Module- 4

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

Ans: SQL stands for **Structured Query Language**.
It is a **standard programming language** specifically designed for managing and manipulating relational databases.

What SQL is:

- SQL allows you to **communicate with a database** to perform operations like:
  - **Create** databases and tables.
  - **Insert** data into tables.
  - **Retrieve** (query) data with conditions.
  - **Update** existing records.
  - **Delete** records.
  - **Control** access to data (permissions).

Why SQL is essential in database management:

1. **Universal Language for Databases**
   - Almost all relational database systems (like MySQL, PostgreSQL, Oracle, SQL Server, SQLite) use SQL, making it the common standard.
2. **Efficient Data Handling**
   - SQL enables **fast querying** and retrieval of large amounts of data with just a few commands.
3. **Data Manipulation**
   - It simplifies adding, modifying, and deleting data without needing complex programming.
4. **Data Security and Access Control**
   - SQL provides **user roles and permissions** to protect sensitive data.

5. **Data Integrity**
    - With constraints (like PRIMARY KEY, FOREIGN KEY, UNIQUE), SQL maintains accuracy and consistency of stored data.
6. **Scalability**
    - SQL databases can handle data ranging from small applications to enterprise-level systems with millions of records.

(2). Explain the difference between DBMS and RDBMS.

Ans: **DBMS (Database Management System)**

- A **software system** used to store, manage, and retrieve data.
- Data can be stored in **files, hierarchical forms, or navigational models**.
- Examples: **Microsoft Access, dBASE, File System**.

RDBMS (Relational Database Management System)

- A **type of DBMS** that stores data in the form of **tables (rows and columns)**.
- Uses **relationships (keys)** to link data between tables.
- Based on **E. F. Codd's relational model**.
- Examples: **MySQL, Oracle, PostgreSQL, SQL Server**.

Feature	**DBMS**	**RDBMS**
Data Storage	Stores data as files or hierarchical structures.	Stores data in **tables (rows & columns)**.
Relationships	Does **not support relationships** between data.	Supports **relationships** using primary & foreign keys.
Normalization	No normalization (may lead to redundancy).	Supports **normalization** to reduce redundancy.

Feature	DBMS	RDBMS
Data Integrity	Limited constraints on data.	Provides strong constraints (e.g., PRIMARY KEY, FOREIGN KEY, UNIQUE).
Transactions	Does not fully support **ACID properties**.	Fully supports **ACID properties** (Atomicity, Consistency, Isolation, Durability).
Data Volume	Suitable for small amounts of data.	Suitable for large-scale applications with huge data.
Examples	MS Access, dBASE, File system.	MySQL, Oracle, PostgreSQL, SQL Server.

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

Ans: **Role of SQL in Managing Relational Databases**

SQL (Structured Query Language) is the **standard language** used to interact with relational databases. It acts as a **bridge** between users/programs and the database engine.

1. Data Definition (DDL – Data Definition Language)

SQL helps in **defining the structure** of a relational database.

- Create, modify, or delete databases and tables.
- Commands: CREATE, ALTER, DROP.

2. Data Manipulation (DML – Data Manipulation Language)

SQL lets you **insert, update, delete, and retrieve** data stored in relational tables.

- Commands: INSERT, UPDATE, DELETE, SELECT.

3. Data Querying (DQL – Data Query Language)

The most important role of SQL is **querying relational data**.

- Retrieve specific records using conditions, sorting, grouping, and joins.

- Command: SELECT.

4. Data Control (DCL – Data Control Language)

SQL manages **permissions and security** of relational databases.

- Commands: GRANT, REVOKE.

5. Transaction Management (TCL – Transaction Control Language)

SQL ensures **data consistency** in relational databases using transactions.

- Commands: COMMIT, ROLLBACK, SAVEPOINT.

(4). What are the key features of SQL?

Ans: **Key Features of SQL**

1. **Declarative Language**
    - SQL is **declarative**, meaning you **specify *what* you want**, not *how* to get it.
    - Example: SELECT Name FROM Students WHERE Age > 18;
2. **Data Definition (DDL)**
    - SQL can **define database structures** like tables, indexes, and schemas.
    - Commands: CREATE, ALTER, DROP.
3. **Data Manipulation (DML)**
    - SQL allows you to **insert, update, and delete data** in relational tables.
    - Commands: INSERT, UPDATE, DELETE.
4. **Data Querying (DQL)**
    - SQL can **retrieve specific data** from one or more tables using queries.
    - Command: SELECT with **filtering, sorting, grouping, and joins**.
5. **Data Control (DCL)**

   - SQL can **control user access and permissions** on databases.
   - Commands: GRANT, REVOKE.
6. **Transaction Management (TCL)**
   - SQL supports **transaction handling** to maintain data integrity.
   - Commands: COMMIT, ROLLBACK, SAVEPOINT.
7. **Handling Large Volumes of Data**
   - SQL can efficiently **manage, search, and manipulate large datasets** in relational databases.
8. **Data Integrity & Constraints**
   - SQL supports **constraints** to ensure **accuracy and consistency** of data.
   - Examples: PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK.
9. **Standardized Language**
   - SQL is a **universal language** supported by almost all RDBMS systems like **MySQL, Oracle, PostgreSQL, SQL Server**.
10. **Support for Complex Queries**
    - SQL allows **joins, subqueries, nested queries, and aggregate functions** for advanced data analysis.

(5). What are the basic components of SQL syntax?

Ans: **1. Keywords**

- Reserved words that define actions in SQL.
- Examples: SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, WHERE, FROM.
- **Note:** Keywords are **not case-sensitive** in SQL, but conventionally written in uppercase.

---

2. Identifiers

- Names given to **database objects** like tables, columns, views, indexes.
- Examples: Students (table), StudentID, Name (columns).
- Rules:
    - Must start with a letter.
    - Can include letters, numbers, and underscores.
    - Cannot be a reserved keyword.

3. Clauses

- Parts of SQL statements that **specify conditions or actions**.
- Examples:
    - SELECT ... FROM ... WHERE ...
    - ORDER BY, GROUP BY, HAVING

4. Expressions

- Combine **columns, operators, and values** to produce a result.
- Examples:
    - Age > 18
    - Salary * 12

5. Predicates

- Used to **filter data based on conditions**.
- Examples:
    - =, <>, >, <, BETWEEN, LIKE, IN

6. Functions

- Predefined operations to **manipulate data**.
- Examples:
    - Aggregate functions: SUM(), AVG(), COUNT(), MAX(), MIN()

    - String functions: CONCAT(), UPPER(), LOWER()

7. Operators

- Symbols used to perform **arithmetic, logical, or comparison operations**.
- Examples:
    - Arithmetic: +, -, *, /
    - Comparison: =, <, >
    - Logical: AND, OR, NOT

8. Statements

- Complete SQL commands that perform an action.
- Examples:
    - SELECT Name FROM Students WHERE Age > 18;
    - INSERT INTO Students (StudentID, Name) VALUES (1, 'Harshil');

A **basic SQL statement** generally follows this pattern:

```
SELECT column1, column2
FROM table_name
WHERE condition
ORDER BY column1;
```

Keywords: SELECT, FROM, WHERE, ORDER BY

Identifiers: column1, column2, table_name

Expressions & Predicates: condition

Statement ends with a semicolon (;).

(6). Write the general structure of an SQL SELECT statement.

