Module 4 – Introduction to DBMS

- ## Introduction to SQL

1. What is SQL, and why is it essential in database management?

ANS:

SQL is a **domain-specific language** designed to interact with databases. It provides a systematic way to access and manage large amounts of data efficiently. It acts as a communication bridge between the user and the database server.

1. **Data Organization and Storage:**
   SQL helps in organizing data into tables, which can be related through keys. This structure ensures that data is stored systematically and can be easily accessed when needed.
2. **Data Integrity and Accuracy:**
   SQL supports constraints like **PRIMARY KEY**, **FOREIGN KEY**, **UNIQUE**, and **CHECK**, which maintain the accuracy and consistency of data.
3. **Security:**
   SQL provides strong security mechanisms by allowing the database administrator to grant or restrict user access to sensitive data.
4. **Portability and Standardization:**
   SQL is an **ANSI (American National Standards Institute)** standard language, meaning SQL commands work across most relational databases with minimal changes.
5. **Scalability and Efficiency:**
   SQL databases can handle large volumes of data and multiple user requests simultaneously, making them ideal for enterprise-level applications.

6. **Support for Data Analysis:**
   SQL supports **aggregate functions** like COUNT(), SUM(), AVG(), MIN(), and MAX() that help in summarizing and analyzing large datasets efficiently.

2. Explain the difference between DBMS and RDBMS.

ANS:

- **DBMS (Database Management System):**
  A DBMS is a software that manages data in a file-based system. It provides a way to store and retrieve data but does not enforce relationships among data.
  Example: Microsoft Access, dBase, and FoxPro.
- **RDBMS (Relational Database Management System):**
  An RDBMS is a type of DBMS based on the **relational model** proposed by **E.F. Codd**. It stores data in multiple related tables connected through **primary keys and foreign keys**.
  Example: MySQL, Oracle, PostgreSQL, and SQL Server.

Feature	DBMS	RDBMS
Data Storage Format	Stores data as files or single tables.	Stores data in multiple related tables.
Data Relationship	No relationships among data.	Maintains relationships between tables using keys.
Normalization	Does not support normalization.	Supports normalization to reduce data redundancy.
Data Redundancy	High redundancy due to lack of relationships.	Low redundancy due to normalization and relational structure.

Feature	DBMS	RDBMS
Data Integrity	Does not enforce data integrity constraints.	Enforces integrity through primary key, foreign key, and constraints.
Data Security	Security is managed at the application level.	Provides advanced security through privileges and user roles.
Scalability	Suitable for small-scale applications.	Suitable for large and complex applications.
Examples	dBase, FileMaker, MS Access (basic).	MySQL, Oracle, PostgreSQL, SQL Server.

3. Describe the role of SQL in managing relational databases.

ANS:

SQL (Structured Query Language) is a **declarative language** that allows users to communicate with databases. It provides a consistent way to define database structures, manipulate data, and control access. It was standardized by **ANSI (American National Standards Institute)** and **ISO (International Organization for Standardization)** to ensure compatibility across different database systems.

Role of SQL in Managing Relational Databases:

SQL plays multiple roles in the effective management of relational databases. Its functionality can be divided into several categories:

1. Data Definition (DDL – Data Definition Language):

SQL allows the creation and modification of database structures such as tables, views, indexes, and schemas.
Common DDL commands include:

- CREATE – To create new database objects.
- ALTER – To modify existing objects.
- DROP – To delete objects.

Example:

```
CREATE TABLE Students (
StudentID INT PRIMARY KEY,
Name VARCHAR(50),
Age INT
);
```

This command creates a table named *Students* with three columns.

2. Data Manipulation (DML – Data Manipulation Language):

DML commands are used to insert, modify, or delete data from database tables.
Common DML commands are:

- INSERT – Adds new data.
- UPDATE – Modifies existing data.
- DELETE – Removes data.
- SELECT – Retrieves data.

Example:

```
SELECT Name, Age FROM Students WHERE Age > 18;
```

This retrieves data of students older than 18 years.

3. Data Querying and Retrieval:

SQL provides powerful querying capabilities to extract specific information from large datasets. With clauses like **WHERE**, **ORDER BY**, **GROUP BY**, and

HAVING, users can filter, sort, and group data efficiently.
This makes SQL extremely useful for **data analysis and reporting**.

4. Data Control (DCL – Data Control Language):

SQL manages user access and permissions through DCL commands.

- GRANT – Gives access privileges to users.
- REVOKE – Removes granted privileges.

This ensures that only authorized users can perform specific actions, maintaining **data security**.

5. Transaction Management (TCL – Transaction Control Language):

SQL maintains **data integrity and consistency** through transaction control.

- COMMIT – Saves all changes permanently.
- ROLLBACK – Reverts changes in case of errors.
- SAVEPOINT – Marks a point to roll back to if needed.

4. What are the key features of SQL?

ANS:

1. Data Definition Capability:

SQL provides a set of **Data Definition Language (DDL)** commands to define the structure of a database.
Commands like CREATE, ALTER, and DROP help in creating and modifying database objects such as tables, views, and indexes.
Example:

```
CREATE TABLE Employees (
  ID INT PRIMARY KEY,
```

Name VARCHAR(50),

Salary DECIMAL(10,2)

);

2. Data Manipulation Capability:

Using **Data Manipulation Language (DML)** commands, SQL allows users to insert, modify, and delete data.
Commands include INSERT, UPDATE, DELETE, and SELECT.
These operations make it easy to manage large volumes of data efficiently.

Example:

UPDATE Employees SET Salary = 60000 WHERE ID = 101;

3. Powerful Data Querying:

SQL provides a very powerful query capability to fetch specific data from large datasets using the SELECT statement.
It supports various clauses like WHERE, GROUP BY, ORDER BY, and HAVING for filtering, sorting, and grouping data.
Example:

SELECT Name, Salary FROM Employees WHERE Salary > 50000 ORDER BY Salary DESC;

4. Data Integrity and Accuracy:

SQL ensures data accuracy by supporting **constraints** such as **PRIMARY KEY**, **FOREIGN KEY**, **UNIQUE**, **NOT NULL**, and **CHECK**.
These constraints prevent invalid or duplicate data entries, maintaining the consistency and reliability of the database.

5. Data Security and Access Control:

SQL includes **Data Control Language (DCL)** commands such as GRANT and REVOKE to manage access rights and permissions.
Database administrators can give or restrict access to specific users, ensuring sensitive data is protected.

6. Transaction Control:

SQL supports **Transaction Control Language (TCL)** to handle transactions safely and ensure database integrity.
Commands like COMMIT, ROLLBACK, and SAVEPOINT help in managing multi-step operations while following the **ACID properties** (Atomicity, Consistency, Isolation, Durability).

