Module 4 – Introduction to DBMS

Introduction to SQL

Theory Questions:

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

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

**SQL (Structured Query Language)** is a **standard programming language** specifically designed for **managing and manipulating relational databases**. It allows users to **create, read, update, and delete** (CRUD) data in a structured and efficient manner.

**🔹 Why SQL is Essential in Database Management:**

**1. Data Querying**

* SQL is used to retrieve data from databases using the SELECT statement.
* Example:
* SELECT name, age FROM students WHERE grade = 'A';

**2. Data Manipulation**

* SQL enables **inserting, updating, and deleting** data.
* Examples:
* INSERT INTO students (name, age) VALUES ('Amit', 18);
* UPDATE students SET age = 19 WHERE name = 'Amit';
* DELETE FROM students WHERE age < 16;

**3. Data Definition**

* SQL is used to **create, alter, and delete** database structures (tables, views, indexes).
* Example:
* CREATE TABLE students (
* id INT PRIMARY KEY,
* name VARCHAR(50),
* age INT,
* grade CHAR(1)
* );

**4. Data Control**

* SQL supports **user access control** and **permissions** with commands like GRANT and REVOKE.

**5. Standardized Language**

* SQL is used by most relational database systems (like MySQL, PostgreSQL, SQL Server, Oracle), making it a universal skill for database professionals.

**6. Data Integrity and Relationships**

* SQL supports **constraints**, **primary/foreign keys**, and **relationships**, ensuring data remains accurate and meaningful.

**✅ Summary:**

**SQL is essential** because it serves as the bridge between users and relational databases, enabling the efficient management, organization, and retrieval of structured data. It’s the foundation of data-driven applications and business intelligence systems.

1. Explain the difference between DBMS and RDBMS.

### Ans. 2. Difference Between DBMS and RDBMS

| **Feature** | **DBMS (Database Management System)** | **RDBMS (Relational Database Management System)** |
| --- | --- | --- |
| **Definition** | Software used to store and manage data in a structured format | A type of DBMS that stores data in the form of **related tables** |
| **Data Structure** | Stores data as **files** or **collections of records** | Stores data in **tables** (rows and columns) |
| **Data Relationship** | No direct support for relationships between data | Supports relationships using **primary and foreign keys** |
| **Normalization** | Not enforced | Uses **normalization** to reduce data redundancy |
| **Data Integrity** | Limited support for constraints and data integrity | Strong support with **constraints** like PRIMARY KEY, FOREIGN KEY, UNIQUE, etc. |
| **Examples** | File System, XML DB, dBase | MySQL, PostgreSQL, Oracle, SQL Server |
| **Concurrency & Transactions** | Basic or no support | Full support for **ACID properties** (Atomicity, Consistency, Isolation, Durability) |
| **Security** | Less secure, basic access control | High level of security with **user roles and permissions** |
| **Scalability and Performance** | Suitable for small data sets | Designed for large-scale, complex data management |

**✅ Summary:**

* **DBMS** is a general system for managing databases, which can be file-based.
* **RDBMS** is a more advanced, structured version of DBMS that uses **tables and relationships**, making it ideal for **modern, relational data**.

**All RDBMS are DBMS, but not all DBMS are RDBMS.**

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

### Ans. ✅ Role of SQL in Managing Relational Databases

**SQL (Structured Query Language)** plays a **central role** in managing **relational databases** by providing a standardized way to **interact with, manipulate, and control** data stored in **tables**. Here's how SQL helps manage relational databases effectively:

**🔹 1. Data Querying**

SQL allows users to **retrieve specific data** from one or more related tables.

* Example:
* SELECT name, salary FROM employees WHERE department = 'HR';

**🔹 2. Data Manipulation**

SQL lets users **add, update, or delete** records in tables using **DML (Data Manipulation Language)** commands.

* Examples:
* INSERT INTO students (name, age) VALUES ('Riya', 19);
* UPDATE students SET age = 20 WHERE name = 'Riya';
* DELETE FROM students WHERE age < 18;

**🔹 3. Data Definition**

SQL is used to **create and modify** database structures (tables, columns, constraints) through **DDL (Data Definition Language)**.

* Example:
* CREATE TABLE courses (
* course\_id INT PRIMARY KEY,
* course\_name VARCHAR(100)
* );

**🔹 4. Data Control**

SQL helps manage **user access and security** through **DCL (Data Control Language)**.

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

**🔹 5. Data Integrity**

SQL supports **constraints** to ensure **valid and consistent data**.

* Examples:
  + PRIMARY KEY: Unique identifier
  + FOREIGN KEY: Maintains relationships
  + NOT NULL, UNIQUE, CHECK

**🔹 6. Transaction Management**

Using **TCL (Transaction Control Language)**, SQL handles multiple operations as a single **atomic transaction**, ensuring data reliability.

* Example:
* BEGIN;
* UPDATE accounts SET balance = balance - 500 WHERE id = 1;
* UPDATE accounts SET balance = balance + 500 WHERE id = 2;
* COMMIT;

**✅ Summary:**

SQL is the **backbone of relational database management**. It enables:

* Efficient data storage and retrieval
* Accurate and secure manipulation
* Structural and user control
* Relationship maintenance between tables

Without SQL, managing relational databases would be extremely complex and inefficient.

1. What are the key features of SQL?

### Ans. ✅ Key Features of SQL (Structured Query Language):

SQL is a powerful and standardized language used for managing relational databases. Its features make it essential for storing, retrieving, and manipulating data.

**🔹 1. Data Querying**

* SQL allows users to **fetch data** from one or more tables using SELECT.
* Example:
* SELECT name, age FROM students WHERE grade = 'A';

**🔹 2. Data Manipulation**

* SQL provides commands to **insert, update, and delete** data.
* Commands: INSERT, UPDATE, DELETE
* Example:
* INSERT INTO employees (name, salary) VALUES ('Amit', 50000);

**🔹 3. Data Definition**

* SQL defines and modifies database structures like tables and schemas.
* Commands: CREATE, ALTER, DROP
* Example:
* CREATE TABLE products (
* id INT PRIMARY KEY,
* name VARCHAR(100),
* price DECIMAL(10, 2)
* );

**🔹 4. Data Control**

* Controls **access and permissions** on data.
* Commands: GRANT, REVOKE
* Example:
* GRANT SELECT ON students TO user1;

**🔹 5. Transaction Control**

* Ensures **data integrity** through atomic transactions.
* Commands: BEGIN, COMMIT, ROLLBACK
* Example:
* BEGIN;
* UPDATE accounts SET balance = balance - 100 WHERE id = 1;
* COMMIT;

**🔹 6. Data Integrity and Constraints**

* SQL supports constraints like:
  + PRIMARY KEY
  + FOREIGN KEY
  + UNIQUE
  + NOT NULL
  + CHECK
* These help maintain **accuracy and consistency** of data.

**🔹 7. Support for Relational Databases**

* SQL is designed to manage **tables with relationships**, using keys and joins.

**🔹 8. Standardized Language**

* SQL is standardized by **ANSI and ISO**, and supported by most RDBMSs like:
  + MySQL, PostgreSQL, Oracle, SQL Server

**🔹 9. High-Level Language**

* SQL is **declarative**: you specify **what** data you want, not **how** to get it.

**✅ Summary Table:**

| **Feature** | **Description** |
| --- | --- |
| Data Querying | Retrieve data using SELECT |
| Data Manipulation | Modify data using INSERT, UPDATE, DELETE |
| Data Definition | Define tables and schemas with CREATE, ALTER |
| Access Control | Manage permissions with GRANT, REVOKE |
| Transaction Management | Handle transactions using COMMIT, ROLLBACK |
| Integrity Constraints | Enforce rules using keys and constraints |
| Relational Support | Manage related data through keys and joins |
| Standardization | Follows ANSI/ISO standards |

**✅ Conclusion:**

SQL's powerful, standardized, and easy-to-use features make it the core language for managing relational databases efficiently.

2. SQL Syntax

Theory Questions:

* 1. What are the basic components of SQL syntax?

### Ans. ✅ 1. Basic Components of SQL Syntax

SQL (Structured Query Language) uses a clear and structured syntax for working with relational databases. Here are the **basic components** that make up SQL syntax:

**🔹 1. SQL Statements / Commands**

SQL is composed of various types of statements grouped into categories:

| **Category** | **Common Commands** | **Purpose** |
| --- | --- | --- |
| **DQL** (Data Query Language) | SELECT | Retrieve data from tables |
| **DML** (Data Manipulation Language) | INSERT, UPDATE, DELETE | Modify data |
| **DDL** (Data Definition Language) | CREATE, ALTER, DROP | Define or modify table structure |
| **DCL** (Data Control Language) | GRANT, REVOKE | Manage user permissions |
| **TCL** (Transaction Control Language) | COMMIT, ROLLBACK, SAVEPOINT | Handle transactions |

**🔹 2. Clauses**

Clauses add conditions or structure to SQL statements.

* Examples:
  + WHERE: Filters records
  + ORDER BY: Sorts result set
  + GROUP BY: Groups rows with the same values
  + HAVING: Filters groups (used with GROUP BY)

SELECT name, age FROM students WHERE grade = 'A' ORDER BY age;

**🔹 3. Expressions**

Expressions calculate values or perform operations.

* Example:
* SELECT salary \* 1.10 AS increased\_salary FROM employees;

**🔹 4. Predicates**

Predicates evaluate to true/false and are used in conditions.

* Examples: =, >, <, IN, BETWEEN, LIKE
* SELECT \* FROM orders WHERE amount BETWEEN 100 AND 500;

**🔹 5. Identifiers**

Names of database objects:

* **Tables**, **columns**, **databases**, **indexes**, etc.
* Example:
* SELECT student\_name FROM student\_details;

**🔹 6. Keywords**

Reserved words that have special meaning in SQL.

* Examples: SELECT, FROM, WHERE, INSERT, JOIN

**🔹 7. Operators**

Used for comparisons, calculations, or logic.

* **Arithmetic**: +, -, \*, /
* **Comparison**: =, !=, <, >, <=, >=
* **Logical**: AND, OR, NOT

**🔹 8. Literals**

Constants or fixed values used in queries.

