Module 4 – Introduction to DBMS

Introduction to SQL

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

SQL (Structured Query Language) is a standard programming language used to **manage and manipulate relational databases**. It allows users to perform operations like storing, retrieving, updating, and deleting data.

Key Points:

1. **Definition:** SQL is a language designed for interacting with databases, especially relational databases (like MySQL, PostgreSQL, SQL Server, Oracle).
2. **Essential Functions in Database Management:**
   - **Data Retrieval:** SELECT statements let you query specific information from tables.
   - **Data Manipulation:** INSERT, UPDATE, DELETE commands allow adding, modifying, and removing data.
   - **Data Definition:** CREATE, ALTER, DROP let you define or change the structure of database tables.
   - **Data Control:** GRANT and REVOKE help manage access permissions for security.

Why SQL is Essential:

- **Efficiency:** Quickly handles large amounts of data.
- **Standardization:** SQL works across many database systems.
- **Accuracy:** Reduces errors in data handling through structured commands.
- **Data Integrity:** Ensures relationships and rules between data are maintained.

2. Explain the difference between DBMS and RDBMS.

Feature	DBMS (Database Management System)	RDBMS (Relational Database Management System)
Data Storage	Stores data as files (hierarchical or flat files)	Stores data in **tables** (rows and columns)
Data Relationship	Limited or no relationships between data	Maintains **relationships** using keys (primary & foreign keys)
Data Integrity	Less strict; may not enforce constraints	Enforces **data integrity** using constraints (e.g., UNIQUE, NOT NULL)
Normalization	Usually not supported	Supports **normalization** to reduce data redundancy
Examples	Microsoft Access, File System	MySQL, Oracle, SQL Server, PostgreSQL
Complex Queries	Limited support	Supports **complex queries** using SQL
Multi-user Access	Less efficient for multiple users	Designed for **concurrent multi-user access**

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

1. Data Retrieval

- SQL lets you **query data** from one or more tables using the SELECT statement.
- Example: Fetching all employees with a salary > 50,000.

---

2. Data Manipulation

- SQL provides commands to **add, update, or delete data**:
  - INSERT → Add new records

    - UPDATE → Modify existing records
    - DELETE → Remove records

---

3. Data Definition

- SQL defines and modifies **database structure** using:
    - CREATE TABLE → Create a new table
    - ALTER TABLE → Modify table structure
    - DROP TABLE → Delete a table

---

4. Data Control and Security

- SQL manages **access permissions**:
    - GRANT → Give users access rights
    - REVOKE → Remove access rights
- Ensures only authorized users can perform certain operations.

---

5. Maintaining Data Integrity

- SQL enforces **rules and constraints** in relational databases:
    - Primary keys, foreign keys
    - UNIQUE, NOT NULL, CHECK constraints

4. What are the key features of SQL?

1. Data Querying

- SQL allows you to **retrieve specific data** from one or more tables using the SELECT statement.

2. Data Manipulation

- Supports operations to **add, update, or delete records** using INSERT, UPDATE, and DELETE.

3. Data Definition

- Lets you **define or modify database structures** using CREATE, ALTER, and DROP commands.

4. Data Control

- SQL can **control user access and permissions** with GRANT and REVOKE commands for security.

5. Transaction Management

- Supports **transactions** to ensure data consistency (COMMIT, ROLLBACK).

6. Data Integrity

- Enforces **rules and constraints** like primary keys, foreign keys, UNIQUE, NOT NULL, and CHECK to maintain correct data.

7. Multi-User Access

- Allows **multiple users to access and manipulate data simultaneously** without conflicts.

8. Standardized Language

- SQL is **widely supported across different relational database systems**, making it portable and standard.

2. SQL Syntax

1. What are the basic components of SQL syntax?

1. Keywords / Commands

- Reserved words used to perform operations.
- Example: SELECT, INSERT, UPDATE, DELETE, CREATE, DROP.

2. Clauses

- Provide conditions or modify SQL commands.

- Example: WHERE, ORDER BY, GROUP BY, HAVING.

3. **Expressions**
    - Formulas or calculations that return values.
    - Example: salary * 0.1, price + tax.
4. **Predicates / Conditions**
    - Logical tests that return TRUE, FALSE, or UNKNOWN.
    - Example: age > 18, name = 'John'.
5. **Identifiers**
    - Names given to database objects like tables, columns, views, etc.
    - Example: students, employee_id.
6. **Literals / Constants**
    - Fixed values used in queries.
    - Example: numbers (100), strings ('India'), dates ('2025-08-31').
7. **Operators**
    - Symbols used to perform operations on data.
    - Example: =, <, >, AND, OR, LIKE, IN.
8. **Comments**
    - Notes inside SQL code, ignored during execution.
    - Example:
        - Single-line: -- this is a comment
        - Multi-line: /* comment */

2. Write the general structure of an SQL SELECT statement

The **general structure of an SQL SELECT statement** is:

```
SELECT column1, column2, ...

FROM table_name
```

```
[WHERE condition]

[GROUP BY column_list]

[HAVING condition]

[ORDER BY column_list [ASC | DESC]];
```

