatically executed when existing data in a table is modified.It is used to track changes, compare old and new values, or restrict invalid updates.
- **DELETE Trigger :** A DELETE trigger is automatically activated when a row is removed from a table.It is mainly used for auditing, logging deleted data, or maintaining data consistency.

Example Program

```
CREATE OR REPLACE TRIGGER emp_insert_trg
AFTER INSERT
ON employees
FOR EACH ROW
```

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('New employee inserted with ID: ' || :NEW.employee_id);
END;
```

Example Program

```
CREATE OR REPLACE TRIGGER emp_insert_trg
AFTER INSERT
ON employees
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('New employee inserted with ID: ' || :NEW.employee_id);
END;
```

Example Program :*Display a message when an employee record is deleted.*

```
CREATE OR REPLACE TRIGGER emp_delete_trg
AFTER DELETE
ON employees
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('Employee deleted with ID: ' || :OLD.employee_id);
END;
```

3. Based on Level

- **Row-Level Trigger –**A **row-level trigger** is executed **once for each row** affected by a DML statement such as INSERT, UPDATE, or DELETE. It uses the FOR EACH ROW clause and allows access to **old and new values** of the row.
- **Statement-Level Trigger –**A **statement-level trigger** is executed **once for an entire SQL statement**, regardless of how many rows are affected by that statement. It is used when an action needs to be performed **only once per operation**, such as logging or enforcing rules at the statement level.

Example (Row-Level Trigger):

```
CREATE OR REPLACE TRIGGER salary_check_trg
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
  IF :NEW.salary < :OLD.salary THEN
    RAISE_APPLICATION_ERROR(-20001, 'Salary cannot be reduced!');
  END IF;
```

END;

CREATE OR REPLACE TRIGGER salary_check_trg
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
IF :NEW.salary < :OLD.salary THEN
RAISE_APPLICATION_ERROR(-20001, 'Salary cannot be reduced!');
END IF;
END;
If we try to reduce any employee's salary, this trigger prevents it.

Example (Statement-Level Trigger):
CREATE OR REPLACE TRIGGER emp_update_stmt_trg
AFTER UPDATE
ON employees
BEGIN
DBMS_OUTPUT.PUT_LINE('Employee table updated successfully');
END;

Execution Example
UPDATE employees
SET salary = salary + 1000
WHERE department = 'IT';
Output
Employee table updated successfully

If 50 rows are updated in one SQL statement, this trigger fires ***only once****.*

Short Questions

1. Define **CHAR** and **VARCHAR2** data types.
2. What is **PRIMARY KEY constraint**?
3. What does **NOT NULL** constraint mean?
4. What is the purpose of **ORDER BY** clause?
5. What is the use of **GROUP BY** clause?
6. Difference between **COUNT()** and **SUM()** functions.
7. Name two **logical operators** in SQL.
8. Define **INNER JOIN**.
9. What is a **CROSS JOIN**?

10. Differentiate between **Nested and Correlated Subquery**.
11. Give one example of a **Set Operator**.
12. How do you **create a view** in SQL?
13. How do you **delete a view** in SQL?
14. Define **PL/SQL**.
15. Name two **data types in PL/SQL**.
16. What is a **cursor** in PL/SQL?
17. What is a **procedure**?
18. Define **function** in PL/SQL.
19. What is a **package**?
20. Define **row-level trigger**.
21. Define **statement-level trigger**.

Long Questions

1. SQL Basics

1. Explain the **different data types in SQL** with examples.
2. Discuss **Integrity Constraints** with examples.
3. Explain the use of **ORDER BY, GROUP BY, and HAVING** clauses with examples.
4. Explain **aggregate functions** (COUNT, SUM, AVG, MIN, MAX) with examples.
5. Discuss **logical and special operators** in SQL with examples.
6. Explain **set operators** (UNION, INTERSECT, MINUS) with examples.
7. Explain **different types of joins** (Inner, Outer, Cross) with examples.
8. Explain **nested and correlated subqueries** with examples.

2. Views

9. Explain how to **create, modify, and delete views** with examples.
10. Discuss the advantages and limitations of using **views**.

3. PL/SQL

11. Explain the **structure of a PL/SQL block** with an example.
12. Explain **control structures and iterative controls** in PL/SQL with examples.
13. Explain **cursors in PL/SQL** with types and examples.
14. Explain **procedures and functions** in PL/SQL with examples.
15. Discuss **packages in PL/SQL** with advantages.
16. Explain **exception handling** in PL/SQL with examples.
17. Explain **row-level and statement-level triggers** with examples

Unit – IV: Transaction, Recovery & Security

1.1 Transaction Management

Transaction Management is a part of the Database Management System (DBMS) that controls how database operations are performed safely. It ensures that all database operations are executed correctly and the database remains in a consistent state even when failures occur or many users access the database at the same time.

A **transaction** is a group of one or more database operations that are treated as a single logical unit. These operations must either be completed fully or not executed at all.

In DBMS, a transaction represents a real-world activity. Each transaction may include operations such as reading data, inserting data, updating data, or deleting data. The result of a transaction should always maintain the correctness of the database.

If any operation inside the transaction fails, the entire transaction is cancelled and the database is returned to its previous state.

Need for Transaction Management

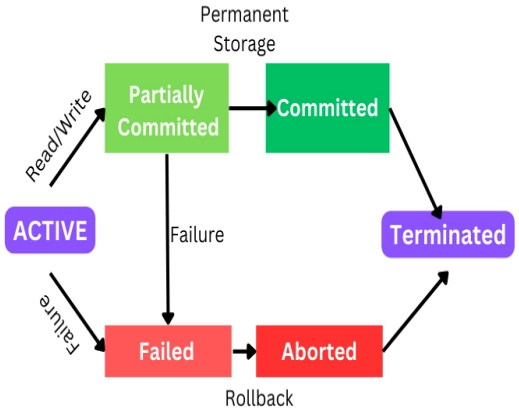
Transaction management is required to:

- Avoid partial updates in the database
- Maintain data accuracy and reliability
- Handle system failures like power loss or crashes
- Allow multiple users to access data safely at the same time

Without transaction management, the database may enter an incorrect or inconsistent state.

1.2 Transaction States in DBMS

During execution, a transaction passes through several states. These states describe the life cycle of a transaction from start to completion.



1. Active State

A transaction is in the **active state** when it begins execution and performs read and write operations on the database. In this state, the transaction is still executing its operations. If no error occurs, it continues execution; otherwise, it may move to the failed state.

2. Partially Committed State

After the transaction executes its final operation, it enters the **partially committed state**. At this

stage, all statements have been executed, but the changes are not yet permanently saved in the database. The transaction can still be undone if a failure occurs.

3. Committed State

A transaction reaches the **committed state** when all its changes are successfully written to the database and stored permanently. Once committed, the effects of the transaction cannot be undone. This state ensures the durability of the transaction.

4. Failed State

A transaction enters the **failed state** when an error occurs during execution, such as a system crash, logical error, or resource failure. In this state, the transaction cannot proceed further.

5. Aborted State

After failure, the transaction moves to the **aborted state**. In this state, all changes made by the transaction are rolled back, and the database is restored to its previous consistent state. The transaction may either be restarted or permanently terminated.

1.3 ACID Properties of Transaction

A transaction should satisfy **ACID properties**:

1. ***Atomicity*** *– All operations succeed or none do.*
2. ***Consistency*** *– Database remains in a valid state after transaction.*
3. ***Isolation*** *– Concurrent transactions do not interfere with each other.*
4. ***Durability*** *– Once committed, changes are permanent even if system fails.*

In a Database Management System (DBMS), a transaction represents a logical unit of work that performs one or more database operations. To ensure correctness, reliability, and safety of data, every transaction must follow the **ACID properties**. These properties protect the database from errors, failures, and simultaneous access by multiple users.

1. Atomicity

Atomicity means that a transaction is treated as a single, indivisible unit. All the operations of a transaction must be executed completely, or none of them should be executed. Partial execution of a transaction is not allowed. If any operation inside the transaction fails due to an error, system crash, or power failure, the entire transaction is rolled back and the database is restored to its previous state.

Atomicity ensures that the database does not contain incomplete or half-finished data.

Example:

Consider an online money transfer from Account A to Account B. The amount must be deducted from Account A and added to Account B. If the amount is deducted from Account A but the system crashes before adding it to Account B, atomicity ensures that the deducted amount is restored to Account A. Thus, either both steps occur or neither occurs.

2. Consistency

Consistency ensures that a transaction takes the database from one valid state to another valid state. After the completion of a transaction, all database rules, constraints, and integrity conditions must be satisfied. A transaction should not violate primary key constraints, foreign key constraints, or any business rules defined in the database.

Consistency guarantees the correctness of data stored in the database.

Example:

In a banking system, the balance of an account should never become negative if such a rule is defined. When a withdrawal transaction is performed, consistency ensures that the withdrawal is allowed only if sufficient balance is available. After the transaction, the database remains in a valid and consistent state.

3. Isolation

Isolation means that when multiple transactions are executed at the same time, each transaction should behave as if it is the only transaction in the system. The intermediate results of a transaction should not be visible to other transactions until it is completed. This prevents problems such as dirty reads, lost updates, and inconsistent data access.

Isolation ensures that concurrent transactions do not interfere with one another.

Example:

Suppose two users are trying to book the last available seat in a train at the same time. Isolation ensures that only one user can successfully book the seat, while the other user is informed that the seat is no longer available. The database prevents both users from booking the same seat.

4. Durability

Durability guarantees that once a transaction is successfully committed, its changes are permanently saved in the database. Even if a system crash, hardware failure, or power outage occurs after the commit operation, the committed data will not be lost. The DBMS uses techniques such as logging and backups to ensure durability.

Durability provides confidence that committed data is safe and reliable.

Example:
After successfully booking a flight ticket online, the booking confirmation is shown to the user. Even if the system crashes immediately after confirmation, the ticket details remain stored in the database and can be retrieved later using the booking reference number.

2. Concurrency Control in DBMS

2.1Concurrency Control Problems:

In a database system, **concurrency** means that **multiple transactions are executed at the same time**. Concurrency improves system performance and resource utilization. However, if concurrent transactions are not properly controlled, they may interfere with each other and cause **incorrect or inconsistent results**. These issues are known as **concurrency control problems**.

1. Dirty Read: A dirty read occurs when one transaction reads data that has been modified by another transaction but not yet committed. If the second transaction is rolled back, the first transaction would have read invalid (dirty) data.

Example:

- **Transaction T1** updates a customer's balance from ₹10,000 to ₹12,000 but has not committed yet.
- **Transaction T2** reads the updated balance of ₹12,000.
- Now T1 **rolls back** (maybe due to an error), and the balance goes back to ₹10,000.
- But T2 has already used the **dirty value ₹12,000**, which is incorrect.

Understanding Purpose: A bank worker sees a deposit in your account before the system finalizes it. Later, the deposit is canceled, but the worker already processed something using the incorrect amount.

2. Lost Update: A lost update occurs when two or more transactions read the same data item and update it independently, but one transaction's update overwrites the other, causing one update to be lost.

In a concurrent transaction environment, multiple transactions may access the same data item at the same time. If these transactions read the same initial value and then perform updates without proper synchronization, the database may accept both updates. However, the update performed later overwrites the earlier one. As a result, the effect of the first transaction is not reflected in the final database state.

This problem occurs due to the absence of proper write coordination between concurrent transactions.

Example

- Transaction T1 reads account balance = ₹10,000.
- Transaction T2 also reads account balance = ₹10,000.
- T1 updates the balance to ₹11,000 and commits.
- T2 updates the balance to ₹12,000 and commits.

The final balance becomes ₹12,000 instead of ₹13,000.
Thus, the update made by T1 is lost.

Understanding Purpose: Two bank clerks update your account at the same time without coordination; one clerk's update is lost.

3. Non-Repeatable Read: A non-repeatable read occurs when a transaction reads the same data item more than once and obtains different values because another transaction modifies and commits that data item between the two reads.

In a concurrent transaction environment, a transaction expects the data it reads to remain unchanged during its execution. However, if another transaction updates the same data item and commits while the first transaction is still active, subsequent reads of that data item return a different value. As a result, the read operation is not repeatable, leading to inconsistency within the transaction.

This problem arises because the database does not guarantee that data values remain stable for the duration of a transaction.

Example

- Transaction T1 reads a product price as ₹500.
- Transaction T2 updates the price to ₹600 and commits.
- Transaction T1 reads the same product price again and obtains ₹600.

Thus, T1 reads different values for the same data item during its execution.
Understanding Purpose: You're reading an online product's price while someone from the marketing team updates the price behind the scenes.

4. Phantom Read: A **phantom read** occurs when a transaction executes the same query more than once and obtains **different sets of rows** because another transaction has **inserted or deleted records** that satisfy the query condition between the two executions.

Unlike non-repeatable reads, phantom reads involve changes in the **number of rows returned**, not just changes in existing values.

Explanation

In a concurrent environment, a transaction may read a group of rows that satisfy a certain condition. If another transaction inserts or deletes rows that also satisfy the same condition and commits, the first transaction, when repeating the query, may see additional or missing rows. These newly appearing or disappearing rows are called **phantoms**.

This happens because the database does not restrict **range-based operations**, such as queries using conditions (WHERE, BETWEEN, >, <).

Example

- Transaction **T1** executes:
  SELECT * FROM orders WHERE amount > 1000
  → Returns 3 rows.
- Transaction **T2** inserts a new order with amount = 2000 and commits.
- Transaction **T1** executes the same query again.
  → Returns 4 rows.

The extra row returned in the second execution is a **phantom row**.

Understanding Purpose: You prepare a list of students with marks > 90. Meanwhile, a new result comes in and someone adds a new topper. When you run the same list again, the new student appears — a **phantom** record.