* Example: 'John', 100, TRUE
* SELECT \* FROM users WHERE name = 'John';

**✅ Example SQL Query Breakdown:**

SELECT name, age -- Columns to display (Identifiers)

FROM students -- Table name (Identifier)

WHERE age > 18 -- Condition (Predicate + Operator)

ORDER BY name ASC; -- Sorting (Clause + Keyword)

**✅ Summary Table:**

| **Component** | **Role** |
| --- | --- |
| **Statements** | Perform actions (e.g., SELECT, INSERT) |
| **Clauses** | Add detail to statements (WHERE, ORDER BY) |
| **Expressions** | Perform calculations |
| **Predicates** | Define conditions |
| **Identifiers** | Name of tables, columns |
| **Keywords** | SQL reserved words |
| **Operators** | Compare or combine values |
| **Literals** | Fixed data values |

**✅ Conclusion:**

Understanding the **basic components of SQL syntax** helps in writing accurate, readable, and effective SQL queries for database management and data analysis.

* 1. Write the general structure of an SQL SELECT statement.

### Ans. ✅ 2. General Structure of an SQL SELECT Statement

The SELECT statement is used to **retrieve data** from one or more tables in a relational database.

**🔹 Basic Syntax:**

SELECT column1, column2, ...

FROM table\_name

[WHERE condition]

[GROUP BY column]

[HAVING condition]

[ORDER BY column [ASC|DESC]]

[LIMIT number];

**🔹 Components Explained:**

| **Clause** | **Description** |
| --- | --- |
| SELECT | Specifies the columns to retrieve. Use \* to select all columns. |
| FROM | Specifies the table(s) from which to retrieve data. |
| WHERE *(Optional)* | Filters rows based on a condition. |
| GROUP BY *(Optional)* | Groups rows that have the same values into summary rows. |
| HAVING *(Optional)* | Filters groups created by GROUP BY. |
| ORDER BY *(Optional)* | Sorts the result set by one or more columns. |
| LIMIT *(Optional, used in MySQL, PostgreSQL, etc.)* | Limits the number of rows returned. |

**🔹 Example:**

SELECT name, department, AVG(salary) AS avg\_salary

FROM employees

WHERE salary > 30000

GROUP BY department

HAVING AVG(salary) > 40000

ORDER BY avg\_salary DESC

LIMIT 5;

**✅ What This Does:**

* Selects names and departments from employees.
* Filters only those with salary > 30,000.
* Groups them by department and calculates average salary.
* Only includes groups with average salary > 40,000.
* Sorts results in descending order of average salary.
* Returns only the top 5 results.

**✅ Summary:**

The SELECT statement is the **most commonly used** SQL command, and its general structure supports **flexible querying**, **filtering**, **grouping**, and **sorting** of data.

* 1. Explain the role of clauses in SQL statements.

### Ans. ✅ 3. Role of Clauses in SQL Statements

**Clauses** in SQL are **building blocks** of SQL statements. They define **what data to act on**, **how to filter it**, **how to group or sort it**, and more. Each clause performs a **specific role** in shaping the behavior and outcome of an SQL query.

**🔹 Key SQL Clauses and Their Roles**

| **Clause** | **Purpose / Role** | **Example Usage** |
| --- | --- | --- |
| SELECT | Specifies the **columns** to be retrieved | SELECT name, age |
| FROM | Specifies the **table(s)** to retrieve data from | FROM students |
| WHERE | Filters rows based on **conditions** | WHERE age > 18 |
| GROUP BY | Groups rows sharing a value in a column | GROUP BY department |
| HAVING | Filters **groups** (used with GROUP BY) | HAVING COUNT(\*) > 5 |
| ORDER BY | Sorts the result set by one or more columns | ORDER BY salary DESC |
| LIMIT / TOP | Limits the number of rows returned (DB-specific) | LIMIT 10 (MySQL) or TOP 5 (SQL Server) |
| JOIN | Combines rows from two or more tables based on related columns | INNER JOIN orders ON customers.id = orders.customer\_id |

**🔹 Example SQL Query with Clauses:**

SELECT department, AVG(salary) AS avg\_salary

FROM employees

WHERE salary > 30000

GROUP BY department

HAVING AVG(salary) > 50000

ORDER BY avg\_salary DESC

LIMIT 3;

**🔍 Explanation:**

* **SELECT**: Chooses department and average salary.
* **FROM**: Specifies the employees table.
* **WHERE**: Filters out employees with salaries ≤ 30,000.
* **GROUP BY**: Groups data by department.
* **HAVING**: Further filters groups with average salary > 50,000.
* **ORDER BY**: Sorts the result by avg\_salary in descending order.
* **LIMIT**: Returns only the top 3 records.

**✅ Summary:**

Clauses define **what data is selected**, **from where**, **under what conditions**, **how it is grouped or sorted**, and **how many results are returned**. They make SQL a **flexible and powerful tool** for querying relational databases.

* 1. SQL Constraints

Theory Questions:

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

### Ans. ✅ 1. What Are Constraints in SQL?

**Constraints** in SQL are **rules applied to columns** in a table to **enforce data integrity, accuracy, and consistency**. They help ensure that the data stored in the database follows certain rules and conditions.

**✅ Common Types of SQL Constraints:**

| **Constraint** | **Description** | **Example** |
| --- | --- | --- |
| **PRIMARY KEY** | Uniquely identifies each row in a table. Cannot be NULL. | id INT PRIMARY KEY |
| **FOREIGN KEY** | Ensures referential integrity by linking one table’s column to another table’s primary key. | FOREIGN KEY (dept\_id) REFERENCES departments(id) |
| **UNIQUE** | Ensures that all values in a column are unique. Allows NULL. | email VARCHAR(100) UNIQUE |
| **NOT NULL** | Ensures that a column **cannot have NULL values**. | name VARCHAR(50) NOT NULL |
| **CHECK** | Ensures that values meet a specific condition. | salary INT CHECK (salary > 0) |
| **DEFAULT** | Sets a **default value** for a column if none is provided. | status VARCHAR(20) DEFAULT 'Active' |

**🔹 1. PRIMARY KEY**

* Combines NOT NULL and UNIQUE.
* Each table should have **only one** primary key (can be a combination of columns).

CREATE TABLE students (

id INT PRIMARY KEY,

name VARCHAR(50)

);

**🔹 2. FOREIGN KEY**

* Links rows between **two tables**.
* Enforces **referential integrity**.

CREATE TABLE orders (

order\_id INT PRIMARY KEY,

customer\_id INT,

FOREIGN KEY (customer\_id) REFERENCES customers(id)

);

**🔹 3. UNIQUE**

* Ensures all values in a column are different.

CREATE TABLE employees (

emp\_id INT,

email VARCHAR(100) UNIQUE

);

**🔹 4. NOT NULL**

* Prevents inserting NULL into a column.

CREATE TABLE products (

product\_id INT,

name VARCHAR(100) NOT NULL

);

**🔹 5. CHECK**

* Validates values based on an expression.

CREATE TABLE accounts (

account\_id INT,

balance DECIMAL(10, 2) CHECK (balance >= 0)

);

**🔹 6. DEFAULT**

* Assigns a default value if none is provided during insertion.

CREATE TABLE users (

user\_id INT,

status VARCHAR(20) DEFAULT 'Active'

);

**✅ Summary:**

| **Constraint** | **Enforces** |
| --- | --- |
| PRIMARY KEY | Row uniqueness and identity |
| FOREIGN KEY | Valid cross-table references |
| UNIQUE | Column-level uniqueness |
| NOT NULL | Mandatory data entry |
| CHECK | Logical conditions on values |
| DEFAULT | Automatic values when none given |

Constraints help maintain **data quality** and **reliability** in SQL databases.

* + 1. How do PRIMARY KEY and FOREIGN KEY constraints differ? 3. What is the role of NOT NULL and UNIQUE constraints?

### Ans. ✅ Differences Between PRIMARY KEY and FOREIGN KEY Constraints:

| **Aspect** | **PRIMARY KEY** | **FOREIGN KEY** |
| --- | --- | --- |
| **Purpose** | Uniquely identifies each record in a table. | Enforces relationships between two tables. |
| **Uniqueness** | Must be **unique** for every row. | May contain **duplicate** values. |
| **NULL Values** | **Cannot** be NULL. | **Can** have NULL values (optional relationship). |
| **Location** | Defined **within the table** it identifies. | Refers to a **primary key** in another (or same) table. |
| **Usage** | Ensures **row-level uniqueness**. | Ensures **referential integrity** between tables. |

**🔹 Example:**

-- Table with PRIMARY KEY

CREATE TABLE departments (

dept\_id INT PRIMARY KEY,

dept\_name VARCHAR(50)

);

-- Table with FOREIGN KEY referring to the above

CREATE TABLE employees (

emp\_id INT PRIMARY KEY,

emp\_name VARCHAR(100),

dept\_id INT,

FOREIGN KEY (dept\_id) REFERENCES departments(dept\_id)

);

**✅ Roles of NOT NULL and UNIQUE Constraints:**

**🔹 NOT NULL Constraint**

* Ensures that a column **must have a value**.
* Prevents insertion of NULL into the specified column.
* Helps enforce **mandatory data entry**.

CREATE TABLE users (

user\_id INT,

username VARCHAR(50) NOT NULL

);

**🔹 UNIQUE Constraint**

* Ensures that **no two rows** in a table have the same value in a particular column (or combination of columns).
* Allows **only one NULL** value (depending on DBMS).

CREATE TABLE users (

email VARCHAR(100) UNIQUE

);

**✅ Summary:**

| **Constraint** | **Role** |
| --- | --- |
| **PRIMARY KEY** | Ensures each row is unique and identifiable. |
| **FOREIGN KEY** | Maintains referential link between tables. |
| **NOT NULL** | Ensures a column cannot be left empty. |
| **UNIQUE** | Ensures all values in a column are distinct. |

These constraints **preserve data integrity** and ensure **logical accuracy** of the database.

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

Theory Questions:

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

### Ans. ✅ 1. SQL Data Definition Language (DDL)

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

**🔹 Key Purposes of DDL:**

* Create and design database schema.
* Modify structure (not data).
* Remove or delete database objects.

**🔹 Common DDL Commands:**

| **Command** | **Description** | **Example** |
| --- | --- | --- |
| CREATE | Creates a new database object (table, view, etc.) | CREATE TABLE students (id INT, name VARCHAR(50)); |
| ALTER | Modifies an existing object (e.g., add/drop a column) | ALTER TABLE students ADD email VARCHAR(100); |
| DROP | Deletes a database object permanently | DROP TABLE students; |
| TRUNCATE | Removes all data from a table (structure remains) | TRUNCATE TABLE students; |
| RENAME | Changes the name of a database object | RENAME TABLE students TO learners; |

**✅ Example DDL Usage:**

-- Creating a table

CREATE TABLE employees (

emp\_id INT PRIMARY KEY,

emp\_name VARCHAR(100),

salary DECIMAL(10,2)

);

-- Altering the table to add a new column

ALTER TABLE employees ADD hire\_date DATE;

-- Removing the table

DROP TABLE employees;

**✅ Summary:**

| **Feature** | **Details** |
| --- | --- |
| **Type of SQL** | Data Definition Language (DDL) |
| **Main Function** | Defines and manages **structure** of database objects |
| **Does it modify data?** | ❌ No (It modifies structure, not the data itself) |

**DDL is essential** for setting up and maintaining the **blueprint of a relational database**.

* + 1. Explain the CREATE command and its syntax.

