**ASSIGNMENT**

**Module 4 – Introduction to DBMS**

**Theory Exercise**

**Introduction to SQL**

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

Structured query language (SQL) is a programming language for storing and processing information in a relational database. A relational database stores information in tabular form, with rows and columns representing different data attributes and the various relationships between the data values.

It allows users to **store, retrieve, manipulate, and control access** to data stored in database systems like MySQL, Oracle, PostgreSQL, and SQL Server.

**SQL is the backbone of database management** — it empowers you to organize, secure, and analyze data effectively, forming the foundation of modern applications and business intelligence systems.

Why SQL Is Essential in Database Management:

1. **Data Retrieval (Querying Data)**  
SQL enables users to extract specific information from large datasets efficiently using commands like

SELECT \* FROM employees WHERE department = 'HR';

2. **Data Manipulation**  
You can **insert, update, and delete** records easily:

INSERT INTO books VALUES (101, 'SQL Basics', 'John Doe', 2022, 500);

UPDATE members SET email = 'newmail@example.com' WHERE member\_id = 5;

DELETE FROM books WHERE price > 1000;

3. **Data Definition (Creating and Modifying Structure)**  
SQL allows you to define and change the **structure** of a database:

CREATE TABLE members (

member\_id INT PRIMARY KEY,

name VARCHAR(50),

email VARCHAR(100)

);

ALTER TABLE members ADD date\_of\_membership DATE;

4. **Data Control (Security and Access)**  
SQL provides commands to **grant and revoke permissions**, ensuring data security:

GRANT SELECT ON books TO librarian;

REVOKE INSERT ON books FROM librarian;

5. **Transaction Control**  
SQL ensures **data integrity** through transactions using commands like:

COMMIT; -- Save changes permanently

ROLLBACK; -- Undo changes if something goes wrong

SAVEPOINT; -- Create a temporary rollback point

6. **Portability and Standardization**  
SQL is **universally supported** by all major relational database systems, making it a **portable and standardized** skill.

**2. Explain the difference between DBMS and RDBMS**.

**DBMS vs RDBMS**

| **Feature** | **DBMS (Database Management System)** | **RDBMS (Relational Database Management System)** |
| --- | --- | --- |
| **Definition** | DBMS is software that manages and organizes data in a database. | RDBMS is an advanced type of DBMS that stores data in tables (relations) and maintains relationships between them. |
| **Data Storage Format** | Data is stored in **files** (e.g., hierarchical or network models). | Data is stored in **tables** with rows and columns. |
| **Data Relationships** | Does **not** establish relationships between data. | Uses **foreign keys** to establish relationships between tables. |
| **Normalization** | Normalization is **not supported**. | Supports **data normalization** to reduce redundancy. |
| **Integrity Constraints** | Constraints like **Primary Key**, **Foreign Key**, **Unique**, etc., are **not enforced**. | Supports and enforces **integrity constraints** to maintain data accuracy. |
| **Examples** | Microsoft Access (basic version), XML Database, File System | MySQL, Oracle, PostgreSQL, SQL Server |
| **Data Access** | Allows data access for a **single user** at a time (mostly). | Allows **multiple users** to access data simultaneously. |
| **Security** | Basic security features. | Provides **advanced security and authorization** mechanisms. |

### ****In Simple Terms****

* **DBMS** = A system for storing and managing data.
* **RDBMS** = A system for storing and managing **related** data in a **structured, table-based** format with **relationships and rules**.

### ****Example****

* **DBMS Example:** A simple file storing student records (like a spreadsheet).
* **RDBMS Example:** A set of tables — one for **Students**, another for **Courses**, linked by a **Student\_ID**, showing how each student is enrolled in specific courses.

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

**Role of SQL in Managing Relational Databases**

**SQL (Structured Query Language)** plays a **central role** in managing relational databases.  
It is the **standard language** used to interact with, organize, and control data stored in **RDBMS** (Relational Database Management Systems) such as MySQL, Oracle, PostgreSQL, and SQL Server.

### ****Main Roles of SQL in Database Management****

#### 1. ****Data Definition (DDL – Data Definition Language)****

SQL defines and modifies the **structure** of database objects like tables, views, and indexes.

**Commands:**

CREATE TABLE students (

student\_id INT PRIMARY KEY,

name VARCHAR(50),

course VARCHAR(50)

);

ALTER TABLE students ADD email VARCHAR(100);

DROP TABLE old\_students;

#### 2. ****Data Manipulation (DML – Data Manipulation Language)****

SQL allows **inserting, updating, deleting, and retrieving** data from tables.

**Commands:**

INSERT INTO students VALUES (101, 'Amit', 'Science', 'amit@example.com');

UPDATE students SET course = 'Maths' WHERE student\_id = 101;

DELETE FROM students WHERE student\_id = 101;

#### 3. ****Data Querying (SELECT Statements)****

The **SELECT** command retrieves specific data from one or more tables.

**Example:**

SELECT name, course FROM students WHERE course = 'Science';

#### 4. ****Data Control (DCL – Data Control Language)****

SQL controls **user access and permissions** to ensure data security.

**Commands:**

GRANT SELECT, INSERT ON students TO librarian;

REVOKE INSERT ON students FROM librarian;

#### 5. ****Transaction Control (TCL – Transaction Control Language)****

SQL manages **transactions** to maintain data consistency and integrity.

**Commands:**

BEGIN;

INSERT INTO students VALUES (102, 'Ravi', 'English', 'ravi@example.com');

SAVEPOINT add\_student;

ROLLBACK TO add\_student;

COMMIT;

#### 6. ****Data Integrity and Relationships****

SQL enforces **rules and relationships** between tables using **primary keys, foreign keys, and constraints.**

**Example:**

ALTER TABLE enrollments

ADD CONSTRAINT fk\_student FOREIGN KEY (student\_id) REFERENCES students(student\_id);

**4. What are the key features of SQL?**

SQL is the **standard language** used to manage and manipulate data in **relational databases**.  
It provides powerful features that make data handling efficient, consistent, and secure.

**1. Data Definition**

SQL allows you to define and manage the structure of a database using DDL (Data Definition Language) commands.  
**Examples:**

CREATE TABLE employees (

emp\_id INT PRIMARY KEY,

name VARCHAR(50),

salary DECIMAL(10,2)

);

ALTER TABLE employees ADD department VARCHAR(30);

**2. Data Manipulation**

You can insert, update, delete, and retrieve data using DML (Data Manipulation Language) commands.  
**Examples:**

INSERT INTO employees VALUES (1, 'Amit', 50000, 'HR');

UPDATE employees SET salary = 55000 WHERE emp\_id = 1;

DELETE FROM employees WHERE emp\_id = 1;

**3. Data Querying**

SQL provides powerful **query capabilities** to **retrieve and filter** data easily.  
**Example:**

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

**4. Data Control**

SQL ensures **security and controlled access** to the database using **DCL (Data Control Language)** commands.  
**Examples:**

GRANT SELECT, UPDATE ON employees TO hr\_manager;

REVOKE UPDATE ON employees FROM hr\_manager;

**5. Transaction Control**

SQL supports **transactions** to maintain **data integrity** during multiple operations.  
**Commands:**

BEGIN;

UPDATE employees SET salary = salary + 5000 WHERE department = 'IT';

COMMIT;

ROLLBACK;

**6. Data Integrity**

SQL enforces **rules and constraints** to maintain **accuracy and reliability** of data.  
**Examples:**

CREATE TABLE students (

student\_id INT PRIMARY KEY,

name VARCHAR(50) NOT NULL,

email VARCHAR(100) UNIQUE

);

**7. Portability and Standardization**

SQL is an ANSI standard language, meaning it works with almost all relational database systems (MySQL, Oracle, PostgreSQL, SQL Server, etc.).

**8. Scalability and Performance**

SQL supports large databases and complex queries, making it suitable for enterprise-level applications.

**2. SQL Syntax:**

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

The **basic components of SQL syntax** are the fundamental building blocks used to write and execute SQL commands for managing and querying databases.

**1. Keywords / Commands**

These are predefined words in SQL used to perform specific operations.  
**Examples:**

* SELECT, INSERT, UPDATE, DELETE – Data manipulation
* CREATE, ALTER, DROP – Database and table management
* WHERE, GROUP BY, ORDER BY – Filtering and sorting data

**2. Identifiers**

Names given to database objects such as tables, columns, or databases.  
**Examples:**

CREATE TABLE employees (emp\_id INT, emp\_name VARCHAR(50));

Here, employees, emp\_id, and emp\_name are identifiers.