5. Uncommitted Data Problem

The **uncommitted data problem** occurs when a transaction uses intermediate results produced by another transaction that has not yet completed. These intermediate results may later be undone if the transaction fails.

This problem is closely related to dirty reads but emphasizes the use of **incomplete transaction results**.

Example:-

- Transaction T1 deducts ₹500 from an account (not yet committed)
- Transaction T2 checks the balance and sees the reduced amount
- Transaction T1 fails and rolls back

Transaction T2 has used data that was never finalized.

Problem:-

- Decisions are made using incomplete data

- Leads to incorrect and inconsistent results

Concurrency control ensures that transactions can access **only committed and stable data**.

6. Inconsistent Data Problem

The **inconsistent data problem** occurs when a transaction reads some data values before another transaction completes and other values after it completes. This gives the transaction an inconsistent view of the database.

Example:-

- Transaction T1 transfers ₹500 from Ravi to Kavya
- Transaction T2 reads Ravi's balance before the transfer
- T1 completes the transfer
- Transaction T2 reads Kavya's balance after the transfer

Transaction T2 sees **half-updated data**, which is logically inconsistent.

Problem:-

- The database state observed by the transaction is incorrect
- Computations based on such data become unreliable

Concurrency control ensures that a transaction sees the database **either before or after** another transaction, but never in the middle of execution.

2.2 Concurrency Control Protocols in DBMS

Concurrency control protocols are a set of rules used by a Database Management System (DBMS) to control the simultaneous execution of multiple transactions. In a multi-user environment, many transactions may try to access the same data at the same time. Without proper control, this can lead to incorrect results and data inconsistency.

Concurrency control ensures that even though transactions execute concurrently, the final result of execution is correct and equivalent to some serial order of transactions.

Concurrency control is closely related to the Isolation property of ACID. It ensures that transactions do not interfere with each other while executing.

In a multi-user database system, many transactions may try to access and update the same data at the same time. Allowing transactions to execute concurrently improves system performance and reduces waiting time. However, **without proper concurrency control**, concurrent execution can lead to incorrect results and data inconsistency.

Concurrency control is required to ensure that **simultaneous transactions do not interfere with each other** and that the database remains correct and reliable.

Need for Concurrency Control in DBMS

In a multi-user database system, several transactions may execute simultaneously. While concurrent execution improves system performance and resource utilization, it can also lead to serious problems if proper control mechanisms are not applied. **Concurrency control** is required to ensure that simultaneous transactions do not interfere with each other and that the database remains consistent and correct.

When concurrency control is not used, the following problems may occur:

Advantages of Concurrency Control Protocols in DBMS

1. Maintains Data Consistency: Concurrency control ensures that multiple transactions accessing the same data do not cause inconsistencies. It guarantees that the database remains in a correct and valid state even when many transactions execute at the same time.

2. Prevents Concurrency Problems: Concurrency control protocols prevent common problems such as:

- Lost updates
- Dirty reads
- Uncommitted data access
- Inconsistent data retrieval

By controlling the order of execution, these problems are avoided.

3. Ensures Transaction Isolation: Concurrency control enforces the **Isolation** property of ACID. Each transaction behaves as if it is executing alone in the system, even though many transactions may be running concurrently.

4. Guarantees Serializability: These protocols ensure that the result of concurrent execution is equivalent to some serial execution of transactions. This guarantees correctness while still allowing parallel execution.

5. Improves Database Reliability: By avoiding incorrect updates and inconsistent reads, concurrency control improves the reliability and trustworthiness of the database system.

6. Supports Multi-User Environment: Concurrency control allows multiple users to access the database at the same time without interfering with each other. This is essential for real-world applications like banking systems, airline reservations, and online shopping platforms.

7. Better System Performance: Although concurrency control restricts some operations, it still allows safe parallel execution of transactions. This improves system throughput compared to executing all transactions strictly one after another.

8. Protects Data Integrity: By enforcing proper execution order and access control, concurrency control protocols protect the integrity of the data stored in the database.

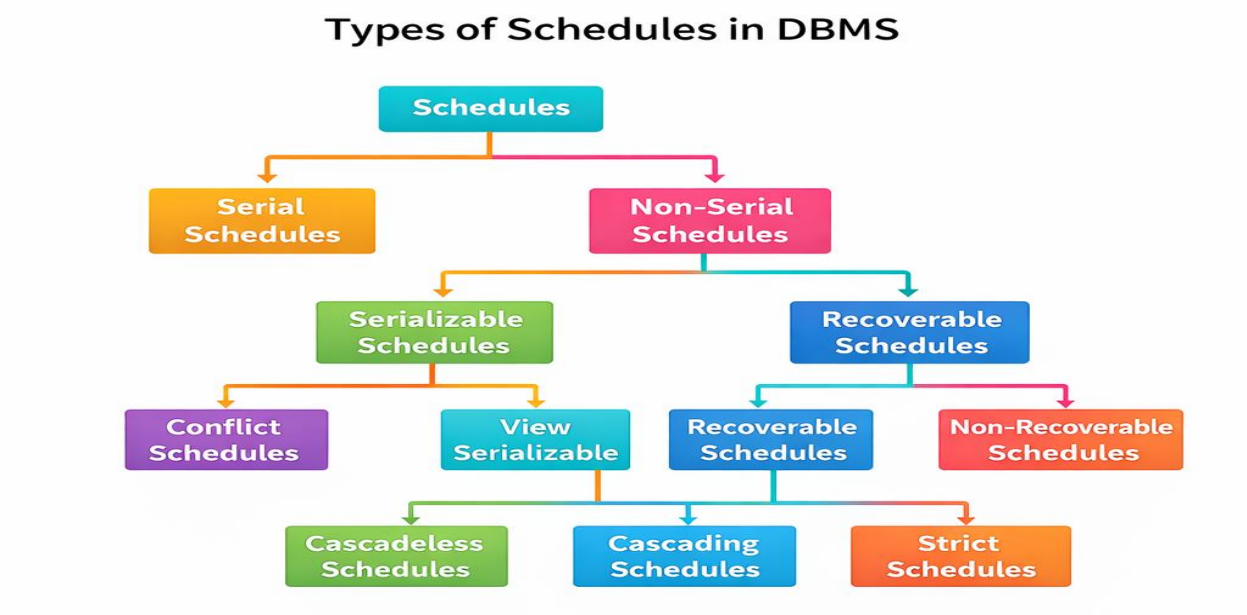
3. Schedules in DBMS

In a Database Management System (DBMS), many users may use the database at the same time. Each user request is handled as a **transaction**. A transaction is a group of operations that read or update data in the database.

When more than one transaction executes at the same time, the DBMS must decide **the order in which their operations are performed**. This order of execution of transaction operations is called a **schedule**.

A schedule shows how the DBMS arranges the operations of different transactions during execution. The main purpose of a schedule is to ensure that the database remains **correct and consistent**, even when multiple transactions run together.

Schedules are important because executing transactions one by one is slow. To improve performance, DBMS allows transactions to run together. However, this may cause conflicts. Therefore, schedules help us analyze whether concurrent execution produces correct results or not.



Types of Schedules

Based on the way transactions are executed, schedules are mainly divided into:

1. **Serial Schedules**
2. **Non-Serial Schedules**

3.1. Serial Schedules

A **serial schedule** is a schedule in which transactions are executed **one after another**. In this type of schedule, one transaction must complete all its operations before the next transaction starts. There is **no overlapping or interleaving** of operations.

Because transactions do not run at the same time, serial schedules are **easy to understand** and always maintain database consistency. Since only one transaction accesses the database at a time, there is **no chance of data conflict**.

However, serial schedules are **not efficient** in real database systems. While one transaction is running, other transactions must wait. This leads to poor utilization of system resources such as CPU and memory. Hence, serial schedules are mainly used for **theoretical understanding**, not for practical execution.

Example of Serial Schedule

Consider two transactions:

- **Transaction T1** updates account **A**
- **Transaction T2** updates account **B**

In a serial schedule, the DBMS executes **T1 completely first**, and only after T1 finishes, **T2** starts.

Order of execution:

T1: Read(A) → Write(A) → Commit

T2: Read(B) → Write(B) → Commit

Here, all operations of T1 are completed before T2 begins.

Therefore, this execution order is called a **serial schedule**.

3.2. Non-Serial Schedules

A **non-serial schedule** is a schedule in which the operations of **two or more transactions are executed at the same time**. In this type of schedule, one transaction does **not** need to finish completely before another transaction starts. Instead, the DBMS **mixes (interleaves)** the operations of different transactions.

Non-serial schedules are used to **improve system performance**. By allowing transactions to run together, the DBMS can reduce waiting time and make better use of system resources such as CPU and memory.

However, because transactions execute together, non-serial schedules may cause **data inconsistency** if proper control is not used. Therefore, non-serial schedules must follow certain rules like **serializability** and **recoverability** to ensure correct results.

Example of Non-Serial Schedule

Consider two transactions:

- **Transaction T1** updates account **A**
- **Transaction T2** updates account **B**

In a non-serial schedule, the DBMS allows both transactions to execute together by interleaving their operations.

Order of execution:

T1: Read(A)
T2: Read(B)
T1: Write(A)
T2: Write(B)
Commit T1
Commit T2

Here:

- T1 starts first
- Before T1 finishes, T2 also starts
- Operations of T1 and T2 are mixed

This execution order is called a **non-serial schedule**.

3.2.1 Serializable Schedules

As studied earlier, **non-serial schedules** allow transactions to execute concurrently in order to improve system performance. However, not all non-serial schedules give correct results. Some schedules may cause **data inconsistency** due to improper interleaving of operations.

To overcome this problem, the DBMS uses the concept of **serializable schedules**.

A **serializable schedule** is a non-serial schedule whose **final result is the same as the result of some serial schedule**. Although transactions are executed at the same time, the effect on the database is exactly the same as if the transactions were executed one after another.

In simple words, a serializable schedule **behaves like a serial schedule in terms of result**, but **works faster** because transactions run concurrently.

Serializable schedules are considered **correct and safe schedules** in DBMS because they maintain database consistency while allowing concurrency.

Example of Serializable Schedule

Consider two transactions:

- **Transaction T1** updates account **A**
- **Transaction T2** updates account **B**

Since T1 and T2 operate on **different data items**, their operations do not conflict.

Non-Serial Execution:

T1: Read(A)
T2: Read(B)
T1: Write(A)

T2: Write(B)
Commit T1
Commit T2
This schedule is **non-serial** because the operations of T1 and T2 are interleaved.
However, the final result is the same as executing the transactions in the following **serial order**:
T1 → then → T2
Therefore, this non-serial schedule is called a **serializable schedule**.

Classification of Serializable Schedules

Serializable schedules are mainly classified into two types:

1. **Conflict Serializable Schedules**
2. **View Serializable Schedules**

1 Conflict Serializability:

A **non-serial schedule** is said to be **conflict serializable** if it can be converted into a **serial schedule** by repeatedly swapping **non-conflicting operations**.
The final serial schedule obtained after swapping must produce the **same result** as the original schedule.

In simple terms, if the order of conflicting operations is preserved and only non-conflicting operations are swapped, then the schedule is conflict serializable.

Conflicting Operations

Two operations are called as **conflicting operations** if all the following conditions hold true for them:

- Both the operations belong to different transactions
- Both the operations are on the same data item
- At least one of the two operations is a **write** operation

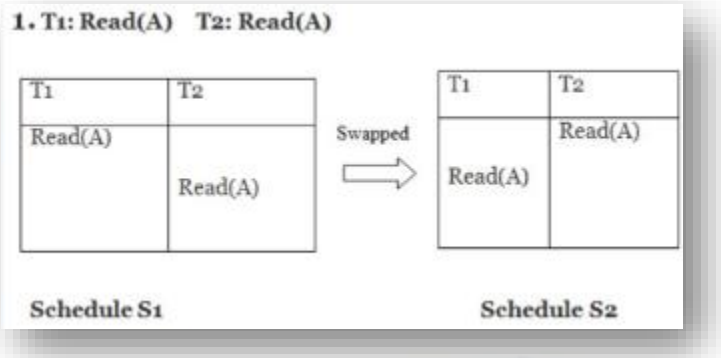
There are three types of conflicting operations:

- **Write-Read (WR) conflict:** Wi(X) and Rj(X)
- **Read-Write (RW) conflict:** Ri(X) and Wj(X)
- **Write-Write (WW) conflict:** Wi(X) and Wj(X)

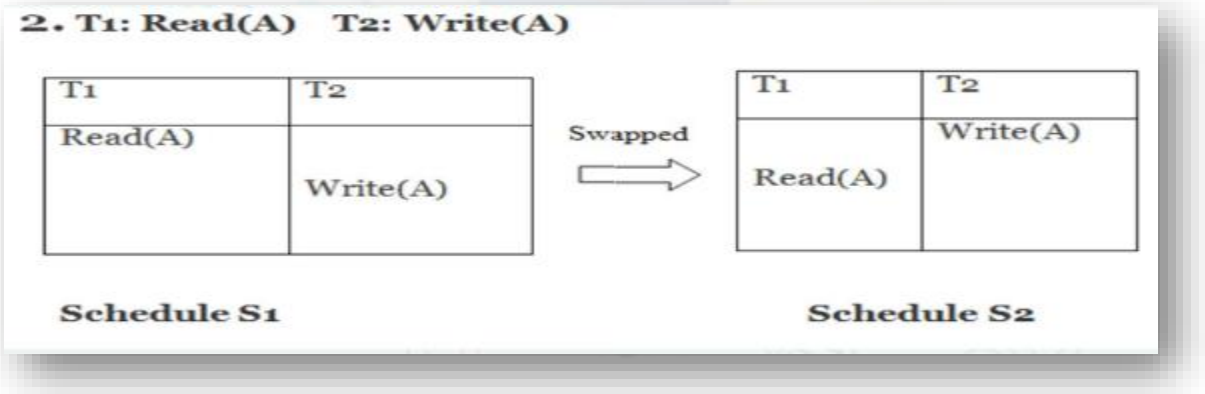
Operations that are not conflicting (and can be swapped) are:

- Read-Read (RR) operations on the same data item.