Explanation of parts:

1. **SELECT** → specifies the columns to retrieve.
    - Example: SELECT name, age
2. **FROM** → specifies the table(s) from which to retrieve data.
    - Example: FROM employees
3. **WHERE** (optional) → filters rows based on a condition.
    - Example: WHERE age > 25
4. **GROUP BY** (optional) → groups rows with the same values into summary rows.
    - Example: GROUP BY department_id
5. **HAVING** (optional) → filters groups (like WHERE but for groups).
    - Example: HAVING COUNT(*) > 5
6. **ORDER BY** (optional) → sorts the result set.
    - Example: ORDER BY salary DESC

Example full query:

```
SELECT department_id, COUNT(*) AS total_employees

FROM employees

WHERE salary > 30000

GROUP BY department_id

HAVING COUNT(*) > 5
```

```
ORDER BY total_employees DESC;
```

3. Explain the role of clauses in SQL statements.

Role of Clauses in SQL

1. SELECT Clause – Specifies which columns (or expressions) to retrieve.

```
SELECT name, age
FROM students;
```

2. FROM Clause – Defines the table (or tables) from which the data will be selected.

```
SELECT *
FROM employees;
```

3. WHERE Clause – Filters rows based on conditions.

```
SELECT *
FROM orders
WHERE amount > 500;
```

4. GROUP BY Clause – Groups rows that have the same values into summary rows.

```
SELECT department, COUNT(*)
FROM employees
GROUP BY department;
```

5. HAVING Clause – Applies conditions on grouped data (like a filter for GROUP BY).

```
SELECT department, COUNT(*)
FROM employees
GROUP BY department
HAVING COUNT(*) > 10;
```

6. ORDER BY Clause – Sorts the result set in ascending or descending order.

```
    SELECT name, salary
    FROM employees
    ORDER BY salary DESC;
```

7. JOIN Clause – Combines rows from two or more tables based on related columns.

```
    SELECT employees.name, departments.department_name
    FROM employees
JOIN departments ON employees.dept_id = departments.id;
```

3. SQL Constraints

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

What are Constraints in SQL?

In SQL, **constraints** are rules applied to columns in a table to **enforce data integrity and consistency**.
They ensure that only valid data is entered into the database.

Types of Constraints in SQL

1. **NOT NULL Constraint**
    - Ensures that a column cannot have a NULL value.
    - Example:
    - CREATE TABLE Students (
    -    student_id INT NOT NULL,
    -    name VARCHAR(50) NOT NULL
    -);
    - Here, student_id and name must always have values.

---

2. **UNIQUE Constraint**

   - Ensures that all values in a column are **unique** (no duplicates).
   - Example:
   - CREATE TABLE Employees (
   -     emp_id INT UNIQUE,
   -     email VARCHAR(100) UNIQUE
   -);
   - No two employees can have the same emp_id or email.

---

3. **PRIMARY KEY Constraint**

   - Uniquely identifies each row in a table.
   - A primary key is a combination of **NOT NULL** and **UNIQUE**.
   - Example:
   - CREATE TABLE Customers (
   -     customer_id INT PRIMARY KEY,
   -     name VARCHAR(100)
   -);
   - customer_id uniquely identifies each customer.

---

4. **FOREIGN KEY Constraint**

   - Links two tables by referencing the **primary key** of another table.
   - Ensures **referential integrity** (the relationship between tables remains valid).
   - Example:
   - CREATE TABLE Orders (

- order_id INT PRIMARY KEY,
- customer_id INT,
- FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
-);
- Every customer_id in Orders must exist in Customers.

---

5. **CHECK Constraint**
    - Ensures that values in a column meet a specific condition.
    - Example:
    - CREATE TABLE Products (
    - product_id INT PRIMARY KEY,
    - price DECIMAL(10,2) CHECK (price > 0)
    -);
    - price must always be greater than 0.

---

6. **DEFAULT Constraint**
    - Provides a default value for a column if no value is specified.
    - Example:
    - CREATE TABLE Accounts (
    - acc_id INT PRIMARY KEY,
    - balance DECIMAL(10,2) DEFAULT 0
    -);
    - If no balance is given, it will be set to 0 automatically.

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

```
-- Customers table with PRIMARY KEY
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100)
);
-- Orders table with FOREIGN KEY
CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
);
```

customer_id in Customers uniquely identifies each customer (PRIMARY KEY).

customer_id in Orders ensures that every order is linked to a valid customer (FOREIGN KEY).

In short:

Primary Key → Unique identity of a record.

Foreign Key → Creates a relationship between two tables.

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

NOT NULL Constraint

- Ensures a column **cannot have NULL (empty) values**.
- Role: Guarantees that important fields always contain data.
- Example:
- name VARCHAR(50) NOT NULL

Every row must have a value for name.