### Ans. ✅ 2. Explain the CREATE Command and Its Syntax

The **CREATE** command in SQL is part of the **Data Definition Language (DDL)** and is used to **create new database objects**, such as:

* Tables
* Databases
* Views
* Indexes
* Schemas
* Stored procedures (in some DBMSs)

**🔹 Most Common Usage: Creating a Table**

**✅ Syntax:**

CREATE TABLE table\_name (

column1 datatype [constraint],

column2 datatype [constraint],

...

);

* table\_name: Name of the table you want to create.
* column: The name of a column.
* datatype: Data type of the column (e.g., INT, VARCHAR, DATE).
* constraint: (Optional) Constraints like PRIMARY KEY, NOT NULL, UNIQUE, etc.

**🔹 Example:**

CREATE TABLE students (

student\_id INT PRIMARY KEY,

name VARCHAR(100) NOT NULL,

email VARCHAR(100) UNIQUE,

age INT,

enrollment\_date DATE DEFAULT CURRENT\_DATE

);

**Explanation:**

* student\_id is the primary key and cannot be null or duplicated.
* name must be provided (NOT NULL).
* email must be unique.
* enrollment\_date gets the current date by default.

**🔹 Other CREATE Usages:**

**✅ Create a Database:**

CREATE DATABASE school;

**✅ Create a View:**

CREATE VIEW student\_names AS

SELECT student\_id, name FROM students;

**✅ Create an Index:**

CREATE INDEX idx\_name ON students(name);

**✅ Summary:**

| **Command** | **Purpose** |
| --- | --- |
| CREATE TABLE | Creates a new table |
| CREATE DATABASE | Creates a new database |
| CREATE VIEW | Creates a virtual table based on a query |
| CREATE INDEX | Improves the performance of SELECT queries |

The CREATE command is fundamental to **setting up the structure** of a relational database.

* + 1. What is the purpose of specifying data types and constraints during table creation?

### Ans. ✅ 3. **Purpose of Specifying Data Types and Constraints During Table Creation**

When creating a table in SQL, **data types** and **constraints** play a critical role in defining the **structure**, **validity**, and **integrity** of the data stored.

### 🔹 **1. Purpose of Data Types**

Data types define the **kind of data** that each column can hold.

#### ✅ Why Use Data Types:

| **Purpose** | **Explanation** |
| --- | --- |
| 🔹 **Storage Efficiency** | Helps the database engine allocate the correct amount of memory. |
| 🔹 **Data Integrity** | Ensures only valid data is inserted (e.g., numbers into numeric fields). |
| 🔹 **Validation** | Prevents accidental data entry errors (e.g., text in a numeric field). |
| 🔹 **Enables Operations** | Allows proper use of operators (e.g., comparisons, sorting, arithmetic). |

#### ✅ Example:

age INT, -- Only integers allowed

name VARCHAR(100), -- String with max length 100

price DECIMAL(10,2) -- Numeric with 2 decimal places

### 🔹 **2. Purpose of Constraints**

Constraints enforce **rules** on the data in columns to maintain **accuracy**, **uniqueness**, and **relationships** between tables.

#### ✅ Common Constraints and Their Purpose:

| **Constraint** | **Purpose** |
| --- | --- |
| PRIMARY KEY | Uniquely identifies each row; no NULLs. |
| FOREIGN KEY | Maintains relationships between tables. |
| NOT NULL | Ensures a column must have a value. |
| UNIQUE | Prevents duplicate values in a column. |
| CHECK | Ensures values meet a condition. |
| DEFAULT | Assigns a default value if none is provided. |

#### ✅ Example:

CREATE TABLE products (

product\_id INT PRIMARY KEY,

name VARCHAR(50) NOT NULL,

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

stock INT DEFAULT 0

);

### ✅ Summary:

| **Aspect** | **Why It's Important** |
| --- | --- |
| **Data Types** | Define **what kind of data** can be stored in a column. |
| **Constraints** | Define **rules and relationships** to enforce data validity and consistency. |

By using **appropriate data types and constraints**, you ensure your database is **well-structured, reliable, and error-resistant** from the start.

* 1. ALTER Command

Theory Questions:

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

### Ans. ✅ 1. **What is the Use of the** ALTER **Command in SQL?**

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

### 🔹 **Main Uses of** ALTER**:**

| **Use Case** | **Example** |
| --- | --- |
| Add a new column | ADD COLUMN |
| Modify an existing column | MODIFY or ALTER COLUMN |
| Delete a column | DROP COLUMN |
| Rename a column or table | RENAME |
| Add or remove constraints | ADD CONSTRAINT, DROP CONSTRAINT |

### ✅ **Syntax and Examples**

#### 🔹 Add a New Column:

ALTER TABLE students ADD email VARCHAR(100);

#### 🔹 Modify Data Type or Size of a Column:

ALTER TABLE students MODIFY name VARCHAR(150); -- MySQL

-- OR

ALTER TABLE students ALTER COLUMN name TYPE VARCHAR(150); -- PostgreSQL

#### 🔹 Drop a Column:

ALTER TABLE students DROP COLUMN email;

#### 🔹 Rename a Table:

ALTER TABLE students RENAME TO learners;

#### 🔹 Add a Constraint:

ALTER TABLE students ADD CONSTRAINT pk\_student\_id PRIMARY KEY (student\_id);

#### 🔹 Drop a Constraint:

ALTER TABLE students DROP CONSTRAINT pk\_student\_id;

⚠️ Note: Syntax may vary slightly between SQL databases (MySQL, PostgreSQL, SQL Server, Oracle).

### ✅ Summary:

| **Feature** | **Description** |
| --- | --- |
| Command Type | DDL (Data Definition Language) |
| Primary Function | Modify the structure of an existing table |
| Common Actions | Add/modify/drop columns or constraints, rename |

The ALTER command is **essential for maintaining and evolving database schemas** without losing existing data.

* + 1. How can you add, modify, and drop columns from a table using ALTER?

### Ans. ✅ 2. Using ALTER to Add, Modify, and Drop Columns in SQL

The ALTER TABLE command is used to **change the structure** of an existing table by **adding**, **modifying**, or **dropping** columns.

**🔹 A. Add a Column**

**✅ Syntax:**

ALTER TABLE table\_name

ADD column\_name datatype [constraint];

**✅ Example:**

ALTER TABLE employees

ADD email VARCHAR(100);

🔹 Adds a new column email to the employees table.

**🔹 B. Modify an Existing Column**

**✅ Syntax (varies slightly by DBMS):**

* **MySQL:**

ALTER TABLE table\_name

MODIFY column\_name new\_datatype;

* **PostgreSQL / SQL Server:**

ALTER TABLE table\_name

ALTER COLUMN column\_name TYPE new\_datatype;

**✅ Example (MySQL):**

ALTER TABLE employees

MODIFY email VARCHAR(150);

🔹 Changes the email column size from 100 to 150 characters.

**🔹 C. Drop a Column**

**✅ Syntax:**

ALTER TABLE table\_name

DROP COLUMN column\_name;

**✅ Example:**

ALTER TABLE employees

DROP COLUMN email;

🔹 Deletes the email column from the employees table.

**✅ Summary Table:**

| **Action** | **SQL Command** |
| --- | --- |
| Add Column | ALTER TABLE table\_name ADD column\_name datatype; |
| Modify Column | ALTER TABLE table\_name MODIFY/ALTER COLUMN...; |
| Drop Column | ALTER TABLE table\_name DROP COLUMN column\_name; |

These commands allow **flexible updates** to your table structure **without deleting or recreating** the entire table.

DROP Command

Theory Questions:

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

### Ans. ✅ 1. What Is the Function of the DROP Command in SQL?

The **DROP** command in SQL is a **Data Definition Language (DDL)** command used to **permanently delete database objects**, such as:

* Tables
* Databases
* Views
* Indexes
* Stored Procedures (in some DBMSs)

**🔹 Key Functions of DROP:**

| **Function** | **Effect** |
| --- | --- |
| **Deletes database objects** | Removes the object completely from the database. |
| **Deletes structure and data** | Erases both the **schema** and the **data** stored within the object. |
| **Irreversible** | Once executed, the operation **cannot be undone** (no rollback). |

**✅ Common Syntax and Examples:**

**🔹 Drop a Table:**

DROP TABLE students;

🔸 Deletes the students table and all its data permanently.

**🔹 Drop a Database:**

DROP DATABASE school;

🔸 Deletes the entire school database and all tables inside it.

**🔹 Drop a View:**

DROP VIEW student\_view;

**🔹 Drop an Index (MySQL):**

DROP INDEX index\_name ON table\_name;

**⚠️ Important Notes:**

* Use DROP **with caution** — it **cannot be undone**.
* Ensure that the object exists before using DROP to avoid errors.
* Use IF EXISTS (supported in many DBMSs) for safer execution:

DROP TABLE IF EXISTS students;

**✅ Summary:**

| **Command** | **Purpose** |
| --- | --- |
| DROP TABLE | Deletes an entire table and its data |
| DROP DATABASE | Removes a database and all its contents |
| DROP VIEW | Removes a stored SQL query (view) |
| DROP INDEX | Removes an index used for faster searches |

The DROP command is a **powerful tool** for managing database objects but should be used **carefully** due to its permanent nature.

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

### Ans. ✅ 2. Implications of Dropping a Table from a Database

The DROP TABLE command in SQL **permanently removes a table**, including its **structure**, **data**, and associated **constraints or indexes**.

**🔹 Key Implications:**

| **Implication** | **Description** |
| --- | --- |
| 🔴 **Data Loss** | All records (rows) in the table are permanently deleted. |
| 🔴 **Structure Loss** | The table's definition (columns, data types, constraints) is erased. |
| 🔴 **Constraint Removal** | All constraints like PRIMARY KEY, FOREIGN KEY, NOT NULL, etc., are dropped. |
| 🔴 **Index Removal** | Indexes associated with the table are also deleted. |
| 🔴 **Dependency Errors** | If the table is referenced by a FOREIGN KEY in another table, dropping it may cause errors unless constraints are removed first. |
| 🔴 **Cannot Be Rolled Back (in most cases)** | Once dropped, the action is **permanent** unless inside a transaction and supported by the DBMS. |

**✅ Example:**

DROP TABLE students;

🔸 This will permanently remove the students table and all of its data.