- Any operations on different data items.

- Any operations within the same transaction



Here S1 and S2 same that means non conflict



Here S1 not equal S2 that means it is conflict

Conflict Equivalent

In the **conflict equivalent**, one can be transformed to another by swapping **non-conflicting operations**. In the given example, S2 is **conflict equivalent** to S1 (S1 can be converted to S2 by swapping **non-conflicting operations**).

Two schedules are said to be conflict equivalent if and only if:

1. They contain the same set of the transaction.
2. If each pair of conflict operations are ordered in the same way.

Non-serial schedule

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

Schedule S1

Serial Schedule

T1	T2
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)

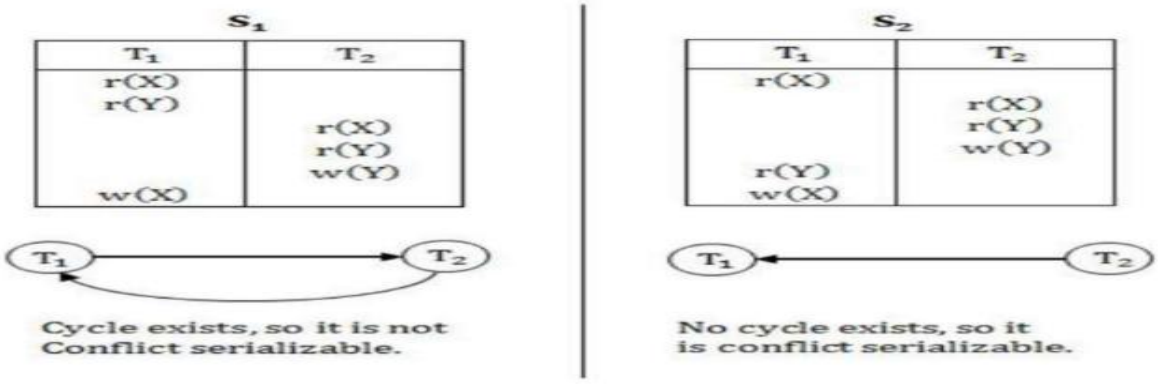
Schedule S2

Schedule S2 is a **serial schedule** because, in this, all operations of **T1** are performed before starting any operation of **T2**. Schedule S1 can be transformed into a serial schedule by swapping non-conflicting operations of **S1**.

S1 is Conflict Serializable

After swapping of non-conflict operations, the schedule S1 becomes:

T1	T2
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)



2. View Serializability

We can say that a schedule is **Conflict-Serializable** (means serial and consistent) iff its corresponding **precedence graph** does not have any loop/cycle.

There may be some schedules that are not **Conflict-Serializable** but still gives a consistent result because the concept of **Conflict-Serializability** becomes limited when the **Precedence Graph** of a schedule contains a **loop/cycle**. In such a case we cannot predict whether a schedule would be consistent or inconsistent.

So, to address such cases we brought the concept of **View-Serializability** because we did not want to confine the concept serializability only to **Conflict-Serializability**.

A schedule is said to be **View-Serializable** if it is **view equivalent** to a **Serial Schedule** (where no interleaving of transactions is possible).

View Serializability: A schedule is called **view serializable** if it is **view equivalent** to a **serial schedule** (**no overlapping transactions**).

How to check view serializability:

First of all, check whether the given schedule is **Non-Conflict Serializable** or **Conflict-Serializable** –

- If the given schedule is **conflict serializable** (means its **precedence graph** does **not contain any loop/cycle**), then the given schedule **must be a view serializable**. **Stop and submit your final answer.**
- If the given schedule is **non-conflict serializable**, then it may or may not be **view serializable**. We cannot predict it just by using the concept of **conflict serializability**, so we need to look at the below cases.

- **If no blind write exists**, then the schedule must be a **Non-View-Serializable schedule**. **Stop and submit your final answer.**

  [Note :Blind write: Performing the **Writing** operation (**updation**), without **reading** operation, such write operation is known as a **blind write**.]

- **If there exists any blind write**, then, in that case, the schedule may or may not be **view serializable**. So we need to look at the below cases. Because, if it does not contain any **blind-write**, we can surely state that the schedule would not be **View-Serializable**.
- **If the above two conditions do not work** (means we have tried the above 2 conditions, then we have come to this step). Then, draw a **precedence graph** using those

dependencies. If no **cycle/loop** exists in the graph, then the schedule would be a **View-Serializable** otherwise **not**.

Conditions for View Equivalence

Two schedules S1 and S2 are said to be **view-equivalent** if below conditions are satisfied:

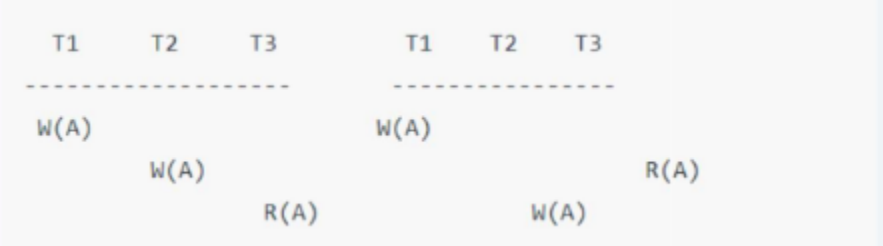
1) Initial Read:

If a **transaction T1 reading data item A** from database in **S1** then in **S2** also **T1 should read A from database**.

Transaction T2 is reading A from database

2) Updated Read

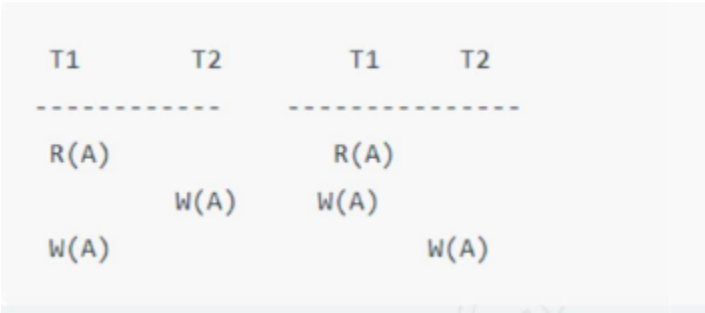
If **Ti is reading A** which is **updated by Tj in S1** then in **S2 also Ti should read A which is updated by Tj**.



Above two schedules are **not view-equivalent** as in **S1**: **T3 is reading A updated by T2**, in **S2 T3 is reading A updated by T1**.

3) Final Write operation

If a **transaction T1 updated A at last in S1**, then in **S2 also T1 should perform final write operations**.



Above two schedules are **not view-equivalent** as **Final write operation in S1 is done by T1** while in **S2 done by T2**.

Schedule S1 |

T1	T2	T3
Read(A) Write(A)	Write(A)	Write(A)

T1	T2	T3
Read(A) Write(A)	Write(A)	Write(A)

Step 1: final updation on data items
In both schedules S and S1, there is **no read** except the **initial read** that's why we don't need to check that condition.
Step 2: Initial Read
The **initial read operation in S is done by T1** and in **S1, it is also done by T1**.
Step 3: Final Write
The **final write operation in S is done by T3** and in **S1, it is also done by T3**.

So, **S and S1 are view Equivalent**.

The first schedule **S1 satisfies all three conditions**, so we don't need to check another schedule.
Hence, **view equivalent serial schedule is: T1 → T2 → T3**

3.2.2 Non-Serializable Schedule

A **non-serializable schedule** is a non-serial schedule that **cannot be transformed into any serial schedule**. The result produced by such a schedule is **not equivalent** to the result of executing the same transactions in any serial order.

In a concurrent database system, operations of multiple transactions are often **interleaved** to improve system performance. However, concurrency is considered correct only when the outcome of execution is the same as that of **some serial execution**.

If the interleaving of operations leads to a result that **differs from all possible serial orders**, then the schedule is called **non-serializable**. Such schedules violate the **isolation property** of transactions and may leave the database in an **incorrect or inconsistent state**.

Classification of Non-Serializable Schedules

Non-serializable schedules are classified into the following types:

1. **Recoverable Schedules**
2. **Non-Recoverable Schedules**

Characteristics of Non-Serializable Schedules

- Operations of multiple transactions are interleaved
- The schedule cannot be rearranged to any serial order
- Conflicting operations occur in an improper sequence
- May lead to data inconsistency and incorrect results

Example of a Non-Serializable Schedule

Consider two transactions **T1** and **T2** operating on the same data item **A**.

Transactions

T1

- Read(A)
- A = A + 10
- Write(A)

T2

- Read(A)
- A = A × 2
- Write(A)

Assume the **initial value of A = 10**.

Non-Serializable Schedule S

Step	Transaction	Operation	Value of A
1	T1	Read(A)	10
2	T2	Read(A)	10
3	T1	Write(A = 20)	20
4	T2	Write(A = 20)	20

Final value of A = 20

Check Against Serial Schedules

Serial Order 1: T1 → T2

- T1: A = 10 + 10 = 20
- T2: A = 20 × 2 = 40

Final value = 40

Serial Order 2: T2 → T1

- T2: A = 10 × 2 = 20
- T1: A = 20 + 10 = 30

  Final value = 30

Conclusion from the Example

- Final value from schedule S = **20**
- Final values from serial schedules = **40 or 30**
- The result of schedule S **does not match any serial execution**

Therefore, **schedule S is a non-serializable schedule**.

Non-serializable schedules may lead to **inconsistency** in the database.

3.2.2.1. Recoverable Schedules

A recoverable schedule is a schedule in which a transaction commits only after all transactions whose data it has read have committed. This ensures that if a transaction depends on another transaction, it does not commit before the transaction it depends on.

In other words, if transaction T1 reads a value written by transaction T2, then T1 must commit after T2 commits.

In a concurrent database system, transactions may read data values produced by other transactions. If a transaction commits before the transaction that wrote the data it read commits, and the writing transaction later fails, the database may enter an inconsistent state.

Recoverable schedules prevent this situation by enforcing a commit dependency rule. If the writing transaction fails, the reading transaction can also be rolled back, thereby maintaining database consistency.

Why Recoverable Schedules Are Needed

- To prevent committing incorrect or uncommitted data
- To ensure safe recovery after transaction failures
- To maintain database consistency

Example of a Recoverable Schedule

Time	Transaction
1	T1: Write(A = 50)
2	T2: Read(A)
3	T1: Commit
4	T2: Commit

- *Transaction T2 reads the value of A written by T1.*
- *T2 commits only after T1 commits.*

- *If T1 fails before committing, T2 can be rolled back.*
- *Hence, the schedule is recoverable.*

Types of Recoverable Schedules

Recoverable schedules are classified into three types:

1. ***Cascading Schedule***
2. ***Cascadeless Schedule***
3. ***Strict Schedule***

a) Cascading Schedule

A cascading schedule is a type of recoverable schedule in which a transaction reads data written by another transaction before it commits. If the writing transaction fails and is rolled back, all transactions that read its uncommitted data must also be rolled back. This chain of rollbacks is called a cascading rollback.

In a cascading schedule, transactions are allowed to read uncommitted (dirty) data from other transactions. Although the schedule is recoverable, it can lead to multiple rollbacks if one transaction fails. This makes recovery complex and costly.

Example

Time	Transaction
1	T1: Write(A = 50)
2	T2: Read(A) → reads 50 (uncommitted)
3	T1: Rollback

Explanation of Example

- Transaction T2 reads a value written by T1 before T1 commits.
- When T1 fails and rolls back, its update to A is undone.
- Since T2 read invalid data, T2 must also roll back.
- This causes a cascading rollback.

Problems with Cascading Schedules

- Multiple transactions may be rolled back
- Recovery becomes complicated
- System performance is reduced

Prevention

- Cascading schedules can be prevented using cascadeless schedules or strict schedules.
- Strict Two-Phase Locking (Strict 2PL) avoids cascading rollbacks by not allowing reads of uncommitted data.

b) Cascadeless Schedule

A cascadeless schedule is a type of recoverable schedule in which a transaction is allowed to read a data item only after the transaction that last wrote that data item has committed. As a result, transactions never read uncommitted (dirty) data.

In a cascadeless schedule, the DBMS ensures that read operations occur only on committed values. Since no transaction reads uncommitted data, the failure of one transaction does not force other transactions to roll back. This completely eliminates cascading rollbacks, making recovery simpler and more efficient**.**

Example

Time	*Transaction*
1	*T1: Write(A = 50)*
2	*T1: Commit*
3	*T2: Read(A) → reads 50 (committed)*

Explanation of Example

- Transaction T2 reads data item A only after T1 commits.
- If T1 were to fail before commit, T2 would not be allowed to read A.
- Therefore, no transaction depends on uncommitted data.

Advantages of Cascadeless Schedules

- Prevents cascading rollbacks
- Simplifies recovery process
- Ensures higher data consistency

c) Strict Schedule

A strict schedule is a type of recoverable schedule in which a transaction is not allowed to read or write a data item until the transaction that last wrote that data item has committed or rolled back.

Strict schedules enforce stronger rules than cascadeless schedules. They prevent both reading and overwriting of uncommitted data. This ensures that no transaction ever accesses a data item that is in an unstable or temporary state.
Because of this restriction, strict schedules completely avoid dirty reads, lost updates, and cascading rollbacks, making recovery simple and reliable.