Ans: General Syntax :-

```
SELECT column1, column2, ...
FROM table_name
WHERE condition
GROUP BY column1, column2, ...
HAVING condition
ORDER BY column1 [ASC|DESC];
```

Explanation of Each Clause

1. **SELECT**
   - Specifies the columns you want to retrieve.
   - Example: SELECT Name, Age
2. **FROM**
   - Specifies the table from which to retrieve data.
   - Example: FROM Students
3. **WHERE** *(optional)*
   - Filters records based on a condition.
   - Example: WHERE Age > 18
4. **GROUP BY** *(optional)*
   - Groups rows that have the same values in specified columns, often used with aggregate functions.
   - Example: GROUP BY Department
5. **HAVING** *(optional)*
   - Filters groups based on a condition (used with GROUP BY).
   - Example: HAVING COUNT(StudentID) > 5
6. **ORDER BY** *(optional)*
   - Sorts the result set in ascending (ASC) or descending (DESC) order.
   - Example: ORDER BY Age DESC

(7). Explain the role of clauses in SQL statements.

Ans: **What is a Clause?**

A **clause** is a **component of an SQL statement** that performs a specific function, such as **filtering data, grouping rows, or specifying the table to query**. Most SQL statements are made up of multiple clauses.

Key Clauses in SQL and Their Roles

1. SELECT Clause

- **Role:** Specifies the columns or expressions to retrieve from the table(s).
- **Example:**

```
SELECT Name, Age
FROM Students;
```

2. FROM Clause

- **Role:** Indicates the table(s) from which to fetch the data.
- **Example:**

```
SELECT Name
FROM Students;
```

3. WHERE Clause *(optional)*

- **Role:** Filters rows based on a specified condition.
- **Example:**

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

4. GROUP BY Clause *(optional)*

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

```
SELECT Department, COUNT(*)
FROM Students
GROUP BY Department;
```

5. HAVING Clause *(optional)*

- **Role:** Filters groups created by GROUP BY based on a condition.

- **Example:**

```sql
SELECT Department, COUNT(*)
FROM Students
GROUP BY Department
HAVING COUNT(*) > 5;
```

6. ORDER BY Clause *(optional)*

- **Role:** Sorts the result set by one or more columns, in **ascending (ASC)** or **descending (DESC)** order.
- **Example:**

```sql
SELECT Name, Age
FROM Students
ORDER BY Age DESC;
```

(8). What are constraints in SQL? List and explain the different types of constraints

Ans: **What are Constraints?**

Constraints are **rules applied to table columns** to **enforce data integrity and consistency** in a relational database.
They ensure that the data entered into the table is **valid, accurate, and reliable**.

Constraints can be applied **when creating a table** (CREATE TABLE) or **after the table is created** (ALTER TABLE).

Types of Constraints in SQL

1. PRIMARY KEY

- **Role:** Uniquely identifies each record in a table.
- **Rules:**
  - Must be **unique** for each row.
  - Cannot contain NULL values.
- **Example:**

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

2. FOREIGN KEY

- **Role:** Ensures **referential integrity** by linking a column to a **primary key in another table**.
- **Example:**

```sql
CREATE TABLE Enrollment (
    EnrollmentID INT PRIMARY KEY,
    StudentID INT,
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID)
);
```

3. UNIQUE

- **Role:** Ensures all values in a column are **distinct**.
- **Difference from Primary Key:** Can allow one NULL value; a table can have **multiple UNIQUE constraints**.
- **Example:**

```sql
CREATE TABLE Students (
    Email VARCHAR(50) UNIQUE
);
```

4. NOT NULL

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

```sql
CREATE TABLE Students (
    Name VARCHAR(50) NOT NULL
);
```

5. CHECK

- **Role:** Ensures that **column values meet a specific condition**.
- **Example:**

```
CREATE TABLE Students (
    Age INT CHECK (Age >= 18)
);
```

6. DEFAULT

- **Role:** Provides a **default value** for a column if no value is specified during insertion.
- **Example:**

```
CREATE TABLE Students (
    Status VARCHAR(10) DEFAULT 'Active'
);
```

7. AUTO_INCREMENT / IDENTITY *(specific to some RDBMS)*

- **Role:** Automatically generates a unique number for new rows, often used for primary keys.
- **Example (MySQL):**

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

(9). How do PRIMARY KEY and FOREIGN KEY constraints differ?

Ans: **PRIMARY KEY**

- **Definition**: A constraint that uniquely identifies each record (row) in a table.
- **Properties**:
    1. A table can have **only one primary key** (though it may consist of multiple columns = *composite key*).

2. The column(s) defined as a primary key must be **unique** and **NOT NULL**.
3. It ensures **entity integrity** (each row is unique).

- **Example**:

```sql
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    name VARCHAR(50),
    age INT
);
```

- Here, student_id uniquely identifies each student.

FOREIGN KEY

- **Definition**: A constraint that links one table's column(s) to the **PRIMARY KEY (or UNIQUE key)** of another table.
- **Properties**:
    1. A table can have **multiple foreign keys**.
    2. A foreign key column may allow **NULL values** (unless otherwise restricted).
    3. It ensures **referential integrity** (values in the foreign key column must exist in the referenced table).
- **Example**:

```sql
CREATE TABLE Enrollments (
    enrollment_id INT PRIMARY KEY,
    student_id INT,
    course_id INT,
    FOREIGN KEY (student_id) REFERENCES Students(student_id)
```

Here, student_id in Enrollments must match a valid student_id in Students.

Primary Key = identity of a record in its own table

Foreign Key = link to a record in another table

(10). What is the role of NOT NULL and UNIQUE constraints?

Ans: **1. NOT NULL Constraint**

- **Role**: Ensures that a column **cannot have NULL values**.
- **Purpose**: Guarantees that every row has a valid (non-missing) value in that column.
- **Key Point**: It enforces **mandatory data entry** for that field.
- **Example**:

```
CREATE TABLE Employees (
    emp_id INT NOT NULL,
    name VARCHAR(50) NOT NULL,
    department VARCHAR(30)
);
```

- Here:
    - emp_id and name must always have values.
    - department can be NULL if not provided.

2. UNIQUE Constraint

- **Role**: Ensures that all values in a column (or group of columns) are **unique across rows**.
- **Purpose**: Prevents duplicate entries in a column or a set of columns.
- **Key Point**: Unlike PRIMARY KEY, a table can have **multiple UNIQUE constraints**, and **NULL values are allowed** (but only one NULL per column in most databases).
- **Example**:

```
CREATE TABLE Users (
    user_id INT PRIMARY KEY,
    email VARCHAR(100) UNIQUE,
    phone VARCHAR(15) UNIQUE
);
```

Here:

- Each email must be different.
- Each phone must be different.
- Both email and phone could have NULL (but not duplicates).

In short:

- **NOT NULL** → "This field cannot be empty."
- **UNIQUE** → "This field's value must not repeat."

(11). Define the SQL Data Definition Language (DDL).

Ans: **SQL Data Definition Language (DDL)**

Definition:
DDL is a subset of SQL commands used to **define, modify, and manage the structure of database objects** such as tables, schemas, indexes, and views.

It deals with the **schema (blueprint)** of the database, not the data inside it.

Key Features of DDL

1. Used to **create, alter, and delete** database structures.
2. Changes made using DDL are **permanent** and **auto-committed** (cannot be rolled back in most databases).
3. Operates on **database objects** rather than the data stored within them.

Main DDL Commands

1. **CREATE** – Creates new database objects

```sql
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    age INT
);
```

2. **ALTER** – Modifies an existing database object

```sql
ALTER TABLE Students ADD email VARCHAR(100);
```

3. **DROP** – Deletes an existing database object (structure + data)

```
DROP TABLE Students;
```

4. **TRUNCATE** – Removes all rows from a table but keeps its structure

```
TRUNCATE TABLE Students;
```

5. **RENAME** – Renames a database object (supported in some DBMS)

```
RENAME TABLE Students TO Learners;
```

(12). Explain the CREATE command and its syntax.