**3. Clauses**

Parts of SQL statements that perform specific functions.  
**Examples:**

FROM — specifies the table to query

WHERE — filters records

GROUP BY — groups rows

ORDER BY — sorts results

**4. Expressions**

Combinations of symbols, column names, and values used to calculate or evaluate data.  
**Examples:**

salary \* 0.1 -- Expression to calculate 10% of salary

price + tax -- Expression to find total cost

**5. Predicates**

Used to specify conditions that can be **TRUE**, **FALSE**, or **UNKNOWN**.  
**Examples:**

WHERE salary > 50000

WHERE department = 'HR'

**6. Operators**

Symbols that perform operations on expressions or values.  
**Types:**

**Arithmetic:** +, -, \*, /

**Comparison:** =, <>, >, <, >=, <=

**Logical:** AND, OR, NOT

**7. Literals**

Fixed data values written directly in SQL statements.  
**Examples:**

'John' -- String literal

50000 -- Numeric literal

'2025-10-20' -- Date literal

**8. Semicolon (;)**

Used to **end an SQL statement** (especially when executing multiple statements).  
**Example:**

SELECT \* FROM employees;

**Example Combining All Components:**

SELECT emp\_name, salary \* 0.1 AS bonus

FROM employees

WHERE department = 'Sales'

ORDER BY salary DESC;

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

The **general structure of an SQL** SELECT **statement** defines how to retrieve data from one or more tables in a database.

**General Structure:**

SELECT [DISTINCT] column\_list

FROM table\_name

[WHERE condition]

[GROUP BY column\_list]

[HAVING condition]

[ORDER BY column\_list [ASC | DESC]];

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

The **role of clauses in SQL statements** is to define the **specific actions, conditions, and structure** of a query.  
Each clause serves a distinct purpose in telling the database **what to do**, **where to do it**, and **how to display the results**.

**Common SQL Clauses and Their Roles**

| **Clause** | **Role / Function** | **Example** |
| --- | --- | --- |
| **SELECT** | Specifies the columns or expressions to retrieve from the table. | SELECT name, salary |
| **FROM** | Defines the table(s) from which to retrieve data. | FROM employees |
| **WHERE** | Filters rows based on a condition before grouping or aggregation. | WHERE salary > 50000 |
| **GROUP BY** | Groups rows that have the same values in specified columns (used with aggregate functions like SUM, AVG). | GROUP BY department |
| **HAVING** | Applies conditions to groups created by GROUP BY. | HAVING AVG(salary) > 60000 |
| **ORDER BY** | Sorts the result set in ascending (ASC) or descending (DESC) order. | ORDER BY salary DESC |
| **LIMIT / TOP** *(varies by SQL dialect)* | Restricts the number of rows returned. | LIMIT 5 (MySQL) / SELECT TOP 5 (SQL Server) |

**3. SQL Constraints**

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

A constraint is a restriction or condition applied to a column or a table to maintain the correctness and validity of the data stored.

**Types of Constraints in SQL**

| **Constraint Type** | **Description** | **Example Syntax** |
| --- | --- | --- |
| **1. NOT NULL** | Ensures that a column **cannot have NULL values**. Every record must contain a value in that column. | sql CREATE TABLE employees ( emp\_id INT NOT NULL, emp\_name VARCHAR(50) NOT NULL ); |
| **2. UNIQUE** | Ensures that all values in a column (or combination of columns) are **unique** — no duplicates allowed. | sql CREATE TABLE students ( roll\_no INT UNIQUE, email VARCHAR(100) UNIQUE ); |
| **3. PRIMARY KEY** | Combines **NOT NULL** and **UNIQUE** — uniquely identifies each record in a table. Each table can have **only one primary key**. | sql CREATE TABLE books ( book\_id INT PRIMARY KEY, title VARCHAR(100) ); |
| **4. FOREIGN KEY** | Maintains **referential integrity** between two tables by linking a column in one table to the **primary key** of another. | sql CREATE TABLE orders ( order\_id INT PRIMARY KEY, customer\_id INT, FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id) ); |
| **5. CHECK** | Ensures that values in a column meet a **specific condition**. | sql CREATE TABLE employees ( emp\_id INT PRIMARY KEY, salary DECIMAL(10,2) CHECK (salary > 0) ); |
| **6. DEFAULT** | Assigns a **default value** to a column when no value is provided. | sql CREATE TABLE members ( join\_date DATE DEFAULT CURRENT\_DATE ); |

* Constraints can be applied at two levels:
  + **Column-level:** applied to a single column.
  + **Table-level:** applied to multiple columns (like composite keys).
* Constraints can be added **during table creation** or **after table creation** using the ALTER TABLE statement.

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

Both **PRIMARY KEY** and **FOREIGN KEY** are essential SQL constraints used to maintain **data integrity** and define **relationships** between tables — but they serve **different purposes**.

**1. Definition**

| **Constraint** | **Description** |
| --- | --- |
| **PRIMARY KEY** | Uniquely identifies each record in a table. Ensures that no two rows have the same key value and that the key column(s) cannot be NULL. |
| **FOREIGN KEY** | Establishes a **relationship** between two tables by referencing the **PRIMARY KEY** of another table. Ensures referential integrity. |

**2. Key Differences**

| **Basis** | **PRIMARY KEY** | **FOREIGN KEY** |
| --- | --- | --- |
| **Purpose** | Uniquely identifies a record within a table. | Links two tables and enforces referential integrity. |
| **Uniqueness** | Must contain **unique** values. | Can contain **duplicate** values. |
| **NULL Values** | **Cannot** contain NULL values. | **Can** contain NULL values (if the relationship allows missing references). |
| **Number per Table** | Each table can have **only one** primary key. | A table can have **multiple foreign keys**. |
| **Defined On** | Defined in the **same table** where the record is uniquely identified. | Defined in a **child table** to reference the parent table’s primary key. |
| **Relation** | Acts as the **parent key** in a relationship. | Acts as the **child key** in a relationship. |
| **Automatic Indexing** | Automatically creates an index. | Does **not** automatically create an index (can be added manually). |

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

### **Role of** NOT NULL **and** UNIQUE **Constraints in SQL**

Both NOT NULL and UNIQUE constraints are used to **maintain data integrity and accuracy** in a database, but they serve **different purposes.**

### ****1.**** NOT NULL ****Constraint****

The NOT NULL constraint ensures that a column **must always have a value —** it **cannot contain NULL** (empty) entries.

#### ****Role / Purpose:****

* Guarantees that **important columns are never left blank.**
* Ensures **data completeness** by forcing users to enter a value for that field.

#### ****Example:****

CREATE TABLE employees (

emp\_id INT PRIMARY KEY,

emp\_name VARCHAR(50) NOT NULL,

salary DECIMAL(10,2) NOT NULL

);

**Explanation:**

* emp\_name and salary cannot be left empty when inserting a record.

INSERT INTO employees (emp\_id, emp\_name, salary)

VALUES (101, NULL, 50000); -- ❌ Error! emp\_name cannot be NULL

### ****2.**** UNIQUE ****Constraint****

The UNIQUE constraint ensures that **all values in a column are distinct** — no two rows can have the same value in that column.

#### ****Role / Purpose:****

* Prevents **duplicate entries** in a column.
* Ensures **data uniqueness** across records.

#### ****Example:****

CREATE TABLE students (

roll\_no INT UNIQUE,

email VARCHAR(100) UNIQUE

);

**Explanation:**

* Each roll\_no and email must be different for every student.
* **Key Difference Between NOT NULL and UNIQUE**

| **Feature** | **NOT NULL** | **UNIQUE** |
| --- | --- | --- |
| **Purpose** | Ensures a column **always has a value** | Ensures **no duplicate values** in a column |
| **NULL Allowed** | ❌ Not allowed | ✅ Allowed (but only one NULL value per column, depending on DBMS) |
| **Focus** | Data completeness | Data uniqueness |
| **Can be Combined** | Yes — both can be applied to the same column | Example: email VARCHAR(100) NOT NULL UNIQUE |

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

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

**SQL Data Definition Language (DDL)** is a subset of SQL commands used to **define, modify, and manage the structure of database objects** such as tables, schemas, indexes, and views.**DDL** is used to **create and change the structure** of the database — not the data stored in it.

