Module 4 – Introduction to DBMS

THEORY

Introduction to SQL

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

SQL (Structured Query Language) is a standard programming language used to store, manage, and retrieve data from relational databases.

Why SQL is essential:

It allows you to **create** and **manage** database structures (tables, views, schemas).

It lets you **insert**, **update**, **delete**, and **retrieve** data efficiently.

It provides a powerful way to **query and analyze data** using conditions, grouping, sorting, and joins.

It is used by all major relational database systems like MySQL, PostgreSQL, Oracle, SQL Server, and SQLite.

SQL ensures **data integrity**, **security**, and supports **transactions**.

2. Explain the difference between DBMS and RDBMS.

Difference between DBMS and RDBMS

Feature	**DBMS (Database Management System)**	**RDBMS (Relational Database Management System)**
Data Model	Stores data as files or hierarchical/network structures	Stores data in **tables (rows & columns)**

Feature	DBMS (Database Management System)	RDBMS (Relational Database Management System)
Relationships	No relationships between data	Supports **relationships using foreign keys**
Data Integrity	Limited	High—enforced using **constraints** (PK, FK, Unique)
Normalization	Not required	Normalization is essential to avoid redundancy
Complex Queries	Limited query capability	Powerful querying using SQL
Examples	File system, XML DB	MySQL, PostgreSQL, Oracle, SQL Server

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

SQL plays multiple roles in relational databases:

Data Definition (DDL)

Used to **define structures**:

```
CREATE TABLE
```

```
ALTER TABLE
```

```
DROP TABLE
```

Data Manipulation (DML)

Used to **insert, modify, delete** data:

```
INSERT
```

```
UPDATE
```

```
DELETE
```

Data Querying (DQL)

Used to **retrieve data**:

```
SELECT
```

Data Control (DCL)

Used to control **access & permissions**:

```
GRANT
REVOKE
```

Transaction Control (TCL)

Used to maintain **data integrity**:

```
COMMIT
ROLLBACK
```

Overall role:

SQL acts as a **bridge** that allows users/applications to communicate with relational databases to manage data efficiently and securely.

4. What are the key features of SQL?

Easy to Learn & Use – uses simple English-like syntax.

Standardized Language – accepted globally for relational databases.

Data Querying – powerful querying using filters, joins, grouping, etc.

Data Manipulation – add, update, delete data.

Data Definition – create and modify database structures.

Data Integrity – constraints ensure clean and correct data.

Transaction Control – ensures reliable and consistent data changes.

Multi-user Support – handles many users simultaneously.

Security Features – permissions, roles, access control.

2. SQL Syntax

1.What are the basic components of SQL syntax?

SQL syntax is built using the following basic components:

Keywords

Reserved words used to perform actions
Examples: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `WHERE`, `FROM`, `JOIN`.

Identifiers

Names given to database objects
Examples: table names, column names, database names.

Operators

Used for comparisons and calculations
Examples:

Comparison: `=`, `>`, `<`, `>=`, `<=`, `<>`

Logical: `AND`, `OR`, `NOT`

Arithmetic: `+`, `-`, `*`, `/`

Clauses

Parts of SQL statements that perform specific tasks
Examples: `WHERE`, `ORDER BY`, `GROUP BY`, `HAVING`.

Expressions

Formulas or conditions that return values
Example: `salary * 2`,`price > 100`.

Statements

Complete SQL instructions
Examples:

`SELECT` (query data)

`INSERT` (add data)

`UPDATE` (modify data)

`DELETE` (remove data)

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

```
SELECT column1, column2, ...

FROM table_name

WHERE condition

GROUP BY column

HAVING condition

ORDER BY column ASC|DESC;
```

Explanation of order:

SELECT → What you want to retrieve

FROM → Table(s) where data exists

WHERE → Filter rows

GROUP BY → Group rows

HAVING → Filter groups

ORDER BY → Sort results

3. Explain the role of clauses in SQL statements

Clauses are the building blocks of SQL statements.

Each clause performs a specific function to define how data should be handled.