Ans: **CREATE Command in SQL**

Definition

The CREATE command in SQL is a **DDL (Data Definition Language)** command used to **create new database objects** such as:

- Databases
- Tables
- Views
- Indexes
- Schemas

The most common usage is to **create a table**.

General Syntax for CREATE TABLE

```
CREATE TABLE table_name (
    column1 datatype [constraint],
    column2 datatype [constraint],
    column3 datatype [constraint],
    ...
);
```

Explanation

- **table_name** → Name of the new table.
- **column1, column2, ...** → Names of the columns.
- **datatype** → Type of data allowed (e.g., INT, VARCHAR, DATE, FLOAT).
- **constraint** (optional) → Rules on columns (e.g., NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY, CHECK, DEFAULT).

Example 1: Creating a Students Table

```
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    age INT CHECK(age >= 18),
    email VARCHAR(100) UNIQUE,
    admission_date DATE DEFAULT CURRENT_DATE
);
```

Explanation of Example

- student_id → primary key, must be unique & not null.
- name → cannot be null.
- age → must be 18 or above.
- email → must be unique.
- admission_date → default value is the current system date.

Other CREATE Variants

1. **Create Database**

```
CREATE DATABASE SchoolDB;
```

2. Create View

```
CREATE VIEW student_view AS
SELECT student_id, name FROM Students;
```

3. Create Index

```sql
CREATE INDEX idx_name ON Students(name);
```

(13). What is the purpose of specifying data types and constraints during table creation?

Ans: **1. Purpose of Specifying Data Types**

When you create a table, each column must have a **data type** that defines:

- **What kind of values** can be stored (e.g., numbers, text, dates).
- **How much memory/storage** is needed.
- **What operations** are allowed (e.g., you can add numbers but not text).

Benefits

- **Data Accuracy**: Prevents storing invalid data (e.g., "abc" in an INT column).
- **Efficient Storage**: Allocates space based on the type (e.g., INT vs VARCHAR(50)).
- **Performance**: Makes queries and indexing faster because the database knows the type of data.

Example:

```sql
age INT,           -- Only whole numbers allowed
name VARCHAR(50), -- Text up to 50 characters
dob DATE          -- Must be a valid date
```

2. Purpose of Specifying Constraints

Constraints are **rules applied to columns** to enforce **data integrity** (correctness and consistency).

Common Constraints & Their Roles

- **NOT NULL** → Value must always be provided.
- **UNIQUE** → No duplicate values allowed.

- **PRIMARY KEY** → Uniquely identifies each row (implies NOT NULL + UNIQUE).
- **FOREIGN KEY** → Ensures a relationship with another table.
- **CHECK** → Restricts values based on a condition.
- **DEFAULT** → Provides a default value if none is given.

Example:

```sql
CREATE TABLE Students (
    student_id INT PRIMARY KEY,          -- Unique identifier
    name VARCHAR(50) NOT NULL,           -- Cannot be empty
    age INT CHECK(age >= 18),            -- Must be 18 or above
    email VARCHAR(100) UNIQUE,           -- No duplicate emails
    admission_date DATE DEFAULT CURRENT_DATE -- Auto-filled if not given
);
```

In short

- **Data Types** = Define *what kind of data* can be stored and optimize performance.
- **Constraints** = Define *rules* to keep the data accurate, valid, and consistent.

(14). What is the use of the ALTER command in SQL?

Ans: **ALTER Command in SQL**

Definition

The ALTER command is a **DDL (Data Definition Language)** command used to **modify the structure of an existing database object** (most commonly a table).

It does **not delete or recreate** the object; instead, it makes structural changes.

Uses of ALTER Command

1. **Add a New Column**

```sql
ALTER TABLE Students
ADD phone VARCHAR(15);
```

2. Modify an Existing Column (Data Type / Size / Constraints)

```
ALTER TABLE Students
MODIFY name VARCHAR(100);
```

→ Changes the size of the name column to 100 characters.

(In some DBMS like SQL Server, the keyword is ALTER COLUMN instead of MODIFY.)

3. Rename a Column

```
ALTER TABLE Students
RENAME COLUMN phone TO contact_number;
```

→ Renames column phone to contact_number.
(Syntax may vary by DBMS: MySQL, Oracle, SQL Server differ.)

4. Drop (Delete) a Column

```
ALTER TABLE Students
DROP COLUMN age;
```

→ Removes the age column completely.

5. Add Constraints

```
ALTER TABLE Students
ADD CONSTRAINT uq_email UNIQUE (email);
```

6. Drop Constraints

```
ALTER TABLE Students
DROP CONSTRAINT uq_email;
```

→ Removes the constraint uq_email.

7. Rename the Table

```
ALTER TABLE Students
RENAME TO Learners;
```

→ Renames the table Students to Learners.

In short

The ALTER command is used to **change the schema (structure)** of an existing table by:

- Adding, modifying, or deleting columns
- Adding or dropping constraints
- Renaming columns or the table itself

(15). How can you add, modify, and drop columns from a table using ALTER?

Ans: **1. Add a Column**

You can add one or more new columns to a table.

Syntax:

```
ALTER TABLE table_name
ADD column_name datatype [constraint];
```

Example:

```
ALTER TABLE Students
ADD phone VARCHAR(15);
```

Adds a new column phone of type VARCHAR(15).

2. Modify an Existing Column

You can change the **datatype, size, or constraints** of a column.

Syntax (MySQL / Oracle):

```
ALTER TABLE table_name
MODIFY column_name new_datatype [constraint];
```

Syntax (SQL Server / PostgreSQL):

```
ALTER TABLE table_name
ALTER COLUMN column_name new_datatype [constraint];
```

Example:

```
ALTER TABLE Students
MODIFY name VARCHAR(100) NOT NULL;    -- MySQL / Oracle
```

```
ALTER TABLE Students
ALTER COLUMN name VARCHAR(100) NOT NULL; -- SQL Server
```

Changes the name column to allow up to 100 characters and makes it NOT NULL.

3. Drop (Delete) a Column

You can remove an existing column from the table.

Syntax:

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

Example:

```
ALTER TABLE Students
DROP COLUMN age;
```

Deletes the age column completely.

In short

- **ADD** → Add a new column.
- **MODIFY / ALTER** → Change column datatype, size, or constraints.
- **DROP** → Remove a column.

(16). What is the function of the DROP command in SQL?

Ans: **DROP Command in SQL**

Definition

The DROP command is a **DDL (Data Definition Language)** command used to **delete database objects permanently**.

Once dropped, the object and all the data inside it are **lost forever** (cannot be rolled back in most databases unless you have backups).

Functions of DROP Command

The main function is to **remove database objects** such as:

- **Database**
- **Table**
- **View**
- **Index**
- **Schema**
- **Trigger / Procedure** (in some DBMS)

Common Uses with Examples

1. **Drop a Table**

```
DROP TABLE Students;
```

Deletes the table Students along with all its data and structure.

2. Drop a Database

```
DROP DATABASE SchoolDB;
```

Removes the entire SchoolDB database and everything inside it (tables, views, indexes, etc.).

3. Drop a View

```
DROP VIEW student_view;
```

4. Drop an Index

```
DROP INDEX idx_name;
```

Removes the index idx_name from the table.

Important Notes

- **DROP = Permanent deletion** of the object and its data.
- **TRUNCATE** only removes data but keeps the structure.
- **DELETE** removes data row by row (can be rolled back if in a transaction).

(17). What are the implications of dropping a table from a database?