* DDL commands define how data is stored in the database.
* They operate on database objects (tables, constraints, schemas, etc.).
* Every DDL command results in an automatic commit — changes are saved permanently.
* DDL focuses on schema definition and management, not data manipulation.

**Common DDL Commands:**

| **Command** | **Purpose** | **Example** |
| --- | --- | --- |
| **CREATE** | Creates new database objects (like tables, views, or databases). | sql CREATE TABLE students (id INT, name VARCHAR(50)); |
| **ALTER** | Modifies existing database objects (adds or removes columns). | sql ALTER TABLE students ADD email VARCHAR(100); |
| **DROP** | Deletes database objects permanently. | sql DROP TABLE students; |
| **TRUNCATE** | Removes all records from a table but keeps its structure. | sql TRUNCATE TABLE students; |
| **RENAME** | Changes the name of a database object. | sql RENAME TABLE students TO learners; |

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

The CREATE command in SQL is used to **define and create new database objects** — for example, creating a new table or database structure to store data.

### ****Common Uses of the**** CREATE ****Command****

You can use CREATE to make:

* A **database**
* A **table**
* A **view**
* An **index**
* A **schema**

### ****1. Syntax for Creating a Table****

CREATE TABLE table\_name (

column1 datatype [constraint],

column2 datatype [constraint],

...

);

#### ****Explanation of Syntax:****

| **Part** | **Description** |
| --- | --- |
| CREATE TABLE | Command to create a new table. |
| table\_name | Name of the table to be created. |
| column1, column2, ... | Names of the table’s columns. |
| datatype | Specifies the type of data each column will store (e.g., INT, VARCHAR, DATE, DECIMAL). |
| [constraint] | Optional — defines rules such as PRIMARY KEY, NOT NULL, UNIQUE, etc. |

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

Purpose of Data Types

A **data type** defines the **kind of values** that can be stored in a column — such as numbers, text, or dates.

**Purpose / Importance:**

| **Purpose** | **Explanation** | **Example** |
| --- | --- | --- |
| **Ensures valid data storage** | Restricts the type of data entered (e.g., numbers in numeric fields only). | age INT → only whole numbers allowed. |
| **Controls memory usage** | Allocates appropriate storage space for each column type. | CHAR(10) uses fixed space; VARCHAR(10) uses only what’s needed. |
| **Improves performance** | Optimizes query speed by using suitable data types. | INT comparisons are faster than text comparisons. |
| **Supports correct operations** | Enables valid calculations and comparisons. | You can sum values in DECIMAL columns but not in VARCHAR. |

### ****Example:****

CREATE TABLE students (

student\_id INT,

student\_name VARCHAR(50),

date\_of\_birth DATE,

marks DECIMAL(5,2)

);

Each column stores only the **specific type** of data defined.

## ****2. Purpose of Constraints****

### ****Definition:****

**Constraints** are rules applied to columns to ensure **data integrity** and **validity** in a table.

### ****Purpose / Importance:****

| **Constraint Type** | **Purpose** | **Example** |
| --- | --- | --- |
| **NOT NULL** | Ensures that a column cannot have empty (NULL) values. | student\_name VARCHAR(50) NOT NULL |
| **UNIQUE** | Ensures that no two rows have the same value in that column. | email VARCHAR(100) UNIQUE |
| **PRIMARY KEY** | Uniquely identifies each record in the table. | student\_id INT PRIMARY KEY |
| **FOREIGN KEY** | Maintains relationships between tables. | FOREIGN KEY (dept\_id) REFERENCES department(dept\_id) |
| **CHECK** | Ensures that values meet a specific condition. | marks CHECK (marks >= 0 AND marks <= 100) |
| **DEFAULT** | Assigns a default value when none is provided. | admission\_date DATE DEFAULT CURRENT\_DATE |

### ****Example:****

CREATE TABLE employees (

emp\_id INT PRIMARY KEY,

emp\_name VARCHAR(50) NOT NULL,

age INT CHECK (age >= 18),

email VARCHAR(100) UNIQUE,

join\_date DATE DEFAULT CURRENT\_DATE

);

**Explanation:**

* Data types ensure proper storage format (e.g., INT, VARCHAR, DATE).
* Constraints enforce rules (e.g., no duplicate emp\_id, age must be ≥ 18).

**5. ALTER Command**

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

The **ALTER** command is a **DDL (Data Definition Language)** command used to **modify the structure of existing database objects**, mainly tables.

**Purpose / Uses of ALTER**

The ALTER command allows you to:

1. Add new columns to an existing table.
2. Modify existing columns (change data type, size, or constraints).
3. Drop columns from a table.
4. Add or drop constraints (e.g., PRIMARY KEY, FOREIGN KEY, CHECK).
5. Rename a table or column (in some SQL dialects).

**Syntax**

-- Add a new column

ALTER TABLE table\_name

ADD column\_name datatype [constraint];

-- Modify an existing column

ALTER TABLE table\_name

MODIFY column\_name new\_datatype [constraint];

-- Drop a column

ALTER TABLE table\_name

DROP COLUMN column\_name;

-- Add a constraint

ALTER TABLE table\_name

ADD CONSTRAINT constraint\_name constraint\_type (column\_name);

-- Drop a constraint

ALTER TABLE table\_name

DROP CONSTRAINT constraint\_name;

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

## ****1. Adding a Column****

You can add a new column to an existing table using the ADD keyword.

### ****Syntax:****

ALTER TABLE table\_name

ADD column\_name datatype [constraint];

### ****Example:****

ALTER TABLE employees

ADD department VARCHAR(30);

* Adds a new column department of type VARCHAR(30) to the employees table.
* You can also add constraints:

ALTER TABLE employees

ADD join\_date DATE NOT NULL;

## ****2. Modifying a Column****

You can modify the **data type, size, or constraints** of an existing column using the MODIFY (MySQL, Oracle) or ALTER COLUMN (SQL Server) keyword.

### ****Syntax (MySQL / Oracle):****

ALTER TABLE table\_name

MODIFY column\_name new\_datatype [constraint];

### ****Syntax (SQL Server):****

ALTER TABLE table\_name

ALTER COLUMN column\_name new\_datatype [constraint];

### ****Example:****

-- MySQL / Oracle

ALTER TABLE employees

MODIFY salary DECIMAL(12,2) NOT NULL;

-- SQL Server

ALTER TABLE employees

ALTER COLUMN salary DECIMAL(12,2) NOT NULL;

* Changes the salary column to allow up to 12 digits with 2 decimal places and sets it as NOT NULL.

## ****3. Dropping a Column****

You can remove a column from a table using the DROP COLUMN clause.

### ****Syntax:****

ALTER TABLE table\_name

DROP COLUMN column\_name;

### ****Example:****

ALTER TABLE employees

DROP COLUMN join\_date;

* Removes the join\_date column from the employees table.
* Note: Dropping a column **deletes all data** stored in that column permanently.

**6. DROP Command**

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

The DROP **command** in SQL is used to **permanently delete database objects** such as a **table, database, view, index, or trigger** from the database.

**Function of the DROP Command**

* It removes an existing database object completely.
* It frees up storage space used by that object.
* It deletes all data, structure, and related constraints of the object.

**Common Syntax**

DROP TABLE table\_name;

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

When you **drop a table** from a database using the DROP TABLE command, it has **serious and permanent effects**. Here are the key **implications**:

**1. Permanent Deletion of Data**

All records (rows) stored in the table are permanently removed.

The deleted data cannot be recovered unless a backup exists.

**2. Loss of Table Structure**

* The **table structure (schema)**, including columns, data types, and constraints, is **deleted**.
* You must **recreate** the table if you want to use it again.

**3. Removal of Associated Constraints and Dependencies**

* Any **primary keys, foreign keys, unique constraints, or indexes** associated with the table are also **deleted**.
* If the table is referenced by **foreign keys** in other tables, you may get an error unless you drop or disable those constraints first.

**4. Impact on Relationships**

* If other tables or views depend on the dropped table, those objects may become **invalid** or **fail to work**.
* For example, dropping a table used in a **view** or **stored procedure** can cause errors.

**5. Storage Space is Freed**

* The disk space previously occupied by the table and its indexes is **released back** to the database system.

**6. Requires Proper Privileges**

* Only users with **DROP privilege** (e.g., database administrator) can perform this operation.

**7. Data Manipulation Language (DML)**

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

**1. INSERT Command**

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

**Syntax:**

INSERT INTO table\_name (column1, column2, column3, ...)