Common clauses and their roles:

FROM Clause

Specifies the table(s) to retrieve data from.

WHERE Clause

Filters rows before grouping — used to apply conditions.
Example: `WHERE age > 18`

GROUP BY Clause

Groups rows that have the same values in specified columns.
Example: `GROUP BY department`

HAVING Clause

Filters groups created by `GROUP BY`.
Example: `HAVING COUNT(*) > 5`

ORDER BY Clause

Sorts the output in ascending or descending order.
Example: `ORDER BY salary DESC`

LIMIT / OFFSET

Limits the number of rows returned (supported in MySQL, PostgreSQL)

3. SQL Constraints

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

Constraints are rules applied to table columns to ensure the **accuracy**, **validity**, and **integrity** of data in a database.

They prevent invalid data from being inserted and maintain relational consistency.

Types of SQL Constraints

1. NOT NULL

Ensures a column **cannot have NULL values**.

Every row must contain a valid value for that column.

2. UNIQUE

Ensures that **all values in a column are different**.

Allows **one NULL** (varies by DBMS).

3. PRIMARY KEY

Uniquely identifies each row in a table.

Combines **NOT NULL + UNIQUE**.

A table can have **only one primary key**, but it can be **composite** (multiple columns).

4. FOREIGN KEY

Enforces a **relationship** between two tables.

Ensures values in a column **match existing values** in the referenced (parent) table.

Maintains **referential integrity**.

5. CHECK

Ensures values satisfy a specific condition.

Example: `age >= 18`

6. DEFAULT

Sets a default value when no value is provided.

Example: `status DEFAULT 'active'`

7. INDEX (not exactly a constraint, but related)

Speeds up searching and querying.

Does not enforce rules, but often categorized with constraints.

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

Feature	**PRIMARY KEY**	**FOREIGN KEY**
Purpose	Uniquely identifies each row in the table	Creates a link between two tables
Uniqueness	Must be **unique**	Can have **duplicate** values
NULL allowed?	**Not allowed**	Allowed depending on requirements
Location	Defined in the **child table** itself	References a primary key of the **parent table**
Table count	Only **one** per table	A table can have **many** foreign

Feature	**PRIMARY KEY**	**FOREIGN KEY**
		keys

Summary:

Primary Key = identity of a record

Foreign Key = connection to another table's primary key

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

NOT NULL Constraint

Ensures that a column **must always have a value**.

Prevents storing incomplete data.

Example:

```
name VARCHAR(50) NOT NULL
```

UNIQUE Constraint

Ensures that **no two rows** have the same value in that column.

Helps maintain **uniqueness** (e.g., email, username).

Example:

```
email VARCHAR(100) UNIQUE
```

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

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

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

DDL commands affect the **database schema** and are automatically **committed** (permanent changes).

Common DDL commands:

`CREATE` – create database objects

`ALTER` – modify existing objects

`DROP` – delete database objects

`TRUNCATE` – remove all rows from a table (structure remains)

`RENAME` – rename a table or object

2. Explain the CREATE command and its syntax.

The **CREATE** command is used to **create new database objects**, most commonly **tables**.

Syntax for creating a table:

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

Explanation:

table_name → Name of the table you want to create.

column1, column2 → Columns (fields) in the table.

datatype → Type of data the column will store (e.g., INT, VARCHAR, DATE).

constraint → Rules applied to ensure data integrity (e.g., NOT NULL, PRIMARY KEY).

Example:

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

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

Purpose of Data Types

Data types define:

The **kind of data** stored (number, text, date, etc.)

The **size** of data allowed

How much **storage space** is used

Allowed **operations** (e.g., math on numbers only)

Example:
`INT`, `VARCHAR(50)`, `DATE`, `FLOAT`

Benefits:

Prevents incorrect data (e.g., text in an age column)

Improves performance and storage efficiency

Purpose of Constraints

Constraints enforce **rules** to maintain data integrity and accuracy.