7. Portability and Standardization:

SQL is a **standardized language**, meaning that SQL code written for one RDBMS (like MySQL) can often be used on another (like PostgreSQL) with minor modifications.
It is platform-independent and can run on different operating systems and environments.

8. Support for Functions and Joins:

SQL supports **aggregate functions** like SUM(), AVG(), COUNT(), MIN(), and MAX() for data analysis.
It also allows **joining tables** using commands like INNER JOIN, LEFT JOIN, and RIGHT JOIN, enabling retrieval of related data from multiple tables.

9. High Performance and Scalability:

SQL databases are optimized to handle large volumes of data and multiple concurrent users efficiently.
Indexing, optimization, and caching techniques improve data retrieval speed and scalability.

- **SQL Syntax**

1. What are the basic components of SQL syntax?

ANS:

Basic Components of SQL Syntax:

1. SQL Statements:

An SQL statement is a complete command that performs a particular action in the database.
Each statement ends with a **semicolon (;)**.
Example:

SELECT * FROM Students;

Here, SELECT is the SQL statement used to retrieve data.

2. SQL Clauses:

Clauses are the **building blocks of SQL statements**. They define the conditions and structure of the query.
Common clauses include:

- SELECT – Specifies columns to retrieve.
- FROM – Specifies the table(s) to fetch data from.
- WHERE – Filters records based on conditions.
- GROUP BY – Groups rows that have the same values.
- ORDER BY – Sorts data in ascending or descending order.
- HAVING – Applies conditions to grouped data.

Example:

SELECT Name, Age

FROM Students

WHERE Age > 18

ORDER BY Name;

3. SQL Keywords:

SQL uses **predefined reserved words**, known as keywords, that have special meaning. Examples include:
SELECT, INSERT, UPDATE, DELETE, CREATE, ALTER, DROP, FROM, WHERE, AND, OR, NOT, etc.
These keywords must be used correctly and are not case-sensitive (though conventionally written in uppercase for readability).

4. SQL Expressions:

An expression is a **combination of values, columns, and operators** that produces a single value.
Expressions are often used in WHERE or SELECT clauses.
Example:

SELECT Name, Salary*12 AS Annual_Salary FROM Employees;

Here, Salary*12 is an expression that calculates the annual salary.

5. SQL Operators:

Operators are used to perform operations on data. They are divided into different types:

- **Arithmetic Operators:** +, -, *, /, %
- **Comparison Operators:** =, >, <, >=, <=, <>
- **Logical Operators:** AND, OR, NOT
  Example:

SELECT * FROM Employees WHERE Salary > 50000 AND Department = 'IT';

6. SQL Functions:

SQL provides built-in functions to perform operations on data.

- **Aggregate Functions:** SUM(), AVG(), COUNT(), MAX(), MIN()
- **String Functions:** UPPER(), LOWER(), CONCAT()
- **Date Functions:** NOW(), CURDATE(), YEAR()
  Example:

SELECT COUNT(*) FROM Students WHERE Age > 18;

7. SQL Comments:

Comments are used to explain SQL code and are ignored by the database engine.

- **Single-line comment:** Starts with --
- **Multi-line comment:** Written between /* ... */
  Example:

-- This query retrieves student names

SELECT Name FROM Students;

8. Case Sensitivity:

In most SQL databases:

- **SQL keywords** are **not case-sensitive** (SELECT = select),
- **Table and column names** may or may not be case-sensitive, depending on the database and operating system.

Example of a Complete SQL Query:

SELECT Name, Department, Salary

FROM Employees

WHERE Salary > 40000

ORDER BY Salary DESC;

Explanation:

- SELECT – Retrieves data.
- FROM – Specifies the table.
- WHERE – Filters data.
- ORDER BY – Sorts the output.

2. Write the general structure of an SQL SELECT statement.

ANS:

The **SELECT statement** in SQL is used to **query records** from a database table based on specific criteria. It follows a structured syntax with several optional clauses to refine the output.

General Structure of SQL SELECT Statement:

SELECT [DISTINCT] column1, column2, ...

FROM table_name

[WHERE condition]

[GROUP BY column_list]

[HAVING condition]

[ORDER BY column_list [ASC | DESC]];

Explanation of Each Clause:

1. SELECT Clause:

- Specifies the **columns (fields)** to be retrieved from a table.

- To fetch all columns, an asterisk * is used.
- The DISTINCT keyword can be used to eliminate duplicate values.
  Example:

SELECT DISTINCT Department FROM Employees;

2. FROM Clause:

- Defines the **table(s)** from which the data will be retrieved.
- When multiple tables are used, they can be combined using **JOIN** operations.
  Example:

SELECT Name, Salary FROM Employees;

3. WHERE Clause:

- Filters the rows based on a specific **condition**.
- Only records satisfying the condition will be displayed.
  Example:

SELECT Name, Salary FROM Employees WHERE Department = 'IT';

4. GROUP BY Clause:

- Groups rows that have the same values in specified columns.
- Commonly used with **aggregate functions** like SUM(), COUNT(), AVG(), etc.
  Example:

SELECT Department, AVG(Salary)

FROM Employees

GROUP BY Department;

5. HAVING Clause:

- Works like the WHERE clause but is used **after grouping** to filter groups based on aggregate results.
  Example:

SELECT Department, COUNT(*) AS Emp_Count

FROM Employees

GROUP BY Department

HAVING COUNT(*) > 5;

6. ORDER BY Clause:

- Specifies the order in which the retrieved records are displayed.
- Sorting can be in **ascending (ASC)** or **descending (DESC)** order.
  Example:

SELECT Name, Salary FROM Employees ORDER BY Salary DESC;

Complete Example of SELECT Statement:

SELECT Name, Department, AVG(Salary) AS Average_Salary

FROM Employees

WHERE Department IS NOT NULL

GROUP BY Department

HAVING AVG(Salary) > 50000

ORDER BY Average_Salary DESC;

Explanation:

- Retrieves the average salary of employees in each department.
- Ignores rows with no department assigned.
- Displays only departments with an average salary greater than ₹50,000.

- Results are shown in descending order of average salary.

3. Explain the role of clauses in SQL statements.

ANS:

A **clause** in SQL is a part of a statement that performs a particular function or specifies an action to be taken on the data.
For example, in a SELECT query, clauses such as FROM, WHERE, and ORDER BY define which table to use, what condition to apply, and how to sort the results.

Role of Clauses in SQL:

Clauses provide structure and control to SQL statements. They define **what data to retrieve**, **from where**, **under what conditions**, and **in what order**.
They make SQL flexible and precise, allowing users to extract exactly the data they need.

Major Clauses in SQL and Their Roles:

1. SELECT Clause:

- Specifies **which columns or fields** should be displayed in the output.
- Can include column names, expressions, or functions.
- DISTINCT can be used to eliminate duplicate rows.