Example

Time	Transaction
1	T1: Write(A = 50)
2	T2: Read(B) (no conflict with A)
3	T1: Commit
4	T2: Write(A = 60)

Explanation of Example

- *Transaction T2 is not allowed to read or write A until T1 commits.*
- *After T1 commits, T2 safely writes A.*
- *The schedule is strict because access to A is delayed until the previous writer completes.*

Advantages of Strict Schedules

- Prevents dirty reads and lost updates
- Avoids cascading rollbacks
- Simplifies recovery
- Most preferred in practical DBMS implementations

3.2.2.2 Non-Recoverable Schedule

A non-recoverable schedule is a schedule in which a transaction commits after reading data written by another transaction that has not yet committed. If the writing transaction later fails and rolls back, the committed transaction cannot be undone, leading to permanent inconsistency in the database.

In a non-recoverable schedule, a transaction is allowed to read uncommitted (dirty) data and commit before the transaction that produced that data commits. If the writing transaction later aborts, the database contains results based on invalid data, and recovery becomes impossible.

Such schedules violate database correctness and are never allowed in DBMS.

Example

Time	Transaction
1	T1: Write(A = 50)
2	T2: Read(A) → reads 50 (uncommitted)
3	T2: Commit
4	T1: Rollback

Explanation of Example

- Transaction T2 reads data written by T1 before T1 commits.
- T2 commits while T1 is still uncommitted.

- When T1 rolls back, its update is undone.
- Since T2 has already committed using invalid data, the database becomes inconsistent.

Problems with Non-Recoverable Schedules

- Causes permanent database inconsistency
- Recovery is not possible
- Violates transaction atomicity and isolation

Prevention

- Prevented by enforcing recoverable schedules
- Cascadeless and strict schedules completely avoid this problem
- Strict Two-Phase Locking (Strict 2PL) prevents non-recoverable schedules

4.Locking Protocols

4.1 Locking-Based Protocols

Locking-based protocols are a type of **concurrency control mechanism** used in a Database Management System (DBMS). These protocols control the simultaneous execution of transactions by using **locks** on data items. A transaction must acquire the appropriate lock before accessing a data item, and the lock must be released after the operation is completed.

The main purpose of locking-based protocols is to **prevent conflicts among transactions** and ensure **data consistency and correctness** in a multi-user environment.

- A **lock** is a mechanism that restricts access to a data item.
- When a transaction holds a lock on a data item, other transactions may be restricted from accessing that item.
- Locks ensure that conflicting operations (read–write or write–write) do not occur at the same time.

1. Shared (S) Lock: A **Shared Lock (S Lock)** is used when a transaction wants to **read** a data item. It allows multiple transactions to read the same data item at the same time. The main purpose of a shared lock is to **permit concurrent reading** of data without allowing any modification.

Rules of Shared Lock

- A transaction must acquire a shared lock before reading a data item.
- Multiple transactions can hold shared locks on the same data item simultaneously.
- A transaction holding a shared lock **cannot write** to the data item.
- Shared locks are compatible with other shared locks.
-

Example: Shared Lock

Two transactions, **T1** and **T2**, want to read data item **A**.

Transaction	Operation
T1	S-Lock(A), Read(A)

T2	S-Lock(A), Read(A)
T1	Release(A)
T2	Release(A)

- *Both T1 and T2 request a shared lock on data item A.*
- *Since shared locks allow multiple readers, both locks are granted.*
- *T1 and T2 read the same value of A at the same time.*
- *No conflict occurs because neither transaction modifies the data.*
- *This improves system performance by allowing parallel read operations.*

Importance of Shared Lock

- Increases concurrency by allowing multiple read operations
- Prevents accidental modification of data during reading
- Ensures data consistency while improving efficiency

2. Exclusive (X) Lock: An **Exclusive Lock (X Lock)** is used when a transaction wants to **write or update** a data item. It provides exclusive access to the data item.The purpose of an exclusive lock is to **prevent all other transactions from reading or writing** a data item while it is being modified.

Rules of Exclusive Lock

- A transaction must acquire an exclusive lock before writing a data item.
- Only one transaction can hold an exclusive lock on a data item at any time.
- While an exclusive lock is held, no other transaction can read or write the data item.
- Exclusive locks are not compatible with any other locks.

Example: Exclusive Lock

Two transactions, **T1** and **T2**, want to write data item **A**.

Transaction	Operation
T1	X-Lock(A), Write(A = 50)
T2	Wait (cannot acquire X-Lock)
T1	Release(A)
T2	X-Lock(A), Write(A = 60)
T2	Release(A)

- *T1 acquires*
- *an exclusive lock on A and updates its value.*
- *T2 requests an exclusive lock but must wait because T1 already holds it.*
- *After T1 releases the lock, T2 acquires the exclusive lock and performs its update.*
- *This ensures that no other transaction reads or writes* ***uncommitted or partial data****.*

Importance of Exclusive Lock

- Prevents dirty reads
- Avoids lost updates
- Ensures correctness of write operations

- Maintains database integrity

4.2 Two-Phase Locking (2PL)

Two-Phase Locking (2PL) is a **lock-based concurrency control protocol** used in a Database Management System (DBMS) to control the execution of concurrent transactions. Its main objective is to ensure that the **final result of concurrent transactions is correct and consistent**, as if the transactions were executed one after another in some serial order.

2PL is one of the most important protocols because it guarantees **conflict serializability**, which is essential for maintaining database correctness in a multi-user environment.

Why Two-Phase Locking Is Required

When multiple transactions execute at the same time, problems such as: may occur.

- Lost updates
- Dirty reads
- Uncommitted data access
- Inconsistent data

Simple locking alone is not sufficient to prevent all these issues.

Two-Phase Locking introduces rules on when locks can be acquired and released, which helps prevent such anomalies and ensures safe concurrent execution.

Role of Two-Phase Locking

Two-Phase Locking solves these issues by introducing strict rules for lock handling:

- A transaction must first acquire all required locks before releasing any lock.
- Once a transaction releases a lock, it cannot acquire any new lock.

This divides the transaction into:

- *A **growing phase** (lock acquisition)*
- *A **shrinking phase** (lock release)*

1. Growing Phase (Lock Acquisition Phase)

- The transaction **acquires all the locks it needs**
- Locks may be **shared (S)** or **exclusive (X)**
- **No lock is released** during this phase
- The number of locks held by the transaction only increases

This phase prepares the transaction for safe execution by securing required data items.

2. Shrinking Phase (Lock Release Phase)

- The transaction **starts releasing locks**
- **No new lock can be acquired** in this phase
- Once a lock is released, the transaction cannot request any more locks

This phase completes the transaction and frees resources for other transactions.

Types of Two-Phase Locking (2PL) Protocols

Two-Phase Locking (2PL) can be implemented in different forms depending on **when locks are released**. These variations control the level of isolation and recovery support provided by the DBMS.

The main types of Two-Phase Locking are:

1. ***Basic Two-Phase Locking (Basic 2PL)***
2. ***Strict Two-Phase Locking (Strict 2PL)***
3. ***Rigorous Two-Phase Locking (Rigorous 2PL)***

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

- All locks are **acquired during the growing phase**.
- Locks are **released during the shrinking phase**, even **before the transaction commits**.
- Once a lock is released, no new lock can be acquired.

Basic 2PL ensures **conflict serializability**, but it does **not guarantee recoverability**.

Issue with Basic 2PL

The main problem with Basic 2PL is that it allows other transactions to read uncommitted data.

- Dirty reads may occur.
- Cascading rollbacks may happen.
  (Meaning of Cascading RollbackA **cascading rollback** occurs when the rollback of one transaction forces other dependent transactions to also roll back because they have read uncommitted data.)
- Recovery becomes complex and costly

Example: Cascading Rollback in Basic 2PL

Time	Transaction	Operation
1	T1	X-Lock(A), Write(A = 50)
2	T2	S-Lock(A), Read(A = 50) (uncommitted)
3	T1	Rollback
4	T2	Must Rollback

- *T1 acquires an exclusive lock on A and updates its value to 50.*
- *T1 releases the lock during the shrinking phase, even though it has not committed yet.*
- *T2 acquires a shared lock on A and reads the value 50, which is uncommitted.*

- *T1 fails and rolls back, restoring A to its original value.*
- *Since T2 read incorrect (uncommitted) data, it must also roll back.*
- *This rollback of T2 occurs because of T1's failure, creating a cascading effect.*
- *This chain reaction of rollbacks is called cascading rollback.*

2.Strict Two-Phase Locking (Strict 2PL)

Strict Two-Phase Locking (Strict 2PL) is an improved and more reliable version of the basic Two-Phase Locking protocol. It is a lock-based concurrency control protocol that not only guarantees conflict serializability but also ensures recoverability of transactions.

Strict 2PL is widely used in real-world database systems because it prevents dirty reads and cascading rollbacks, which are major problems in Basic 2PL.

Rules of Strict Two-Phase Locking

Strict 2PL follows all the rules of Two-Phase Locking, with an additional restriction:

- All locks are acquired during the growing phase.
- Locks are released during the shrinking phase.
- All exclusive (X) locks are held until the transaction commits or rolls back.
- Shared (S) locks may be released earlier.
- Once a lock is released, no new lock can be acquired.

Because exclusive locks are held until commit, other transactions cannot read or write uncommitted data

Key Feature of Strict 2PL

The most important feature of Strict 2PL is:

A transaction never releases its exclusive locks until it finishes (commit or rollback).

This ensures that:

- Uncommitted data is never accessed by other transactions.
- The database remains consistent during concurrent execution.

Advantages of Strict Two-Phase Locking

Strict 2PL provides the following advantages:

- Prevents dirty reads
- Avoids cascading rollbacks
- Guarantees conflict serializability
- Guarantees recoverability
- Simplifies the recovery process
- Widely used in practical DBMS implementations

Example: Strict 2PL Preventing Cascading Rollback

Transactions T1 and T2

Time	Transaction	Operation
1	T1	X-Lock(A), Write(A = 50)
2	T2	Requests S-Lock(A) → Wait
3	T1	Commit, Release X-Lock(A)
4	T2	S-Lock(A), Read(A = 50)

- *Transaction T1 acquires an exclusive lock on data item A and updates its value to 50.*
- *Because Strict 2PL is used, T1 holds the exclusive lock until commit.*
- *Transaction T2 requests a shared lock on A but must wait until T1 completes.*
- *T1 commits successfully and releases the exclusive lock.*
- *T2 then acquires the shared lock and reads committed data only.*

Since T2 reads only committed data, no cascading rollback occurs.

Limitations of Strict Two-Phase Locking

Although Strict 2PL is safer, it still has some drawbacks:

- Deadlocks may occur when transactions wait for each other's locks
- Reduced concurrency due to longer lock holding
- Lock management overhead

However, these disadvantages are acceptable in most systems because data correctness is more important than maximum concurrency.

3. Rigorous Two-Phase Locking (Rigorous 2PL)

Rigorous Two-Phase Locking is the **most strict and conservative form** of the Two-Phase Locking protocol used for concurrency control in DBMS. It provides the **highest level of isolation** by enforcing very strict rules on when locks can be released.

Rule

- All locks, including both shared (S) and exclusive (X) locks, are held until the transaction commits or rolls back.
- No lock is released during transaction execution.
- All locks are released together only after the transaction completes.

Advantages

- Provides the **simplest and safest recovery mechanism**.
- Completely avoids **dirty reads**.

- Prevents **cascading rollbacks**.
- Ensures very high data consistency and correctness.

Although it is the most restrictive form of 2PL, it is highly reliable.

Example: Rigorous 2PL

Transactions T1 and T2

Time	Transaction	Operation
1	T1	X-Lock(A), Write(A = 50)
2	T1	S-Lock(B), Read(B = 20)
3	T2	Requests X-Lock(A) → **Wait**
4	T1	Commit, release all locks
5	T2	X-Lock(A), Proceeds

- *Transaction **T1** acquires both exclusive and shared locks and holds them throughout its execution.*
- *Transaction **T2** cannot acquire any lock on data item A while T1 is active.*
- *T2 must wait until T1 commits and releases all locks.*
- *After T1 commits, T2 acquires the required lock and proceeds safely.*
- *This guarantees that T2 accesses only **committed and consistent data**.*

4.3 Timestamp-Based Protocols

In **Timestamp-Based Protocols**, each transaction is assigned a **unique timestamp** at the moment it starts execution. This timestamp determines the **priority and execution order** of transactions.

Transactions with **smaller timestamps are older** and are given higher priority than newer transactions.

The protocol ensures that all conflicting operations occur in **timestamp order**, making concurrent execution equivalent to serial execution **without using locks**.

Key Features of Timestamp-Based Protocols

- Each transaction gets a unique timestamp.
- Older transactions have higher priority.
- Ensures **serializability without locking**.
- **Deadlocks are avoided** because transactions never wait.
- Transactions violating timestamp order are **aborted and restarted**.

Key Concepts

1. Timestamps

- Each transaction **T** is assigned a timestamp **TS(T)**.
- Timestamp may be generated using:
    - System clock time, or

- An increasing counter.

- Timestamps create a **total ordering** of transactions.

2. Read and Write Timestamps of Data Items

For each data item **X**, the DBMS maintains:

- **Read Timestamp (RTS(X))**
  → The largest timestamp of any transaction that has read X.
- **Write Timestamp (WTS(X))**
  → The largest timestamp of any transaction that has written X.

These values help decide whether an operation is allowed.