VALUES (value1, value2, value3, ...);

**Example:**

INSERT INTO members (member\_id, member\_name, email)

VALUES (101, 'John Doe', 'john@example.com');

Adds a new record to the members table.

**2. UPDATE Command**

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

**Syntax:**

UPDATE table\_name

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

WHERE condition;

**Example:**

UPDATE books

SET price = 600

WHERE book\_id = 5;

Updates the price of the book whose book\_id is 5.

**3. DELETE Command**

**Purpose:**  
The DELETE command is used to **remove one or more records** from a table.

**Syntax:**

DELETE FROM table\_name

WHERE condition;

**Example:**

DELETE FROM members

WHERE member\_id = 101;

Deletes the record of the member with ID 101.

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

The **WHERE clause** is **very important** in UPDATE and DELETE operations because it **specifies which rows** in the table should be affected by the command.

Without it, **all rows** in the table will be updated or deleted — often leading to **data loss** or **unintended changes**.

### ****importance of WHERE Clause****

#### 1. ****Limits the scope of the operation****

* It ensures that **only specific records** matching the given condition are updated or deleted.
* Prevents unwanted changes to all data in the table.

#### 2. ****Prevents accidental data loss****

* Without a WHERE clause, SQL applies the command to **every row**, which can **erase or modify all records**.

#### 3. ****Maintains data integrity****

* It helps you **target the correct records**, ensuring that only valid and intended data is modified.

#### 4. ****Supports conditional updates or deletions****

* You can apply **logical conditions** (like comparisons, ranges, or pattern matching) to filter rows precisely.

**8. Data Query Language (DQL)**

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

The SELECT statement is used to **fetch specific columns and rows** from a table based on given conditions.  
It allows you to **view, filter, sort, and analyze** data without modifying it.

**Basic Syntax**

SELECT column1, column2, ...

FROM table\_name

WHERE condition;

**Example**

SELECT title, author, price

FROM books

WHERE price > 500;

Retrieves the **title**, **author**, and **price** of all books priced **above 500**.

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

## ****1. WHERE Clause****

### ****Purpose:****

The **WHERE clause** is used to **filter records** — it specifies **which rows** should be included in the result based on a condition.

### ****Syntax:****

SELECT column1, column2, ...

FROM table\_name

WHERE condition;

### ****Example:****

SELECT \*

FROM books

WHERE price > 500;

This retrieves only those books whose **price is greater than 500.**

### ****Key Points:****

* WHERE works with SELECT, UPDATE, and DELETE statements.
* It uses **comparison operators** (=, <, >, <=, >=, !=) and **logical operators** (AND, OR, NOT) to filter data.
* Filtering happens **before** sorting or grouping.

## ****2. ORDER BY Clause****

### ****Purpose:****

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

### ****Syntax:****

SELECT column1, column2, ...

FROM table\_name

ORDER BY column\_name [ASC | DESC];

### ****Example:****

SELECT title, price

FROM books

ORDER BY price DESC;

This displays book titles sorted by **price in descending order** (highest first).

**9. Data Control Language (DCL)**

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

## ****1. GRANT Command****

### ****Purpose:****

The GRANT command is used to **give specific privileges** (permissions) to a user or role so they can perform certain operations on database objects (like tables, views, or procedures).

### ****Syntax:****

GRANT privilege\_name

ON object\_name

TO user\_name;

### ****Example:****

GRANT SELECT, INSERT

ON books

TO librarian;

This allows the user librarian to **view** (SELECT) and **add** (INSERT) records in the books table.

### ****Common Privileges:****

| **Privilege** | **Description** |
| --- | --- |
| **SELECT** | Allows reading data from a table |
| **INSERT** | Allows adding new rows |
| **UPDATE** | Allows modifying existing rows |
| **DELETE** | Allows removing rows |
| **ALL PRIVILEGES** | Grants all permissions on the object |

## ****2. REVOKE Command****

### ****Purpose:****

The REVOKE command is used to **remove or cancel privileges** that were previously granted to a user.

### ****Syntax:****

REVOKE privilege\_name

ON object\_name

FROM user\_name;

### ****Example:****

REVOKE INSERT

ON books

FROM librarian;

This removes the **INSERT** permission from the user librarian, meaning they can no longer add new books.

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

## ****1. Managing Privileges with**** GRANT

### ****Purpose:****

To **assign permissions** to a user or role so they can perform specific actions (like SELECT, INSERT, UPDATE, DELETE) on database objects.

### ****Syntax:****

GRANT privilege\_name(s)

ON object\_name

TO user\_name;

### ****Example 1: Give basic access****

GRANT SELECT, INSERT

ON books

TO librarian;

The user librarian can now **view** and **add** records in the books table.

### ****Example 2: Give full access****

GRANT ALL PRIVILEGES

ON members

TO admin;

The user **admin** can **perform all operations** (SELECT, INSERT, UPDATE, DELETE, etc.) on the members table.

### ****Example 3: Grant privileges to multiple users****

GRANT SELECT

ON books

TO user1, user2;

Both **user1** and **user2** can read data from books.

## ****2. Managing Privileges with**** REVOKE

### ****Purpose:****

To **remove previously granted permissions** from a user or role, preventing them from performing certain actions.

### ****Syntax:****

REVOKE privilege\_name(s)

ON object\_name

FROM user\_name;

### ****Example 1: Remove one privilege****

REVOKE INSERT

ON books

FROM librarian;

The user librarian can no longer insert new records.

**10. Transaction Control Language (TCL)**

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

## ****1. COMMIT Command****

### ****Purpose:****

The COMMIT command is used to **save all the changes** made during the current transaction **permanently** to the database.

Once you commit, the changes **cannot be undone**.

### ****Syntax:****

COMMIT;

### ****Example:****

INSERT INTO books VALUES (11, 'SQL Guide', 'Alex Smith', 2024, 550);

UPDATE books SET price = 600 WHERE book\_id = 5;

COMMIT;

Both the **insert** and **update** operations are permanently saved in the database.

### ****Key Points:****

* Ends the current transaction and **makes changes permanent**.
* Frees locks on the affected rows or tables.
* Once committed, the changes **cannot be rolled back**.

## ****2. ROLLBACK Command****

### ****Purpose:****

The ROLLBACK command is used to **undo changes** made during the current transaction, **reverting the database** to its last committed state.

### ****Syntax:****

ROLLBACK;

### ****Example:****

INSERT INTO books VALUES (12, 'Database Design', 'Emma Lee', 2025, 700);

ROLLBACK;

The inserted record will **not be saved**, as the transaction is rolled back.

## ****3. COMMIT vs ROLLBACK****

| **Feature** | **COMMIT** | **ROLLBACK** |
| --- | --- | --- |
| **Purpose** | Saves changes permanently | Undoes uncommitted changes |
| **Reversibility** | Cannot be undone | Can undo changes since the last COMMIT |
| **Effect on Transaction** | Ends the current transaction and makes changes visible to others | Cancels the current transaction |
| **Example Use** | After successful data entry | On error or invalid data |

## ****Example of Both in a Transaction****

START TRANSACTION;

INSERT INTO members VALUES (201, 'John Doe', '2025-10-22', 'john@example.com');

UPDATE members SET email = 'john\_new@example.com' WHERE member\_id = 201;

-- If all good

COMMIT;

-- If something goes wrong

ROLLBACK;

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

A **transaction** groups multiple SQL statements together so that **either all succeed or none do**.  
It ensures that the database remains in a **consistent state**, even if a failure (like power loss or error) occurs.

## ****How Transactions Are Managed****

SQL databases manage transactions using **Transaction Control Language (TCL)** commands:  
START TRANSACTION, COMMIT, ROLLBACK, and SAVEPOINT.

### ****Step-by-Step Transaction Flow****

#### ****1. Start a Transaction****

A transaction begins automatically when you execute the first SQL statement,  
or explicitly with:

START TRANSACTION;

or

BEGIN;

This marks the start of a transaction block.

#### ****2. Perform SQL Operations****

You can now execute multiple **DML (Data Manipulation Language)** commands such as:

INSERT INTO books VALUES (10, 'SQL Basics', 'John Doe', 2024, 500);

UPDATE books SET price = 550 WHERE book\_id = 10;

These changes are **temporary** until you commit.

#### ****3. Use SAVEPOINT (Optional)****

You can create a **savepoint** — a marker within a transaction to which you can roll back partially.