Example:

SELECT DISTINCT Department FROM Employees;

Role: Determines the data to be retrieved.

2. FROM Clause:

- Specifies **the table(s)** from which data will be selected.

- When multiple tables are involved, JOIN operations can be used.
  Example:

SELECT Name, Salary FROM Employees;

Role: Indicates the data source for the query.

3. WHERE Clause:

- Filters records based on a specified **condition**.
- Only rows that satisfy the condition are retrieved or affected.
  Example:

SELECT Name, Salary FROM Employees WHERE Department = 'IT';

Role: Helps in retrieving **specific** data rather than all records.

4. GROUP BY Clause:

- Groups rows that have the same values in specified columns.
- Commonly used with aggregate functions like SUM(), AVG(), COUNT(), etc.
  Example:

SELECT Department, AVG(Salary)

FROM Employees

GROUP BY Department;

Role: Summarizes data into groups for analysis.

5. HAVING Clause:

- Works like the WHERE clause but is used to filter groups instead of individual rows.
- Used along with GROUP BY to apply conditions to aggregated data.
  Example:

SELECT Department, COUNT(*) AS Emp_Count

FROM Employees

GROUP BY Department

HAVING COUNT(*) > 5;

Role: Filters data after grouping to refine the result set.

6. ORDER BY Clause:

- Specifies the order in which the retrieved records are displayed.
- Sorting can be done in **ascending (ASC)** or **descending (DESC)** order.
  Example:

SELECT Name, Salary FROM Employees ORDER BY Salary DESC;

Role: Organizes the query result to improve readability and analysis.

7. LIMIT Clause (specific to MySQL):

- Restricts the **number of rows** returned by a query.
  Example:

SELECT * FROM Employees LIMIT 5;

Role: Controls the output size for efficiency.

Importance of Clauses in SQL Statements:

1. **Structure and Organization:** Clauses make SQL statements more readable and logically organized.
2. **Data Precision:** They allow fine-tuned control over what data is selected or modified.
3. **Flexibility:** Different clauses can be combined to form complex and powerful queries.

4. **Performance Optimization:** Clauses like WHERE and LIMIT help reduce data processing overhead by filtering results early.
5. **Data Analysis:** Clauses such as GROUP BY and HAVING are vital for summarizing and analyzing large datasets.

- **SQL Constraints**

1. **What are constraints in SQL? List and explain the different types of constraints.**

ANS:

Constraints in SQL are **rules applied to table columns** to maintain **data accuracy, integrity, and consistency** in the database.

2. **NOT NULL:** Ensures that a column **cannot have NULL values.**

Name VARCHAR(50) NOT NULL;

3. **UNIQUE:** Ensures all values in a column are **distinct.**

Email VARCHAR(100) UNIQUE;

4. **PRIMARY KEY:** Combines **NOT NULL + UNIQUE**; uniquely identifies each record in a table.

PRIMARY KEY (Student_ID);

5. **FOREIGN KEY:** Creates a link between two tables by referencing the **primary key of another table.**

FOREIGN KEY (Dept_ID) REFERENCES Department(Dept_ID);

6. **CHECK:** Ensures that a column value meets a specific condition.

CHECK (Age >= 18);

7. **DEFAULT:** Assigns a default value to a column if no value is provided.

Salary DECIMAL(10,2) DEFAULT 10000;

2. How do PRIMARY KEY and FOREIGN KEY constraints differ?

ANS:

Both **PRIMARY KEY** and **FOREIGN KEY** are important constraints in SQL used to maintain **data integrity** and **relationships** between tables.

Aspect	**PRIMARY KEY**	**FOREIGN KEY**
Purpose	Uniquely identifies each record in a table.	Creates a link between two tables.
Uniqueness	Must contain **unique and non-null** values.	Can contain **duplicate or null** values.
Table Type	Exists in the **parent (main)** table.	Exists in the **child (referencing)** table.
Number per Table	Only **one** primary key allowed.	Multiple foreign keys can exist.
Data Integrity	Ensures **entity integrity** (unique rows).	Ensures **referential integrity** (valid relationships).

Example:

```
CREATE TABLE Department (
  Dept_ID INT PRIMARY KEY,
  Dept_Name VARCHAR(50)
);
```

```
CREATE TABLE Employee (
  Emp_ID INT PRIMARY KEY,
  Emp_Name VARCHAR(50),
  Dept_ID INT,
  FOREIGN KEY (Dept_ID) REFERENCES Department(Dept_ID)
);
```

Explanation:

- Dept_ID in **Department** is the **PRIMARY KEY** — uniquely identifies each department.
- Dept_ID in **Employee** is a **FOREIGN KEY** — links employees to their department.

3. What is the role of NOT NULL and UNIQUE constraints?

ANS:

Constraints in SQL ensure the **accuracy and consistency** of data. Among them, **NOT NULL** and **UNIQUE** are two commonly used column-level constraints that control how data is stored in a table.

1. NOT NULL Constraint:

- Ensures that a column **cannot contain NULL (empty) values.**
- Used when a column must always have a valid value.
- Helps prevent incomplete or missing data entries.

Example:

```
CREATE TABLE Students (
  Student_ID INT NOT NULL,
  Name VARCHAR(50) NOT NULL
);
```

Here, both Student_ID and Name must always have a value.

2. UNIQUE Constraint:

- Ensures that all values in a column are **distinct** — no duplicates are allowed.
- Allows NULL values unless combined with NOT NULL.
- Used for columns like email, phone number, etc.

Example:

```
CREATE TABLE Employees (
  Emp_ID INT UNIQUE,
  Email VARCHAR(100) UNIQUE
);
```

- **Main SQL Commands and Sub-commands (DDL)**

1. **Define the SQL Data Definition Language (DDL).**

ANS:

Data Definition Language (DDL) in SQL is a set of commands used to **define, modify, and manage the structure** of database objects such as tables, schemas, indexes, and views.

It deals with the **creation and organization** of database structures rather than the manipulation of data within them.

Main DDL Commands:

- **CREATE:**
  Used to **create** new database objects like tables, databases, or views.

```
CREATE TABLE Students (
  Student_ID INT PRIMARY KEY,
  Name VARCHAR(50),
  Age INT
);
```

- **ALTER:**
  Used to **modify** an existing table structure — for example, adding or removing columns.

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

- **DROP:**
  Deletes an existing database object permanently.

```
DROP TABLE Students;
```

- **TRUNCATE:**
  Removes all records from a table but **retains the structure** for future use.

```
TRUNCATE TABLE Students;
```

2. **Explain the CREATE command and its syntax.**

ANS:

The **CREATE** command in SQL is a **Data Definition Language (DDL)** statement used to **create new database objects**, such as databases, tables, views, or indexes.
It defines the **structure and properties** of the object being created.