Ans: **Implications of Dropping a Table**

1. **Permanent Loss of Data**
   - All rows (data) in the table are deleted permanently.
   - Unless you have a backup, the data cannot be recovered.
2. **Loss of Table Structure**
   - The table definition (columns, data types, constraints, indexes) is removed.
   - You would have to **recreate the table** to use it again.
3. **Loss of Dependent Objects**
   - Any objects that depend on the table (like **views, triggers, indexes, or foreign key references**) may also be affected.
   - Some DBMS will prevent dropping if foreign key constraints exist, or will **cascade delete** dependent objects if specified.
4. **Impact on Applications**
   - Queries, reports, or applications relying on the table will **fail** after the table is dropped.
5. **Cannot be Rolled Back**
   - In most DBMS, DROP TABLE is **auto-committed**.
   - This means you **cannot undo** the operation with a simple rollback unless the database supports transactional DDL or you restore from a backup.

Example

```
DROP TABLE Students;
```

Implications:

- The Students table is completely removed.
- All student records are lost.
- Any view or query using Students will stop working.
- Any indexes or constraints on Students are gone.

(18). Define the INSERT, UPDATE, and DELETE commands in SQL.

Ans: **1. INSERT Command**

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

Syntax:

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

Example:

```
INSERT INTO Students (student_id, name, age)
VALUES (1, 'Harshil', 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.
You usually combine it with a WHERE clause to update specific rows.

Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

Example:

```
UPDATE Students
SET age = 21
WHERE student_id = 1;
```

Changes the age of the student with student_id = 1 to 21.

3. DELETE Command

Definition:
The DELETE command is used to **remove rows** from a table.
You usually combine it with a WHERE clause to delete specific rows. Without WHERE, all rows are deleted.

Syntax:

```
DELETE FROM table_name
WHERE condition;
```

Example:

```
DELETE FROM Students
WHERE student_id = 1;
```

Removes the student record with student_id = 1 from the table.

In short:

- **INSERT → Add new data**
- **UPDATE → Change existing data**
- **DELETE → Remove data**

(19). What is the importance of the WHERE clause in UPDATE and DELETE operations?

Ans: **1. Purpose of the WHERE Clause**

The WHERE clause **specifies which rows** should be affected by an UPDATE or DELETE operation. Without it, **all rows in the table** will be modified or deleted.

2. Importance in UPDATE

- **Controls which rows are updated**
- Prevents unintended changes to the entire table
- Ensures data integrity by targeting only the desired records

Example:

```
UPDATE Students
SET age = 21
WHERE student_id = 1;
```

Only the student with student_id = 1 will have their age updated.

Without WHERE:

```
UPDATE Students
SET age = 21;
```

3. Importance in DELETE

- **Controls which rows are deleted**
- Prevents accidental loss of all data in the table

Example:

```
DELETE FROM Students
WHERE student_id = 1;
```

Deletes only the student with student_id = 1.

Without WHERE:

```
DELETE FROM Students;
```

All rows in the Students table are deleted — a potentially catastrophic mistake.

(20). What is the SELECT statement, and how is it used to query data?

Ans: **1. Definition of SELECT Statement**

The SELECT statement is a **Data Query Language (DQL)** command in SQL used to **retrieve data from one or more tables** in a database.

- It **does not modify data**, only **fetches it**.
- You can use it to filter, sort, aggregate, and display data in different ways.

2. Basic Syntax

```sql
SELECT column1, column2, ...
FROM table_name
WHERE condition
GROUP BY column
HAVING condition
ORDER BY column ASC|DESC;
```

Clause	**Purpose**
SELECT	Specifies columns to retrieve (use * for all columns)
FROM	Specifies the table(s) to query
WHERE	Filters rows based on a condition
GROUP BY	Groups rows for aggregate functions (e.g., SUM, COUNT)
HAVING	Filters groups after aggregation
ORDER BY	Sorts the results in ascending (ASC) or descending (DESC) order

3. Basic Examples

a) Select all columns

```sql
SELECT *
FROM Students;
```

Retrieves all columns and all rows from the Students table.

b) Select specific columns

```
SELECT name, age
FROM Students;
```

Retrieves only the name and age columns.

c) Using WHERE clause

```
SELECT name, age
FROM Students
WHERE age >= 18;
```

Retrieves students aged 18 or above.

d) Sorting results

```
SELECT name, age
FROM Students
ORDER BY age DESC;
```

Lists students sorted by age from highest to lowest.

e) Using aggregate functions

```
SELECT COUNT(*) AS total_students
FROM Students;
```

Returns the total number of students.

(21). Explain the use of the ORDER BY and WHERE clauses in SQL queries.

Ans: **1. WHERE Clause**

Purpose

- The WHERE clause is used to **filter rows** returned by a query.
- Only the rows that satisfy the **condition** are retrieved.

Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Example

```
SELECT name, age
FROM Students
WHERE age >= 18;
```

Retrieves only students whose age is 18 or older.

Key Points

- Can use **comparison operators** (=, >, <, >=, <=, <>)
- Can use **logical operators** (AND, OR, NOT)
- Can filter with **pattern matching** (LIKE) or **ranges** (BETWEEN … AND …)

Example with AND/OR:

```
SELECT name, age
FROM Students
WHERE age >= 18 AND department = 'Science';
```

2. ORDER BY Clause

Purpose

- The ORDER BY clause is used to **sort the result set** based on one or more columns.
- Sorting can be **ascending (ASC)** or **descending (DESC)**.
- Default is **ascending**.

Syntax

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC];
```

Example

```sql
SELECT name, age
FROM Students
ORDER BY age DESC;
```

Retrieves all students sorted by age from **highest to lowest**.

Multiple column sorting example:

```sql
SELECT name, age, department
FROM Students
ORDER BY department ASC, age DESC;
```

First sorts by department alphabetically, then by age in descending order within each department.

3. Using WHERE and ORDER BY Together

```sql
SELECT name, age
FROM Students
WHERE age >= 18
ORDER BY age ASC;
```

In short:

- **WHERE = filter rows**
- **ORDER BY = sort rows**
- Together, they allow you to **get exactly the data you need, in the order you want**.

(22). What is the purpose of GRANT and REVOKE in SQL?

Ans: **1. GRANT Command**

Purpose

The GRANT command in SQL is used to **give specific privileges (permissions) to a user or role** on database objects like tables, views, or procedures.

- It controls **what actions a user can perform**.

- Helps in **database security and access control**.

Syntax

```
GRANT privilege_type
ON object_name
TO user_name
[WITH GRANT OPTION];
```

Common Privileges

- SELECT → Read data
- INSERT → Add new rows
- UPDATE → Modify existing rows
- DELETE → Delete rows
- ALL → All privileges

Example

```
GRANT SELECT, INSERT ON Students TO user1;
```

Allows user1 to **read and insert** data in the Students table.

2. REVOKE Command

Purpose

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

- Ensures users **lose access when no longer authorized**.
- Maintains **security and control** over database operations.

Syntax

```
REVOKE privilege_type
ON object_name
FROM user_name;
```

Example

```sql
REVOKE INSERT ON Students FROM user1;
```

Removes the ability of user1 to insert data into the Students table.

In short:

- **GRANT = give permissions**
- **REVOKE = take permissions away**
- Together, they allow **fine-grained access control** in SQL.

(23). How do you manage privileges using these commands?

Ans: **1. GRANT Command**

- Purpose: Assigns specific privileges to a user or role.
- Syntax:

```sql
GRANT privilege_list
ON object_name
TO user_name;
```

Example:

```sql
GRANT SELECT, INSERT
ON Employees
TO John;
```

2. REVOKE Command

- Purpose: Removes previously granted privileges from a user or role.
- Syntax:

```sql
REVOKE privilege_list
ON object_name
FROM user_name;
```

Example:

```sql
REVOKE INSERT
ON Employees
FROM John;
```

3. Types of Privileges

Some common privileges you can manage:

- **SELECT** → Read data from a table/view.
- **INSERT** → Add new rows into a table.
- **UPDATE** → Modify existing rows.
- **DELETE** → Remove rows.
- **ALL PRIVILEGES** → Grants every available privilege on the object.
- **GRANT OPTION** → Allows the user to grant their privileges to others.