**⚠️ Best Practices:**

* ✅ Always **back up data** before dropping critical tables.
* ✅ Use DROP TABLE IF EXISTS table\_name; to avoid runtime errors.
* ✅ Ensure **referential integrity** by handling dependencies (e.g., dropping foreign keys first).

**✅ Summary:**

| **Action** | **Impact** |
| --- | --- |
| Drop Table | Deletes structure, data, constraints, indexes |
| Permanent? | Yes — usually **cannot be undone** |
| Affects other tables? | Yes, if foreign key dependencies exist |

**Dropping a table** is a powerful but destructive operation. It should be done **only when the table is no longer needed** and **after verifying dependencies and backups**.

Data Manipulation Language (DML)

Theory Questions:

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

### Ans. ✅ 1. **Definitions of INSERT, UPDATE, and DELETE Commands in SQL**

The INSERT, UPDATE, and DELETE commands are part of **Data Manipulation Language (DML)** in SQL. These commands are used to **manage data within existing tables**.

### 🔹 **1.** INSERT **Command**

Used to **add new records** (rows) into a table.

#### ✅ Syntax:

INSERT INTO table\_name (column1, column2, ...)

VALUES (value1, value2, ...);

#### ✅ Example:

INSERT INTO students (student\_id, name, age)

VALUES (1, 'Alice', 20);

### 🔹 **2.** UPDATE **Command**

Used to **modify existing records** in a table.

#### ✅ Syntax:

UPDATE table\_name

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

WHERE condition;

#### ✅ Example:

UPDATE students

SET age = 21

WHERE student\_id = 1;

⚠️ Always use the WHERE clause to avoid updating all records unintentionally.

### 🔹 **3.** DELETE **Command**

Used to **remove existing records** from a table.

#### ✅ Syntax:

DELETE FROM table\_name

WHERE condition;

#### ✅ Example:

DELETE FROM students

WHERE student\_id = 1;

⚠️ Without WHERE, all records in the table will be deleted.

### ✅ Summary Table:

| **Command** | **Purpose** | **Caution** |
| --- | --- | --- |
| INSERT | Adds new data to a table | Ensure data types match |
| UPDATE | Changes data in existing rows | Use WHERE to avoid mass updates |
| DELETE | Removes data from a table | Use WHERE to avoid full deletion |

These commands are essential for **managing the data lifecycle** in a relational database.

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

### Ans. ✅ 2. **Importance of the** WHERE **Clause in** UPDATE **and** DELETE **Operations**

The **WHERE clause** in SQL is **critical** when using UPDATE and DELETE commands, as it **specifies which rows** should be affected.

### 🔹 **Why is the** WHERE **Clause Important?**

| **Reason** | **Explanation** |
| --- | --- |
| ✅ **Targets Specific Rows** | It ensures only the intended records are updated or deleted. |
| ✅ **Prevents Unintended Changes** | Without WHERE, **all rows** in the table will be affected. |
| ✅ **Protects Data Integrity** | Helps avoid data loss or incorrect updates. |
| ✅ **Improves Accuracy** | Focuses the operation based on conditions (e.g., id = 5). |

### 🔸 **Examples:**

#### ✅ **With** WHERE **– Safe Operation**

UPDATE employees

SET salary = 60000

WHERE emp\_id = 101;

🔹 Only the employee with emp\_id = 101 will have their salary updated.

DELETE FROM employees

WHERE emp\_id = 101;

🔹 Only the row with emp\_id = 101 will be deleted.

#### ❌ **Without** WHERE **– Risky Operation**

UPDATE employees

SET salary = 60000;

⚠️ **All employees** will now have a salary of 60000.

DELETE FROM employees;

⚠️ **All employee records** will be permanently deleted.

### ✅ Summary:

| **Command** | **With WHERE** | **Without WHERE** |
| --- | --- | --- |
| UPDATE | Updates specific rows | Updates **all** rows |
| DELETE | Deletes specific rows | Deletes **all** rows |

🔔 **Conclusion:**  
Always use the WHERE clause in UPDATE and DELETE to **protect your data and avoid accidental full-table modifications**.

Data Query Language (DQL)

Theory Questions:

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

### Ans. ✅ 1. What is the SELECT Statement and How Is It Used to Query Data?

The **SELECT statement** is the most commonly used command in SQL. It is part of the **Data Query Language (DQL)** and is used to **retrieve data** from one or more tables in a database.

**🔹 Purpose of SELECT:**

To **fetch specific columns, rows, or calculations** from a table based on certain conditions.

**✅ Basic Syntax:**

SELECT column1, column2, ...

FROM table\_name

[WHERE condition]

[ORDER BY column]

[GROUP BY column]

[LIMIT number];

* SELECT: Specifies which columns to retrieve.
* FROM: Specifies the table to query.
* WHERE: Filters rows based on conditions (optional).
* ORDER BY: Sorts the result (optional).
* GROUP BY: Groups rows with the same values (optional).
* LIMIT: Restricts the number of results (optional).

**🔸 Example 1: Fetch All Columns**

SELECT \* FROM students;

Returns **all columns and all rows** from the students table.

**🔸 Example 2: Fetch Specific Columns**

SELECT name, age FROM students;

Returns only the name and age columns.

**🔸 Example 3: Filtered Results Using WHERE**

SELECT name, age FROM students

WHERE age > 18;

Returns names and ages of students who are older than 18.

**🔸 Example 4: Ordered Results**

SELECT name, age FROM students

ORDER BY age DESC;

Returns names and ages of students, sorted by age in descending order.

**✅ Summary:**

| **Keyword** | **Function** |
| --- | --- |
| SELECT | Chooses columns to display |
| FROM | Indicates the source table |
| WHERE | Filters rows based on conditions |
| ORDER BY | Sorts the result set |
| GROUP BY | Groups results (usually with aggregation) |
| LIMIT | Restricts number of rows returned |

🔔 The SELECT statement is essential for **viewing, analyzing, and reporting** data from a database.

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

### Ans. ✅ 2. **Use of** ORDER BY **and** WHERE **Clauses in SQL Queries**

The ORDER BY and WHERE clauses are commonly used in SQL SELECT statements to **filter** and **sort** the result set.

### 🔹 **1. WHERE Clause – Filtering Rows**

The WHERE clause is used to **specify a condition** that must be met for rows to be included in the result.

#### ✅ Syntax:

SELECT column1, column2

FROM table\_name

WHERE condition;

#### ✅ Example:

SELECT name, age

FROM students

WHERE age > 18;

🔸 Returns only those students whose age is greater than 18.

### 🔹 **2. ORDER BY Clause – Sorting Rows**

The ORDER BY clause is used to **sort the result set** based on one or more columns, in **ascending (ASC)** or **descending (DESC)** order.

#### ✅ Syntax:

SELECT column1, column2

FROM table\_name

ORDER BY column1 [ASC|DESC];

#### ✅ Example:

SELECT name, age

FROM students

ORDER BY age DESC;

🔸 Returns all students sorted by age in descending order.

### 🔹 **Using WHERE and ORDER BY Together**

#### ✅ Example:

SELECT name, age

FROM students

WHERE age > 18

ORDER BY name ASC;

🔸 Filters students over 18 and sorts them alphabetically by name.

### ✅ Summary Table:

| **Clause** | **Purpose** | **Common Use** |
| --- | --- | --- |
| WHERE | Filters records based on a condition | WHERE age > 18 |
| ORDER BY | Sorts the filtered result | ORDER BY name ASC |

Data Control Language (DCL)

Theory Questions:

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

### Ans. ✅ 1. **Purpose of** GRANT **and** REVOKE **in SQL**

The **GRANT** and **REVOKE** statements are part of **Data Control Language (DCL)** in SQL. They are used to **control access permissions** on database objects such as tables, views, procedures, and more.

### 🔹 **1. GRANT – Give Permissions**

The GRANT command is used to **assign specific privileges** to one or more users.

#### ✅ Syntax:

GRANT privilege [, privilege...]

ON object\_name

TO user\_name;

#### ✅ Example:

GRANT SELECT, INSERT

ON students

TO user1;

🔸 This allows user1 to **view (SELECT)** and **add data (INSERT)** into the students table.

### 🔹 **2. REVOKE – Remove Permissions**

The REVOKE command is used to **take back** previously granted privileges from users.

#### ✅ Syntax:

REVOKE privilege [, privilege...]

ON object\_name

FROM user\_name;

#### ✅ Example:

REVOKE INSERT

ON students

FROM user1;

🔸 This removes user1's ability to insert data into the students table.

### ✅ Common Privileges That Can Be Granted or Revoked:

| **Privilege** | **Description** |
| --- | --- |
| SELECT | Read data from a table/view |
| INSERT | Add new rows into a table |
| UPDATE | Modify existing data |
| DELETE | Remove data from a table |
| ALL | Grants all available privileges |

### ✅ Summary:

| **Command** | **Function** |
| --- | --- |
| GRANT | Assigns specific access rights to a user |
| REVOKE | Removes access rights from a user |

🔐 These commands are essential for **database security**, ensuring that users have access **only to the data and operations they need**.

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

### Ans. ✅ 2. Managing Privileges Using GRANT and REVOKE Commands in SQL

Privileges control **what actions users can perform** on database objects like tables, views, or procedures. You can manage these privileges using the **GRANT** and **REVOKE** commands.

**🔹 A. Granting Privileges (GRANT)**

You use GRANT to **assign specific access rights** to users or roles.

**✅ Syntax:**

GRANT privilege [, privilege...]

ON object\_name

TO user\_name [WITH GRANT OPTION];

* privilege: The permission (e.g., SELECT, INSERT, UPDATE).
* object\_name: The table or view the privilege applies to.
* WITH GRANT OPTION: Allows the user to grant the same privileges to others.

**✅ Example:**

GRANT SELECT, INSERT ON employees TO user1;

🔸 user1 can now read and insert data into the employees table.

**🔹 B. Revoking Privileges (REVOKE)**

You use REVOKE to **remove specific privileges** from users or roles.

**✅ Syntax:**

REVOKE privilege [, privilege...]

ON object\_name

FROM user\_name;

**✅ Example:**

REVOKE INSERT ON employees FROM user1;

🔸 user1 can no longer insert data into the employees table.