Rules for Read and Write Operations

Read Rule: Read(Ti, X)

A transaction **Ti** is allowed to read data item **X** if:

- **TS(Ti) ≥ WTS(X)** → Read is allowed
- Otherwise → **Ti is aborted and restarted**

If allowed, update:

- RTS(X) = max(RTS(X), TS(Ti))

Write Rule: Write(Ti, X)

A transaction **Ti** is allowed to write data item **X** if:

- **TS(Ti) ≥ RTS(X)** and
- **TS(Ti) ≥ WTS(X)**

If either condition fails:

- The transaction **Ti is aborted and restarted**

If allowed, update:

- WTS(X) = TS(Ti)

Example of Timestamp-Based Protocol

Given:

- **TS(T1) = 10** (older transaction)
- **TS(T2) = 20** (newer transaction)
- Initially: RTS(A) = 0, WTS(A) = 0

Case 1: Normal Execution

Time	Transaction	Operation	Action
1	T1	Write(A = 50)	Allowed, WTS(A) = 10
2	T2	Read(A)	20 ≥ 10 → Allowed, RTS(A) = 20
3	T2	Write(A = 60)	20 ≥ RTS & WTS → Allowed

✓ Execution follows timestamp order.

Case 2: Timestamp Violation

Time	Transaction	Operation	Action
1	T2	Write(A = 60)	TS(T2)=20 < WTS(A)=10 → **Abort**

❌ Transaction is aborted because timestamp order is violated.

Advantages of Timestamp-Based Protocols

- **No deadlocks** (transactions never wait)
- Guarantees **serializability**
- Simple logic for concurrency control
- Suitable for systems with frequent read operations

Disadvantages of Timestamp-Based Protocols

- **Frequent rollbacks** if conflicts are high
- Wasted computation due to aborted transactions

4.4 Validation-Based (Optimistic) Protocols

Validation-Based Protocols, also known as Optimistic Concurrency Control (OCC), are a type of concurrency control mechanism used in DBMS. These protocols are based on the assumption that conflicts among transactions occur rarely. Therefore, transactions are allowed to execute without locking data items, and correctness is checked only at the time of commit.

Unlike lock-based protocols, optimistic methods do not prevent conflicts in advance. Instead, they detect conflicts after transaction execution and allow a transaction to commit only if it does not violate serializability.

- Each transaction executes freely without acquiring locks.
- Updates are performed on a private workspace, not directly on the database.
- Before committing, the transaction is validated to check for conflicts.
- If validation succeeds, the transaction commits.
- If validation fails, the transaction is aborted and restarted.

This approach increases concurrency and avoids unnecessary blocking.

1. Read Phase (Execution Phase)

The transaction executes normally and performs read and write operations.

- Data items are read from the database.
- All write operations are performed locally in a private workspace.
- The actual database is not modified.

- No locks are acquired.

This phase allows maximum concurrency because transactions do not block each other.

2. Validation Phase (Commit-Time Check)

Before committing, the DBMS checks whether the transaction conflicts with other transactions that have already committed.
To ensure that committing this transaction will not violate serializability.

What Is Checked:

- Whether the data read by the transaction was modified by others.
- Whether the transaction's writes conflict with others' reads or writes.

Outcome:

- If validation passes → transaction is allowed to commit.
- If validation fails → transaction is aborted and restarted.

This phase decides the success or failure of the transaction.

3. Write Phase (Update Phase)

If validation is successful:

- All changes from the private workspace are written to the database.
- The transaction commits.

If validation fails:

- The transaction is rolled back.
- It restarts with a new timestamp.

Validation Rules

Let Ti be the transaction being validated and Tj be a transaction that committed earlier.
Transaction Ti can commit if any one of the following conditions is true:

1. Ti completes before Tj starts
   → No overlap, so no conflict.
2. Ti reads and writes data items that Tj did not write
   → Ti is independent of Tj.
3. Ti writes data items that Tj did not read
   → No read–write conflict.

If none of these conditions hold, a conflict exists and Ti must abort.
Example

Transactions T1 and T2:

Transaction	Operation
T1	Read(A), Read(B), Write(A=50)
T2	Read(B), Write(B=30)

In this example, two transactions T1 and T2 execute concurrently using the validation-based (optimistic) concurrency control protocol. During the read phase, both transactions read the required data items and perform their updates in a private workspace without using locks. Transaction T1 reads data items A and B and prepares to update A, while transaction T2 reads B and prepares to update B. In the validation phase, the DBMS checks for conflicts between the transactions. Since T1 updates only A and T2 updates only B, there is no conflict between their read and write sets. Therefore, both transactions pass validation. In the write phase, T1 writes A = 50 and T2 writes B = 30 to the database and commit successfully. This example shows that transactions can execute concurrently without conflicts when they access different data items.

5.Deadlock in DBMS

5.1 Dead Lock:

A deadlock in a Database Management System (DBMS) is a situation in which two or more transactions are waiting indefinitely for each other to release locks, and none of them can proceed further.

In a deadlock, every transaction in the waiting cycle holds at least one resource and is waiting for a resource held by another transaction in the same cycle.

In a multi-user database environment, transactions often require locks on data items before performing read or write operations. When transactions acquire locks in an improper order, a circular waiting condition may occur. Since no transaction is willing to release its lock until it gets the required lock, all involved transactions remain blocked forever. This situation is known as a deadlock.

Deadlocks do not resolve automatically and must be handled explicitly by the DBMS.

Example of Deadlock

Consider two transactions T1 and T2 and two data items A and B.

Time	Transaction	Operation
1	T1	X-Lock(A)
2	T2	X-Lock(B)
3	T1	Requests X-Lock(B) → Wait
4	T2	Requests X-Lock(A) → Wait

- T1 holds lock on A and waits for B.
- T2 holds lock on B and waits for A.
- Neither transaction can proceed → Deadlock occurs.

5.2 Necessary Conditions for Deadlock

Deadlock occurs only if all four conditions hold simultaneously:

1. *Mutual Exclusion*:

Mutual exclusion means that at least one resource in the system can be held by only one transaction at a time. When a transaction acquires such a resource, no other transaction can use it until the first transaction releases it. This condition is necessary for deadlock because if resources were shareable, transactions would not have to wait for one another.

Example:
If transaction T1 holds an exclusive lock (X-lock) on data item A for writing, transaction T2 cannot access A until T1 releases the lock. Since A cannot be shared, mutual exclusion exists.

2. *Hold and Wait*:

The hold and wait condition occurs when a transaction already holds one or more resources and requests additional resources that are currently held by other transactions. While waiting for the new resource, the transaction continues to hold its existing resources, which may block other transactions.

Example:
Transaction T1 holds a lock on data item A and requests a lock on data item B. At the same time, T2 holds a lock on B. T1 waits for B while still holding A, creating a hold-and-wait situation.

3. No Preemption

No preemption means that a resource cannot be forcibly taken away from a transaction. A resource can be released only voluntarily by the transaction after it finishes using it. This condition contributes to deadlock because waiting transactions cannot obtain resources unless the holding transaction releases them.

Example:
If transaction T1 has acquired a lock on data item A, the DBMS cannot forcibly remove that lock and give it to T2. T2 must wait until T1 releases the lock, which enforces the no-preemption condition.

4. Circular Wait

Circular wait occurs when a cycle of transactions exists such that each transaction is waiting for a resource held by the next transaction in the cycle. This circular dependency causes all transactions in the cycle to wait indefinitely.

Example:
Transaction T1 holds a lock on A and waits for B, transaction T2 holds a lock on B and waits for C, and transaction T3 holds a lock on C and waits for A. This forms a circular chain, resulting in a deadlock.

5.3 Deadlock Handling Techniques in DBMS

A deadlock occurs when two or more transactions wait indefinitely for resources held by each other. Since deadlocks do not resolve automatically, a Database Management System (DBMS) must use specific techniques to handle or control deadlocks. The main deadlock handling techniques are:

1. ***Deadlock Prevention***
2. ***Deadlock Avoidance***
3. ***Deadlock Detection and Recovery***

1. Techniques for Deadlock Prevention

Eliminating the Hold and Wait Condition

The hold and wait condition is one of the necessary conditions for deadlock, where a transaction holds at least one resource while waiting for additional resources. Deadlocks arise because multiple transactions may partially acquire resources and then wait indefinitely for others held by different transactions.

To eliminate this condition, the system enforces a strict resource allocation policy: a transaction must request all the resources it will need before it begins execution. The transaction is allowed to start only if every required resource is available at the same time. If even one required resource is unavailable, the transaction is not granted any resource and is forced to wait.

This approach ensures that a transaction never enters a state where it holds some resources and waits for others, thereby completely removing the possibility of deadlock caused by the hold and wait condition.

In normal execution, transactions often request resources incrementally, depending on the execution path. This incremental approach increases system flexibility but also opens the door to deadlock situations.

By contrast, eliminating the hold and wait condition requires the system to predict and declare all future resource needs in advance. This transforms resource allocation into an all-or-nothing strategy:

- Either the transaction gets all required resources together, or
- It gets none at all and remains in the waiting state.

Because transactions do not retain partial ownership of resources, circular waiting chains cannot form, which directly prevents deadlock

Example: Consider a transaction that requires exclusive access to data items A and B to complete its execution.

- The transaction must request locks on both A and B simultaneously.
- If both locks are available, the transaction acquires them and proceeds.
- If either A or B is already locked, the transaction waits without acquiring any lock.

Since the transaction does not hold any resource while waiting, it cannot block other transactions, and deadlock is avoided.

Advantages

- Completely eliminates deadlocks caused by the hold and wait condition
- Guarantees a deadlock-free execution environment
- Simplifies deadlock handling since detection and recovery mechanisms are not required

Disadvantages

- Leads to poor resource utilization, as resources may remain idle while transactions wait for all required resources to become available
- Transactions may hold resources longer than necessary, even if some resources are not used immediately
- Requires transactions to know all resource requirements in advance, which is often impractical in real-world applications
- Reduces system concurrency, especially in complex or long-running transactions

2.Allowing Resource Pre-emption

The no pre-emption condition is one of the essential conditions required for deadlock. It states that once a resource is allocated to a transaction, the system cannot forcibly take it back until the

transaction voluntarily releases it after completing its execution. This restriction can lead to deadlock when multiple transactions hold resources while waiting for additional ones.

To eliminate this condition, the system introduces resource pre-emption. In this approach, the system is given the authority to forcibly withdraw resources from a transaction under specific conditions. When a transaction that is already holding some resources requests another resource that is currently unavailable, the system pre-empts all resources held by that transaction. The transaction is then rolled back to a safe state and restarted later, when the required resources are available.

This technique prevents deadlock by ensuring that a transaction never remains in a waiting state while holding resources. By forcing the transaction to release its resources, the system breaks the possibility of circular waiting, which is a key requirement for deadlock to occur.

Example

Consider two transactions, T1 and T2:

- T1 holds a lock on data item A.
- T1 then requests a lock on B, which is currently held by T2.

Since B is unavailable, the system forces T1 to release A and places T1 in a waiting or restart state. This allows T2 to continue execution without being blocked, thereby preventing a circular wait between the two transactions.

Advantages

- Helps in breaking deadlock situations at an early stage
- Prevents transactions from holding resources while waiting
- Improves system safety in environments where rollback is possible

Disadvantages

- Causes additional overhead due to frequent rollback and restart of transactions
- May reduce system performance if pre-emption occurs often
- Not all resources can be pre-empted, such as non-recoverable or physical resources
- May lead to starvation if a transaction is repeatedly pre-empted

3. Eliminating Circular Wait Condition (Resource Ordering)

The circular wait condition is one of the necessary conditions for deadlock. It occurs when a set of transactions form a closed chain, where each transaction is waiting for a resource held by the next transaction in the chain. As long as this circular dependency exists, none of the transactions can proceed, resulting in deadlock.

To prevent the circular wait condition, the system enforces a global ordering of resources. Under this approach, every resource in the system is assigned a unique order or priority. Transactions are then required to request resources strictly in increasing order of this predefined sequence and are not allowed to request resources in the reverse order.

This rule ensures that transactions follow a consistent and predictable resource acquisition pattern. Since all transactions move in the same direction when requesting resources, it becomes impossible to form a cycle in the resource allocation graph, thereby completely eliminating circular waiting.

Example

Assume the system defines a global resource order as:

A → B → C

- A transaction is allowed to request A first, then B, and then C.
- The transaction is not allowed to request A after holding B, or B after holding C.

Because all transactions follow the same order, a circular wait among transactions cannot occur.

Advantages

- Completely prevents circular waiting, ensuring deadlock-free execution
- Simple and deterministic approach once the resource order is defined
- **No need for deadlock detection or recovery mechanisms**

Disadvantages

- Requires careful and correct design of the global resource ordering
- May lead to reduced concurrency, as transactions may have to wait for lower-ordered resources even when higher-ordered ones are free
- Not flexible in systems where resource requirements change dynamically

4. Timestamp-Based Prevention (Optional Advanced Method)

Timestamp-based deadlock prevention is an advanced technique in which each transaction is assigned a unique timestamp at the time it enters the system. This timestamp represents the age of the transaction and is used to decide whether a transaction should wait or be rolled back when a resource conflict occurs.

Instead of allowing transactions to wait indefinitely and form deadlocks, the system uses timestamps to control the waiting behavior. When a transaction requests a resource that is currently held by another transaction, the system compares their timestamps and applies a predefined rule to decide the next action.

This approach prevents deadlock by ensuring that circular waiting never forms, as waiting and rollback decisions are strictly governed by transaction age.

Methods Used in Timestamp-Based Prevention

1. Wait–Die Scheme

In the Wait–Die scheme, transaction age determines whether a transaction waits or aborts:

- If an older transaction requests a resource held by a younger transaction, the older transaction is allowed to wait.
- If a younger transaction requests a resource held by an older transaction, the younger transaction is aborted (dies) and restarted later with the same timestamp.

This method prevents deadlock because younger transactions never wait for older ones, eliminating circular waiting.

2. Wound–Wait Scheme

In the Wound–Wait scheme, the rules are slightly reversed:

- If an older transaction requests a resource held by a younger transaction, the older transaction forces the younger one to roll back (wounds it).