In summary:

- Use **GRANT** to give users access.
- Use **REVOKE** to take away access.

(24). What is the purpose of the COMMIT and ROLLBACK commands in SQL?

Ans: In SQL, **COMMIT** and **ROLLBACK** are **transaction control commands**. They are used to manage changes made by DML statements like INSERT, UPDATE, and DELETE.

COMMIT

- **Purpose**: Saves all changes made during the current transaction permanently to the database.
- Once committed, changes **cannot be undone**.
- Example:

```sql
BEGIN TRANSACTION;

UPDATE Employees
SET Salary = Salary * 1.10
WHERE Department = 'Sales';

COMMIT;
```

ROLLBACK

- **Purpose**: Undoes all changes made during the current transaction (since the last COMMIT or SAVEPOINT).
- Example:

```sql
BEGIN TRANSACTION;

DELETE FROM Employees WHERE Department = 'HR';

ROLLBACK;
```

Key Points

- A **transaction** is a unit of work that can include multiple SQL statements.
- Until you issue COMMIT, the changes are **temporary** and visible only to you.
- ROLLBACK helps recover from mistakes before committing.

In summary:

- **COMMIT** → "Save changes permanently."
- **ROLLBACK** → "Undo changes since last commit."

(25). Explain how transactions are managed in SQL databases.

Ans: Transactions are a **core concept in SQL databases** that ensure data integrity and consistency when multiple operations happen together

What is a Transaction?

A **transaction** is a sequence of one or more SQL statements executed as a **single logical unit of work**.
Either **all changes happen** (commit) or **none happen** (rollback).

Transaction Properties (ACID)

Transactions follow the **ACID** properties:

1. **Atomicity** → "All or nothing."

   - Either the entire transaction is completed or none of it is.

2. **Consistency** → Database moves from one valid state to another valid state.
3. **Isolation** → Multiple transactions run independently without interfering with each other.
4. **Durability** → Once committed, changes are permanent, even if the system crashes.

Transaction Management Commands

1. **START / BEGIN TRANSACTION**
   - Marks the beginning of a transaction.

```
BEGIN TRANSACTION;
```

2. **COMMIT**

- Saves all the changes made in the transaction permanently.

```
COMMIT;
```

3. **ROLLBACK**

- Cancels all the changes made in the transaction (since the last commit).

```
ROLLBACK;
```

4. **SAVEPOINT**

- Creates a checkpoint inside a transaction. You can roll back only to that point instead of the full transaction.

```
SAVEPOINT sp1;

UPDATE Accounts SET Balance = Balance - 100 WHERE AccID = 1;
ROLLBACK TO sp1;  -- undo only changes after sp1
```

5. **SET TRANSACTION**

- Defines properties like isolation level.

```sql
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Example of Transaction Management

```sql
BEGIN TRANSACTION;

UPDATE Accounts
SET Balance = Balance - 500
WHERE AccID = 10
                 Ask ChatGPT

UPDATE Accounts
SET Balance = Balance + 500
WHERE AccID = 202;

IF @@ERROR <> 0   -- (in SQL Server; other DBs use similar checks)
   ROLLBACK;
ELSE
   COMMIT;
```

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

Ans: **What is a JOIN?**

A **JOIN** in SQL is used to combine rows from two or more tables based on a related column between them (usually a **primary key–foreign key** relationship).

General syntax:

```sql
SELECT columns
FROM table1
JOIN table2
ON table1.column = table2.column;
```

Types of JOINs

1. INNER JOIN

- Returns **only the rows that have matching values** in both tables.
- If no match is found, the row is excluded.

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

2. LEFT JOIN (or LEFT OUTER JOIN)

- Returns **all rows from the left table** and the **matching rows from the right table**.
- If no match exists, NULL is returned for columns of the right table.

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

3. RIGHT JOIN (or RIGHT OUTER JOIN)

- Returns **all rows from the right table** and the **matching rows from the left table**.
- If no match exists, NULL is returned for columns of the left table.

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

4. FULL OUTER JOIN

- Returns **all rows when there is a match in either table**.
- If no match exists, NULL is returned for the missing side.

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

(27). How are joins used to combine data from multiple tables?

Ans: General Syntax of a JOIN

```
SELECT table1.column, table2.column
FROM table1
JOIN table2
ON table1.common_column = table2.common_column;
```

table1 and table2 → the tables to combine.

common_column → the column that defines the relationship.

1. **INNER JOIN**

```
SELECT Employees.Name, Departments.DeptName
FROM Employees
INNER JOIN Departments
ON Employees.DeptID = Departments.DeptID;
```

ombines rows **only when DeptID matches in both tables**.
Result: Alice → HR, Bob → Sales.

2. **LEFT JOIN**

```
SELECT Employees.Name, Departments.DeptName
FROM Employees
LEFT JOIN Departments
ON Employees.DeptID = Departments.DeptID;
```

Takes **all employees** and adds department info if available.
Result: Alice → HR, Bob → Sales, Charlie → NULL.

3. **RIGHT JOIN**

```
SELECT Employees.Name, Departments.DeptName
FROM Employees
RIGHT JOIN Departments
ON Employees.DeptID = Departments.DeptID;
```

Takes **all departments** and adds employee info if available.
Result: Alice → HR, Bob → Sales, NULL → Marketing.

4. FULL OUTER JOIN

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

Combines **all employees and all departments** (matches where possible, NULL otherwise).
Result: Alice → HR, Bob → Sales, Charlie → NULL, NULL → Marketing.

Why Use Joins?

- To avoid **data duplication** (normalize databases).
- To retrieve **related information** from different tables.
- To answer real-world queries like:
  - "List all employees with their department names."
  - "Show all departments, even those with no employees."

(28). 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 **arrange identical data into groups**. It's most useful when used with **aggregate functions** (like COUNT, SUM, AVG, MAX, MIN) to perform calculations **on each group of data**, rather than the whole table.

Syntax

```sql
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE condition
GROUP BY column_name;
```

1. **Counting grouped data**

```
SELECT Product, COUNT(*) AS TotalOrders
FROM Sales
GROUP BY Product;
```

Groups rows by Product and counts orders.

Product	TotalOrders
Laptop	2
Phone	2
Tablet	1

2. **Summing grouped data**

```
SELECT Product, SUM(Quantity) AS TotalQuantity
FROM Sales
GROUP BY Product;
```

Finds total quantity sold per product.

Result:

Product	TotalQuantity
Laptop	3
Phone	5
Tablet	4

3. **Average price per product**

```
SELECT Product, AVG(Price) AS AvgPrice
FROM Sales
GROUP BY Product;
```

Product	AvgPrice
Laptop	800
Phone	500
Tablet	300

In summary:

- GROUP BY groups rows based on column values.
- Aggregate functions summarize each group (e.g., total sales per product).
- Use HAVING to filter grouped results (like WHERE for groups).

(29). Explain the difference between GROUP BY and ORDER BY

Ans: Both **GROUP BY** and **ORDER BY** are SQL clauses, but they serve **different purposes**:

GROUP BY

- **Purpose**: Groups rows that have the same values into summary rows.
- Usually used with **aggregate functions** (COUNT, SUM, AVG, MAX, MIN).
- It **changes the number of rows** returned (because rows are grouped).
- **Focus**: Summarization.

Example:

```sql
SELECT Product, SUM(Quantity) AS TotalQuantity
FROM Sales
GROUP BY Product;
```

Groups sales by product and shows total quantity for each.

Product	TotalQuantity
Laptop	3
Phone	5
Tablet	4

ORDER BY

- **Purpose**: Sorts the result set by one or more columns.
- Can be applied to **raw columns or aggregate results**.
- It does **not reduce rows**, only rearranges them.
- **Focus**: Presentation (ordering).

Example:

```
SELECT Product, Quantity
FROM Sales
ORDER BY Quantity DESC;
```

Product	Quantity
Tablet	4
Phone	3
Laptop	2
Phone	2
Laptop	1

Using Both Together:

```
SELECT Product, SUM(Quantity) AS TotalQuantity
FROM Sales
GROUP BY Product
ORDER BY TotalQuantity DESC;
```

Product	**TotalQuantity**
Phone	5
Tablet	4
Laptop	3

GROUP BY → Groups data into categories for aggregation.

ORDER BY → Sorts data (including grouped/aggregated results).

(30). 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 **precompiled collection of one or more SQL statements** (such as SELECT, INSERT, UPDATE, DELETE, or control-flow statements) that are stored in the database and can be executed as a single unit.

It acts like a function in programming: you define it once, and then you can call it multiple times with different inputs.

Key Features of a Stored Procedure:

- **Precompiled:** SQL statements inside the procedure are parsed and optimized once, which can improve performance.
- **Reusable:** Can be executed multiple times without rewriting the SQL.
- **Parameterization:** Can accept **input parameters** (to provide values), **output parameters** (to return values), or both.
- **Encapsulation:** Groups multiple SQL operations into one unit.
- **Security & Access Control:** Permissions can be granted to execute the procedure without giving direct access to the underlying tables.
- **Example:**
- **Standard SQL Query:**

```
SELECT * FROM Employees WHERE DepartmentID = 5;
```

Stored Procedure:

```
CREATE PROCEDURE GetEmployeesByDept
    @DeptID INT
AS
BEGIN
    SELECT * FROM Employees WHERE DepartmentID = @DeptID;
END;
```

Execution:

```
EXEC GetEmployeesByDept @DeptID = 5;
```

In short:
A **standard SQL query** is a one-time command, while a **stored procedure** is a reusable, parameterized, and precompiled program stored in the database.

(31). Explain the advantages of using stored procedures.

Ans: **1. Better Performance**

- Stored procedures are **precompiled and cached** by the database.
- SQL statements inside them don't need to be parsed and optimized every time they run.
- This makes repeated execution faster compared to ad-hoc queries.

2. Reusability

- Once written, a stored procedure can be **executed multiple times** by different applications or users.
- Saves time by avoiding duplication of SQL code.

3. Maintainability

- Business logic is centralized in the database.
- If requirements change, only the procedure needs to be updated—no need to modify application code everywhere.

4. Security

- Users can be granted permission to **execute the procedure** without giving them direct access to the underlying tables.
- Sensitive queries remain hidden from the client, reducing security risks.

5. Reduced Network Traffic

- Instead of sending multiple SQL statements over the network, the client can just call the procedure once.
- This lowers **network load**, especially for complex operations.

6. Transaction Management

- Stored procedures can include multiple SQL statements under a **single transaction**.
- Using COMMIT and ROLLBACK, they ensure **data consistency** and prevent partial updates.

7. Encapsulation of Logic

- Complex business rules can be packaged inside a procedure.
- This separates **database logic from application logic**, making systems more modular.

Example:
A money transfer between two bank accounts is best done in a stored procedure.
It can:

1. Subtract balance from one account,
2. Add balance to another,
3. Ensure both happen in one transaction (all or nothing).

This prevents data corruption if something goes wrong mid-process.

(32). What is a view in SQL, and how is it different from a table?

Ans: **Key Features of a View:**

- **Virtual table:** Data is not stored permanently; it is generated dynamically when queried.

- **Based on a query:** A view can use SELECT statements with joins, filters, or aggregations.
- **Reusable:** You can use a view like a table in your queries (SELECT * FROM ViewName).
- **Security:** Can restrict users from accessing certain columns or rows of the base tables.

You can create a view to show only IT employees:

```sql
CREATE VIEW IT_Employees AS
SELECT EmployeeID, Name, Salary
FROM Employees
WHERE Department = 'IT';
```

Querying the view:

```sql
SELECT * FROM IT_Employees;
```

Feature	Table	View
Data storage	Stores data physically	No physical storage; virtual
Definition	Actual database object with rows and columns	Defined by a SQL query
Updatability	Can be inserted/updated/deleted directly	Some views are read-only; updatable views have restrictions
Purpose	Stores actual data	Simplifies complex queries, hides data, or provides security
Performance	Direct access to data	Depends on underlying tables; may be slower for complex queries

Feature	**Table**	**View**
Security	Access controlled at table level	Can restrict access to certain columns/rows

In short:

- A **table** is a real storage structure in the database.
- A **view** is a **virtual table** created to simplify queries, enhance security, or present data in a specific format.

(33). Explain the advantages of using views in SQL databases.

Ans: **1. Simplify Complex Queries**

- Views can **encapsulate complex queries** involving joins, aggregations, or filters.
- Users can query the view directly without writing the complicated SQL each time.

Example:
Instead of writing a multi-table join every time, you can create a view:

```
CREATE VIEW EmployeeSalary AS
SELECT Name, Department, Salary
FROM Employees
JOIN Departments ON Employees.DepartmentID = Departments.ID;
```

Users can now just SELECT * FROM EmployeeSalary.

2. Enhance Security

- Views allow you to **restrict access** to certain columns or rows of a table.
- Users can be granted access to the view without giving direct access to the underlying tables.

Example:
A view can show employee names and departments but hide salaries:

```sql
CREATE VIEW EmployeePublicInfo AS
SELECT Name, Department
FROM Employees;
```

3. Data Abstraction

- Views **hide the complexity of the underlying database structure**.
- Users do not need to know table names or join conditions.

4. Reusability

- Once created, a view can be **used multiple times** in queries.
- This reduces code duplication and ensures consistency in results.

5. Easier Maintenance

- If the underlying table structure changes, you can **update the view** instead of rewriting queries in multiple applications.

6. Logical Data Independence

- Views provide a **level of abstraction** that separates the logical presentation of data from physical storage.
- This makes it easier to change the schema without affecting end-users.

7. Aggregated or Summarized Data

- Views can be used to provide **pre-aggregated data**, like totals or averages, for reporting purposes.

Example:

```sql
CREATE VIEW DepartmentSalary AS
SELECT Department, AVG(Salary) AS AvgSalary
FROM Employees
GROUP BY Department;
```

(34). Explain the difference between INSERT, UPDATE, and DELETE triggers.

Ans: **1. INSERT Trigger**

- **When it fires:** Automatically runs **after or before a new row is inserted** into a table.
- **Purpose:** Can be used to enforce business rules, populate audit logs, or validate data before insertion.
- **Example:**

```
CREATE TRIGGER trg_InsertEmployee
AFTER INSERT ON Employees
FOR EACH ROW
BEGIN
    INSERT INTO AuditLog(EmployeeID, Action, ActionDate)
    VALUES (NEW.EmployeeID, 'Inserted', NOW());
END;
```

- **Explanation:** Every time a new employee is added, a record is added to the AuditLog table.

2. UPDATE Trigger

- **When it fires:** Automatically runs **after or before an existing row is updated** in a table.
- **Purpose:** Can enforce constraints, track changes, or maintain history of old and new values.
- **Example:**