UNIQUE Constraint

- Ensures all values in a column are **different (no duplicates)**.
- Role: Maintains **uniqueness** of data in a column.
- Example:
- email VARCHAR(100) UNIQUE

No two rows can have the same email.

In short:

- **NOT NULL** → Prevents missing values.
- **UNIQUE** → Prevents duplicate values.

4. Main SQL Commands and Sub-commands (DDL)

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

Definition:
SQL **Data Definition Language (DDL)** is the part of SQL used to **define, create, modify, and delete** the structure of database objects such as tables, schemas, indexes, and views.

Role of DDL

- It deals with the **structure** of the database, not the data inside it.
- DDL commands are usually auto-committed (changes are permanent once executed).

Common DDL Commands

1. CREATE – Creates a new database object (e.g., table, view).

```
CREATE TABLE Students (

student_id INT PRIMARY KEY,
    name VARCHAR(50) NOT NULL
);
```

2. ALTER – Modifies an existing database object.

```
ALTER TABLE Students ADD age INT;
```

3. DROP – Deletes a database object permanently.

```
DROP TABLE Students;
```

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

```
TRUNCATE TABLE Students;
```

5. RENAME – Changes the name of a database object.

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

2. Explain the CREATE command and its syntax

Definition

The **CREATE** command is a **DDL (Data Definition Language)** command used to create new database objects such as **databases, tables, views, indexes, or schemas**.

Its most common use is to **create tables**.

Syntax of CREATE TABLE

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

Explanation

- **table_name** → Name of the new table.
- **column1, column2, …** → Names of the table's columns.
- **datatype** → Defines the type of data (e.g., INT, VARCHAR, DATE).
- **constraint** → Rules like PRIMARY KEY, NOT NULL, UNIQUE, CHECK, etc.

Example

```
CREATE TABLE Employees (
   emp_id INT PRIMARY KEY,
   name VARCHAR(100) NOT NULL,
   email VARCHAR(100) UNIQUE,
   salary DECIMAL(10,2) CHECK (salary > 0),
   department_id INT
);
```

This creates a table **Employees** with constraints:

- emp_id → Primary Key
- name → Cannot be NULL
- email → Must be unique
- salary → Must be greater than 0

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

1. Data Types

- Define **what kind of data** a column can hold (e.g., INT, VARCHAR, DATE).
- Purpose:
    - Ensure **correct storage** (numbers, text, dates, etc.).

    - Optimize **memory usage** and performance.
    - Prevent invalid data entry (e.g., you cannot insert text into an INT column).

Example:

age INT,

name VARCHAR(50)

- age can only store numbers.
- name can store up to 50 characters.

---

2. Constraints

- Define **rules** for the data stored in the table.
- Purpose:
    - Maintain **data accuracy and integrity**.
    - Enforce **business rules** (e.g., salary must be > 0).
    - Control uniqueness, relationships, and validity.

Example:

email VARCHAR(100) UNIQUE,

salary DECIMAL(10,2) CHECK (salary > 0),

emp_id INT PRIMARY KEY

- email must be unique.
- salary must always be greater than 0.
- emp_id uniquely identifies each employee.

5. ALTER Command

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

Definition:
The **ALTER** command is a **DDL (Data Definition Language)** command used to **modify an existing database object**, most commonly a **table**, without deleting it.

Uses of ALTER Command

1. **Add a new column**

   ```
   ALTER TABLE Employees ADD age INT;
   ```

   Adds a new column age to the table.

2. **Modify (change) a column's data type or size**

   ```
   ALTER TABLE Employees MODIFY name VARCHAR(150);
   ```

   Changes the size of the name column to 150 characters.

3. **Rename a column** (syntax may differ by DBMS)

   ```
   ALTER TABLE Employees RENAME COLUMN name TO full_name;
   ```

   Renames name to full_name.

4. **Drop (remove) a column**

   ```
   ALTER TABLE Employees DROP COLUMN age;
   ```

   Deletes the age column from the table.

5. **Add or drop constraints**

   ```
   ALTER TABLE Employees ADD CONSTRAINT unique_email UNIQUE(email);
   ```

   Adds a unique constraint to email.

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

ALTER Command for Columns in SQL

1. **Add a Column**

```
ALTER TABLE table_name ADD column_name datatype;
```

Example:

```
ALTER TABLE Employees ADD age INT;
```

2. **Modify a Column**

```
ALTER TABLE table_name MODIFY column_name new_datatype;
```

Example:

```
ALTER TABLE Employees MODIFY name VARCHAR(150);
```

3. **Drop a Column**

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

Example:

```
ALTER TABLE Employees DROP COLUMN age;
```

6. DROP Command

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

Definition:
The **DROP** command is a DDL (Data Definition Language) command used to **delete an existing database object permanently** (such as a table, database, view, or index).

Functions / Uses:

1. **Drop a table** – removes the table and all its data.

```
DROP TABLE Employees;
```

2. **Drop a database** – removes the entire database.

```
DROP DATABASE CompanyDB;
```

3. **Drop other objects** – can also delete views, indexes, or constraints.