SAVEPOINT sp1;

UPDATE books SET price = 600 WHERE book\_id = 10;

If something goes wrong after this, you can roll back only to sp1.

#### ****4. Commit or Rollback****

* **Commit the Transaction**
* COMMIT;

Saves all changes permanently to the database.

* **Rollback the Transaction**
* ROLLBACK;

❌ Cancels all changes made since the last COMMIT or START TRANSACTION.

* **Rollback to a Savepoint**
* ROLLBACK TO sp1;

Undoes changes made after the savepoint.

## ****ACID Properties of Transactions****

Transactions follow **ACID** principles to maintain data integrity:

| **Property** | **Meaning** | **Description** |
| --- | --- | --- |
| **A – Atomicity** | “All or nothing” | Entire transaction succeeds or fails as a unit. |
| **C – Consistency** | Valid state | Database moves from one valid state to another. |
| **I – Isolation** | Independence | Transactions don’t interfere with each other. |
| **D – Durability** | Permanence | Once committed, changes are permanent even after a crash. |

## ****Example: Managing a Transaction****

START TRANSACTION;

INSERT INTO members VALUES (301, 'Alice', '2025-10-22', 'alice@example.com');

UPDATE books SET price = price - 50 WHERE book\_id = 3;

-- If all operations succeed

COMMIT;

-- If any error occurs

ROLLBACK;

This ensures that either **both** the insert and update happen, or **neither** do — maintaining data consistency.

## ****In Summary****

| **Step** | **Command** | **Purpose** |
| --- | --- | --- |
| 1 | START TRANSACTION | Begin a new transaction |
| 2 | SAVEPOINT | Create a checkpoint within a transaction |
| 3 | COMMIT | Save all changes permanently |
| 4 | ROLLBACK | Undo uncommitted changes |
| 5 | ROLLBACK TO savepoint | Undo part of the transaction |

**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?**

## ****1. Concept of JOIN****

* Tables in relational databases are often related using **primary key–foreign key relationships**.
* A **JOIN** lets you query these related tables together instead of separately.

**Basic Syntax:**

SELECT table1.column1, table2.column2, ...

FROM table1

JOIN table2

ON table1.common\_column = table2.common\_column;

## ****2. Types of JOINs****

| **JOIN Type** | **Description** | **Result** |
| --- | --- | --- |
| **INNER JOIN** | Returns only the rows that **match in both tables** | Intersection of tables |
| **LEFT JOIN (LEFT OUTER JOIN)** | Returns **all rows from the left table**, and matching rows from the right table. If no match, NULL is returned for right table columns | Left table + matching right table rows |
| **RIGHT JOIN (RIGHT OUTER JOIN)** | Returns **all rows from the right table**, and matching rows from the left table. If no match, NULL is returned for left table columns | Right table + matching left table rows |
| **FULL OUTER JOIN** | Returns **all rows from both tables**, with NULLs in places where there is no match | Union of both tables with NULLs for missing matches |

### ****3. Examples****

Assume we have two tables:

**employees**

| **emp\_id** | **emp\_name** | **dept\_id** |
| --- | --- | --- |
| 1 | Alice | 10 |
| 2 | Bob | 20 |
| 3 | Charlie | 30 |

**departments**

| **dept\_id** | **dept\_name** |
| --- | --- |
| 10 | HR |
| 20 | IT |
| 40 | Sales |

#### ****INNER JOIN****

SELECT emp\_name, dept\_name

FROM employees

INNER JOIN departments

ON employees.dept\_id = departments.dept\_id;

**Result:** Only matching rows

| **emp\_name** | **dept\_name** |
| --- | --- |
| Alice | HR |
| Bob | IT |

#### ****LEFT JOIN****

SELECT emp\_name, dept\_name

FROM employees

LEFT JOIN departments

ON employees.dept\_id = departments.dept\_id;

**Result:** All employees, even if no department match

| **emp\_name** | **dept\_name** |
| --- | --- |
| Alice | HR |
| Bob | IT |
| Charlie | NULL |

#### ****RIGHT JOIN****

SELECT emp\_name, dept\_name

FROM employees

RIGHT JOIN departments

ON employees.dept\_id = departments.dept\_id;

**Result:** All departments, even if no employee match

| **emp\_name** | **dept\_name** |
| --- | --- |
| Alice | HR |
| Bob | IT |
| NULL | Sales |

#### ****FULL OUTER JOIN**** (not supported in MySQL directly, but works in other databases like SQL Server or PostgreSQL)

SELECT emp\_name, dept\_name

FROM employees

FULL OUTER JOIN departments

ON employees.dept\_id = departments.dept\_id;

**Result:** All employees and all departments

| **emp\_name** | **dept\_name** |
| --- | --- |
| Alice | HR |
| Bob | IT |
| Charlie | NULL |
| NULL | Sales |

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

1. **Identify the relationship between tables**
   * Usually, one table has a **primary key** (unique identifier).
   * Another table has a **foreign key** that references this primary key.
2. **Use a JOIN clause in your SELECT statement**
   * Specify the type of join (INNER, LEFT, RIGHT, FULL OUTER) depending on how you want unmatched rows to be handled.
   * Use the ON keyword to define the condition that matches rows between tables.
3. **Retrieve combined data**
   * The result set contains columns from **all joined tables**.
   * Only rows that satisfy the join condition (or all rows depending on join type) are returned.

## ****Example: Combining Two Tables****

**Tables:**

**employees**

| **emp\_id** | **emp\_name** | **dept\_id** |
| --- | --- | --- |
| 1 | Alice | 10 |
| 2 | Bob | 20 |
| 3 | Charlie | 30 |

**departments**

| **dept\_id** | **dept\_name** |
| --- | --- |
| 10 | HR |
| 20 | IT |
| 40 | Sales |

### ****INNER JOIN**** — only matching rows

SELECT emp\_name, dept\_name

FROM employees

INNER JOIN departments

ON employees.dept\_id = departments.dept\_id;

**Result:**

| **emp\_name** | **dept\_name** |
| --- | --- |
| Alice | HR |
| Bob | IT |

### ****LEFT JOIN**** — all rows from left table

SELECT emp\_name, dept\_name

FROM employees

LEFT JOIN departments

ON employees.dept\_id = departments.dept\_id;

**Result:**

| **emp\_name** | **dept\_name** |
| --- | --- |
| Alice | HR |
| Bob | IT |
| Charlie | NULL |

### ****RIGHT JOIN**** — all rows from right table

SELECT emp\_name, dept\_name

FROM employees

RIGHT JOIN departments

ON employees.dept\_id = departments.dept\_id;

**Result:**

| **emp\_name** | **dept\_name** |
| --- | --- |
| Alice | HR |
| Bob | IT |
| NULL | Sales |

### ****FULL OUTER JOIN**** — all rows from both tables

(Works in SQL Server/PostgreSQL, not directly in MySQL)

SELECT emp\_name, dept\_name

FROM employees

FULL OUTER JOIN departments

ON employees.dept\_id = departments.dept\_id;

**Result:**

| **emp\_name** | **dept\_name** |
| --- | --- |
| Alice | HR |
| Bob | IT |
| Charlie | NULL |
| NULL | Sales |

**12. SQL Group By**

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

The **GROUP BY clause** in SQL is used to **group rows that have the same values in one or more columns** into **summary rows**, often to perform **aggregate calculations** like total, average, count, etc.

It is commonly used with **aggregate functions** such as COUNT(), SUM(), AVG(), MAX(), and MIN() to **compute results for each group**.

**Syntax:**

SELECT column1, column2, AGGREGATE\_FUNCTION(column3)

FROM table\_name

WHERE condition

GROUP BY column1, column2;

* column1, column2 → the columns you want to group by.
* AGGREGATE\_FUNCTION → functions like SUM(), COUNT(), AVG(), MAX(), MIN().
* WHERE clause is optional (to filter rows before grouping).