```
CREATE TRIGGER trg_UpdateEmployeeSalary
AFTER UPDATE ON Employees
FOR EACH ROW
BEGIN
    INSERT INTO SalaryHistory(EmployeeID, OldSalary, NewSalary, ChangeDate)
    VALUES (OLD.EmployeeID, OLD.Salary, NEW.Salary, NOW());
END;
```

- **Explanation:** When an employee's salary is updated, the old and new salaries are recorded in a history table.

3. DELETE Trigger

- **When it fires:** Automatically runs **after or before a row is deleted** from a table.
- **Purpose:** Can prevent deletion of critical data, enforce referential integrity, or log deleted records.
- **Example:**

```sql
CREATE TRIGGER trg_DeleteEmployee
BEFORE DELETE ON Employees
FOR EACH ROW
BEGIN
    INSERT INTO DeletedEmployees(EmployeeID, Name, DeletedDate)
    VALUES (OLD.EmployeeID, OLD.Name, NOW());
END;
```

- **Explanation:** Before an employee is deleted, their information is saved in a backup table.

In short:

- **INSERT triggers** → fire on adding new rows.
- **UPDATE triggers** → fire on modifying existing rows.
- **DELETE triggers** → fire on removing rows.

(35). What is PL/SQL, and how does it extend SQL's capabilities?

Ans: PL/SQL (Procedural Language/Structured Query Language) is Oracle Corporation's procedural extension to standard SQL. It combines SQL's data manipulation power with procedural programming features, allowing developers to write more complex, modular, and efficient database programs.

Here's a detailed breakdown:

1. Definition

- **SQL** is a declarative language used to query, insert, update, or delete data in a relational database.
- **PL/SQL** extends SQL by adding **procedural constructs** like loops, conditionals, variables, constants, and exceptions.
- Essentially, PL/SQL allows you to **write programs that include SQL statements**, execute them together, and handle logic flow.

2. How PL/SQL Extends SQL

SQL alone cannot handle procedural logic, error handling, or complex calculations easily. PL/SQL adds:

Feature	**Description**	**Example Use**
Variables and Constants	Store temporary data and reuse it in logic.	DECLARE v_salary NUMBER;
Control Structures	Conditional statements (IF...ELSE) and loops (FOR, WHILE).	Calculate bonuses based on salary tiers.
Procedures and Functions	Modular blocks of code for reuse.	CREATE PROCEDURE IncreaseSalary(...)
Cursors	Process multiple rows returned by a query, one at a time.	Loop through employees to apply a raise.
Exception Handling	Handle runtime errors gracefully.	Catch divide-by-zero or data-not-found errors.
Triggers	Automatic execution of code in response to database events.	Update audit tables when a row is inserted.
Packages	Group related procedures, functions, and variables.	Organize all employee-related operations in one package.

3. Structure of PL/SQL Program

A PL/SQL block has three parts:

```
DECLARE
   -- Declaration section (variables, constants)
BEGIN
   -- Execution section (SQL and procedural logic)
EXCEPTION
   -- Exception handling section
END;
```

Example:

```
DECLARE
   v_salary NUMBER := 50000;
   v_bonus  NUMBER;
BEGIN
   IF v_salary > 40000 THEN
      v_bonus := v_salary * 0.10;
   ELSE
      v_bonus := v_salary * 0.05;
   END IF;
   DBMS_OUTPUT.PUT_LINE('Bonus: ' || v_bonus);
EXCEPTION
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('An error occurred.');
END;
```

Here, PL/SQL:

- Declares variables
- Performs conditional logic
- Prints output
- Handles exceptions

4. Advantages of PL/SQL over SQL

1. **Procedural Capabilities** – Loops, conditionals, and modularity.

2. **Improved Performance** – Blocks of SQL can be executed in a single call, reducing network traffic.
3. **Error Handling** – Catch and manage runtime errors with exceptions.
4. **Code Reusability** – Functions, procedures, and packages can be reused across applications.
5. **Tight Integration with SQL** – You can embed SQL statements directly within procedural logic.

(36). List and explain the benefits of using PL/SQL.

Ans: **1. Procedural Capabilities**

- **Explanation:** Unlike standard SQL, PL/SQL allows procedural programming with **loops, conditionals, and sequential execution**.
- **Benefit:** Enables complex business logic to be implemented directly in the database.
- **Example:** Calculating employee bonuses differently based on salary tiers using IF...ELSE statements.

---

2. Tight Integration with SQL

- **Explanation:** PL/SQL can embed SQL statements directly within procedural code.
- **Benefit:** You can **manipulate data efficiently** and perform complex queries without switching between different languages.
- **Example:** Selecting rows from a table and processing each row in a loop using a cursor.

---

3. Improved Performance

- **Explanation:** PL/SQL blocks execute **as a single unit**, reducing the number of database calls.
- **Benefit:** Reduces network traffic and improves execution speed.

- **Example:** A loop that updates 1,000 records can be executed in one PL/SQL block instead of 1,000 individual SQL statements.

4. Modularity and Code Reusability

- **Explanation:** PL/SQL allows the creation of **procedures, functions, and packages**.
- **Benefit:** You can reuse code across multiple applications, making maintenance easier.
- **Example:** A CalculateTax function can be called wherever tax computation is required.

5. Exception Handling

- **Explanation:** PL/SQL provides **structured error handling** using EXCEPTION blocks.
- **Benefit:** Ensures that runtime errors are managed gracefully without crashing the application.
- **Example:** Handling division by zero or missing data errors with custom messages.

6. Security

- **Explanation:** PL/SQL allows you to **hide business logic** in the database and control access via privileges.
- **Benefit:** Sensitive operations can be executed without exposing the underlying SQL code to end users.
- **Example:** A stored procedure can update salary records without giving direct UPDATE access to the employees table.

7. Maintainability

- **Explanation:** Using **packages, procedures, and functions**, PL/SQL code is organized and modular.
- **Benefit:** Easier to debug, update, and maintain large applications.

- **Example:** All payroll-related procedures can reside in a single package, making updates straightforward.

8. Support for Triggers

- **Explanation:** PL/SQL supports **triggers**, which automatically execute when certain events occur in the database.
- **Benefit:** Automates repetitive tasks and ensures data integrity.
- **Example:** Automatically updating an audit table whenever a record is inserted or modified.

9. Portability

- **Explanation:** PL/SQL programs can run on any Oracle database without changes.
- **Benefit:** Applications are database-independent within Oracle environments, facilitating deployment across multiple systems.

(37). What are control structures in PL/SQL? Explain the IF-THEN and LOOP control structures.

Ans: PL/SQL supports three main types of control structures:

1. **Sequential control** – Executes statements in order (default).
2. **Conditional control** – Executes statements based on conditions (e.g., IF...THEN).
3. **Looping control** – Repeats statements multiple times (e.g., LOOP, WHILE, FOR).

1. IF-THEN Control Structure

The **IF-THEN** structure allows you to execute a block of code **only if a specified condition is true**.

Syntax:

```
IF condition THEN
   -- statements to execute if condition is true
ELSIF another_condition THEN
   -- statements if another condition is true
ELSE
   -- statements if no conditions are true
END IF;
```

Example:

```
DECLARE
   v_salary NUMBER := 45000;
   v_bonus  NUMBER;
BEGIN
   IF v_salary > 40000 THEN
      v_bonus := v_salary * 0.10;
   ELSE
      v_bonus := v_salary * 0.05;
   END IF;

   DBMS_OUTPUT.PUT_LINE('Bonus: ' || v_bonus);
END;
```

Explanation:

- **Checks if v_salary > 40000.**
- **If true, assigns 10% as bonus; otherwise, assigns 5%.**
- **Only one block executes based on the condition.**

2. LOOP Control Structure

The LOOP structure allows you to repeat a block of code multiple times. There are several types:

a) Basic LOOP

```
LOOP
   -- statements
   EXIT WHEN condition;  -- condition to exit the loop
END LOOP;
```

Example:

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

Explanation:

- Prints numbers 1 to 5.
- EXIT WHEN determines when the loop stops.

b) FOR LOOP

Used when the number of iterations is known.