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

1. **Permanent Deletion** – The table structure and all its data are removed permanently.

2. **Loss of Constraints/Indexes** – Any **primary key, foreign key, indexes, or triggers** defined on the table are also deleted.
3. **Broken Relationships** – If other tables reference it using **FOREIGN KEYS**, dropping may cause **referential integrity issues** (or be restricted until constraints are removed).
4. **No Rollback (in most DBMS)** – Once dropped, the table cannot be recovered unless a backup exists.

7. Data Manipulation Language (DML)

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

INSERT, UPDATE, and DELETE Commands in SQL

1. **INSERT Command**

- **Definition:** Used to **add new records (rows)** into a table.
- **Example:**
- INSERT INTO Students (student_id, name, age)
- VALUES (1, 'Rahul', 20);

---

2. **UPDATE Command**

- **Definition:** Used to **modify existing records** in a table.
- **Example:**
- UPDATE Students
- SET age = 21
- WHERE student_id = 1;

---

3. **DELETE Command**

- **Definition:** Used to **remove records (rows)** from a table.
- **Example:**

- DELETE FROM Students
- WHERE student_id = 1;

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

Importance of WHERE Clause in UPDATE and DELETE

- The **WHERE clause** specifies the **condition** that determines **which rows** will be updated or deleted.
- Without a WHERE clause:
    - **UPDATE** → changes **all rows** in the table.
    - **DELETE** → removes **all rows** from the table.

Examples

1. **UPDATE with WHERE**

UPDATE Employees

SET salary = salary + 1000

WHERE emp_id = 101;

Only the employee with emp_id = 101 gets a salary increment.

Without WHERE: all employees' salaries would increase.

---

2. **DELETE with WHERE**

DELETE FROM Employees

WHERE department = 'HR';

Only HR employees are deleted.

Without WHERE: all employees in the table would be deleted.

8. Data Query Language (DQL)

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

Definition:
The **SELECT** statement is a **DML command** used to **retrieve data** from one or more tables in a database.
It is the most commonly used SQL statement for querying data.

Basic Syntax

```
SELECT column1, column2, ...

FROM table_name

WHERE condition;
```

Explanation of Clauses

- **SELECT** → specifies the columns to display.
- **FROM** → specifies the table to fetch data from.
- **WHERE** → (optional) filters rows based on conditions.
- **ORDER BY / GROUP BY / HAVING** → (optional) further organize the results.

Examples

1. Retrieve all data from a table:

   ```
   SELECT * FROM Students;
   ```

2. Retrieve specific columns:

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

3. Retrieve with condition:

   ```
   SELECT name, age FROM Students

   WHERE age > 18;
   ```

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

WHERE Clause

- **Purpose:** Filters rows based on a **condition**.
- Ensures only the records that meet the condition are retrieved.
- **Example:**
- SELECT name, age
- FROM Students
- WHERE age > 18;

Returns only students older than 18.

ORDER BY Clause

- **Purpose:** Sorts the result set in **ascending (ASC)** or **descending (DESC)** order.
- Can sort by one or multiple columns.
- **Example:**
- SELECT name, age
- FROM Students
- ORDER BY age DESC;

Returns students sorted by age from highest to lowest.

9. Data Control Language (DCL)

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

GRANT and REVOKE in SQL

1. **GRANT Command**

- **Purpose:** Used to **give specific privileges (permissions)** to users on database objects (tables, views, etc.).
- **Example:**
- GRANT SELECT, INSERT ON Students TO user1;

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

---

2. **REVOKE Command**

- **Purpose:** Used to **remove previously granted privileges** from users.
- **Example:**
- REVOKE INSERT ON Students FROM user1;

Removes the **INSERT** privilege from user1 on the Students table.

2. How do you manage privileges using these commands?

Managing Privileges with GRANT and REVOKE

1. **Granting Privileges (GRANT)**

- Used to **assign permissions** to a user or role.
- Syntax:
- GRANT privilege_list
- ON object_name
- TO user_name;
- Example:
- GRANT SELECT, INSERT
- ON Students
- TO user1;

user1 can **view** and **add** records in Students.

---

2. **Revoking Privileges (REVOKE)**

- Used to **remove permissions** from a user or role.
- Syntax:
- REVOKE privilege_list
- ON object_name
- FROM user_name;
- Example:
- REVOKE INSERT
- ON Students
- FROM user1;

user1 can no longer **insert** records but still has SELECT access.

10. Transaction Control Language (TCL)

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

COMMIT and ROLLBACK in SQL

1. **COMMIT Command**

- **Purpose:** Saves all the changes made by DML statements (INSERT, UPDATE, DELETE) permanently in the database.
- Once committed, the changes **cannot be undone**.
- **Example:**
- UPDATE Employees SET salary = salary + 1000 WHERE emp_id = 101;
- COMMIT;

Salary update is permanently saved.