- If a younger transaction requests a resource held by an older transaction, the younger transaction is allowed to wait.

This approach favors older transactions and helps them complete quickly.

How Timestamp-Based Prevention Avoids Deadlock

By controlling who waits and who rolls back based on timestamps, the system ensures that:

- Waiting is always unidirectional (based on age)
- Cycles cannot form in the waiting graph
- Deadlocks are completely prevented

Advantages of Deadlock Prevention

- Deadlocks never occur, as all necessary conditions are prevented
- System behavior becomes predictable and controlled
- Recovery mechanisms are simple, as deadlock detection is unnecessary

Disadvantages of Deadlock Prevention

- Reduced concurrency, since strict rules limit resource sharing
- Low resource utilization, as transactions may wait unnecessarily
- Increased waiting and starvation possibilities, especially for younger transactions

2. Deadlock Avoidance in DBMS

Deadlock avoidance is a technique used in a Database Management System (DBMS) to ensure that the system does not enter a deadlock situation. In this approach, the DBMS carefully examines each resource request before granting it. A resource is allocated only if the system remains in a safe state after the allocation.

Unlike deadlock prevention, deadlock avoidance does not strictly restrict how transactions request resources. Instead, it allows flexibility but makes intelligent decisions at runtime to avoid deadlock.

Safe State

A system is said to be in a safe state if there exists at least one order of transaction execution in which all transactions can complete successfully without causing deadlock. This order is known as a safe sequence.

If the system is in a safe state, deadlock will not occur.

Unsafe State:

A system is in an unsafe state if no safe sequence exists. An unsafe state does not mean that a deadlock has already occurred, but it indicates that the system may lead to deadlock if further resource requests are granted.

Deadlock avoidance ensures that the system never moves from a safe state to an unsafe state.

Working of Deadlock Avoidance

In deadlock avoidance, the DBMS must know the maximum resource requirements of each transaction in advance. When a transaction requests a resource, the system temporarily assumes that the resource is allocated and checks whether the resulting state is safe.

- If the system remains safe, the request is granted
- If the system becomes unsafe, the request is postponed

This decision process is repeated for every resource request.

Example

Consider two transactions, T1 and T2, competing for a limited number of resources.

- T1 may need a maximum of 6 units of a resource
- T2 may need a maximum of 4 units of the same resource
- Total available resources = 10 units

Suppose T1 currently holds 4 units and requests 2 more units.

The system checks whether the remaining resources are sufficient for T2 to complete its execution.

- If the check shows that both T1 and T2 can finish in some order, the request is granted
- If not, the request is delayed

By allowing only safe allocations, the system avoids deadlock.

The Banker's Algorithm is a popular method used for deadlock avoidance. It works by checking whether granting a resource request keeps the system in a safe state.

The Banker's Algorithm

The Banker's Algorithm is a deadlock avoidance algorithm used in a Database Management System (DBMS). It is called the Banker's Algorithm because it works in a way similar to a banker who lends money carefully, ensuring that all customers can repay their loans without causing loss.

The main goal of the Banker's Algorithm is to allocate resources safely so that the system never enters a deadlock state.

Before granting a resource request, the system checks whether the allocation will keep the system in a safe state.

If the system remains safe after granting the request, the resource is allocated.

If the system becomes unsafe, the request is postponed.

Thus, the algorithm avoids deadlock by allowing only safe resource allocations.

To apply the Banker's Algorithm, the system maintains the following information:

1. **Available**
   The number of available instances of each resource.
2. **Maximum (Max)**
   The maximum number of resources each transaction may need.
3. **Allocation**
   The number of resources currently allocated to each transaction.
4. **Need**
   The remaining resources required by each transaction.

$$\text{Need} = \text{Max} - \text{Allocation}$$

Safe State and Safe Sequence

- A safe state is a state in which there exists at least one order of transaction execution such that all transactions can complete successfully.
- This order is called a safe sequence.
- If no safe sequence exists, the system is in an unsafe state.

The Banker's Algorithm ensures that the system always remains in a safe state**.**

Working of the Banker's Algorithm

When a transaction requests resources, the system performs the following steps:

1. Check if the requested resources are less than or equal to the transaction's remaining need.
2. Check if the requested resources are available.
3. Temporarily allocate the resources.
4. Check whether the system is still in a safe state.
5. If the state is safe, the allocation is confirmed.
6. If the state is unsafe, the allocation is rolled back and the request is denied.

Example

Assume a system has 10 units of a single resource.

Two transactions: T1 and T2

Transaction	Maximum	Allocated	Need
T1	7	3	4
T2	5	2	3

Available resources = 10 − (3 + 2) = 5
Now, suppose T1 requests 2 more units.

- Requested (2) ≤ Need of T1 (4) ✓
- Requested (2) ≤ Available (5) ✓

After temporary allocation:

- T1 Allocation = 5, Need = 2
- Available = 3

The system checks if T2 can complete with the remaining resources.
Since T2 needs only 3 units and 3 are available, T2 can complete, followed by T1.
Hence, the system is in a safe state, and the request is granted.

6. Database Recovery management in DBMS

6 .1 Database Recovery

Database Recovery is the process of restoring a database to a correct and consistent state after a failure such as a system crash, power failure, disk failure, or software error. The main objective of recovery is to ensure that all committed transactions are permanently reflected in the database, while the effects of uncommitted or incomplete transactions are removed.

In simple terms, recovery guarantees that the database obeys the ACID properties, especially Atomicity and Durability, even in the presence of failures.

Why Database Recovery Is Needed

Failures are unavoidable in real-world systems. When a failure occurs during transaction execution, the database may end up in:

1. **System Crashes:** Power failure, OS crash, or DBMS crash can leave transactions incomplete.
2. **Transaction Failures:** Errors like invalid input, constraint violation, or divide by zero.
3. **Media Failures:** Hard disk or storage failure can corrupt data.
4. **Concurrency Issues:** Multiple transactions executing simultaneously may cause inconsistencies.

Without recovery mechanisms, it becomes impossible to determine which operations were completed correctly and which were not. Recovery ensures that the database always returns to a safe and reliable state.

6.2 Types of Failures That Require Recovery

1. System Crash:

A system crash occurs when the database system stops working due to power failure, operating system crash, or hardware malfunction while transactions are in execution. In this case, the contents of main memory are lost, but the data stored on disk remains safe. Some transactions may have completed, while others may be partially executed.

Example: A transaction updates a customer's balance and the system suddenly shuts down due to power failure. After restart, recovery ensures that only committed updates remain, and incomplete transactions are undone.

3. *Transaction Failure:*

A transaction failure happens when a transaction cannot complete its execution due to logical errors, constraint violations, invalid input, or explicit aborts caused by deadlock or system conditions. Even though the system continues running, the failed transaction must be rolled back to maintain database consistency.

Example: A bank transaction tries to withdraw ₹50,000 from an account that has only ₹20,000. The system detects the error and aborts the transaction, restoring the account balance to its original value.

4. ***Disk Failure:*** Disk failure occurs when there is physical damage or corruption of storage devices, such as hard disk crashes, head crashes, or controller failures. This type of failure is severe because database files may be permanently lost. Recovery in this case requires the use of backups and archived logs to restore the database.

   Example: A database server's hard disk crashes and all stored tables are lost. The database administrator restores the latest backup and applies logs to recover recent committed transactions.

5. *Application Errors:*

Application errors occur due to bugs in application programs, incorrect logic, or improper user input, which may incorrectly modify the database even though the DBMS is functioning correctly. These errors can lead to inconsistent or wrong data and must be corrected using recovery techniques.

Example: A faulty billing program mistakenly deducts charges twice from customer accounts. Recovery is used to undo the incorrect updates and restore the correct balances**.**

6.3 Recovery Techniques

Recovery techniques in a Database Management System (DBMS) are strategies used to restore a database to a correct and consistent state after a failure. Failures can occur due to **hardware problems, software bugs, system crashes, power outages, or human errors**. Recovery mechanisms are essential to maintain **data integrity** and ensure that all completed transactions are saved permanently while incomplete transactions do not affect the database.

A reliable recovery system ensures:

- Committed transactions are permanently recorded.
- Uncommitted or failed transactions are undone.
- The database remains consistent even after system failures.

The commonly used recovery techniques are **Transaction Log, Checkpointing, Backup & Restore, and Shadow Paging**.

1. Transaction Log

A **Transaction Log** is a sequential record of all operations performed by transactions in the database. It is primarily used for **UNDO** (rollback incomplete transactions) and **REDO** (reapply committed transactions) in case of failures. The log stores details such as the transaction start, operations performed, old and new values of data items, and the transaction's commit or abort status.

- Before any change is applied to the database, it is first recorded in the transaction log.
- If a system crash occurs, the log is used to redo committed transactions and undo uncommitted ones.

Example:

- Transaction T1: Debit ₹500 from Account A (original balance ₹1000)
- Transaction T2: Deposit ₹200 into Account B (original balance ₹300)

Log entries:

Transaction	Operation	Old Value	New Value
T1	Debit A	1000	500
T2	Deposit B	300	500

- If the system crashes after T1 debited ₹500 but before commit → **UNDO T1**, restoring Account A to ₹1000.
- If the crash occurs after T1 committed → **REDO T1**, ensuring the debit remains recorded.

2. Checkpointing

Checkpointing is the process of periodically saving the current state of the database. It reduces the amount of work required during recovery by marking a known consistent state. Only transactions executed after the last checkpoint need to be considered during recovery, which speeds up the process.

- At regular intervals, the DBMS writes all modified data from memory to disk.
- A checkpoint record is added to the transaction log.
- During recovery, the system starts from the most recent checkpoint instead of scanning the entire log.

Example:

- At 10:00 AM, the database saves a snapshot of all tables.
- Transactions T1, T2, T3 execute after the checkpoint.
- If the system crashes at 10:30 AM, recovery only needs to process the logs for T1, T2, and T3, not the earlier transactions.

Checkpointing speeds up recovery because the system does not have to scan the entire log file.

3. Backup & Restore

Backup and Restore is a preventive recovery technique designed to protect data from major failures such as disk crashes, cyberattacks, accidental deletion, or natural disasters. A backup is a copy of the database stored securely on another storage device, tape, or cloud. **Restore** involves recovering the database from this backup.

- Full or partial backups are taken at regular intervals (daily, weekly).
- If the main database fails, the backup is restored.
- Any transactions occurring after the last backup are reapplied using the transaction log (**REDO**) to update the database to the most recent state.

Example:

- A bank takes a daily backup of its database at 11:59 PM.
- If the main server crashes the next day, the database can be restored from this backup.

- Any transactions that occurred after the backup and before the crash can be reapplied using the transaction log (**REDO**) to bring the database to the most recent state.

This method protects against permanent data loss caused by hardware failure or disasters.

4. Shadow Paging

Shadow Paging is a recovery method that maintains a shadow copy of the database pages. Updates are applied to the shadow pages first, while the original pages remain unchanged.

- During a transaction, changes are made to shadow pages only.
- If the transaction commits, the shadow pages replace the main database pages.
- If the system crashes before commit, the shadow pages are discarded and the main database remains unchanged.

Example:

- Transaction T1 updates Account A: Balance ₹1000 → ₹800.
- The update is applied to the shadow page only.
- If T1 commits → shadow page replaces main database → Balance A = ₹800.
- If the system crashes before commit → main database still shows ₹1000.

6.3.1 Log-Based Recovery as a Recovery Technique in DBMS

Recovery techniques in a Database Management System (DBMS) are methods used to restore the database to a **correct and consistent state after failures**. Failures may occur due to system crashes, power failures, hardware faults, or software errors. Among the various recovery techniques, **Log-Based Recovery** is one of the **most fundamental and widely used** methods.
Log-Based Recovery belongs to the category of **recovery techniques** because it provides a **systematic and reliable approach** to database recovery using **transaction logs**. By analyzing log records, the DBMS can **undo incomplete transactions** and **redo committed transactions**, thereby ensuring data integrity and consistency.

Log-Based Recovery in DBMS

Log-Based Recovery is a recovery technique in which the DBMS uses **log information** to restore the database after a failure. The core principle behind this technique is:
Every change made to the database must be recorded in a log before it is actually applied.
The log acts as a **permanent and ordered record of transaction activities**, allowing the DBMS to determine what actions were completed and what actions were left unfinished at the time of failure.

Need for Log-Based Recovery

During normal operation, multiple transactions execute concurrently and update database items. If a failure occurs during execution:

- Some transactions may have **completed successfully**
- Some transactions may be **partially executed**
- Some updates may exist only in **main memory**

Without log-based recovery, such failures could leave the database in an **inconsistent state**. Log-based recovery ensures that:

- **Completed (committed) transactions are preserved**
- **Incomplete (uncommitted) transactions do not affect the database**

Log File and Its Role

A **log file** is a sequential file stored on stable storage that records all important transaction-related events. It provides sufficient information to **reconstruct the database state** after a crash.

Information Stored in the Log

Each log record typically contains:

- Transaction identifier
- Operation type (start, update, commit, abort)
- Data item modified
- Old value (before update)
- New value (after update)

The log maintains the **exact order of operations**, which is essential for correct recovery.

Write-Ahead Logging (WAL)

Write-Ahead Logging is a fundamental rule of log-based recovery.

According to WAL:

- A log record describing a change must be written to disk **before** the corresponding database page is written.
- This guarantees that recovery information is never lost, even if a crash occurs immediately after the update.

WAL ensures that the log always reflects the most recent state of transaction execution.

UNDO and REDO Operations

Log-based recovery relies on two basic recovery actions: **UNDO** and **REDO**.

UNDO Operation

UNDO reverses the effects of **uncommitted transactions**.