Purpose:

- To create **databases** and **tables** for storing data.
- To define **columns, data types, and constraints**.
- To organize and manage data efficiently within a relational database.

General Syntax:

```
CREATE TABLE table_name (
  column1 datatype [constraint],
  column2 datatype [constraint],
  ...
);
```

Example:

```
CREATE TABLE Students (
  Student_ID INT PRIMARY KEY,
  Name VARCHAR(50) NOT NULL,
  Age INT CHECK (Age >= 18),
  Email VARCHAR(100) UNIQUE
);
```

Explanation:

- Creates a table named **Students**.
- Student_ID is the **primary key** (unique identifier).
- Name cannot be empty (**NOT NULL**).
- Age must be **18 or above**.

- Email must be **unique** for every student.

3. **What is the purpose of specifying data types and constraints during table creation?**

ANS:

When creating a table in SQL, each column must have a **data type** and can have **constraints** to define the kind of data it will store and the rules it must follow. These ensure **data accuracy, integrity, and consistency** within the database.

1. Purpose of Data Types:

Data types define **what kind of values** a column can store, such as numbers, text, or dates.
They help the database use storage efficiently and prevent invalid data entry.

Common SQL Data Types:

- **INT:** For storing integer numbers.
- **VARCHAR(n):** For variable-length text (e.g., names, emails).
- **DATE:** For storing date values.
- **DECIMAL(p, q):** For precise decimal numbers (e.g., salary, price).

2. Purpose of Constraints:

Constraints are **rules applied to columns** to ensure valid and consistent data.

Common Constraints:

- **NOT NULL:** Prevents empty values.
- **UNIQUE:** Ensures all values are distinct.
- **PRIMARY KEY:** Uniquely identifies each record.
- **FOREIGN KEY:** Maintains relationships between tables.
- **CHECK:** Ensures values meet a specific condition.

- **ALTER Command**

1. **What is the use of the ALTER command in SQL?**

ANS:

The **ALTER** command in SQL is a **Data Definition Language (DDL)** statement used to **modify the structure of an existing database object**, usually a table, **without deleting or recreating it**.

Purpose:

The ALTER command allows database administrators or developers to make structural changes to a table after it has been created.

It is commonly used to:

- **Add new columns**
- **Modify existing columns** (data type, size, or constraints)
- **Rename columns or tables**
- **Drop columns or constraints**

Common Uses and Syntax:

- **Add a New Column:**

ALTER TABLE Students ADD Email VARCHAR(100);

→ Adds a new column named Email to the Students table.

- **Modify a Column:**

ALTER TABLE Students MODIFY Age INT;

→ Changes the data type or size of the Age column.

- **Rename a Column:**

ALTER TABLE Students RENAME COLUMN Name TO FullName;

→ Renames the column Name to FullName.

- **Drop a Column:**

ALTER TABLE Students DROP COLUMN Email;

→ Deletes the Email column permanently.

- **Add or Drop a Constraint:**

ALTER TABLE Students ADD CONSTRAINT chk_age CHECK (Age >= 18);

ALTER TABLE Students DROP CONSTRAINT chk_age;

2. **How can you add, modify, and drop columns from a table using ALTER?**

ANS:

The **ALTER** command in SQL is used to **change the structure of an existing table** without deleting it.
It allows you to **add**, **modify**, or **remove** columns as per changing requirements.

Add a Column:

Used to insert a **new column** into an existing table.

Example:

ALTER TABLE Students

ADD Email VARCHAR(100);

→ Adds a new column Email to the Students table.

2. Modify a Column:

Used to **change the data type, size, or constraint** of an existing column.

Example:

ALTER TABLE Students

MODIFY Name VARCHAR(80) NOT NULL;

→Changes the column Name to hold up to 80 characters and disallows NULL values.

3. Drop a Column:

Used to **delete an existing column** permanently from a table.

Example:

ALTER TABLE Students

DROP COLUMN Email;

→ Removes the Email column from the table.

- **DROP Command**

1. **What is the function of the DROP command in SQL?**

ANS:

The **DROP** command in SQL is a **Data Definition Language (DDL)** statement used to **permanently delete database objects** such as **tables, databases, views, or indexes**.
Once dropped, the object and all the data stored within it are **completely removed** from the database.

Common Uses and Syntax:

- **Drop a Table:**

DROP TABLE table_name;

Example:

DROP TABLE Students;

→ Permanently deletes the table Students along with all its data and structure.

- **Drop a Database:**

DROP DATABASE database_name;

Example:

DROP DATABASE CollegeDB;

→ Deletes the entire database CollegeDB and all objects inside it.

- **Drop a View or Index:**

DROP VIEW view_name;

DROP INDEX index_name;

2. **What are the implications of dropping a table from a database?**

ANS:

When a table is **dropped** using the SQL DROP TABLE command, it is **permanently removed** from the database along with all its **data, structure, and dependencies**.

1. **Permanent Data Loss:**

- All records stored in the table are **deleted permanently**.
- Once dropped, the data **cannot be recovered** unless a backup exists.
- DROP TABLE Students;

2. **Loss of Table Structure:**

- The **schema (column definitions, data types, and constraints)** of the table is also deleted.
- You would have to **recreate the table** to use it again.

3. **Effect on Relationships:**

- If the table is **referenced by a FOREIGN KEY** in another table, the DROP command may **fail** unless the referencing table is dropped first.
- Dropping a parent table can **break referential integrity** between related tables.

4. **Removal of Constraints and Indexes:**

- All **constraints (PRIMARY KEY, FOREIGN KEY, CHECK, etc.)** and **indexes** linked to the table are also removed.

- **Data Manipulation Language (DML)**

1. **Define the INSERT, UPDATE, and DELETE commands in SQL.**

ANS:

These three commands are part of SQL's **Data Manipulation Language (DML)**, used to **add, modify, and remove data** from database tables.

1. INSERT Command

Definition:
The INSERT command is used to **add new records (rows)** into a table.

Syntax:

INSERT INTO table_name (column1, column2, ...)

VALUES (value1, value2, ...);

Example:

INSERT INTO Students (Student_ID, Name, Age)

VALUES (101, 'Rahul', 20);

→Adds a new student record to the Students table.

2. UPDATE Command

Definition:
The UPDATE command is used to **modify existing records** in a table.

Syntax:

UPDATE table_name

SET column1 = value1, column2 = value2, ...

WHERE condition;

Example:

UPDATE Students

SET Age = 21

WHERE Student_ID = 101;

→Updates the Age of the student with Student_ID 101 to 21.

Note: Always use a WHERE clause to avoid updating all rows unintentionally.