---

2. **ROLLBACK Command**

- **Purpose:** Undoes (cancels) all changes made by DML statements since the last COMMIT or SAVEPOINT.
- Restores the database to its previous state.
- **Example:**
- DELETE FROM Employees WHERE department = 'HR';
- ROLLBACK;

The delete action is undone, and HR employees remain in the table.

2. Explain how transactions are managed in SQL databases

Definition:
A **transaction** is a sequence of one or more SQL operations (like INSERT, UPDATE, DELETE) that are executed as a **single logical unit of work**. Transactions ensure that the database remains **consistent and reliable**.

How Transactions Are Managed

1. **BEGIN / START TRANSACTION**
   - Marks the **start** of a transaction.
   - Example:
   - START TRANSACTION;
2. **Execute SQL Statements**
   - Perform operations like INSERT, UPDATE, or DELETE.
   - These changes are **temporary** until committed.
3. **COMMIT**
   - Saves all changes **permanently**.
   - Example:
   - COMMIT;
4. **ROLLBACK**

    - Cancels all changes made since the last COMMIT (or SAVEPOINT).
    - Example:
    - ROLLBACK;

5. **SAVEPOINT (Optional)**
    - Creates a checkpoint inside a transaction.
    - You can roll back only to that point instead of the entire transaction.
    - Example:
    - SAVEPOINT sp1;
    - ROLLBACK TO sp1;

11. 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?

JOIN in SQL

Concept:
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).

👉 Joins allow you to query data that is spread across multiple tables.

Types of Joins

1. INNER JOIN

- Returns only the **matching rows** from both tables.
- Rows without matches are excluded.
- **Example:**

- SELECT Students.name, Courses.course_name
- FROM Students
- INNER JOIN Courses
- ON Students.course_id = Courses.course_id;

---

2. LEFT JOIN (LEFT OUTER JOIN)

- Returns **all rows from the left table**, and the **matching rows from the right table**.
- If no match exists, NULLs are returned for right table columns.
- **Example:**
- SELECT Students.name, Courses.course_name
- FROM Students
- LEFT JOIN Courses
- ON Students.course_id = Courses.course_id;

---

3. RIGHT JOIN (RIGHT OUTER JOIN)

- Returns **all rows from the right table**, and the **matching rows from the left table**.
- If no match exists, NULLs are returned for left table columns.
- **Example:**
- SELECT Students.name, Courses.course_name
- FROM Students
- RIGHT JOIN Courses
- ON Students.course_id = Courses.course_id;

---

4. FULL OUTER JOIN

- Returns **all rows from both tables**, with NULLs where no match exists.
- **Example:**
- SELECT Students.name, Courses.course_name
- FROM Students
- FULL OUTER JOIN Courses

ON Students.course_id = Courses.course_id;

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

How Joins Combine Data from Multiple Tables

- **Joins** are used in SQL to **retrieve data spread across multiple tables** by linking them through a **common column** (usually a primary key in one table and a foreign key in another).
- This allows you to treat data from multiple tables as if it were in a single table.

General Syntax

SELECT columns

FROM table1

JOIN table2

ON table1.common_column = table2.common_column;

Example

Suppose we have two tables:

Students Table

student_id	name	course_id
1	Rahul	101

student_id	name	course_id
2	Meena	102

Courses Table

course_id	course_name
101	Mathematics
102	Computer Sci

Query with JOIN:

```
SELECT Students.name, Courses.course_name
FROM Students
INNER JOIN Courses
ON Students.course_id = Courses.course_id;
```

Result:

name	course_name
Rahul	Mathematics
Meena	Computer Sci

12. SQL Group By

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

Definition

- The GROUP BY clause in SQL is used to **arrange rows into groups** based on the values of one or more columns.
- It is **always used with aggregate functions** (like COUNT(), SUM(), AVG(), MAX(), MIN()) to perform calculations **for each group** of data.

Syntax

```
SELECT column_name, AGGREGATE_FUNCTION(column_name)

FROM table_name

GROUP BY column_name;
```

Example

Suppose we have a **Sales** table:

product_id	category	amount
1	Electronics	5000
2	Clothing	2000
3	Electronics	7000
4	Clothing	3000

Query:

```
SELECT category, SUM(amount) AS total_sales

FROM Sales

GROUP BY category;
```

Result:

category	total_sales
Electronics	12000
Clothing	5000

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

Feature	GROUP BY	ORDER BY
Purpose	Groups rows based on column values	Sorts rows in ascending (ASC) or descending (DESC) order

Feature	GROUP BY	ORDER BY
Works With	Often used with **aggregate functions** (SUM(), AVG(), COUNT(), etc.)	Used for **sorting** query results (no aggregation required)
Output	Returns **one row per group**	Returns **all rows**, but in sorted order
Clause Position	Comes **before ORDER BY** in SQL	Comes **after GROUP BY** in SQL
Example	SELECT dept, AVG(salary) FROM Employees GROUP BY dept; → Gives average salary per department	SELECT * FROM Employees ORDER BY salary DESC; → Lists employees sorted by salary

13. SQL Stored Procedure

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

Definition

- A **stored procedure** is a **precompiled collection of SQL statements** (like SELECT, INSERT, UPDATE, DELETE, etc.) stored in the database.
- It can accept **parameters**, execute complex logic, and return results.
- Instead of writing the same SQL repeatedly, you just **call the procedure**.