**🔹 C. Viewing Current Privileges**

Some systems allow you to query system tables or use commands like:

SHOW GRANTS FOR user1;

Or:

SELECT \* FROM information\_schema.role\_table\_grants

WHERE grantee = 'user1';

**🔐 Best Practices for Managing Privileges:**

| **Practice** | **Why It Matters** |
| --- | --- |
| ✅ Grant only needed permissions | Limits accidental or malicious actions |
| ✅ Revoke unused privileges | Reduces security risks |
| ✅ Use roles for group access | Easier to manage access for multiple users |
| ✅ Avoid using GRANT ALL carelessly | It may give more access than required |

**✅ Summary:**

| **Action** | **Command Used** | **Purpose** |
| --- | --- | --- |
| Give access | GRANT | Assigns permissions to users |
| Take back access | REVOKE | Removes permissions from users |
| Control spread | WITH GRANT OPTION | Lets users pass privileges to others |

By properly managing privileges using GRANT and REVOKE, you can ensure **data security, controlled access, and efficient user management** in your SQL database.

Transaction Control Language (TCL)

Theory Questions:

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

### Ans. ✅ 1. Purpose of COMMIT and ROLLBACK Commands in SQL

The COMMIT and ROLLBACK commands are part of **Transaction Control Language (TCL)** in SQL. They are used to manage **transactions**, ensuring **data integrity and consistency**.

**🔹 What is a Transaction?**

A **transaction** is a sequence of one or more SQL operations treated as a **single unit of work**. It must be either **fully completed** or **fully undone**.

**🔹 1. COMMIT – Save the Changes**

* The COMMIT command is used to **permanently save all changes** made by the current transaction to the database.
* Once committed, changes **cannot be undone**.

**✅ Syntax:**

COMMIT;

**✅ Example:**

UPDATE accounts

SET balance = balance - 1000

WHERE acc\_id = 101;

COMMIT;

🔸 This confirms and saves the deduction in balance.

**🔹 2. ROLLBACK – Undo the Changes**

* The ROLLBACK command is used to **undo all changes** made in the current transaction.
* It reverts the database to the **last committed state**.

**✅ Syntax:**

ROLLBACK;

**✅ Example:**

UPDATE accounts

SET balance = balance - 1000

WHERE acc\_id = 101;

ROLLBACK;

🔸 This cancels the balance deduction and restores the previous value.

**🔸 Use Case Scenario:**

BEGIN TRANSACTION;

UPDATE products SET stock = stock - 10 WHERE product\_id = 200;

UPDATE orders SET status = 'Placed' WHERE order\_id = 501;

-- If all operations succeed

COMMIT;

-- If any error occurs

ROLLBACK;

**✅ Summary:**

| **Command** | **Purpose** | **Effect** |
| --- | --- | --- |
| COMMIT | Saves changes made by the transaction | Changes are **permanent** |
| ROLLBACK | Cancels changes made by the transaction | Changes are **undone** |

🔐 These commands are vital for maintaining **data accuracy**, especially in operations involving **multiple interdependent queries**.

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

### Ans. ✅ 2. How Transactions Are Managed in SQL Databases

A **transaction** in SQL is a logical unit of work that contains one or more SQL statements. Transactions ensure that database operations are **executed completely or not at all**, maintaining **data integrity** even in the event of errors or failures.

**🔹 Key Properties of Transactions (ACID):**

| **Property** | **Description** |
| --- | --- |
| **A – Atomicity** | Ensures that all operations in a transaction **either complete fully or not at all**. |
| **C – Consistency** | Guarantees that a transaction brings the database from one valid state to another. |
| **I – Isolation** | Ensures that **concurrent transactions** do not interfere with each other. |
| **D – Durability** | Once committed, changes are **permanently saved**, even if the system crashes. |

**🔹 Basic Transaction Management Commands:**

| **Command** | **Purpose** |
| --- | --- |
| BEGIN or START TRANSACTION | Marks the start of a transaction |
| COMMIT | Saves all changes made in the transaction permanently |
| ROLLBACK | Reverts all changes made in the transaction to the last commit |
| SAVEPOINT | Sets a checkpoint within a transaction |
| ROLLBACK TO SAVEPOINT | Undoes part of a transaction back to a specific point |

**🔹 Transaction Lifecycle:**

1. **Begin a Transaction**
2. START TRANSACTION; -- or BEGIN;
3. **Execute SQL Statements**
4. UPDATE accounts SET balance = balance - 1000 WHERE acc\_id = 1;
5. UPDATE accounts SET balance = balance + 1000 WHERE acc\_id = 2;
6. **Commit or Rollback**
   * If all is good:
   * COMMIT;
   * If there's an error:
   * ROLLBACK;

**🔸 Using SAVEPOINT (Advanced Example):**

START TRANSACTION;

UPDATE accounts SET balance = balance - 1000 WHERE acc\_id = 1;

SAVEPOINT deduct;

UPDATE accounts SET balance = balance + 1000 WHERE acc\_id = 2;

-- Something goes wrong here

ROLLBACK TO deduct;

COMMIT;

🔹 Only the second update is undone; the first remains intact until the final commit.

**✅ Summary:**

| **Feature** | **What it Does** |
| --- | --- |
| BEGIN | Starts a transaction |
| COMMIT | Saves all changes made in the transaction permanently |
| ROLLBACK | Cancels all changes since the last commit |
| SAVEPOINT | Creates a restore point within a transaction |
| ROLLBACK TO SAVEPOINT | Undoes part of a transaction |

🔐 **Transaction management** is essential for ensuring **data accuracy, consistency, and reliability** in multi-step or concurrent operations.

11. SQL Joins

Theory Questions:

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

### Ans. ✅ 1. **Concept of JOIN in SQL**

The **JOIN** operation in SQL is used to **combine rows** from two or more tables **based on a related column** between them, typically using **primary and foreign keys**.

### 🔹 **Why Use JOIN?**

To fetch **related data** that is **stored across multiple tables**, instead of duplicating data in a single table (which violates normalization rules).

### ✅ **Types of JOINs and Their Differences:**

| **Type** | **Description** |
| --- | --- |
| **INNER JOIN** | Returns **only matching rows** from both tables. |
| **LEFT JOIN** | Returns **all rows from the left table** and matching rows from the right. |
| **RIGHT JOIN** | Returns **all rows from the right table** and matching rows from the left. |
| **FULL OUTER JOIN** | Returns **all rows** from both tables, with NULLs where there is no match. |

### 🔸 1. **INNER JOIN**

Returns records with **matching values in both tables**.

SELECT A.name, B.department

FROM Employees A

INNER JOIN Departments B

ON A.dept\_id = B.id;

📌 **Result:** Only employees who belong to a department that exists in Departments.

### 🔸 2. **LEFT JOIN (LEFT OUTER JOIN)**

Returns **all records from the left table**, and matching ones from the right. If no match, NULL.

SELECT A.name, B.department

FROM Employees A

LEFT JOIN Departments B

ON A.dept\_id = B.id;

📌 **Result:** All employees, even those not assigned to any department (NULL for department).

### 🔸 3. **RIGHT JOIN (RIGHT OUTER JOIN)**

Returns **all records from the right table**, and matching ones from the left. If no match, NULL.

SELECT A.name, B.department

FROM Employees A

RIGHT JOIN Departments B

ON A.dept\_id = B.id;

📌 **Result:** All departments, even those with no employees assigned.

### 🔸 4. **FULL OUTER JOIN**

Returns **all records when there is a match in either** left or right table. NULL for missing matches.

SELECT A.name, B.department

FROM Employees A

FULL OUTER JOIN Departments B

ON A.dept\_id = B.id;

📌 **Result:** All employees and all departments. If no match, NULL appears on the side that doesn’t have matching data.

### ✅ Visual Summary:

| **JOIN Type** | **Left Table** | **Right Table** | **Matched?** | **Unmatched Left** | **Unmatched Right** |
| --- | --- | --- | --- | --- | --- |
| INNER JOIN | ✅ | ✅ | ✅ | ❌ | ❌ |
| LEFT JOIN | ✅ | ✅/NULL | ✅ | ✅ | ❌ |
| RIGHT JOIN | ✅/NULL | ✅ | ✅ | ❌ | ✅ |
| FULL OUTER JOIN | ✅/NULL | ✅/NULL | ✅ | ✅ | ✅ |

### ✅ Conclusion:

SQL JOINs are powerful for **querying related data across multiple tables**, allowing flexible and complex data analysis. Understanding the difference between the JOIN types is crucial for getting **accurate results** from your queries.

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

### Ans. ✅ 1. **Concept of JOIN in SQL**

The **JOIN** operation in SQL is used to **combine rows** from two or more tables **based on a related column** between them, typically using **primary and foreign keys**.

### 🔹 **Why Use JOIN?**

To fetch **related data** that is **stored across multiple tables**, instead of duplicating data in a single table (which violates normalization rules).

### ✅ **Types of JOINs and Their Differences:**

| **Type** | **Description** |
| --- | --- |
| **INNER JOIN** | Returns **only matching rows** from both tables. |
| **LEFT JOIN** | Returns **all rows from the left table** and matching rows from the right. |
| **RIGHT JOIN** | Returns **all rows from the right table** and matching rows from the left. |
| **FULL OUTER JOIN** | Returns **all rows** from both tables, with NULLs where there is no match. |

### 🔸 1. **INNER JOIN**

Returns records with **matching values in both tables**.

SELECT A.name, B.department

FROM Employees A

INNER JOIN Departments B

ON A.dept\_id = B.id;

📌 **Result:** Only employees who belong to a department that exists in Departments.

### 🔸 2. **LEFT JOIN (LEFT OUTER JOIN)**

Returns **all records from the left table**, and matching ones from the right. If no match, NULL.

SELECT A.name, B.department

FROM Employees A

LEFT JOIN Departments B

ON A.dept\_id = B.id;

📌 **Result:** All employees, even those not assigned to any department (NULL for department).

### 🔸 3. **RIGHT JOIN (RIGHT OUTER JOIN)**

Returns **all records from the right table**, and matching ones from the left. If no match, NULL.

SELECT A.name, B.department

FROM Employees A

RIGHT JOIN Departments B

ON A.dept\_id = B.id;

📌 **Result:** All departments, even those with no employees assigned.

### 🔸 4. **FULL OUTER JOIN**

Returns **all records when there is a match in either** left or right table. NULL for missing matches.

SELECT A.name, B.department

FROM Employees A

FULL OUTER JOIN Departments B

ON A.dept\_id = B.id;

📌 **Result:** All employees and all departments. If no match, NULL appears on the side that doesn’t have matching data.