3. DELETE Command

Definition:
The DELETE command is used to **remove existing records** from a table.

Syntax:

DELETE FROM table_name

WHERE condition;

Example:

DELETE FROM Students

WHERE Student_ID = 101;

→Deletes the student record with Student_ID 101.

2. **What is the importance of the WHERE clause in UPDATE and DELETE operations?**

ANS:

The WHERE clause in SQL is used to **specify conditions** that determine which rows will be affected by an UPDATE or DELETE statement.

Role in UPDATE Operations:

- Ensures that **only specific records** are modified.
- Prevents unintentional changes to all rows in a table.

Example:

UPDATE Students

SET Age = 21

WHERE Student_ID = 101;

→Only updates the age of the student with Student_ID 101.
Without WHERE, **all students' ages would be set to 21**, which can cause serious data errors.

Role in DELETE Operations:

- Ensures that **only selected records** are removed from the table.
- Prevents accidental deletion of the entire table data.

Example:

DELETE FROM Students

WHERE Age < 18;

→Deletes only students younger than 18.
Without WHERE, **all rows in the Students table would be deleted**.

- **Data Query Language (DQL)**

1. **What is the SELECT statement, and how is it used to query data?**

ANS:

The SELECT statement in SQL is used to **retrieve data from one or more tables** in a relational database. It is the most commonly used command for querying data and allows users to specify exactly which columns and rows to retrieve.

The basic syntax of a SELECT statement is:

SELECT column1, column2, ...

FROM table_name

WHERE condition

ORDER BY column_name;

- SELECT specifies the columns to retrieve.
- FROM indicates the table(s) containing the data.
- WHERE filters rows based on specific conditions.
- ORDER BY sorts the results in ascending or descending order.

Example:

SELECT Name, Age

FROM Students

WHERE Age > 18

ORDER BY Name;

This query retrieves the names and ages of students older than 18 and displays them sorted by name.

2. **Explain the use of the ORDER BY and WHERE clauses in SQL queries.**

ANS:

- The WHERE and ORDER BY clauses are used to **filter and organize data** when querying a database with SQL.
- The WHERE clause is used to **filter rows** based on specific conditions. It allows you to retrieve only the records that meet certain criteria.

Example:

SELECT Name, Age

FROM Students

WHERE Age > 18;

This query retrieves only students whose age is greater than 18. Without WHERE, all rows would be returned.

- The ORDER BY clause is used to **sort the results** of a query in either ascending (ASC) or descending (DESC) order.

Example:

SELECT Name, Age

FROM Students

ORDER BY Age DESC;

This query retrieves all students and displays them sorted by age in descending order.

- **Data Control Language (DCL)**

1. **What is the purpose of GRANT and REVOKE in SQL?**

ANS:

- The GRANT and REVOKE commands in SQL are used to **control user access and permissions** on database objects, ensuring security and proper management of data.
- The GRANT command is used to **give specific privileges** to a user or role, such as SELECT, INSERT, UPDATE, or DELETE on a table.

Example:

GRANT SELECT, INSERT ON Students TO User1;

This allows User1 to view and add records in the Students table.

- The REVOKE command is used to **remove previously granted privileges** from a user or role.

Example:

REVOKE INSERT ON Students FROM User1;

This removes the ability of User1 to insert records into the Students table.

2. **How do you manage privileges using these commands?**

ANS:

- Privileges in SQL are managed by using the GRANT and REVOKE commands to **assign or remove permissions** for users on database objects like tables, views, or procedures.
- The GRANT command is used to **assign privileges** such as SELECT, INSERT, UPDATE, DELETE, or ALL to users. It defines what actions a user is allowed to perform.

Example:

GRANT SELECT, UPDATE ON Employees TO User1;

This allows User1 to view and update records in the Employees table.

The REVOKE command is used to **withdraw privileges** that were previously granted to users.

Example:

REVOKE UPDATE ON Employees FROM User1;

This removes the update privilege from User1.

- **Transaction Control Language (TCL)**

1. **What is the purpose of the COMMIT and ROLLBACK commands in SQL?**

ANS:

- The COMMIT and ROLLBACK commands are used in SQL to manage **transactions**, ensuring data accuracy and consistency in the database.

- A **transaction** is a set of one or more SQL operations that are treated as a single logical unit of work. These commands help confirm or undo the changes made during a transaction.
- The COMMIT command is used to **save all the changes permanently** made by the transaction in the database. Once committed, the changes cannot be undone.

Example:

COMMIT;

- The ROLLBACK command is used to **undo all the changes** made during the current transaction before it was committed. It restores the database to its previous state.

Example:

ROLLBACK;

In short, COMMIT finalizes the transaction, while ROLLBACK cancels it. Together, they help maintain **data integrity** and ensure **error recovery** in SQL operations.

2. **Explain how transactions are managed in SQL databases.**

ANS:

Transactions in SQL are used to ensure that a group of database operations is executed **safely and reliably**. A transaction is a logical unit of work that must be either **fully completed or fully rolled back** to maintain data integrity.

Transactions follow the **ACID properties**:

- **Atomicity:** Ensures all operations in a transaction are completed successfully, or none are applied.
- **Consistency:** Maintains the database in a valid state before and after the transaction.

- **Isolation:** Ensures that concurrent transactions do not interfere with each other.
- **Durability:** Once a transaction is committed, the changes are permanent, even if the system fails.

The main commands used in transaction management are:

- **BEGIN TRANSACTION / START TRANSACTION:** Starts a new transaction.
- **COMMIT:** Saves all changes made during the transaction permanently.
- **ROLLBACK:** Reverts all changes made during the transaction.

Example:

START TRANSACTION;

UPDATE Accounts SET Balance = Balance - 500 WHERE Acc_ID = 1;

UPDATE Accounts SET Balance = Balance + 500 WHERE Acc_ID = 2;

COMMIT;

If an error occurs, ROLLBACK can be used to undo the updates.

- **SQL Joins**

1. **Explain the concept of JOIN in SQL. What is the difference between INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN?**

ANS:

A **JOIN** in SQL is used to **combine data from two or more tables** based on a related column between them. It allows users to retrieve meaningful information spread across multiple tables by establishing a relationship using **primary and foreign keys**.

The general syntax of a JOIN is:

SELECT columns

FROM table1

JOIN table2

ON table1.column = table2.column;

Types of SQL JOINs:

1. **INNER JOIN:**
   Returns only the rows that have **matching values** in both tables.

SELECT * FROM Employees

INNER JOIN Departments

ON Employees.DeptID = Departments.DeptID;

Result: Only employees that belong to an existing department are shown.

2. **LEFT JOIN (LEFT OUTER JOIN):**
   Returns **all rows from the left table** and matching rows from the right table. Unmatched rows in the right table are shown as **NULL**.