Syntax Example

```
CREATE PROCEDURE GetEmployeeDetails
   @DeptID INT
AS
BEGIN
   SELECT name, salary
```

```
    FROM Employees
    WHERE department_id = @DeptID;
END;
```

To run it:

```
EXEC GetEmployeeDetails @DeptID = 2;
```

Difference: Stored Procedure vs Standard SQL Query

Feature	Stored Procedure	Standard SQL Query
Definition	Precompiled set of SQL statements stored in DB	A single SQL statement executed directly
Reusability	Reusable (can be called multiple times)	Must be rewritten/run each time
Performance	Faster (precompiled and cached)	Parsed and executed each time
Parameters	Can accept inputs/outputs	Usually no parameters
Complexity	Can contain logic (loops, conditions, multiple queries)	Generally a single operation
Security	Can restrict direct table access (execute only the procedure)	Users need direct access to tables

2. Explain the advantages of using stored procedures.

Advantages of Using Stored Procedures in SQL

1. **Reusability**
    - Once created, a stored procedure can be **called multiple times** by different programs or users.
    - No need to rewrite the same SQL logic repeatedly.
2. **Improved Performance**

   - Stored procedures are **precompiled** and stored in the database, so they execute faster than standard SQL queries.

3. **Reduced Network Traffic**
   - Multiple SQL statements can be executed in a **single procedure call**, reducing the number of requests sent between client and server.
4. **Enhanced Security**
   - Users can be granted permission to **execute the procedure** without having direct access to the underlying tables.
5. **Maintainability and Modularity**
   - Complex logic can be **organized in one place**, making it easier to update or maintain.
6. **Supports Parameters**
   - Can accept **input and output parameters**, allowing dynamic execution.
7. **Consistency**
   - Ensures **consistent execution** of business rules across applications.

14. SQL View

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

Definition

- A **view** is a **virtual table** that is based on the result of a **SELECT query**.
- It **does not store data physically**; it just displays data from one or more tables.
- You can use views to **simplify complex queries** or **restrict access** to specific columns/rows.

Syntax Example

```
CREATE VIEW EmployeeView AS
SELECT name, salary
FROM Employees
WHERE department_id = 2;
```

To query the view:

```
SELECT * FROM EmployeeView;
```

Difference Between View and Table

Feature	View	Table
Data Storage	Virtual, no physical storage	Stores data physically
Definition	Defined by a SELECT query	Defined by CREATE TABLE
Update	Sometimes updatable (with restrictions)	Always updatable
Purpose	Simplifies queries, restricts access	Stores actual data
Columns/Rows	Can show subset or join of tables	Contains all defined columns and rows

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

1. **Simplifies Complex Queries**
    - Encapsulates complex SELECT statements into a single object.
    - Users can query the view without rewriting complex joins or calculations.
2. **Data Security / Access Control**
    - Restricts access to specific columns or rows.
    - Users can see only the data exposed by the view, not the entire table.
3. **Consistent Data Presentation**

- Ensures **uniform output format** for multiple users or applications.

4. **Reusability**

   - Once created, a view can be reused in multiple queries or reports.

5. **Logical Data Independence**

   - Underlying table structure can change without affecting queries using the view (as long as columns in the view remain).

6. **Aggregation and Summarization**

   - Views can **pre-calculate summaries** or aggregates (e.g., totals, averages) for reporting.

15. SQL Triggers

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

Definition

- A **trigger** is a **special type of stored procedure** that is automatically executed (**fired**) by the database when a **specific event** occurs on a table or view.
- Events can be INSERT, UPDATE, or DELETE.

Purpose of Triggers

- Enforce **business rules** automatically.
- Maintain **audit trails** (track changes).
- Ensure **data integrity**.

Types of Triggers

Type	When it is Fired / Use Case
BEFORE Trigger	Executes **before** an INSERT, UPDATE, or DELETE operation. Used for **validation or modifying values** before changes are saved.
AFTER Trigger	Executes **after** an INSERT, UPDATE, or DELETE. Used for **logging, auditing, or cascading changes**.
INSTEAD OF Trigger	Executes **instead of** an INSERT, UPDATE, or DELETE. Often used on **views** to allow updates through a view.

Example (AFTER INSERT Trigger)

```
CREATE TRIGGER trg_AfterInsertEmployee

AFTER INSERT ON Employees

FOR EACH ROW

BEGIN

  INSERT INTO AuditLog(employee_id, action, action_date)

  VALUES (NEW.emp_id, 'INSERT', NOW());

END;
```

- This trigger logs every new employee added into an **AuditLog** table automatically.