### ✅ Visual Summary:

| **JOIN Type** | **Left Table** | **Right Table** | **Matched?** | **Unmatched Left** | **Unmatched Right** |
| --- | --- | --- | --- | --- | --- |
| INNER JOIN | ✅ | ✅ | ✅ | ❌ | ❌ |
| LEFT JOIN | ✅ | ✅/NULL | ✅ | ✅ | ❌ |
| RIGHT JOIN | ✅/NULL | ✅ | ✅ | ❌ | ✅ |
| FULL OUTER JOIN | ✅/NULL | ✅/NULL | ✅ | ✅ | ✅ |

### ✅ Conclusion:

SQL JOINs are powerful for **querying related data across multiple tables**, allowing flexible and complex data analysis. Understanding the difference between the JOIN types is crucial for getting **accurate results** from your queries.

* 1. SQL Group By

Theory Questions:

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

### Ans. ✅ 1. What is the GROUP BY Clause in SQL?

The GROUP BY clause in SQL is used to **group rows that have the same values** in one or more columns. It is typically used **with aggregate functions** (like COUNT(), SUM(), AVG(), MAX(), MIN()) to **perform calculations on each group** rather than on the entire table.

**🔹 Purpose of GROUP BY:**

To summarize data by **grouping similar records** together and applying **aggregate functions** on each group.

**✅ Syntax:**

SELECT column1, AGGREGATE\_FUNCTION(column2)

FROM table\_name

GROUP BY column1;

**✅ Example 1: Using GROUP BY with COUNT()**

SELECT department, COUNT(\*) AS num\_employees

FROM employees

GROUP BY department;

🔸 Groups all employees by department and counts how many employees are in each department.

**✅ Example 2: Using GROUP BY with SUM()**

SELECT customer\_id, SUM(amount) AS total\_spent

FROM orders

GROUP BY customer

Explain the difference between GROUP BY and ORDER BY.

Ans. In SQL, GROUP BY and ORDER BY are both **clauses** used in SELECT statements, but they serve **different purposes**:

**🔹 GROUP BY – Used for Grouping Data**

* **Purpose:** Organizes rows that have the same values into **groups**.
* **Typically used with aggregate functions** like SUM(), COUNT(), AVG(), etc.
* **Does not sort** the output unless combined with ORDER BY.

**✅ Example:**

SELECT department, COUNT(\*) AS num\_employees

FROM employees

GROUP BY department;

🔸 This groups employees by their department and counts how many are in each group.

**🔹 ORDER BY – Used for Sorting Data**

* **Purpose:** Sorts the result set by one or more columns.
* Can be used with or without GROUP BY.
* Default sort is **ascending (ASC)**; use DESC for descending.

**✅ Example:**

SELECT name, salary

FROM employees

ORDER BY salary DESC;

🔸 This sorts employees by salary in descending order.

**🔄 Using Both Together**

You can use both in a query, like this:

SELECT department, COUNT(\*) AS num\_employees

FROM employees

GROUP BY department

ORDER BY num\_employees DESC;

🔸 This:

1. Groups employees by department,
2. Counts how many employees are in each group,
3. Sorts the result by that count in descending order.

**🔍 Summary Table**

| **Clause** | **Function** | **Commonly Used With** | **Sorts Output?** |
| --- | --- | --- | --- |
| GROUP BY | Groups rows into sets | Aggregate functions | ❌ No |
| ORDER BY | Sorts result rows | Any query | ✅ Yes |

Let me know if you'd like to see a visual example using sample data.

13. SQL Stored Procedure

Theory Questions:

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

### Ans. 🔹 What is a Stored Procedure in SQL?

A **stored procedure** is a **precompiled set of SQL statements** stored in the database that can be executed repeatedly. It can include **logic (IF, loops, variables)** and is used to **perform a task** like inserting data, updating tables, or processing business logic.

**✅ Syntax Example (MySQL):**

DELIMITER //

CREATE PROCEDURE GetEmployeeCount()

BEGIN

SELECT COUNT(\*) FROM employees;

END //

DELIMITER ;

To run it:

CALL GetEmployeeCount();

**🔄 Difference Between Stored Procedure and Standard SQL Query**

| **Feature** | **Stored Procedure** | **Standard SQL Query** |
| --- | --- | --- |
| **Definition** | A named set of SQL statements stored in DB | A one-time SQL statement or query |
| **Reusability** | Can be reused by calling the procedure | Executed once; must be retyped to reuse |
| **Complex Logic** | Supports control flow (IF, loops, variables) | Limited to basic SQL operations |
| **Performance** | Precompiled; may execute faster | Compiled and executed each time |
| **Security** | Can restrict access via procedure calls | Access requires direct SQL privileges |
| **Maintainability** | Easy to maintain centralized logic | Hard to manage if logic is repeated |

**🧠 When to Use Stored Procedures**

* Complex business logic with multiple steps
* Repetitive tasks (like daily reports or batch updates)
* Securing business logic from end users
* Reducing traffic between application and database

Let me know if you’d like an example with parameters or real-world use cases!

1. Explain the advantages of using stored procedures.

Ans. Using **stored procedures** in SQL offers several advantages, especially in large or complex database systems. Here are the key benefits:

**✅ 1. Reusability and Modularity**

* Stored procedures can be **written once and called many times**, reducing duplication.
* They help break complex operations into modular blocks.

**✅ 2. Improved Performance**

* Stored procedures are **precompiled** and stored in the database.
* Execution plans are cached, making repeated execution **faster than sending raw SQL queries**.

**✅ 3. Enhanced Security**

* Users can be granted **permission to execute a procedure** without giving them direct access to the underlying tables.
* This protects sensitive data and enforces **controlled access**.

**✅ 4. Reduced Network Traffic**

* Instead of sending multiple SQL statements over the network, you send **a single procedure call**.
* This is especially beneficial in client-server applications.

**✅ 5. Maintainability**

* Business logic is **centralized** in the database.
* If changes are needed, you only update the procedure, not every application using it.

**✅ 6. Better Error Handling**

* Most SQL dialects allow **error trapping** using constructs like BEGIN...EXCEPTION...END.
* This ensures better control over what happens if something goes wrong.

**✅ 7. Support for Complex Logic**

* Stored procedures support **control-of-flow constructs** such as IF, WHILE, LOOP, etc.
* This makes it possible to implement **advanced logic directly in the database**.

**✅ 8. Consistency**

* Ensures **consistent execution** of tasks, since all users and applications call the same procedure.

**📌 Summary Table**

| **Advantage** | **Description** |
| --- | --- |
| Reusability | Write once, use many times |
| Performance | Precompiled, reduces processing time |
| Security | Limits access to data via controlled procedures |
| Network Efficiency | Minimizes data sent between client and server |
| Maintainability | Centralizes logic for easy updates |
| Error Handling | Offers structured error management |
| Complex Logic Support | Enables programming constructs in SQL |
| Consistency | Uniform behavior across all calls |

Would you like to see a real-world scenario where stored procedures are especially useful?

14. SQL View

Theory Questions:

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

### Ans. 🔹 What is a View in SQL?

A **view** is a **virtual table** in SQL that is based on the result of a SELECT query. It **does not store data physically**; instead, it **displays data** from one or more underlying tables.

**✅ Example:**

CREATE VIEW ActiveEmployees AS

SELECT id, name, department

FROM employees

WHERE status = 'active';

You can now query it like a table:

SELECT \* FROM ActiveEmployees;

**🔄 Difference Between a View and a Table**

| **Feature** | **View** | **Table** |
| --- | --- | --- |
| **Definition** | Virtual table created from a SELECT query | Real table that stores data physically |
| **Data Storage** | Does **not** store data itself | **Stores** data in the database |
| **Updatability** | Read-only or updatable (with conditions) | Always updatable |
| **Purpose** | Simplifies complex queries; enhances security | Stores raw data |
| **Dependency** | Depends on underlying tables | Independent |
| **Performance** | Slightly slower if complex; real-time data | Faster for direct data access |

**📌 Key Uses of Views:**

1. **Simplifying complex queries**  
   → Use a view to encapsulate a join or aggregation.
2. **Enhancing security**  
   → Expose only specific columns to users (hide sensitive info).
3. **Data abstraction**  
   → Create a user-friendly interface over raw tables.
4. **Reusable logic**  
   → Avoid rewriting the same SELECT logic.

**🚫 Example of Security Use Case:**

If you don’t want users to see salaries:

CREATE VIEW EmployeePublicInfo AS

SELECT id, name, department

FROM employees;

They can query EmployeePublicInfo, but won’t see sensitive columns like salary.

Let me know if you’d like to explore **materialized views**, which do store data!

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

### Ans. ✅ Advantages of Using Views in SQL Databases

Views provide several benefits that help with **security, simplicity, performance, and maintainability** in a database system. Here's a breakdown:

**🔹 1. Data Abstraction**

* Views simplify complex SQL queries by hiding joins, subqueries, and calculations behind a simple interface.
* Users can interact with data more easily without needing to understand the underlying schema.

**✅ Example:**

CREATE VIEW SalesSummary AS

SELECT region, SUM(sales) AS total\_sales

FROM orders

GROUP BY region;

**🔹 2. Enhanced Security**

* Views can **restrict access** to specific rows or columns of a table, allowing users to see only what they need.
* Helps implement **role-based access control**.

**✅ Example:**

CREATE VIEW PublicEmployees AS

SELECT name, department

FROM employees;

Hides sensitive data like salary or SSN.

**🔹 3. Reusability & Simplified Querying**

* You can write a complex query once as a view and reuse it multiple times.
* Reduces duplication and improves query readability.

**🔹 4. Logical Data Independence**

* You can change the structure of underlying tables (like column names or joins) without affecting users, as long as the view definition stays consistent.

**🔹 5. Easier Maintenance**

* Centralizes frequently used logic or business rules (like filtering only active records).
* If a change is needed, you update the view definition instead of every query across the application.

**🔹 6. Data Consistency**

* When multiple users or applications rely on the same view, they get **consistent and uniform** results, avoiding discrepancies in business logic.

**🔹 7. Support for Derived Columns**

* Views can contain **calculated fields** (like totals, averages, etc.), which save users from recalculating them in every query.

**🔹 8. Facilitates Reporting**

* Ideal for creating reports and dashboards by organizing data in meaningful formats tailored to reporting tools or business users.