Using the **before image** stored in the log, the DBMS restores data items to their original values.

REDO Operation

REDO reapplies the effects of **committed transactions**.

Using the **after image**, the DBMS ensures that all committed changes are written to the database.

These two operations together guarantee database consistency.

Types of Log Records

Before Image (UNDO Log)

- Stores the value of a data item before it is modified
- Used to undo incomplete or failed transactions

After Image (REDO Log)

- Stores the value of a data item after modification
- Used to redo committed transactions whose updates were not saved to disk

Recovery Process Using Log-Based Recovery

When a system crash occurs, the DBMS performs recovery in the following steps:

1. The log file is scanned to identify committed and uncommitted transactions.
2. For all committed transactions, REDO operations are applied.
3. For all uncommitted transactions, UNDO operations are applied.
4. The database is restored to a consistent and correct state.

This process ensures that the final database state is equivalent to a serial execution of committed transactions.

Example

Consider two transactions, T1 and T2:

- T1 transfers ₹1,000 from Account A to Account B and commits.
- T2 withdraws ₹500 from Account C but crashes before committing.

During recovery:

- Changes made by T1 are redone using the after image.
- Changes made by T2 are undone using the before image.

Thus, only valid transaction effects remain in the database.

6.4 Recovery Facilities:

Recovery facilities are mechanisms provided by a Database Management System (DBMS) to ensure that the database remains **consistent, reliable, and recoverable** even after failures such as system crashes, disk failures, or software errors. These facilities help restore the database to a correct state by **preserving committed transactions** and **removing incomplete transactions**.

The main recovery facilities in DBMS are:

1. **Backup Facility**
2. **Logging Facility**

1. Backup Facility

The **backup facility** is a fundamental recovery support provided by a Database Management System (DBMS) to protect the database from **permanent data loss**. The core idea behind this facility is to **keep additional copies of the database in a safe place**, so that data can be restored if the original database becomes unavailable.

Conceptually, the backup facility works as a **data insurance mechanism**. It does not stop failures from happening, but it ensures that even after serious failures—such as disk crashes, file corruption, or hardware damage—the database can still be recovered. Without backups, recovery from such failures would be impossible.

Backups can be taken for the **entire database** or for **only the modified portions**, depending on system requirements. Since backups are usually taken at fixed intervals, they may not include the most recent updates. Therefore, backups are commonly used together with **transaction logs** to rebuild the database up to the exact point of failure.

Working of Backup Facility

- The DBMS periodically creates backup copies of the database.
- These copies are stored on separate and secure storage media.
- When a major failure occurs, the system restores the database from the **latest available backup**.
- After restoration, **redo operations using log records** are applied to recover recent committed transactions.

This combination ensures that the database is restored to a **correct and consistent state**.

Example

Consider a company that performs a **daily database backup every night**.

- During the day, many transactions update customer records.
- In the afternoon, a disk crash damages the main database.

Recovery steps:

- The system restores the database using the previous night's backup.
- Transaction logs generated after the backup are applied using redo operations.
- The database is recovered to the state just before the disk failure.

As a result, **no committed data is lost**, even though the original database was damaged.

2. Logging Facility

The **logging facility** is a recovery mechanism provided by a Database Management System (DBMS) to **record all important actions performed by transactions**. Its main purpose is to help the system **restore the database to a correct and consistent state after a failure**.

Conceptually, the logging facility works as a **reliable record of database activity**. Whenever a transaction performs an operation such as updating data, the details of that operation are first written into a special file called a **log**. This information is later used during recovery to determine which changes should be **kept** and which should be **reversed**.

Without a logging facility, the DBMS would not be able to identify the status of transactions at the time of failure, and correct recovery would not be possible.

Log Contains

The log is stored on stable storage and usually records:

- Start of a transaction
- Data modification operations
- Old value of the data item
- New value of the data item
- Commit or abort of a transaction

This information allows the DBMS to track the complete history of transaction execution.

Types of Log Information

Before Image (UNDO Information)

The **before image** stores the **old value of a data item before it is modified** by a transaction.
It is used to **undo changes** made by transactions that did not complete successfully.

After Image (REDO Information)

The **after image** stores the **new value of a data item after modification**.
It is used to **redo changes** made by committed transactions that were not written to disk before a system crash.

Working of Logging Facility

1. Before updating the database, the DBMS writes the operation details into the log.
2. The database update is performed only after logging is completed.
3. If a failure occurs:
    - Changes made by **uncommitted transactions** are undone using before images.
    - Changes made by **committed transactions** are redone using after images.
4. The database is restored to a consistent state.

This process ensures correctness even in the presence of failures.

Example

In a banking system, a transaction updates an account balance from ₹10,000 to ₹8,000.

- The old balance (₹10,000) is written to the log before the update.
- The new balance (₹8,000) is written after the update.

If the system crashes:

- If the transaction was not committed, the balance is restored to ₹10,000.
- If the transaction was committed, the balance is set to ₹8,000.

Recovery facilities like the **backup facility** and **logging facility** give the DBMS the necessary support to handle failures. By using these facilities, the DBMS can save copies of data and record changes made by transactions. **Backup and recovery** explains how these saved copies and records are used to restore the database to a correct and consistent state after a failure.

6.5 Backup & Recovery

Backup and recovery is an important part of database reliability and safety. While security techniques protect a database from unauthorized access, backup and recovery protect data from being lost due to failures such as system crashes, disk failures, human mistakes, or natural disasters.

- Backup means creating a safe copy of the database and storing it in a secure location.
- Recovery means restoring the database to a correct and consistent state using the backup and log information after a failure.

Backup and recovery ensure that committed data is not permanently lost and that the organization can continue its operations even after failures.

The main objectives of backup and recovery are:

1. **Data Protection:** Prevent permanent loss of critical data.
2. **Business Continuity:** Ensure the organization can continue operations after a failure.
3. **Disaster Recovery:** Quickly restore databases in case of hardware crashes, malware attacks, or natural disasters.

6.5.1 Types of Backups

Backing up a database means creating a copy of its data so that it can be restored in case of failure, corruption, or accidental deletion. There are **three main types of backups**—full, incremental, and differential—each with its advantages and use cases.

1. Full Backup

A **full backup** is a complete copy of the entire database, including all tables, records, and system information. This type of backup ensures that, in the event of a failure, the entire database can be restored from a single backup file. The main advantage is simplicity, as recovery is straightforward. However, full backups take **more storage space and time** because they copy every piece of data every time a backup is performed.

Example: A bank performs a full backup of its customer database every night. If the main server crashes the next day, the IT team can restore the database from the backup, ensuring that all accounts, transactions, and balances are recovered without loss.

Full backups are best when storage space is not a limitation and when recovery speed is a priority, as the entire database is available in one file.

2. Incremental Backup

An **incremental backup** saves only the data that has **changed since the last backup**, whether it was a full or another incremental backup. Incremental backups are **faster and use less storage**

compared to full backups. However, recovering the database may take longer because multiple incremental backups must be applied in sequence along with the last full backup.

Example: A retail company takes a full backup of its sales database on Sunday. On Monday, an incremental backup is performed, which includes only the new sales transactions made that day. If a server failure occurs on Wednesday, the IT team restores the full backup from Sunday and then applies the incremental backups from Monday and Tuesday to recover all data.

Incremental backups are efficient in terms of storage and speed but require careful management to ensure all incremental files are available during recovery.

3. Differential Backup

A **differential backup** saves all changes made **since the last full backup**, regardless of any incremental backups taken in between. Differential backups are **larger than incremental backups** but smaller than full backups. The main advantage is that recovery is faster than incremental backups because only the last full backup and the latest differential backup are needed.

A university takes a full backup of its student records database on Sunday. On Tuesday, a differential backup is taken, which includes all changes from Monday and Tuesday. If a failure occurs on Tuesday evening, the IT team can restore the database by applying the full backup from Sunday and the differential backup from Tuesday, avoiding the need to apply multiple incremental backups.

7. Threats

Database security is the practice of protecting a database to ensure the **confidentiality, integrity, and availability (CIA)** of the stored data. Confidentiality ensures that sensitive data is only accessible to authorized users, integrity ensures that the data remains accurate and consistent, and availability ensures that the database is accessible when needed. Database security is essential because databases often store critical information such as exam records, banking data, medical records, and business secrets. Without proper security, data can be stolen, misused, corrupted, or lost, leading to serious consequences for both users and organizations.

Types of Threats:

1. **Unauthorized Access** – Hackers accessing confidential data.
    - Example: A student accesses exam results table without permission.
2. **SQL Injection** – Attacker injects malicious SQL queries.
    - Example: `SELECT * FROM Users WHERE username='admin' OR '1'='1';`
3. **Data Corruption/Loss** – Due to software bugs, hardware failure, or viruses.

- Example: Power failure corrupts sales transaction logs.

4. **Denial of Service (DoS)** – Overloading the system to make it unavailable.
   - Example: Continuous fake login attempts crash the DB server.

7.1 Threats to Database Security

A **threat** is any potential event or action that can harm the database system or compromise its data. Threats can be intentional, such as cyberattacks, or unintentional, like system failures. Understanding these threats is crucial for designing effective security measures. Common types of threats include:

1. Unauthorized Access

Unauthorized access occurs when a user or attacker gains access to data without permission. This can happen due to weak passwords, poor access control, or stolen credentials. **Example:** A student logs into the university database and accesses the exam results table of other students without permission. Such access compromises confidentiality and privacy

.

2. SQL Injection

SQL injection is a type of attack where an attacker inserts malicious SQL code into database queries to manipulate or retrieve sensitive data. This usually happens when applications do not properly validate user inputs.

Example: An attacker enters a query like `SELECT * FROM Users WHERE username='admin' OR '1'='1';` into a login form. This can trick the database into giving unauthorized access to all user accounts, bypassing authentication.

3.Data Corruption or Loss:

Data corruption or loss can occur due to software bugs, hardware failures, human error, or malicious viruses. Corrupted data may become inaccurate or unusable, while lost data can lead to serious operational problems.

Example: A sudden power outage during a sales transaction may corrupt the transaction logs, causing loss of sales records. This affects both the integrity and reliability of the database.

4. Denial of Service (DoS)

A **Denial of Service (DoS)** attack targets database availability by overwhelming the system with excessive requests, making it slow or completely unavailable to legitimate users. DoS attacks can disrupt business operations and reduce trust in the system.

Example: A hacker continuously sends fake login requests to a banking server, overloading it

until the database server crashes. Genuine customers are unable to access their accounts during the attack.

8. Database Security

Database security refers to the measures and methods used to protect a database from unauthorized access, misuse, theft, or damage. The main aim is to ensure the **confidentiality, integrity, and availability** of data. This means only authorized users can access the data, the data remains correct and consistent, and it is always available when needed. Security is very important because databases often store sensitive information like bank accounts, personal details, healthcare records, and company secrets.

1. Authentication

Authentication is the process of checking if a user is really who they claim to be. This step ensures that only legitimate users can access the database. Most systems use **usernames and passwords**, while advanced systems may use **biometric verification** or **two-factor authentication (2FA)** for extra security.

Example: A bank employee enters their username and password to log in to the database. If someone else tries with wrong credentials, access is denied, preventing unauthorized use.

2. Authorization

Authorization controls what an authenticated user can do in the database. It sets rules and permissions for each user, such as reading, updating, or deleting data. Even if a user is logged in, they cannot perform tasks they are not allowed to.

Example: In a hospital, a receptionist can view appointments but cannot see detailed medical records. Only doctors and nurses can access and update patient health information.

3. Encryption

Encryption converts data into a coded form so that unauthorized users cannot read it. Only someone with the correct key can decode and understand the data. Encryption can protect data that is **stored in the database** or **sent over a network**.

Example: When a customer pays online, their credit card information is encrypted before being sent. Even if hackers intercept the data, they cannot read it without the encryption key.

4. Access Control

Access control restricts what users can do based on their role or security policies. It ensures users can only perform actions allowed for their role.

Types of Access Control:

- **Discretionary Access Control (DAC):** Users control access to data they own.
- **Mandatory Access Control (MAC):** Access is controlled by system rules and policies.

Example: In a university database, professors can update grades, students can only see their own results, and staff can manage enrollment records.

5. Auditing

Auditing means monitoring and recording database activity. Audit logs record who accessed the database, what actions they performed, and when. This helps detect unusual or unauthorized activity and provides evidence if a security breach occurs.

Example: If a bank employee accesses multiple customer accounts without permission, audit logs help identify the employee and prevent further unauthorized access.

6. Backup and Recovery for Security

Backup and recovery protect data against accidental deletion, hardware failures, malware, or ransomware attacks. Backups are often **encrypted** and stored safely in another location. If data is lost or compromised, it can be restored from the backup.

Example: A company makes daily encrypted backups of its employee records. If the main database is attacked by ransomware, the company can restore the data from the backup without losing important information.

9. RAID - Redundant Array of Independent Disks

RAID, which stands for **Redundant Array of Independent Disks**, is a storage technology used to improve the **performance, reliability, and fault tolerance** of databases and computer systems. Instead of relying on a single hard disk, RAID combines multiple disks into a single logical unit, allowing data to be distributed, duplicated, or striped across disks depending on the RAID level.

RAID is widely used in database systems, servers, and enterprise storage solutions to **prevent data loss** and ensure high availability. It is an important part of database security and disaster recovery because it **reduces the risk of downtime** due to hardware failure.

Objectives of RAID

The main objectives of RAID are:

1. **Data Redundancy:** Protect data by storing copies across multiple disks, so that if one disk fails, data can still be recovered.
2. **Improved Performance:** Multiple disks can read and write data simultaneously, speeding up database operations.
3. **Fault Tolerance:** RAID allows the system to continue operating even when one or more disks fail.