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

Trigger Type	When it Fires	Purpose / Use Case	Example
INSERT Trigger	Fires **when a new row is inserted** into a table	Automatically validate or log new records	Log every new employee added into an audit table

Trigger Type	When it Fires	Purpose / Use Case	Example
UPDATE Trigger	Fires **when an existing row is updated**	Track changes, enforce business rules, maintain history	Record salary changes of employees in a history table
DELETE Trigger	Fires **when a row is deleted**	Prevent accidental deletion or log deletions	Store deleted customer data in a backup table before deletion

In short:

- **INSERT Trigger** → Acts on new rows being added.
- **UPDATE Trigger** → Acts on rows being modified.
- **DELETE Trigger** → Acts on rows being removed.

16. Introduction to PL/SQL

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

PL/SQL

Definition

- **PL/SQL** (Procedural Language/SQL) is **Oracle's procedural extension of SQL**.
- It allows combining **SQL statements with procedural constructs** like variables, loops, conditions, and exceptions.
- Unlike standard SQL, PL/SQL can **process multiple rows, perform calculations, and control program flow**.

How PL/SQL Extends SQL

1. **Procedural Logic**
   - Supports **IF-ELSE**, loops (FOR, WHILE), and case statements.
2. **Variables and Constants**

   - Can declare and use variables to store and manipulate data.

3. **Error Handling**
   - Supports **exception handling** for runtime errors.
4. **Modularity**
   - Allows creation of **procedures, functions, packages, and triggers**.
5. **Performance**
   - Executes multiple SQL statements in a **single block**, reducing network traffic.

Example of a PL/SQL Block

```
DECLARE
  v_salary NUMBER;
BEGIN
  SELECT salary INTO v_salary
  FROM Employees
  WHERE emp_id = 101;

  IF v_salary < 50000 THEN
    UPDATE Employees
    SET salary = salary + 5000
    WHERE emp_id = 101;
  END IF;

  COMMIT;
END;
```

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

Benefits of Using PL/SQL

1. **Combines SQL and Procedural Logic**
   - Allows using **loops, conditions, and variables** along with SQL queries.
   - Makes it possible to write complex programs that SQL alone cannot handle.
2. **Improved Performance**
   - Executes **multiple SQL statements in a single block**, reducing network traffic and improving efficiency.
3. **Error Handling (Exception Management)**
   - Provides **robust error handling** using EXCEPTION blocks to manage runtime errors.
4. **Modularity and Reusability**
   - Supports **procedures, functions, packages, and triggers**.
   - Code can be reused across applications or modules.
5. **Security**
   - Can **restrict direct access to tables** by granting execution rights on procedures or functions instead.
6. **Maintainability**
   - Structured blocks and modular programming make code **easier to read, maintain, and debug**.
7. **Portability**
   - PL/SQL blocks can run on any **Oracle database** without modification.

17. PL/SQL Control Structures

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

Definition

- **Control structures** are used in PL/SQL to **control the flow of execution** in a program.
- They allow the program to make **decisions** or **repeat actions** based on conditions.

---

1. IF-THEN Statement

- Used to **execute a block of code only if a condition is true**.
- **Syntax:**

```
IF condition THEN
  -- statements to execute if condition is true
END IF;
```

- **Example:**

```
DECLARE
  v_salary NUMBER := 40000;
BEGIN
  IF v_salary < 50000 THEN
    DBMS_OUTPUT.PUT_LINE('Salary is below 50,000');
  END IF;
END;
```

- Can also have IF-THEN-ELSE or ELSIF for multiple conditions.

---

2. LOOP Statement

- Used to **execute a block of code repeatedly** until a condition is met.

- **Types of Loops:**
    1. **Basic LOOP** – executes indefinitely until EXIT is used.
    2. **WHILE LOOP** – executes while a condition is true.
    3. **FOR LOOP** – executes a fixed number of times.
- **Example (Basic LOOP):**

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

- **Example (FOR LOOP):**

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

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

How Control Structures Help in PL/SQL

Control structures in PL/SQL allow you to **add logic, decision-making, and repetition** to SQL operations, making queries more **flexible and powerful**.

---

1. Decision Making (IF-THEN / IF-ELSE)

- Lets the program **execute different SQL statements based on conditions**.
- Example: Apply a bonus only if salary < 50,000:

```
IF v_salary < 50000 THEN
   UPDATE Employees SET salary = salary + 5000
   WHERE emp_id = 101;
END IF;
```

---

2. Loops (LOOP, FOR, WHILE)

- Allows repeating **SQL operations multiple times** without rewriting queries.
- Example: Give all employees in a department a salary increment:

```
FOR i IN 1..10 LOOP
   UPDATE Employees SET salary = salary + 1000
   WHERE emp_id = i;
END LOOP;
```

---

3. Combining SQL with Procedural Logic

- Control structures let you **combine multiple queries, conditions, and calculations** in a single PL/SQL block.
- This reduces **redundancy**, improves **maintainability**, and ensures **business rules are consistently applied**.