**How it works with aggregate functions**

1. **COUNT()** – Counts rows in each group.
2. **SUM()** – Adds up numeric values in each group.
3. **AVG()** – Finds the average of numeric values in each group.
4. **MAX() / MIN()** – Finds the maximum or minimum value in each group.
5. **Example 1: Count of employees in each department**
6. Suppose we have a table employees:

| **emp\_id** | **name** | **department** | **salary** |
| --- | --- | --- | --- |
| 1 | Alice | HR | 5000 |
| 2 | Bob | IT | 6000 |
| 3 | Charlie | HR | 5500 |
| 4 | David | IT | 6500 |
| 5 | Eve | Finance | 7000 |

1. Query:
2. SELECT department, COUNT(\*) AS num\_employees
3. FROM employees
4. GROUP BY department;
5. **Result:**

| **department** | **num\_employees** |
| --- | --- |
| HR | 2 |
| IT | 2 |
| Finance | 1 |

1. **Example 2: Total salary per department**
2. SELECT department, SUM(salary) AS total\_salary
3. FROM employees
4. GROUP BY department;
5. **Result:**

| **department** | **total\_salary** |
| --- | --- |
| HR | 10500 |
| IT | 12500 |
| Finance | 7000 |

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

| **Feature** | **GROUP BY** | **ORDER BY** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| **Purpose** | Groups rows that have the same values in one or more columns, usually for aggregate calculations. | Sorts the result set in ascending (ASC) or descending (DESC) order. |

|  |  |  |
| --- | --- | --- |
| **Focus** | Aggregation (combining rows) | Presentation (sorting rows) |

* **GROUP BY** collects rows into **groups** based on the column(s) you specify. Each group can then be summarized using aggregate functions like SUM, COUNT, AVG, etc.
* **ORDER BY** does **not group rows**. It simply **sorts the rows** of the result set according to one or more columns.

SELECT department, SUM(salary) AS total\_salary

FROM employees

GROUP BY department

ORDER BY total\_salary DESC;

GROUP BY department → groups employees by department.

SUM(salary) → calculates total salary per department.

ORDER BY total\_salary DESC → sorts the grouped results from highest to lowest total salary

1. **GROUP BY** changes the **structure** of the data (aggregates multiple rows into one per group).
2. **ORDER BY** changes only the **order** of rows; it does not change the data itself.
3. GROUP BY is often used **with aggregate functions**, while ORDER BY is optional and can be applied to any query.

**13. SQL Stored Procedure**

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

**Stored Procedure:**  
A named set of SQL statements (including SELECT, INSERT, UPDATE, DELETE, and control-flow statements like IF, WHILE, or loops) that can accept **parameters**, perform operations, and return results.

**Standard SQL Query:**  
A single SQL statement executed directly against the database, typically for a one-time operation like retrieving or updating data.

**Key Differences**

| **Feature** | **Stored Procedure** | **Standard SQL Query** |
| --- | --- | --- |
| **Definition** | Precompiled block of SQL statements | Single SQL statement |
| **Reusability** | Can be reused multiple times | Typically used once per execution |
| **Parameters** | Can accept input/output parameters | Cannot accept parameters directly |
| **Complexity** | Can include multiple SQL statements, loops, conditionals | Usually one statement only |
| **Performance** | Precompiled → faster for repeated execution | Compiled at runtime each time |
| **Security** | Can restrict direct access to underlying tables | Direct access needed |
| **Use Cases** | Business logic, batch operations, reporting | Simple data retrieval or modification |

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

Using **stored procedures** in SQL offers several advantages over writing individual SQL queries.

**1. Performance Improvement**

* Stored procedures are **precompiled** and stored in the database.
* The database engine **doesn’t need to parse and compile** the SQL statements every time they are executed, making them **faster for repeated operations**.

**2. Reusability**

* Once created, a stored procedure can be **called multiple times** by different applications or users.
* Promotes **modular programming** and reduces code duplication.

**3. Maintainability**

* Business logic is **centralized in the database**.
* Any change to the logic requires modifying the stored procedure **once**, rather than changing SQL queries in multiple applications.

**4. Security**

* Stored procedures can **restrict direct access to tables**.
* Users can be granted permission to **execute a procedure without giving direct access** to underlying tables.
* Helps prevent **SQL injection attacks** by using parameters instead of dynamically building SQL queries.

**5. Reduced Network Traffic**

* Multiple SQL statements can be executed in a **single call** to the stored procedure.
* This reduces the **number of round-trips** between application and database, especially useful in client-server applications.

**6. Parameterization**

* Stored procedures can accept **input, output, and input/output parameters**.
* This allows **dynamic behavior** without changing the SQL inside the procedure.

**7. Consistency**

* Ensures that **critical business rules** or calculations are applied consistently every time.
* Reduces the risk of errors caused by inconsistent queries in different parts of an application.

**14. SQL View**

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

**View:** A named SQL query stored in the database that behaves like a table. It can **simplify complex queries**, **hide sensitive columns**, and **present data in a customized way**.

**Table:** A physical storage structure in the database that **holds actual data** in rows and columns.

**Key Differences**

| **Feature** | **View** | **Table** |
| --- | --- | --- |
| **Data Storage** | Virtual (no physical storage, except in materialized views) | Physical (stores actual data) |
| **Creation** | Created using CREATE VIEW AS SELECT ... | Created using CREATE TABLE ... |
| **Data Manipulation** | Can be read-only or updatable (depending on complexity) | Fully updatable |
| **Purpose** | Simplify queries, hide columns, present data differently | Store raw data |
| **Performance** | Depends on underlying tables; may be slower for complex views | Optimized for storage and retrieval |
| **Dependency** | Depends on one or more tables | Independent (basic unit of storage) |

**Syntax Example**

**Creating a view to show employee names and salaries in IT department:**

CREATE VIEW IT\_Employees AS

SELECT name, salary

FROM employees

WHERE department = 'IT';

**Querying the view:**

SELECT \* FROM IT\_Employees;

This will display:

| **name** | **salary** |
| --- | --- |
| Bob | 6000 |
| David | 6500 |

**Advantages of Views**

1. **Simplifies complex queries** – you can reuse a view instead of writing long queries repeatedly.
2. **Provides security** – hide sensitive columns from users.
3. **Logical data abstraction** – present data differently from the way it is stored.
4. **Consistency** – ensure that the same business logic or calculations are used every time.

**Simplify Complex Queries**

* Views can **encapsulate complex SQL queries** involving joins, aggregations, or subqueries.
* Users can query the view as if it were a simple table, **avoiding repetition of complex SQL**.

**Example:**

CREATE VIEW IT\_Employees AS

SELECT name, salary

FROM employees

WHERE department = 'IT';

Now users can simply do:

SELECT \* FROM IT\_Employees;

**Enhance Security**

* Views can **restrict access to sensitive columns or rows**.
* Users can be granted permission to access the view without giving direct access to the underlying tables.

**Example:**  
Hide salary column for most users:

CREATE VIEW Employee\_Names AS

SELECT name, department

FROM employees;

**Data Abstraction / Logical Representation**

* Views provide a **logical representation of data** that may differ from how it is physically stored.
* This allows developers to **change the underlying table structure** without affecting applications that use the view.

**Maintain Consistency**

* Using views ensures **consistent calculations or business logic** across multiple queries.
* For example, a view can always calculate total sales per department using the same formula.

**Reusability**

* Once created, a view can be **used in multiple queries**, reducing repetition of SQL code and minimizing errors.

**Aggregation and Summarization**

* Views can provide **pre-aggregated data** to simplify reporting.
* Users can query summaries directly without having to write GROUP BY queries repeatedly.

**Join and Union Simplification**

* Views can **combine data from multiple tables** into a single “virtual table” for easier querying.

**Example:**

CREATE VIEW Employee\_Department AS

SELECT e.name, e.salary, d.department\_name

FROM employees e

JOIN departments d ON e.department\_id = d.department\_id;

Now querying employee info along with department becomes simple:

SELECT \* FROM Employee\_Department;

**15. SQL Triggers**

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

A **trigger** is a set of SQL statements that are automatically executed when a particular event occurs in the database.

**Syntax Example:**

CREATE TRIGGER trigger\_name

BEFORE INSERT ON table\_name

FOR EACH ROW

BEGIN

-- Trigger logic here

END;

### ****Types of Triggers in SQL****

#### 1. ****BEFORE Trigger****

* Executes **before** an INSERT, UPDATE, or DELETE operation.
* Used to **validate or modify data** before it’s written to the database.

**Example:**

CREATE TRIGGER before\_insert\_member

BEFORE INSERT ON members

FOR EACH ROW

BEGIN

IF NEW.date\_of\_membership < '2020-01-01' THEN

SET NEW.date\_of\_membership = CURDATE();

END IF;

END;

#### 2. ****AFTER Trigger****

* Executes **after** an INSERT, UPDATE, or DELETE operation.
* Commonly used for **logging**, **auditing**, or **updating related tables**.