RAID Levels

There are several RAID levels, each offering a different balance between **redundancy, performance, and storage efficiency**:

RAID 0 – Striping

RAID 0, also called striping, works by dividing data into smaller blocks and distributing these blocks across two or more disks. Each disk stores only a portion of the overall data, and all disks operate simultaneously to read or write data. This parallelism significantly increases performance because multiple disks can process data at the same time.

Advantages:

The main advantage of RAID 0 is speed. Because data is read from or written to multiple disks in parallel, the overall read/write performance increases roughly in proportion to the number of disks used. It is also cost-efficient, as it does not require extra disks for redundancy.

Disadvantages:

RAID 0 provides no fault tolerance. If any single disk in the array fails, the data stored across all disks becomes unusable. This is because each disk holds only a portion of the data, and there is no backup or parity to reconstruct lost information.

Example:

Consider a 100 MB file stored using RAID 0 across two disks:

- Disk 1 stores 50 MB of the file.
- Disk 2 stores the other 50 MB.

When reading or writing the file, both disks work simultaneously, effectively doubling the speed compared to a single disk. However, if Disk 1 fails, the entire 100 MB file cannot be reconstructed, leading to total data loss.

"Think of a highway with two lanes carrying cars from point A to B. Half of the cars travel in Lane 1 and the other half in Lane 2. Traffic moves faster because cars are traveling simultaneously in both lanes. However, if Lane 1 is blocked due to an accident, all the cars in that lane cannot reach their destination. Similarly, in RAID 0, data is split across disks for speed, but failure of any single disk causes total data loss."

RAID 1 – Mirroring

RAID 1, also called **mirroring**, works by creating an **exact copy of data on two or more disks**. Every write operation is duplicated across all disks in the array. This ensures that if one disk fails, the mirrored copy on the other disk(s) keeps the data accessible. RAID 1 focuses on **data safety over storage efficiency**.

Example:
A bank stores customer account details using RAID 1 across two disks:

- Disk 1 and Disk 2 both contain the same account balances and transaction history. If Disk 1 fails, the system continues to operate using Disk 2. Once Disk 1 is replaced, data from Disk 2 is copied to it to restore mirroring.

Advantages:

- High fault tolerance: Data remains safe even if a disk fails.
- Fast read performance: Data can be read from any disk in the mirror.
- Easy recovery: Replacing a failed disk and copying from the mirror restores the array quickly.

Disadvantages:

- High storage cost: Requires at least double the storage, as each disk stores a full copy.
- Slightly slower writes: Every write must be performed on all mirrored disks.

*"Think of RAID 1 like **keeping two identical safes with the same valuable item**. If one safe is broken into or damaged, the valuable item is still safe in the other. Similarly, RAID 1 ensures data remains accessible even if one disk fails."*

RAID 2 – Bit-Level Striping with Hamming Code

RAID 2 divides data at the **bit level** and distributes these bits across multiple data disks. In addition to data disks, it uses **separate disks to store error-correcting codes**, specifically **Hamming Code**.

Hamming Code is an error detection and correction method that allows the system to identify and correct single-bit errors. In RAID 2, when data is written, it is split into individual bits and

written across data disks. At the same time, error-correction bits are calculated and stored on dedicated parity disks.

This means RAID 2 does not just detect errors — it can also **correct certain types of errors automatically** without data loss.

Example:
Suppose a 4-bit data value is to be stored:

Data bits: 1 0 1 1

- Each bit is stored on a separate disk.
- Additional disks store Hamming Code bits calculated from the data bits.

If one disk storing a data bit fails or produces an incorrect bit, the Hamming Code stored on the parity disks helps the system **detect and correct the error automatically**.

Advantages:

- Can detect and correct errors automatically.
- Provides high data reliability.
- Suitable for systems where data accuracy is extremely important.

Disadvantages:

- Requires many disks (both data and error-correction disks).
- Complex design and expensive implementation.
- Not commonly used in modern systems because RAID 3, 5, and 6 provide similar benefits more efficiently.

"Think of RAID 2 like a classroom exam system where each student writes part of an answer on separate sheets, and a teacher keeps additional checking sheets that can identify and correct mistakes. If one student makes a small mistake, the teacher's checking sheets help fix it.

Similarly, RAID 2 uses Hamming Code disks to detect and correct errors in data disks."

RAID 3 – Byte-Level Striping with Dedicated Parity

RAID 3 works by dividing data at the **byte level** and distributing these bytes across multiple data disks. In addition to the data disks, RAID 3 uses **one dedicated parity disk** to store parity information.

Parity is a special type of error-checking information calculated from the data stored on the data disks. If one disk fails, the missing data can be reconstructed using the parity information stored on the dedicated parity disk.

Unlike RAID 2 (which uses complex error-correcting codes), RAID 3 uses a simpler parity method. All disks in RAID 3 operate together for every read and write operation, meaning the system treats the array as one synchronized unit.

Example:
Suppose a file is divided into bytes and stored across three data disks:

- Disk 1 stores Byte 1
- Disk 2 stores Byte 2
- Disk 3 stores Byte 3
- Disk 4 (Parity Disk) stores parity information calculated from Bytes 1, 2, and 3

If Disk 2 fails, the system uses the parity disk and the remaining data disks to reconstruct Byte 2. This allows the system to continue operating without losing data.

Advantages:

- Can tolerate failure of one disk.
- Good performance for large, continuous data transfers.
- Provides fault tolerance with relatively simple parity calculation.

Disadvantages:

- The dedicated parity disk can become a bottleneck during write operations because every write must update the parity disk.
- Not efficient for small, independent read/write operations.
- Less commonly used in modern systems compared to RAID 5 and RAID 6.

"Imagine three workers writing different parts of a report, while a fourth worker keeps a summary note that can recreate any missing section. If one worker is absent, the summary note helps reconstruct the missing part. However, every time a change is made, the summary note must also be updated, which can slow things down."

Similarly, RAID 3 uses a dedicated parity disk to reconstruct lost data, but constant updates to that parity disk may reduce performance.

RAID 4 – Block-Level Striping with Dedicated Parity

RAID 4 divides data into **blocks** (larger units than bytes) and distributes these blocks across multiple data disks. In addition to the data disks, RAID 4 uses **one dedicated parity disk** to store parity information.

Parity is calculated from the data blocks stored on the data disks. If any one data disk fails, the missing data block can be reconstructed using the parity information and the remaining disks.

Unlike RAID 3, where data is striped at the byte level and all disks operate together, RAID 4 allows **independent read operations** because data is stored in blocks. This means the system can read from a single disk without involving all disks in the array.

Example:
Suppose a file is divided into blocks:

- Disk 1 stores Block A
- Disk 2 stores Block B
- Disk 3 stores Block C
- Disk 4 (Parity Disk) stores parity information calculated from Blocks A, B, and C

If Disk 2 fails, Block B can be reconstructed using Blocks A and C along with the parity information from Disk 4. The system continues functioning while the failed disk is replaced.

Advantages:

- Can tolerate failure of one disk.
- Better read performance compared to RAID 3 because blocks can be accessed independently.
- Efficient for read-heavy environments.

Disadvantages:

- The dedicated parity disk can become a bottleneck during write operations, as every write requires updating the parity disk.
- If the parity disk fails, the system loses fault tolerance until it is replaced.
- Less commonly used today because RAID 5 distributes parity across all disks, removing the bottleneck issue.

"Imagine three employees working on different sections of a report, while one supervisor keeps a backup summary of all sections. If one employee's work is lost, the supervisor's summary helps recreate it. However, every time any employee updates their section, the supervisor must update the summary too, which can slow down the process."

RAID 5 – Block-Level Striping with Distributed Parity

RAID 5 divides data into **blocks** and distributes these blocks across multiple disks, similar to RAID 4. However, instead of storing parity information on a single dedicated disk, RAID 5 **distributes the parity blocks evenly across all disks in the array**.

Parity is calculated from the data blocks and is used to reconstruct data if one disk fails. By distributing parity across all disks, RAID 5 eliminates the bottleneck problem found in RAID 4, where a single parity disk slows down write operations.

This design provides a **balance between performance, storage efficiency, and fault tolerance**, making RAID 5 one of the most commonly used RAID levels in database systems.

Advantages:

- Can tolerate failure of one disk.
- Good read performance due to block-level striping.
- Better write performance than RAID 4 because parity is distributed.
- Efficient use of storage (only one disk's worth of space is used for parity).

Disadvantages:

- Cannot tolerate more than one disk failure at the same time.
- Write operations are slower than RAID 0 due to parity calculation.
- During disk rebuilding, system performance decreases.

"Imagine a group of employees working on different sections of a project, and instead of one supervisor keeping all backup notes, each employee takes turns keeping backup information. If one employee is absent, the team can reconstruct their work using the distributed backup notes. Since the responsibility of keeping backup information is shared, no single person becomes overloaded."

RAID 6 – Block-Level Striping with Double Distributed Parity

RAID 6 is an extension of RAID 5 that provides higher fault tolerance by using block-level striping with double distributed parity. Like RAID 5, data is divided into blocks and spread across multiple disks. However, RAID 6 stores two independent parity blocks for each stripe instead of one.

These two parity blocks are distributed across all disks, so there is no dedicated parity disk. This design allows the system to survive the failure of two disks at the same time, which makes RAID 6 more reliable than RAID 5.

- Data is split into blocks and striped across all disks
- For each set of data blocks, two different parity values are calculated
- These parity blocks are distributed across different disks in each stripe
- If one or two disks fail, the missing data is reconstructed using the remaining data blocks and the two parity blocks

Because parity is distributed, the load is shared among all disks, avoiding performance bottlenecks.

RAID 6 can tolerate the failure of up to two disks simultaneously.
Even during two disk failures, the system continues to operate and data remains accessible, although performance may be reduced.

Example
Assume a RAID 6 system with five disks: D1, D2, D3, D4, and D5.

- Data blocks are stored across all disks
- Two parity blocks (P and Q) are calculated for each stripe
- The parity blocks rotate among the disks

If any two disks fail (for example, D2 and D4), the system reconstructs their data using the remaining disks and the two parity blocks.

Advantages of RAID 6

- Can tolerate failure of two disks
- High data reliability and fault tolerance
- Good read performance due to block-level striping
- Suitable for large storage systems where disk failure risk is high

Disadvantages of RAID 6

- Write performance is slower than RAID 5 due to double parity calculation
- Requires more storage space (two disks worth used for parity)
- Disk rebuild process is time-consuming and affects performance
- Higher implementation cost

"Imagine a team project where two backup copies of important notes are kept and the responsibility of maintaining them is shared among team members. Even if two members are absent, the team can still recover all the information using the remaining backups. This extra backup provides higher safety but takes more effort to maintain."

RAID 10 (RAID 1 + 0)
RAID 10, also called RAID 1+0, is a hybrid RAID technique that combines the features of RAID 1 (mirroring) and RAID 0 (striping). It provides high performance, high data reliability, and fast recovery, making it one of the most preferred RAID levels for database systems.
In RAID 10, data is first mirrored to provide fault tolerance, and then the mirrored data is striped across multiple disks to improve performance**.**

Working of RAID 10

- Disks are grouped into pairs

- Each pair works as a mirror (same data written to both disks)
- Data is then striped across these mirrored pairs
- Read and write operations occur in parallel, improving speed

Because of mirroring, data is always available even if a disk fails.

Fault Tolerance

RAID 10 can tolerate the failure of one disk in each mirrored pair.

As long as one disk in a mirror pair is working, data remains accessible.

This makes RAID 10 more reliable than RAID 5 and RAID 0.

Example

Assume four disks: D1, D2, D3, and D4.

- D1 and D2 form a mirror pair
- D3 and D4 form another mirror pair
- Data is striped across the two mirror pairs

If disk D1 fails, data is still available from D2.

If disk D3 fails, data is still available from D4.

Advantages of RAID 10

- Very high read and write performance
- Excellent fault tolerance
- Fast data recovery (no parity calculation required)
- Suitable for heavy transaction workloads

Disadvantages of RAID 10

- Requires more disks (minimum 4 disks)
- Storage efficiency is low (only 50% usable)
- Higher cost compared to parity-based RAID levels

"Imagine writing important notes in two identical notebooks (mirroring) and dividing the work across two teams (striping). Even if one notebook is lost, the backup is available, and work progresses quickly because tasks are shared."

Example:

- 4 disks: Disk1 & Disk2 = mirror, Disk3 & Disk4 = mirror.
- Data striped across the mirrored pairs.

Short – Questions

Transaction Management

1. Define a transaction in DBMS.
2. Expand ACID in DBMS.
3. Which ACID property ensures permanent results of a transaction?
4. What is a shared lock?

5. What is an exclusive lock?
6. What is timestamp ordering in DBMS?
7. What are the three phases of optimistic concurrency control?
8. Define conflict serializability.
9. Define deadlock.
10. List four necessary conditions for deadlock.

Database Recovery

11. Why do we need recovery in DBMS?
12. List any two types of failures.
13. Define deferred update method.
14. What is a checkpoint in recovery?
15. What is shadow paging?

Database Security

16. Define a database threat.
17. List any two authorization levels.
18. Define a database view.
19. What is encryption?
20. Expand RAID.

Long– Questions

Transaction Management

1. Explain ACID properties with a money transfer example.
2. Explain lock-based concurrency control with example schedules.
3. Explain timestamp-based concurrency control with step-by-step example.
4. Explain optimistic concurrency control protocol with an example.
5. Differentiate between conflict and view serializability with examples.

Database Recovery

6. Explain different types of failures in DBMS with examples.
7. Differentiate between deferred update and immediate update recovery with examples.

Database Security

8. Explain different types of threats in DBMS with examples.
9. Explain authorization in DBMS with examples of access rights.
10. Explain different RAID levels with diagrams and examples.