**📌 Summary Table**

| **Advantage** | **Description** |
| --- | --- |
| Data Abstraction | Hides complex logic behind simple interface |
| Security | Restricts access to sensitive data |
| Reusability | Allows reuse of complex queries |
| Maintenance | Easy to update logic in one place |
| Logical Independence | Underlying schema changes don’t affect users |
| Consistency | Ensures uniform query results |
| Reporting | Helps in generating structured and user-friendly reports |
| Derived Data Support | Provides calculated columns without repeating logic |

Let me know if you'd like a comparison of **views vs materialized views** next!

15. SQL Triggers

Theory Questions:

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

### Ans. 🔹 What is a Trigger in SQL?

A **trigger** is a special type of stored procedure that **automatically executes** (or "fires") in response to specific **events** on a table or view, such as INSERT, UPDATE, or DELETE.

Triggers are used to **enforce business rules**, **maintain audit trails**, **validate data**, or **automate tasks** without writing manual SQL code every time.

**✅ Syntax Example (MySQL):**

CREATE TRIGGER before\_insert\_employee

BEFORE INSERT ON employees

FOR EACH ROW

BEGIN

SET NEW.created\_at = NOW();

END;

🔹 This trigger automatically sets the created\_at column before a new employee is inserted.

**🔸 Types of Triggers in SQL**

SQL triggers can be classified by **timing** and **event type**:

| **Category** | **Description** |
| --- | --- |
| **BEFORE** | Executes **before** the triggering event (e.g., before INSERT) |
| **AFTER** | Executes **after** the triggering event (e.g., after UPDATE) |
| **INSTEAD OF** | Used **on views** to define alternate actions for triggering event (mostly in SQL Server/Oracle) |

**🔹 By Event:**

1. **INSERT Trigger**  
   Fires when a new row is added.
2. **UPDATE Trigger**  
   Fires when an existing row is modified.
3. **DELETE Trigger**  
   Fires when a row is deleted.

**🧠 Usage Examples for Each Type**

| **Trigger Type** | **When to Use** | **Example Use Case** |
| --- | --- | --- |
| **BEFORE INSERT** | To validate or modify data before saving | Set default values, check constraints |
| **AFTER INSERT** | To update related tables after new data is added | Log insert actions, update counters |
| **BEFORE UPDATE** | To validate data before it's changed | Prevent changing protected fields |
| **AFTER UPDATE** | To audit changes or maintain history | Save old values in an audit table |
| **BEFORE DELETE** | To restrict deletion based on conditions | Block deletion if foreign key exists |
| **AFTER DELETE** | To perform cleanup actions | Log deleted data, adjust totals |
| **INSTEAD OF** (on views) | To allow modification of a view by redirecting the operation | Enable update on a non-updatable view |

**🔐 Why Use Triggers?**

* **Enforce business rules** automatically
* **Audit data changes** (who did what and when)
* **Cascade changes** (update totals, logs, etc.)
* **Prevent invalid operations** or ensure consistency

**⚠️ Considerations**

* Triggers can **slow down performance** if overused.
* Debugging can be tricky since they run in the background.
* Avoid complex logic inside triggers to keep them efficient and predictable.

Let me know if you'd like a full example of a trigger-based **audit logging system**!

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

### Ans. 🔄 Difference Between INSERT, UPDATE, and DELETE Triggers in SQL

Triggers in SQL are special procedures that **automatically execute** when certain changes happen to a table. The **INSERT**, **UPDATE**, and **DELETE** triggers respond to different data modification events.

**🔹 1. INSERT Trigger**

* **Fires when:** A new row is **added** to a table.
* **Purpose:** Validate or modify data before insert, or log new entries.

**✅ Example Use Case:**

Automatically set a created\_at timestamp or log who added the row.

CREATE TRIGGER before\_insert\_employee

BEFORE INSERT ON employees

FOR EACH ROW

BEGIN

SET NEW.created\_at = NOW();

**🔹 2. UPDATE Trigger**

* **Fires when:** An existing row is **modified**.
* **Purpose:** Audit changes, enforce rules, track old and new values.

**✅ Example Use Case:**

Log old and new salary when an employee's record is updated.

CREATE TRIGGER after\_update\_salary

AFTER UPDATE ON employees

FOR EACH ROW

BEGIN

INSERT INTO salary\_changes(employee\_id, old\_salary, new\_salary)

VALUES (OLD.id, OLD.salary, NEW.salary);

END;

**🔹 3. DELETE Trigger**

* **Fires when:** A row is **removed** from a table.
* **Purpose:** Prevent deletion under certain conditions, log deleted records.

**✅ Example Use Case:**

Backup deleted rows to an archive table.

CREATE TRIGGER after\_delete\_employee

AFTER DELETE ON employees

FOR EACH ROW

BEGIN

INSERT INTO deleted\_employees(id, name, department)

VALUES (OLD.id, OLD.name, OLD.department);

END;

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

16. Introduction to PL/SQL

Theory Questions:

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

### Ans. 🔹 What is PL/SQL?

**PL/SQL (Procedural Language/Structured Query Language)** is **Oracle's procedural extension to SQL**. It allows developers to write **program logic** (procedures, functions, loops, conditions) using a programming structure similar to traditional languages like C or Pascal, but integrated with SQL.

🔧 Think of PL/SQL as **SQL + Programming Features** like variables, loops, conditionals, and modular code.

**✅ How PL/SQL Extends SQL's Capabilities**

| **Feature** | **SQL** | **PL/SQL Enhancement** |
| --- | --- | --- |
| **Procedural Logic** | Not supported | Supports IF, LOOP, CASE, etc. |
| **Variables** | Not available | Allows declaring and using variables |
| **Control Structures** | Limited | Full control flow: IF, WHILE, FOR, etc. |
| **Exception Handling** | Not available | Provides EXCEPTION blocks for error handling |
| **Modular Programming** | Not supported | Supports procedures, functions, packages |
| **Code Reuse** | Manual repetition | Enables reuse through stored procedures |

**📦 PL/SQL Block Structure**

DECLARE

v\_salary NUMBER;

BEGIN

SELECT salary INTO v\_salary FROM employees WHERE id = 101;

IF v\_salary < 3000 THEN

UPDATE employees SET salary = 3000 WHERE id = 101;

END IF;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Employee not found.');

END;

* DECLARE – Define variables.
* BEGIN – Main logic.
* EXCEPTION – Handle errors.
* END – Close the block.

**🔄 Key Benefits of PL/SQL Over Plain SQL**

1. **Allows complex decision-making** with IF...THEN...ELSE
2. **Supports loops** (FOR, WHILE) for repeated operations
3. **Handles exceptions gracefully** with EXCEPTION blocks
4. **Improves performance** by reducing network traffic (multiple operations in one block)
5. **Encourages modularity** using stored procedures, functions, and packages

**🧠 Real-World Use Cases**

* Automating business rules
* Writing stored procedures for payroll, billing, etc.
* Triggers for data auditing
* Scheduled data processing tasks

Let me know if you'd like a beginner-friendly PL/SQL program example!

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

### Ans. ✅ Benefits of Using PL/SQL

PL/SQL (Procedural Language/SQL) offers several advantages that enhance the power and flexibility of SQL in Oracle databases. Below are the key benefits:

**🔹 1. Combines SQL with Procedural Logic**

* PL/SQL integrates the power of SQL with programming constructs like:
  + IF...THEN...ELSE
  + LOOP, WHILE
  + CASE, GOTO
* Allows writing full programs that interact with the database in intelligent ways.

**🔹 2. Improved Performance**

* PL/SQL code is **compiled and stored** in the database.
* Reduces **network traffic** by allowing multiple SQL statements to be executed in a single block.
* Better than sending individual queries one by one.

**🔹 3. Supports Modular Programming**

* You can define and reuse:
  + **Procedures**
  + **Functions**
  + **Packages**
* Encourages clean, maintainable, and reusable code.

**🔹 4. Robust Error Handling**

* PL/SQL provides an EXCEPTION block to gracefully handle runtime errors.
* Helps prevent crashes and improves application reliability.

BEGIN

-- some operation

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('No data found!');

END;

**🔹 5. Increased Security**

* You can **restrict direct table access** and allow users to access data only through **procedures or functions**.
* Ensures better control and data protection.

**🔹 6. Code Reusability**

* Write once, use many times.
* Common logic (e.g., tax calculations, validations) can be stored in procedures/functions.

**🔹 7. Easy Maintenance**

* If a business rule changes, update the PL/SQL code in one place.
* No need to modify every client application.

**🔹 8. Tight Integration with Oracle Database**

* PL/SQL is deeply embedded in the Oracle engine.
* Supports advanced Oracle features like:
  + Triggers
  + Cursors
  + Bulk operations (FORALL, BULK COLLECT)

**🔹 9. Supports Triggers and Automation**

* PL/SQL can be used to define triggers that **automatically react** to events (INSERT, UPDATE, DELETE).

**📌 Summary Table**

| **Benefit** | **Description** |
| --- | --- |
| Combines SQL & Logic | Adds decision-making, loops, etc. to SQL |
| Better Performance | Fewer round-trips between client and server |
| Modular Structure | Supports procedures, functions, and packages |
| Error Handling | Manages errors using EXCEPTION blocks |
| Enhanced Security | Data access through controlled procedures |
| Code Reusability | Avoid repeating business logic |
| Easier Maintenance | Centralized logic for easy updates |
| Oracle Integration | Optimized for use with Oracle features |
| Automation via Triggers | Automate business rules and data validations |

Would you like a sample PL/SQL program that demonstrates these features?

17. PL/SQL Control Structures

Theory Questions:

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

### Ans. 🔹 What Are **Control Structures** in PL/SQL?

**Control structures** in PL/SQL are programming constructs that control the **flow of execution** within a PL/SQL block. They help in making decisions, repeating actions, and branching logic based on conditions.

### ✅ Three Main Types of Control Structures in PL/SQL:

1. **Conditional Control** – IF-THEN, IF-THEN-ELSE, CASE
2. **Iterative Control (Loops)** – LOOP, WHILE, FOR
3. **Sequential Control** – GOTO, NULL

## 🔹 1. ****IF-THEN Control Structure****

Used to perform **conditional execution**. It checks a condition and executes a block of statements if the condition is true.

### 🔸 Syntax:

IF condition THEN

-- statements

END IF;

### 🔸 Variants:

* IF ... THEN ... END IF;
* IF ... THEN ... ELSE ... END IF;
* IF ... THEN ... ELSIF ... ELSE ... END IF;

### ✅ Example:

DECLARE

v\_salary NUMBER := 2800;

BEGIN

IF v\_salary < 3000 THEN

DBMS\_OUTPUT.PUT\_LINE('Salary is below threshold.');

END IF;

END;

## 🔹 2. ****LOOP Control Structure****

Used to **repeat a block of code** multiple times.