**Example:**

CREATE TRIGGER after\_book\_insert

AFTER INSERT ON books

FOR EACH ROW

BEGIN

INSERT INTO audit\_log(action, book\_id, timestamp)

VALUES ('Book Added', NEW.book\_id, NOW());

END;

#### 3. ****INSTEAD OF Trigger**** (Mainly in SQL Server and Oracle for Views)

* Executes **in place of** an INSERT, UPDATE, or DELETE on a **view**.
* Used when you want to perform custom logic for modifying data in underlying base tables.

**Example (SQL Server style):**

CREATE TRIGGER update\_view\_instead

ON book\_view

INSTEAD OF UPDATE

AS

BEGIN

UPDATE books

SET title = inserted.title

FROM inserted

WHERE books.book\_id = inserted.book\_id;

END;

### ****When Triggers Are Used****

1. **Automatic auditing/logging** (e.g., tracking who modified a record and when).
2. **Enforcing complex business rules** (e.g., preventing price drop below a minimum).
3. **Maintaining derived or summary data** (e.g., updating totals in another table).
4. **Synchronizing tables** (e.g., cascading changes to related tables).
5. **Validating input data** before insertion or modification.

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

| **Trigger Type** | **When It Executes** | **Available References** | **Common Uses** |
| --- | --- | --- | --- |

|  |  |  |  |
| --- | --- | --- | --- |
| **INSERT** | When a new row is added | NEW (new values only) | Logging new data, setting defaults |

|  |  |  |  |
| --- | --- | --- | --- |
| **UPDATE** | When an existing row is modified | OLD (before values), NEW (after values) | Enforcing rules, auditing changes |

|  |  |  |  |
| --- | --- | --- | --- |
| **DELETE** | When a row is removed | OLD (deleted row values) | Logging deletions, cascading cleanup |

* **INSERT trigger** → reacts to **new data added**.
* **UPDATE trigger** → reacts to **data changes**.
* **DELETE trigger** → reacts to **data removal**.

**16. Introduction to PL/SQL**

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

**PL/SQL (Procedural Language/Structured Query Language)** is **Oracle’s procedural extension of SQL**.  
It allows you to **combine SQL statements with procedural programming constructs** such as **loops, conditions, variables, and error handling**.

**SQL** is a declarative language (you tell what you want to do),  
while **PL/SQL** adds procedural features (you tell how to do it).

Standard SQL can only perform **one statement at a time** and lacks programming features like loops, conditions, and variables.  
PL/SQL extends SQL by allowing developers to write **complex business logic** inside the database itself.

**Key Features of PL/SQL**

| **Feature** | **Description** |
| --- | --- |
| **Block structure** | Code is organized into blocks (DECLARE, BEGIN, EXCEPTION, END). |
| **Variables & Constants** | Allows declaration and use of variables for intermediate processing. |
| **Control structures** | Supports loops (FOR, WHILE), conditional statements (IF-THEN-ELSE). |
| **Exception handling** | Handles runtime errors gracefully. |
| **Cursors** | Allows row-by-row data processing from SQL queries. |
| **Procedures & Functions** | Enables code reusability and modular design. |
| **Triggers & Packages** | Automates actions and groups related logic. |

**How PL/SQL Extends SQL’s Capabilities**

| **Feature** | **SQL** | **PL/SQL** |
| --- | --- | --- |
| **Programming logic** | ❌ Not supported | ✅ Supports conditions, loops, variables |
| **Error handling** | ❌ Limited | ✅ Advanced exception handling |
| **Procedures/Functions** | ❌ No | ✅ Yes, allows modular programming |
| **Performance** | Executes one statement at a time | Executes multiple statements as one block (faster) |
| **Triggers** | ❌ Not available | ✅ Can define triggers for automation |

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

Benefits of Using PL/SQL

**1. Improved Performance**

* **PL/SQL reduces the number of database calls.**  
  Multiple SQL statements can be sent to the database as a **single block**, minimizing communication between the application and the database server.
* This reduces **network traffic** and **execution time**.

**Example:**  
Instead of executing 10 separate INSERT statements from an application, a single PL/SQL block can insert all records at once.

**2. High Productivity**

* PL/SQL combines **SQL** (for data manipulation) with **procedural constructs** like loops, conditions, and variables.
* This makes it easier to write **complex business logic** directly inside the database.

**Example:**  
You can use FOR loops and IF conditions to process multiple rows or apply conditional updates — tasks that SQL alone cannot handle easily.

**3. Better Maintainability and Reusability**

* PL/SQL supports **modular programming** using:
  + **Procedures**
  + **Functions**
  + **Packages**
* These can be stored in the database and reused by multiple applications or users, making maintenance easier.

**Example:**  
A calculate\_discount() function can be reused in many parts of the system without rewriting the logic each time.

**4. Enhanced Security**

* Business logic written in PL/SQL is **stored and executed on the database server**, not on the client side.
* This keeps sensitive data and rules **protected from unauthorized access**.
* You can also **control privileges** at the procedure or package level.

**5. Robust Error Handling**

* PL/SQL provides a built-in **exception handling mechanism** using the EXCEPTION block.
* Developers can define **custom error responses**, ensuring that programs fail gracefully instead of crashing.

**Example:**

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

**6. Tight Integration with SQL**

* PL/SQL is **fully compatible with SQL** — you can use all SQL commands directly within PL/SQL blocks.
* It allows both **DML operations (INSERT, UPDATE, DELETE)** and **DDL operations (CREATE, DROP)** inside procedures.

**7. Portability**

* PL/SQL code is **portable** across all Oracle environments (like SQL\*Plus, Oracle Forms, or Oracle Reports).
* You can run the same PL/SQL program in different applications without modification.

**8. Supports Triggers and Automation**

* PL/SQL enables **automatic execution** of code using **triggers** in response to database events (INSERT, UPDATE, DELETE).
* This helps enforce business rules and maintain data integrity automatically.

**17. PL/SQL Control Structures**

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

**Control structures** in PL/SQL are statements that **control the flow of execution** of a program.  
They determine **how and when** different parts of the PL/SQL block are executed — allowing you to make decisions, repeat actions, and manage logic dynamically.

In simple terms, control structures help PL/SQL behave like a real programming language (similar to Python, C, or Java).

### ****Types of Control Structures in PL/SQL****

There are **three main types**:

1. **Conditional Control** → IF-THEN, IF-THEN-ELSE, IF-THEN-ELSIF
2. **Iterative Control** → LOOP, WHILE LOOP, FOR LOOP
3. **Sequential Control** → GOTO, NULL statements

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

Used for **conditional execution** — to execute certain statements **only if a condition is true**.

### ✅ ****Syntax:****

IF condition THEN

statements;

END IF;

### ✅ ****Example:****

DECLARE

v\_salary NUMBER := 40000;

BEGIN

IF v\_salary < 50000 THEN

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

END IF;

END;

**Explanation:**

* The block checks the condition v\_salary < 50000.
* If it’s true, the message “Salary is below average” is displayed.
* If it’s false, nothing happens.

### ✅ ****Variants of IF:****

#### a) ****IF-THEN-ELSE****

Used when you want to execute one block if a condition is true, and another if it’s false.

IF condition THEN

statements\_if\_true;

ELSE

statements\_if\_false;

END IF;

**Example:**

IF v\_salary > 50000 THEN

DBMS\_OUTPUT.PUT\_LINE('High Salary');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Low Salary');

END IF;

#### b) ****IF-THEN-ELSIF****

Used when there are **multiple conditions**.

IF condition1 THEN

statements1;

ELSIF condition2 THEN

statements2;

ELSE

statements3;

END IF;

**Example:**

IF v\_salary > 70000 THEN

DBMS\_OUTPUT.PUT\_LINE('Excellent Salary');

ELSIF v\_salary BETWEEN 50000 AND 70000 THEN

DBMS\_OUTPUT.PUT\_LINE('Good Salary');

ELSE

DBMS\_OUTPUT.PUT\_LINE('Needs Improvement');

END IF;

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

Used to **repeat** a set of statements **multiple times**.  
You can exit from a loop using the **EXIT** or **EXIT WHEN** statement.