SELECT * FROM Employees

LEFT JOIN Departments

ON Employees.DeptID = Departments.DeptID;

3. **RIGHT JOIN (RIGHT OUTER JOIN):**
   Returns **all rows from the right table** and matching rows from the left table. Unmatched rows in the left table appear as **NULL**.

SELECT * FROM Employees

RIGHT JOIN Departments

ON Employees.DeptID = Departments.DeptID;

4. **FULL OUTER JOIN:**
   Returns **all rows from both tables**, with NULLs where there is no match.

SELECT * FROM Employees

FULL OUTER JOIN Departments

ON Employees.DeptID = Departments.DeptID;

Summary:

- **INNER JOIN** → Only matching records.
- **LEFT JOIN** → All from left + matching from right.
- **RIGHT JOIN** → All from right + matching from left.
- **FULL OUTER JOIN** → All records from both tables.

3. **How are joins used to combine data from multiple tables?**

ANS:

Joins in SQL are used to **combine rows from two or more tables** based on a related column, usually a **primary key–foreign key relationship**. They help retrieve data that is spread across different tables in a relational database.

When two tables share a common column, a **JOIN** operation links them, allowing users to query meaningful combined data.

Syntax:

SELECT columns

FROM table1

JOIN table2

ON table1.common_column = table2.common_column;

Example:
Suppose we have two tables:

- Students(StudentID, Name, DeptID)
- Departments(DeptID, DeptName)

To display student names along with their department names:

SELECT Students.Name, Departments.DeptName

FROM Students

JOIN Departments

ON Students.DeptID = Departments.DeptID;

This query matches rows where DeptID is the same in both tables and combines the related data.

- **SQL Group By**

1. **What is the GROUP BY clause in SQL? How is it used with aggregate functions?**

ANS:

The GROUP BY clause in SQL is used to **group rows that have the same values** in specified columns into summary rows. It is often used along with **aggregate functions** like COUNT(), SUM(), AVG(), MAX(), and MIN() to perform calculations on each group of data.

Purpose:
It helps in generating summarized or categorized reports — for example, total sales per region, average salary per department, etc.

Syntax:

SELECT column_name, aggregate_function(column_name)

FROM table_name

GROUP BY column_name;

Example:
To find the total salary paid in each department:

SELECT DeptID, SUM(Salary) AS TotalSalary

FROM Employees

GROUP BY DeptID;

This query groups employees by department and calculates the total salary for each department.

2. **Explain the difference between GROUP BY and ORDER BY.**

ANS:

Both GROUP BY and ORDER BY are SQL clauses used to organize query results, but they serve **different purposes**.

1. Purpose:

- GROUP BY is used to **group rows** that have the same values in one or more columns, often used with **aggregate functions** like SUM(), AVG(), or COUNT().
- ORDER BY is used to **sort the result set** in ascending (ASC) or descending (DESC) order based on one or more columns.

2. Usage:

- GROUP BY summarizes data.
- ORDER BY arranges data.

Example:

SELECT DeptID, SUM(Salary) AS TotalSalary

FROM Employees

GROUP BY DeptID

ORDER BY TotalSalary DESC;

Here, GROUP BY groups employees by department to calculate total salary per department, and ORDER BY sorts those totals in descending order.

3. Aggregate Functions:

- GROUP BY is typically used with aggregate functions.
- ORDER BY can be used with or without aggregate functions.

4. Output:

- GROUP BY reduces the number of rows by grouping similar data.
- ORDER BY only changes the **order** of rows without reducing them.

In short, GROUP BY **categorizes** data, while ORDER BY **organizes** it.

- **SQL Stored Procedure**

1. **What is a stored procedure in SQL, and how does it differ from a standard SQL query?**

ANS:

A **stored procedure** in SQL is a **predefined set of SQL statements** that are stored and executed on the database server. It allows you to perform operations like inserting, updating, or retrieving data with a single call.

Stored procedures help in **automating repetitive tasks**, **enhancing performance**, and **improving security** by controlling access to underlying data.

Syntax:

```
CREATE PROCEDURE GetEmployeeDetails
AS
BEGIN
  SELECT * FROM Employees;
END;
```

To execute it:

EXEC GetEmployeeDetails;

Difference Between Stored Procedure and Standard SQL Query:

Aspect	Stored Procedure	Standard SQL Query
Definition	A saved and reusable block of SQL statements.	A single SQL statement executed once.
Execution	Executed by calling its name.	Executed directly each time.
Performance	Faster due to precompilation and caching.	Slower as it compiles each time.
Reusability	Can be reused multiple times.	Must be written and executed each time.
Security	Provides controlled access through permissions.	Directly exposes tables and data.

2. **Explain the advantages of using stored procedures.**

ANS:

Stored procedures offer several benefits in terms of **performance, security, and maintainability** in database management systems.

1. Improved Performance:
Stored procedures are **precompiled and stored** in the database, so they execute faster than individual SQL queries. The database engine optimizes and caches the execution plan, reducing processing time.

2. Reusability and Modularity:
Once created, a stored procedure can be **reused multiple times** by different applications or users, promoting code reuse and consistency across the system.

3. Enhanced Security:
Stored procedures help in **restricting direct access** to tables. Users can be given permission to execute procedures without granting access to the underlying data, ensuring data protection.

4. Reduced Network Traffic:
Since multiple SQL statements are executed in a single call to the stored procedure, it **reduces communication** between the application and the database server, improving performance.

5. Easier Maintenance:
Business logic can be centralized inside stored procedures. Any change in logic can be made once in the procedure rather than modifying multiple queries in applications.

6. Error Handling and Control:
Stored procedures allow **conditional logic** and **error handling** using control-of-flow statements like IF, BEGIN, END, and TRY-CATCH.

- **SQL View**

1. **What is a view in SQL, and how is it different from a table?**

ANS:

A **view** in SQL is a **virtual table** that displays data from one or more tables using a **stored SQL query**. It does not store data physically; instead, it presents data dynamically whenever the view is accessed.

Views are mainly used to **simplify complex queries**, **enhance security**, and **provide customized data access** to users.

Syntax:

```
CREATE VIEW ViewName AS
SELECT column1, column2
FROM TableName
WHERE condition;
```

Example:

```
CREATE VIEW ActiveEmployees AS
SELECT Name, Department
FROM Employees
WHERE Status = 'Active';
```

When you query this view using SELECT * FROM ActiveEmployees;, it displays only active employees.

Difference Between View and Table:

Aspect	View	Table
Definition	A virtual table based on a query.	A physical structure that stores data.
Data Storage	Does not store data; retrieves it from base tables.	Stores actual data in the database.
Updation	Generally read-only (some can be updatable).	Fully updatable.
Purpose	Used to simplify queries and restrict data access.	Used to store and manage data.
Performance	Slower, as data is fetched dynamically.	Faster, since data is stored physically.