### 🔸 Basic LOOP Syntax (infinite unless exited):

LOOP

-- statements

EXIT WHEN condition;

END LOOP;

### ✅ Example:

DECLARE

counter NUMBER := 1;

BEGIN

LOOP

DBMS\_OUTPUT.PUT\_LINE('Counter: ' || counter);

counter := counter + 1;

EXIT WHEN counter > 5;

END LOOP;

END;

### 🔸 Other Types of Loops:

| **Loop Type** | **Description** |
| --- | --- |
| LOOP | Repeats until explicitly exited |
| WHILE | Repeats **while** a condition is true |
| FOR | Repeats for a fixed number of iterations |

### 📌 Summary Table

| **Control Structure** | **Purpose** | **Example Use** |
| --- | --- | --- |
| IF-THEN | Conditional execution | Checking values, validating conditions |
| LOOP | Repeats until exit condition | Repeating a block, iterating counters |

Would you like to see WHILE and FOR loops with examples too?

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

### Ans. 🔹 How Control Structures in PL/SQL Help in Writing Complex Queries

Control structures in PL/SQL (such as IF, LOOP, WHILE, FOR, and CASE) allow you to go **beyond the limitations of plain SQL** by adding logic, decision-making, and repetition. This enables you to handle **complex business rules, dynamic operations**, and **conditional processing** that would be difficult—or even impossible—using SQL alone.

**✅ Benefits of Using Control Structures in Complex PL/SQL Queries**

**🔸 1. Conditional Execution (IF, CASE)**

You can perform different actions based on certain conditions or data values.

**📌 Example Use:**

IF customer\_type = 'Premium' THEN

discount := 0.20;

ELSE

discount := 0.10;

END IF;

🔹 Useful when logic depends on specific input values, such as setting different prices, bonuses, or access levels.

**🔸 2. Iterative Logic (LOOP, WHILE, FOR)**

When dealing with **multiple rows**, **batch processing**, or **calculations** that require repetition, loops allow you to handle these efficiently.

**📌 Example Use:**

FOR i IN 1..10 LOOP

INSERT INTO temp\_table VALUES (i, i\*i);

END LOOP;

🔹 Helps process multiple records, generate reports, or populate tables dynamically.

**🔸 3. Error Handling (EXCEPTION)**

Combining control structures with exception blocks helps you handle failures (like missing data or invalid operations) without crashing the program.

**📌 Example:**

BEGIN

SELECT salary INTO v\_sal FROM employees WHERE id = 101;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Employee not found.');

END;

**🔸 4. Dynamic Query Handling**

You can build and execute SQL statements based on conditions using control structures, making the code flexible and dynamic.

IF table\_name = 'employees' THEN

EXECUTE IMMEDIATE 'DELETE FROM employees WHERE status = ''inactive''';

END IF;

**🔸 5. Modular and Readable Code**

Control structures break complex tasks into logical blocks, making code **easier to write, understand, and maintain**.

**📌 Real-World Scenarios Where They Help:**

| **Scenario** | **Control Structure Used** |
| --- | --- |
| Apply different tax rates based on income | IF-THEN-ELSE |
| Loop through 100 customer IDs | FOR LOOP |
| Retry operation if failed | LOOP + EXCEPTION |
| Apply logic only to certain departments | CASE, IF |
| Execute different SQL based on conditions | IF, EXECUTE IMMEDIATE |

**🟩 In Summary:**

Control structures in PL/SQL **transform static SQL into powerful procedural programs** by allowing:

* **Decisions** to be made dynamically
* **Tasks to be repeated**
* **Errors to be managed**
* **Code to be modularized**

👉 They are essential for building **robust, real-world enterprise applications** that go beyond basic data retrieval.

Would you like a complete PL/SQL example demonstrating all control structures together?

18. SQL Cursors

Theory Questions:

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

### Ans. 🔹 What is a **Cursor** in PL/SQL?

A **cursor** in PL/SQL is a **pointer** to the context area that stores the result of a query. It allows you to **retrieve and process query results row by row** — especially useful when handling multiple rows of data returned by a SELECT statement.

### ✅ Why Use Cursors?

* To process each row of a query result individually
* To fetch data into variables for custom logic or processing
* To gain control over complex data manipulation and reporting tasks

## 🔄 Types of Cursors in PL/SQL

| **Cursor Type** | **Description** |
| --- | --- |
| **Implicit Cursor** | Automatically created by Oracle for single-row queries like SELECT INTO, INSERT, UPDATE, or DELETE |
| **Explicit Cursor** | Manually declared by the programmer to handle multi-row SELECT queries |

### 🔹 1. **Implicit Cursor**

Oracle creates it **automatically** when you execute a DML statement or a SELECT INTO query that returns a single row.

#### ✅ Example:

DECLARE

v\_name employees.name%TYPE;

BEGIN

SELECT name INTO v\_name FROM employees WHERE id = 101;

DBMS\_OUTPUT.PUT\_LINE('Employee Name: ' || v\_name);

END;

🔸 Oracle internally uses an **implicit cursor** here.

#### 🔸 Key Attributes (used with SQL):

* SQL%FOUND – TRUE if at least one row is affected
* SQL%NOTFOUND – TRUE if no rows are affected
* SQL%ROWCOUNT – Number of rows affected

### 🔹 2. **Explicit Cursor**

Used when a SELECT statement **returns multiple rows** and you want to **fetch them one by one**.

#### ✅ Steps to Use an 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 FROM employees WHERE department = 'Sales';

v\_name employees.name%TYPE;

BEGIN

OPEN emp\_cursor;

LOOP

FETCH emp\_cursor INTO v\_name;

EXIT WHEN emp\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Employee: ' || v\_name);

END LOOP;

CLOSE emp\_cursor;

END;

### 📊 Comparison: Implicit vs Explicit Cursors

| **Feature** | **Implicit Cursor** | **Explicit Cursor** |
| --- | --- | --- |
| Created by | Oracle automatically | Manually by the programmer |
| Use case | Single-row operations | Multi-row SELECT operations |
| Cursor name | Always SQL | Custom name (e.g., emp\_cursor) |
| Needs open/fetch/close | No | Yes |
| Control | Less (automatic handling) | More (step-by-step data processing) |
| Error handling | Must handle exceptions (NO\_DATA\_FOUND) | Can use loops with %FOUND, %NOTFOUND |

### 🧠 Summary

* Use **implicit cursors** for simple, one-row operations.
* Use **explicit cursors** when you need to loop through **multiple rows** and handle them one at a time with more **custom logic**.

Let me know if you’d like to explore **cursor FOR loops** or **parameterized cursors** next!

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

### Ans. 🔹 When Would You Use an Explicit Cursor Over an Implicit Cursor?

You should use an **explicit cursor** in PL/SQL when:

**✅ 1. You Need to Process Multiple Rows One-by-One**

* **Implicit cursors** are for single-row operations only.
* **Explicit cursors** let you **loop through and process multiple rows** returned by a SELECT query.

📌 Example Use: Displaying all employees in a department, calculating row-based totals, or generating custom reports.

DECLARE

CURSOR emp\_cur IS

SELECT name FROM employees WHERE department = 'HR';

v\_name employees.name%TYPE;

BEGIN

OPEN emp\_cur;

LOOP

FETCH emp\_cur INTO v\_name;

EXIT WHEN emp\_cur%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('Employee: ' || v\_name);

END LOOP;

CLOSE emp\_cur;

END;

**✅ 2. You Need More Control Over Query Execution**

Explicit cursors allow:

* Manual control over **when** the query starts (OPEN)
* Step-by-step **fetching** of rows
* **Custom logic** for each row processed
* Access to **cursor attributes** like:
  + %FOUND
  + %NOTFOUND
  + %ROWCOUNT
  + %ISOPEN

**✅ 3. You Want to Use Cursor Attributes on SELECT Queries**

Only explicit cursors let you use advanced cursor attributes for **SELECT** operations returning multiple rows.

**✅ 4. You Want to Reuse the Cursor or Make it Parameterized**

* Explicit cursors can be **reused** and even **parameterized** (e.g., pass department ID to a cursor).
* Implicit cursors are **created and destroyed automatically**, and cannot be reused.

CURSOR emp\_by\_dept(p\_dept\_id NUMBER) IS

SELECT \* FROM employees WHERE department\_id = p\_dept\_id;

**✅ 5. You're Working with Complex Business Logic**

Explicit cursors help when:

* You need to process data row by row with conditional statements
* You want to **track** how many rows are fetched/processed
* You need to **log or audit** changes row by row

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

👉 In short, use an **explicit cursor** when you need:

* **More than one row**
* **Fine-grained control**
* **Custom logic for each row**

Let me know if you'd like examples of **cursor FOR loops** or **parameterized cursors** to go further!

19. Rollback and Commit Savepoint

Theory Questions:

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

### Ans. 🔹 When Is It Useful to Use SAVEPOINTS in a Database Transaction?

A **SAVEPOINT** in SQL marks a specific point within a transaction that you can **roll back to without affecting the entire transaction**. It is especially useful for **partial undo** of changes in long or complex transactions.

**✅ Key Use Cases for SAVEPOINTS:**

**🔸 1. Handling Partial Errors in Large Transactions**

When a transaction involves **multiple steps**, some of which may fail, a savepoint allows you to:

* Roll back just the failed part
* Keep the successful operations intact

📌 Example:

BEGIN;

UPDATE accounts SET balance = balance - 100 WHERE id = 1;

SAVEPOINT after\_debit;

-- This might fail

UPDATE accounts SET balance = balance + 100 WHERE id = 999;

-- If the second update fails

ROLLBACK TO after\_debit;

COMMIT;

This ensures the debit is undone only if the credit fails.

**🔸 2. Nested Logic or Subtasks**

If your application logic has **nested processes** or **sub-transactions**, savepoints can be used to revert only a part of the logic without aborting the entire process.

**🔸 3. Avoiding Full Rollbacks**

In case of errors or validations failing mid-transaction, using a savepoint avoids rolling back **everything** — you lose less work and can still commit valid changes.

**🔸 4. Transactional Checkpoints**

Savepoints can act as **milestones** or checkpoints in a transaction where everything up to that point is known to be stable or valid.

**🔸 5. Improved Error Recovery**

Instead of catching errors at the end, you can use savepoints to **step back gradually** in the event of complex error scenarios.

**🔹 Syntax Recap:**

SAVEPOINT savepoint\_name;

-- do something risky

ROLLBACK TO savepoint\_name;

-- commit the rest

COMMIT;

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

|  |  |
| --- | --- |
|  |  |
|  |  |