### ****Syntax:****

LOOP

statements;

EXIT WHEN condition;

END LOOP;

### ****Example:****

DECLARE

v\_counter NUMBER := 1;

BEGIN

LOOP

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

v\_counter := v\_counter + 1;

EXIT WHEN v\_counter > 5; -- Stops loop after 5 iterations

END LOOP;

END;

**Explanation:**

* The loop prints the value of v\_counter five times.
* When v\_counter > 5, the loop **exits**.

| **Control Structure** | **Purpose** | **Example Use Case** |
| --- | --- | --- |
| **IF-THEN** | Executes statements when condition is true | Checking if a discount applies |
| **IF-THEN-ELSE** | Executes one of two blocks | Approving or rejecting a request |
| **IF-THEN-ELSIF** | Handles multiple conditions | Classifying salary ranges |
| **LOOP** | Repeats statements until explicitly exited | Displaying or processing records repeatedly |

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

Control structures in PL/SQL — such as **IF-THEN**, **LOOP**, and **WHILE** — make SQL far more powerful and flexible by allowing **decision-making**, **iteration**, and **conditional logic** within the database.

They transform simple SQL queries into **dynamic, intelligent programs** that can handle complex business logic directly inside the database.

**1. They Add Decision-Making Capabilities**

* Using **IF-THEN-ELSE**, PL/SQL can execute different SQL statements based on specific conditions.
* This allows you to apply **business rules** that plain SQL alone cannot handle.

**Example:**

IF v\_salary > 70000 THEN

UPDATE employees SET bonus = 10000 WHERE emp\_id = v\_emp\_id;

ELSE

UPDATE employees SET bonus = 5000 WHERE emp\_id = v\_emp\_id;

END IF;

This logic updates bonuses differently based on salary — something not possible in a single SQL statement without PL/SQL.

**2. They Enable Repetitive Processing (Iteration)**

* Control structures like **LOOP**, **FOR LOOP**, and **WHILE LOOP** let you **execute SQL statements repeatedly**.
* Useful when you need to process multiple rows, records, or tasks in sequence.

**Example:**

FOR rec IN (SELECT emp\_id, salary FROM employees) LOOP

UPDATE employees

SET bonus = salary \* 0.10

WHERE emp\_id = rec.emp\_id;

END LOOP;

Here, PL/SQL iterates through each employee record and updates the bonus — something not achievable with a single SQL query.

**3. They Allow Conditional Query Execution**

* You can **control when and how** a query runs.
* For example, only run an INSERT if a certain condition is met.

**Example:**

IF NOT EXISTS (SELECT \* FROM members WHERE email = 'test@example.com') THEN

INSERT INTO members(member\_name, email) VALUES('Test User', 'test@example.com');

END IF;

This prevents duplicate data by checking conditions before inserting.

**4. They Support Error Handling**

* Control structures combined with **EXCEPTION handling** make complex queries more reliable.
* Errors like “no data found” or “divide by zero” can be handled gracefully without breaking the program.

**Example:**

BEGIN

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

DBMS\_OUTPUT.PUT\_LINE('Salary: ' || v\_salary);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

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

END;

This ensures the query continues running even if an error occurs.

**5. They Help Implement Complex Business Logic**

* PL/SQL allows combining multiple SQL operations (INSERT, UPDATE, DELETE) with decision-making and looping — forming a **complete business transaction**.

**Example:**  
A payroll system can:

1. Calculate salary based on attendance
2. Apply bonuses based on performance
3. Insert a record in the payment history table

All within a single **PL/SQL block**.

**6. They Improve Code Organization and Reusability**

* You can encapsulate complex logic into **procedures**, **functions**, and **packages**.
* This allows reuse of code and makes maintenance easier.

**18. SQL Cursors:**

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

A **cursor** in PL/SQL is a **pointer or handle** that allows you to **process query results row by row**.

When a SQL query (like SELECT) is executed, Oracle temporarily stores the result set in memory.  
A **cursor** acts as a **work area** to access and manipulate each row from that result set.

**Types of Cursors in PL/SQL**

There are **two main types** of cursors:

1. **Implicit Cursor** → Automatically created by Oracle for simple SQL statements.
2. **Explicit Cursor** → Defined manually by the programmer for complex or multi-row queries.

## ****1. Implicit Cursor****

An **implicit cursor** is **automatically created by Oracle** whenever you execute a SQL statement (like INSERT, UPDATE, DELETE, or a single-row SELECT INTO).

## ****2. Explicit Cursor****

An **explicit cursor** is **manually declared and controlled** by the programmer to process **multiple rows** returned by a query.

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

**1. When the Query Returns More Than One Row**

* An **implicit cursor** works automatically for queries that return **a single row**.
* But if your SELECT query returns **multiple rows**, you **must** use an **explicit cursor** — otherwise, Oracle raises the error:

**2. When You Need to Process Each Row Individually**

* Use an explicit cursor when you must **perform an action for every row** — such as displaying data, performing calculations, or inserting into another table.

**3. When You Need Fine-Grained Control Over the Query**

With explicit cursors, you have full control over:

* When the query executes (OPEN)
* How many rows are fetched at a time (FETCH)
* When the cursor closes (CLOSE)

This flexibility is useful for:

* Complex logic
* Conditional fetching
* Nested cursors

**4. When You Want to Use Cursor Attributes for Tracking**

Explicit cursors allow you to use cursor-specific attributes like:

* %ROWCOUNT → number of rows processed
* %FOUND, %NOTFOUND, %ISOPEN → control flow based on cursor state

**5. When You Need to Reuse the Query Multiple Times**

* You can open and close an explicit cursor multiple times in a program to **re-run the same query** with different conditions — which is not possible with implicit cursors.

**19. Rollback and Commit Savepoint**

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

A **SAVEPOINT** in SQL (and PL/SQL) is a **marker or checkpoint** within a transaction that allows you to **partially roll back** the transaction to a specific point **without undoing the entire transaction**.

It helps you control and manage complex transactions **more safely and flexibly**.

**Syntax**

SAVEPOINT savepoint\_name;

## ****How SAVEPOINT Works in Transaction Control****

In transaction management, the main commands are:

| **Command** | **Purpose** |
| --- | --- |
| COMMIT | Makes all changes permanent |
| ROLLBACK | Undoes all uncommitted changes |
| SAVEPOINT | Creates a checkpoint within a transaction |
| ROLLBACK TO savepoint | Undoes changes made after a specific savepoint |

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

**SAVEPOINTS** are useful whenever you’re performing **a long or complex transaction** involving **multiple steps or operations**, and you want the ability to **undo part of it** without losing all your work.

## ****Common Situations Where SAVEPOINTS Are Useful****

### ****1. Complex Transactions with Multiple Logical Steps****

When a single transaction includes several stages (e.g., inserting into multiple related tables), you can use savepoints to revert only a specific stage if it fails.

**Example:**

BEGIN;

INSERT INTO customers VALUES (...);

SAVEPOINT after\_customer;

INSERT INTO orders VALUES (...);

SAVEPOINT after\_order;

INSERT INTO payments VALUES (...);

-- Payment fails

ROLLBACK TO after\_order; -- undo payment only

COMMIT;

✅ Customer and order remain saved, payment is undone.

### ****2. Partial Error Recovery****

When you expect some operations might fail or violate constraints, savepoints let you safely recover **without discarding the entire transaction**.

**Example:**  
If inserting hundreds of rows, and one insert violates a unique constraint —  
you can rollback to a previous savepoint and continue processing others.

### ****3. Maintaining Data Consistency in Conditional Logic****

When performing a set of operations that depend on conditions (like verifying inventory or balances), savepoints help preserve consistent data even if conditions fail midway.

**Example:**

SAVEPOINT check\_inventory;

IF stock < required THEN

ROLLBACK TO check\_inventory;

ELSE

UPDATE stock SET quantity = quantity - required;

END IF;

### ****4. Testing or Debugging Transactions****

While developing or testing, you can use savepoints to experiment with parts of a transaction without committing everything each time.

### ****5. Nested or Hierarchical Operations****

If a transaction includes **sub-operations** (e.g., processing multiple accounts or departments), you can assign savepoints to each sub-operation and rollback only that section if needed.

## ****Benefits of Using SAVEPOINTS****

| **Benefit** | **Explanation** |
| --- | --- |
| **Fine-grained control** | Roll back only a specific portion of a transaction |
| **Error recovery** | Handle errors gracefully without losing all progress |
| **Data integrity** | Protects consistent data state even during partial failures |
| **Improved flexibility** | Allows partial success of a long transaction |
| **Useful in loops or batch operations** | Each batch can have its own savepoint |