2. **Explain the advantages of using views in SQL databases.**

ANS:

Views provide a **virtual table** based on queries and offer several benefits for database management and usability.

1. Simplify Complex Queries:
They allow users to access complex joins, filters, and calculations easily without rewriting the full query.

2. Enhance Security:
Views can restrict access to certain columns or rows, so users see only the data they are allowed to access.

3. Reusability:
A single view can be used in multiple queries or applications, reducing duplicate SQL code.

4. Data Abstraction:
Views hide the complexity of underlying tables, providing a simplified and logical representation of the data.

5. Consistency:
Views always show up-to-date data from the base tables, ensuring consistent results.

6. Easy Maintenance:
Modifications to table structures can be handled through views, minimizing changes in application queries.

- **SQL Triggers**

1. **What is a trigger in SQL? Describe its types and when they are used.**

ANS:

A **trigger** in SQL is a **special kind of stored procedure** that automatically executes in response to certain events on a table or view. Triggers are used to **enforce business rules, maintain data integrity, and automate tasks** without requiring manual intervention.

Syntax (Basic Example):

```
CREATE TRIGGER TriggerName
AFTER INSERT ON TableName
FOR EACH ROW
BEGIN
  -- SQL statements
END;
```

Types of Triggers:

1. **BEFORE Trigger:**

- **Executes before an INSERT, UPDATE, or DELETE operation.**
- **Used to validate or modify data before it is saved to the table.**
  Example: Ensuring a salary is not negative before inserting a record.

2. **AFTER Trigger:**

- **Executes after an INSERT, UPDATE, or DELETE operation.**
- **Used to perform actions after changes, like logging or updating related tables.**
  Example: Automatically updating an audit table after a new employee is added.

3. **INSTEAD OF Trigger:**

- **Replaces the standard action of an operation on a view or table.**
- **Useful for updating complex views that cannot be directly modified.**
  Example: Modifying a view that joins multiple tables.

Uses of Triggers:

- **Automatically enforce data integrity rules.**
- **Maintain audit trails or logs of changes.**
- **Synchronize changes across related tables.**
- **Perform automatic calculations or updates when data changes.**

2. **Explain the difference between INSERT, UPDATE, and DELETE triggers.**

ANS:

Triggers in SQL are automatic actions that execute in response to changes in a table. Depending on the type of data operation, triggers are classified as INSERT, UPDATE, or DELETE triggers.

1. INSERT Trigger:

- **Fires automatically when a new row is added to a table.**
- **Used to validate or modify data before or after insertion or to update related tables.**
  Example: Automatically logging a new employee entry into an audit table.

2. UPDATE Trigger:

- **Fires when an existing row is modified in a table.**
- **Used to check or enforce constraints, update related tables, or maintain historical data.**
  Example: Recording changes in a student's grade whenever it is updated.

3. DELETE Trigger:

- **Fires when a row is deleted from a table.**

- **Used to maintain data integrity, prevent deletion under certain conditions, or log deletions.**
  Example: Archiving deleted customer records into a backup table.

Summary:

- **INSERT trigger → Responds to new data being added.**
- **UPDATE trigger → Responds to changes in existing data.**
- **DELETE trigger → Responds to removal of data.**

- **Introduction to PL/SQL**

1. **What is PL/SQL, and how does it extend SQL's capabilities?**

ANS:

PL/SQL (Procedural Language/Structured Query Language) is Oracle's **procedural extension of SQL**. It combines SQL's data manipulation capabilities with **procedural programming features**, allowing more complex and powerful database operations.

Key Features of PL/SQL:

- Supports **variables, loops, and conditional statements** (IF, FOR, WHILE).
- Allows creation of **procedures, functions, packages, and triggers**.
- Provides **exception handling** to manage runtime errors.
- Enables **modular programming** and code reusability.

How PL/SQL Extends SQL:

- SQL alone is **declarative** and executes single statements, while PL/SQL allows **procedural logic** to control execution flow.
- Enables **batch processing** of multiple SQL statements in a single block.

- Supports **complex business logic**, like loops, calculations, and conditional operations, which cannot be handled by standard SQL alone.
- Allows **error handling**, making programs more robust and reliable.

Example:

```
DECLARE
  total_salary NUMBER;
BEGIN
  SELECT SUM(Salary) INTO total_salary FROM Employees;
  IF total_salary > 100000 THEN
    DBMS_OUTPUT.PUT_LINE('Total salary exceeds limit');
  END IF;
END;
```

2. **List and explain the benefits of using PL/SQL.**

ANS:

PL/SQL extends SQL by adding **procedural features**, which provide several advantages for database programming and management.

1. Improved Performance:
PL/SQL allows **batch execution** of multiple SQL statements in a single block, reducing communication between the application and the database server and improving performance.

2. Procedural Capabilities:
It supports **loops, conditions, and variables**, enabling developers to write complex logic that cannot be achieved with standard SQL alone.

3. Modularity and Reusability:
PL/SQL allows the creation of **procedures, functions, packages, and triggers**, which can be reused across applications, promoting consistent and maintainable code.

4. Exception Handling:
Built-in error handling mechanisms let developers **catch and manage runtime errors**, making applications more robust and reliable.

5. Security and Data Integrity:
By encapsulating business logic in PL/SQL blocks, you can **restrict direct access to tables** and ensure rules and constraints are consistently applied.

6. Integration with SQL:
PL/SQL is fully compatible with SQL, allowing **seamless execution of SQL queries** within procedural blocks for data manipulation.

7. Maintainability:
Centralized code in procedures, functions, and packages simplifies **updates and maintenance**, as changes can be made in one place without modifying multiple queries.

- **PL/SQL Control Structures**

1. **What are control structures in PL/SQL? Explain the IF-THEN and LOOP control structures.**

ANS:

Control structures in PL/SQL are used to **control the flow of execution** within a program. They allow conditional execution, repetition, and decision-making, similar to procedural programming languages.

1. IF-THEN Control Structure:
The IF-THEN structure is used to **execute a block of code only if a specified condition is true**.

Syntax:

```
IF condition THEN
  -- statements to execute
```

```
END IF;
```

Example:

```
DECLARE
  salary NUMBER := 5000;
BEGIN
  IF salary > 4000 THEN
    DBMS_OUTPUT.PUT_LINE('Salary is above 4000');
  END IF;
END;
```

Here, the message is displayed only if the salary is greater than 4000.

2. LOOP Control Structure:

The LOOP structure is used to **execute a block of code repeatedly** until a certain condition is met or an exit statement is encountered.