18. SQL Cursors Theory Questions:

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

Definition

- A **cursor** is a **pointer that allows you to fetch and manipulate rows returned by a query one at a time**.
- Cursors are used when a query returns **multiple rows**, and you want to process them **sequentially** in PL/SQL.

Types of Cursors

1. Implicit Cursor

- Automatically created by PL/SQL when a **single-row DML query** (INSERT, UPDATE, DELETE, SELECT INTO) is executed.
- **No need to declare**; PL/SQL manages it internally.
- **Example:**

```
DECLARE
   v_salary Employees.salary%TYPE;
BEGIN
   SELECT salary INTO v_salary
   FROM Employees
   WHERE emp_id = 101;
   DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary);
END;
```

- Here, PL/SQL automatically creates an implicit cursor for the SELECT INTO statement.

---

2. Explicit Cursor

- Must be **declared, opened, fetched, and closed** manually by the programmer.
- Used for **queries returning multiple rows**.
- **Steps for Explicit Cursor:**
    1. **Declare** the cursor
    2. **Open** the cursor
    3. **Fetch** rows from the cursor
    4. **Close** the cursor
- **Example:**

```
DECLARE
  CURSOR emp_cursor IS
    SELECT name, salary FROM Employees;
  v_name Employees.name%TYPE;
  v_salary Employees.salary%TYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO v_name, v_salary;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_name || ': ' || v_salary);
  END LOOP;
  CLOSE emp_cursor;
END;
```

Key Differences

Feature	Implicit Cursor	Explicit Cursor
Declaration	Automatically created	Must be explicitly declared
Use Case	Single-row queries	Multiple-row queries
Control	Managed by PL/SQL	Controlled by the programmer
Operations	No manual open/fetch/close	Requires open, fetch, close

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

1. **Query Returns Multiple Rows**
   - Implicit cursors handle only **single-row queries**.
   - Example: Fetch and process all employees' salaries one by one.
2. **Need to Fetch Rows Sequentially**
   - When you want to **process each row individually** using a loop, explicit cursors are ideal.
3. **Better Control over Cursor Operations**
   - You can **OPEN, FETCH, and CLOSE** the cursor at your convenience.
   - Allows **conditional processing** of rows.
4. **Use Cursor Attributes**
   - Can utilize %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN to manage logic.

Example Use Case

```
DECLARE
  CURSOR emp_cursor IS
    SELECT name, salary FROM Employees;
  v_name Employees.name%TYPE;
```

```
   v_salary Employees.salary%TYPE;
BEGIN
   OPEN emp_cursor;
   LOOP
      FETCH emp_cursor INTO v_name, v_salary;
      EXIT WHEN emp_cursor%NOTFOUND;
      IF v_salary < 50000 THEN
         DBMS_OUTPUT.PUT_LINE(v_name || ' needs a raise.');
      END IF;
   END LOOP;
   CLOSE emp_cursor;
END;
```

- Here, **each row is checked individually**, which cannot be done with an implicit cursor.

19. Rollback and Commit Savepoint Theory Questions:

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

Definition

- A **SAVEPOINT** is a **marker set within a transaction** that allows you to **partially roll back** the transaction to a specific point without undoing the entire transaction.

- Useful for managing **large transactions** where only part of the changes need to be undone.

Syntax

```
SAVEPOINT savepoint_name;
```

Interaction with ROLLBACK and COMMIT

1. **ROLLBACK TO SAVEPOINT**
   - Undoes all changes **made after the savepoint** but keeps the changes made **before it**.
   - Example:
2. BEGIN;
3. INSERT INTO Employees VALUES (101, 'Rahul', 50000);
4. SAVEPOINT sp1;
5. INSERT INTO Employees VALUES (102, 'Meena', 60000);
6. ROLLBACK TO sp1; -- Only the second insert is undone
7. COMMIT;   -- Changes before sp1 are saved
8. **COMMIT**
   - Saves **all changes in the transaction permanently**, including those **before and after savepoints**.
   - Once committed, the transaction and savepoints are removed.

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

1. **Large Transactions**
   - When a transaction involves **multiple steps or operations**, savepoints allow rolling back only the **problematic part** instead of the whole transaction.
2. **Error Handling**
   - If an error occurs in the middle of a transaction, you can **rollback to a savepoint** instead of discarding all previous successful operations.
3. **Complex Business Logic**

   - Useful when executing **conditional operations** where only some actions need to be undone based on conditions.

4. **Partial Commit Preparation**
   - Helps in **staging changes** before final commit, ensuring that only **valid operations are saved permanently**.

Example

```
BEGIN;

INSERT INTO Orders VALUES (101, 'Laptop', 2);

SAVEPOINT sp1;

INSERT INTO Orders VALUES (102, 'Phone', -5); -- Invalid quantity

ROLLBACK TO sp1;  -- Undo only the invalid insert

COMMIT;          -- Save valid insert permanently
```

- Only the first valid insert is committed; the invalid insert is discarded.