```
FOR i IN 1..5 LOOP
   DBMS_OUTPUT.PUT_LINE('Iteration: ' || i);
END LOOP;
```

Explanation:

- Iterates exactly 5 times.
- i automatically increments from 1 to 5.

c) WHILE LOOP

Repeats as long as a condition is true.

```
DECLARE
   v_counter NUMBER := 1;
BEGIN
   WHILE v_counter <= 5 LOOP
      DBMS_OUTPUT.PUT_LINE('Count: ' || v_counter);
      v_counter := v_counter + 1;
   END LOOP;
END;
```

(38). How do control structures in PL/SQL help in writing complex queries?

Ans: **1. Conditional Logic with IF-THEN/ELSE**

- **Problem in SQL:** You cannot easily execute different actions based on conditions in a single query.
- **How PL/SQL helps:** IF-THEN-ELSE lets you **choose actions dynamically** based on data or input.
- **Example:** Calculating bonuses differently based on salary brackets.

```
IF salary > 50000 THEN
   bonus := salary * 0.10;
ELSE
   bonus := salary * 0.05;
END IF;
```

2. Repetition with LOOP, FOR, and WHILE

- **Problem in SQL:** Processing row-by-row calculations or iterative tasks is difficult in standard SQL without multiple queries.
- **How PL/SQL helps:** Loop structures let you **iterate through query results** or repeat operations as needed.
- **Example:** Giving a raise to each employee in a cursor:

```
FOR emp_rec IN (SELECT * FROM employees) LOOP
   UPDATE employees
   SET salary = salary * 1.05
   WHERE employee_id = emp_rec.employee_id;
END LOOP;
```

3. Combining Multiple SQL Statements

- **Problem in SQL:** Performing multiple operations atomically requires multiple queries, which is inefficient and hard to manage.
- **How PL/SQL helps:** Control structures allow **sequencing multiple SQL statements** within one block.
- **Example:** Insert into an audit table, update salary, and log messages—all in one block:

```
BEGIN
   INSERT INTO audit_log VALUES (...);
   UPDATE employees SET salary = salary * 1.05 WHERE dept_id = 10;
   DBMS_OUTPUT.PUT_LINE('Salaries updated successfully');
END;
```

4. Error Handling with EXCEPTION

- **Problem in SQL:** Standard SQL errors stop execution and cannot be handled gracefully.
- **How PL/SQL helps:** You can **catch and handle exceptions**, allowing complex queries to continue or log errors.
- **Example:**

```
BEGIN
   UPDATE employees SET salary = salary * 1.05;
EXCEPTION
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('No employees found.');
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('An unexpected error occurred.');
END;
```

5. Dynamic Query Execution

- Using control structures, you can **build queries dynamically** and execute them based on conditions or loops. This is not possible with plain SQL.

(39). What is a cursor in PL/SQL? Explain the difference between implicit and explicit cursors.

Ans: **1. What is a Cursor?**

- A cursor **maintains the context area** in memory for a query, storing the result set and tracking which row is currently being processed.
- Without a cursor, PL/SQL cannot process multi-row query results sequentially.
- Think of it as a **bookmark for traversing query results row by row**.

Example Conceptually:

```
SELECT * FROM employees;  -- returns multiple rows
-- Cursor lets you fetch each row individually in PL/SQL
```

2. Types of Cursors

PL/SQL supports **two types of cursors**:

A. Implicit Cursor

- Automatically created by PL/SQL **whenever you execute a SQL statement** like INSERT, UPDATE, DELETE, or a single-row SELECT.
- You **do not explicitly declare or open** it.
- Can handle queries that return **only one row** (or affect a certain number of rows).
- **Attributes** of implicit cursors:

Example:

```
BEGIN
   UPDATE employees
   SET salary = salary * 1.10
   WHERE dept_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;
```

B. Explicit Cursor

- Declared and controlled **manually** by the programmer.
- Used for **queries returning multiple rows** that need to be processed sequentially.
- Steps to use an explicit cursor:
    1. **Declare** the cursor with a SQL query.
    2. **Open** the cursor to execute the query.
    3. **Fetch** rows one by one into variables.
    4. **Close** the cursor to release resources.

Example:

```
DECLARE
   CURSOR emp_cursor IS
      SELECT employee_id, first_name, salary
      FROM employees
      WHERE dept_id = 10;

   v_emp_id employees.employee_id%TYPE;
   v_name   employees.first_name%TYPE;
   v_salary employees.salary%TYPE;
BEGIN
   OPEN emp_cursor;  -- Step 2

   LOOP
      FETCH emp_cursor INTO v_emp_id, v_name, v_salary;  -- Step 3
      EXIT WHEN emp_cursor%NOTFOUND;

      DBMS_OUTPUT.PUT_LINE(v_emp_id || ' ' || v_name || ' ' || v_salary);
   END LOOP;

   CLOSE emp_cursor;  -- Step 4
END;
```

Summary:

- **Cursor:** A pointer to process query results row by row.

- **Implicit Cursor:** Automatic, simple DML or single-row queries.
- **Explicit Cursor:** Manual, for multi-row queries and fine-grained control.

(40). When is it useful to use savepoints in a database transaction?

Ans: **1. When Savepoints Are Useful**

a) Partial Rollback

- If a transaction involves multiple operations, and one operation fails, you can **rollback only to a specific savepoint** rather than undoing the entire transaction.
- **Example:** You insert records into multiple tables. If the third table insertion fails, you can rollback to the savepoint after the second insertion instead of undoing all three insertions.

---

b) Complex Business Logic

- In transactions with **conditional operations**, savepoints allow selective rollback depending on logic conditions.
- **Example:** A banking system transfers funds between multiple accounts. If one transfer fails, only that part can be rolled back without affecting other transfers.

---

c) Error Recovery

- Savepoints provide **fine-grained control for error handling** in transactions.
- They allow you to **recover from predictable errors** and continue executing the rest of the transaction.
- **Example:** Skip a problematic row during batch processing but continue processing others.

d) Testing and Debugging

- When developing or testing stored procedures and transactions, savepoints allow **repeated partial rollbacks** without restarting the entire transaction.
- **Example:** Debugging a multi-step order processing routine.

2. How Savepoints Work

Syntax:

```
SAVEPOINT savepoint_name;

-- Some SQL operations

ROLLBACK TO savepoint_name;  -- Rolls back only to this point
```

Example:

```
BEGIN
   INSERT INTO employees VALUES (101, 'Alice', 'HR');
   SAVEPOINT sp1;

   INSERT INTO employees VALUES (102, 'Bob', 'Finance');
   SAVEPOINT sp2;

   INSERT INTO employees VALUES (103, 'Charlie', 'IT');

   -- Oops, some error in the last insert
   ROLLBACK TO sp2;  -- Undo only the last insert, keep previous ones
END;
```

Explanation:

- sp1 marks the first savepoint, sp2 the second.
- Only the insertion after sp2 is undone; previous insertions remain intact.

3. Key Points

1. Savepoints **cannot be used outside a transaction** (they work after BEGIN TRANSACTION or implicitly in Oracle).

2. You can **set multiple savepoints** within a transaction.
3. Rolling back to a savepoint **does not end the transaction**—you can continue executing subsequent operations.