Syntax:

```
LOOP
  -- statements to execute
  EXIT WHEN condition;
END LOOP;
```

Example:

```
DECLARE
  counter NUMBER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('Counter: ' || counter);
    counter := counter + 1;
    EXIT WHEN counter > 5;
```

```
  END LOOP;
END;
```

This loop prints counter values from 1 to 5.

2. **How do control structures in PL/SQL help in writing complex queries?**

ANS:

Control structures in PL/SQL allow developers to **add logic, conditions, and repetition** to SQL operations, making it possible to handle complex tasks that cannot be achieved with standard SQL alone.

1. Conditional Execution:
Structures like IF-THEN-ELSE allow queries to **execute different SQL statements based on specific conditions**, enabling dynamic decision-making.
Example: Executing different updates for employees depending on their department or salary.

2. Repetition and Iteration:
Loops (LOOP, WHILE, FOR) allow repeated execution of SQL statements over a set of data or until a condition is met.
Example: Processing and updating multiple rows in a table one by one or in batches.

3. Modular Logic:
Control structures can be combined with **procedures, functions, and triggers**, enabling complex workflows and calculations within the database.
Example: Automatically adjusting inventory levels and sending alerts when stock falls below a threshold.

4. Error Handling and Validation:
With EXCEPTION blocks, control structures can handle errors gracefully and ensure data integrity while executing multiple related SQL operations.

- **SQL Cursors**

1. **What is a cursor in PL/SQL? Explain the difference between implicit and explicit cursors.**

ANS:

A **cursor** in PL/SQL is a **pointer that allows you to retrieve and manipulate rows returned by a query one at a time**. Cursors are used when a query returns **multiple rows** and you need to process each row individually.

Types of Cursors:

1. Implicit Cursor:

- Automatically created by PL/SQL when a **SELECT statement returns a single row** or for DML statements (INSERT, UPDATE, DELETE).
- Managed internally by PL/SQL, so you don't need to declare or open it.
- You can access its attributes like %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN.

Example:

```
BEGIN
  UPDATE Employees
  SET Salary = Salary + 500
  WHERE DeptID = 10;
  IF SQL%ROWCOUNT > 0 THEN
    DBMS_OUTPUT.PUT_LINE('Rows updated: ' || SQL%ROWCOUNT);
  END IF;
END;
```

2. Explicit Cursor:

- Must be **declared, opened, fetched, and closed** manually.
- Used when a query returns **multiple rows**, allowing row-by-row processing.
- Provides more control over fetching and processing each record.

Example:

```
DECLARE
  CURSOR emp_cursor IS
    SELECT Name, Salary FROM Employees WHERE DeptID = 10;
  emp_record emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO emp_record;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(emp_record.Name || ': ' || emp_record.Salary);
  END LOOP;
  CLOSE emp_cursor;
END;
```

Aspect	**Implicit Cursor**	**Explicit Cursor**
Declaration	Automatically created	Must be explicitly declared
Usage	Single-row queries or DML	Multi-row queries

Aspect	Implicit Cursor	Explicit Cursor
Control	Limited	Full control over row-by-row processing
Management	Handled by PL/SQL	Must manually OPEN, FETCH, and CLOSE

2. **When would you use an explicit cursor over an implicit one?**

ANS:

An **explicit cursor** is used in PL/SQL when you need **more control over query results**, especially for **queries that return multiple rows**. Implicit cursors are limited to single-row operations or automatic DML handling.

Situations for Using Explicit Cursors:

1. **Processing Multiple Rows:**
   When a query returns more than one row and you want to **process each row individually**.
2. **Custom Control Over Fetching:**
   You need to **open, fetch, and close** the cursor manually, controlling the flow of row-by-row processing.
3. **Complex Business Logic:**
   When each row requires **conditional processing or calculations** before performing an operation.
4. **Accessing Cursor Attributes:**
   Explicit cursors allow you to **check status attributes** like %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN for finer control.

Example:
If you want to print the names and salaries of all employees in a department one by one:

```
DECLARE
  CURSOR emp_cursor IS
```

```
    SELECT Name, Salary FROM Employees WHERE DeptID = 10;
  emp_record emp_cursor%ROWTYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO emp_record;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(emp_record.Name || ': ' || emp_record.Salary);
  END LOOP;
  CLOSE emp_cursor;
END;
```

- **Rollback and Commit Savepoint**

1. **Explain the concept of SAVEPOINT in transaction management. How do ROLLBACK and COMMIT interact with savepoints?**

ANS:

A **SAVEPOINT** in SQL is a **named point within a transaction** that allows partial rollback of changes without affecting the entire transaction. It provides **fine-grained control** over database modifications, enabling you to undo specific parts of a transaction if needed.

Syntax to Create a Savepoint:

```
SAVEPOINT savepoint_name;
```

Interaction with ROLLBACK:

- You can **rollback to a specific savepoint** without undoing all changes made in the transaction.
- This allows selective undoing of operations.

Example:

```
BEGIN TRANSACTION;

INSERT INTO Employees VALUES (101, 'Rahul', 5000);

SAVEPOINT sp1;

UPDATE Employees SET Salary = 5500 WHERE EmpID = 101;

ROLLBACK TO sp1;  -- Undo only the update, insert remains

COMMIT;          -- Commit the remaining changes
```

Interaction with COMMIT:

- When a transaction is committed, **all savepoints within it are removed**, and changes become permanent.
- You cannot rollback to a savepoint after a commit.

Benefits of Savepoints:

- Enable **partial undo** in long transactions.
- Help maintain **data integrity** by allowing recovery from specific errors.
- Improve **flexibility** in transaction management by controlling which operations to keep or discard.

2. **When is it useful to use savepoints in a database transaction?**

ANS:

Savepoints are useful in database transactions when you need **fine-grained control over changes** and the ability to **partially undo operations** without affecting the entire transaction.

1. Long or Complex Transactions:
In transactions involving multiple steps, savepoints allow you to **rollback only the erroneous part** while keeping the correct changes intact.

2. Error Recovery:
If an error occurs midway through a transaction, you can **rollback to a savepoint** instead of discarding all previous work.

3. Conditional Logic in Transactions:
When different operations depend on certain conditions, savepoints allow selective undoing of changes based on **business rules or validations**.

4. Maintaining Data Integrity:
Savepoints help ensure that only **valid and consistent changes** are committed, reducing the risk of partial or inconsistent data updates.

Example:

```
BEGIN TRANSACTION;

INSERT INTO Orders VALUES (101, 'ProductA', 5);

SAVEPOINT sp1;

UPDATE Inventory SET Quantity = Quantity - 5 WHERE ProductID = 'ProductA';

-- Suppose an error occurs

ROLLBACK TO sp1;  -- Undo only the inventory update, order remains

COMMIT;
```