Examples:

`PRIMARY KEY` → unique row identity

`NOT NULL` → value must be provided

`UNIQUE` → no duplicate values

`FOREIGN KEY` → maintains relationships

`CHECK` → ensures logical conditions

`DEFAULT` → automatically assigns values

Benefits:

Prevents duplicate or invalid data

Maintains relationships between tables

Ensures consistency and reliability

5. ALTER Command

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

The **ALTER** command in SQL is used to **modify the structure** of an existing table *without deleting or recreating it*.

Uses of ALTER:

Add new columns

Modify existing columns (data type, size, constraints)

Rename columns

Rename the table

Add or drop constraints

Delete (drop) columns

ALTER belongs to the **DDL (Data Definition Language)** category because it changes the table schema.

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

A. Add a Column

Syntax:

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

Example:

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

B. Modify/Change a Column

Syntax (MySQL / PostgreSQL):

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

Example:

```
ALTER TABLE Students
MODIFY age INT;
```

SQL Server Syntax (using ALTER COLUMN):

```
ALTER TABLE table_name
ALTER COLUMN column_name new_datatype;
```

C. Rename a Column

MySQL / PostgreSQL:

```
ALTER TABLE table_name
RENAME COLUMN old_name TO new_name;
```

Example:

```
ALTER TABLE Students
RENAME COLUMN address TO home_address;
```

D. Drop (Delete) a Column

Syntax:

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

Example:

```
ALTER TABLE Students
DROP COLUMN home_address;
```

Summary Table

Operation	**Syntax**
Add a column	`ALTER TABLE ... ADD column datatype`
Modify a column	`ALTER TABLE ... MODIFY column datatype`
Rename a column	`ALTER TABLE ... RENAME COLUMN old TO new`

Operation	Syntax
Drop a column	`ALTER TABLE ... DROP COLUMN column`

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:

Tables

Databases

Views

Indexes

Stored procedures

Most commonly:

`DROP TABLE` is used to **delete an entire table**, including:

All rows (data)

Table structure (columns, constraints)

Indexes related to that table

This action **cannot be undone**, because DROP is a **DDL command** and is auto-committed.

Syntax:

```
DROP TABLE table_name;
```

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

Dropping a table has several serious and permanent consequences:

A. Permanent Loss of Data

All the data stored in the table is permanently deleted.

It cannot be recovered unless a backup exists.

B. Loss of Table Structure

The table design (columns, data types, constraints) is erased.

You cannot reference the table anymore.

C. Removal of Constraints & Relationships

All **Primary Keys**, **Foreign Keys**, **Unique constraints**, and **Indexes** on the table are deleted.

If other tables reference it via **FOREIGN KEY**, you will get an error unless:

You drop the foreign key first, or

Use `DROP TABLE ... CASCADE` (PostgreSQL/Oracle)

D. Breaks Referential Integrity

If the table is part of relationships, dropping it may break links between tables.

Example:

If `Orders` table references `Customers(customer_id)`

Dropping `Customers` without CASCADE → error

Dropping with CASCADE → all dependent constraints removed

E. Space is Freed

All storage used by the table (data + indexes) is released back to the system.

7. Data Manipulation Language (DML)

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

These three commands belong to **DML (Data Manipulation Language)** and are used to manipulate data stored in tables.

INSERT Command

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

Syntax:

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

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

Example:

```
INSERT INTO Students (id, name, age)

VALUES (1, 'Amit', 20);
```

UPDATE Command

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

Syntax:

```
UPDATE table_name

SET column = new_value

WHERE condition;
```

Example:

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

DELETE Command

The **DELETE** command is used to **remove rows** from a table.

Syntax:

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

Example:

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

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

The **WHERE clause** specifies **which rows** should be updated or deleted.

Without a WHERE clause:

`UPDATE` will modify **all rows** in the table.

`DELETE` will remove **all rows** from the table.

This can lead to **accidental loss of data**.

Importance of WHERE Clause:

1. Prevents unintentional changes

Only specific rows that match the condition will be updated or deleted.

2. Ensures data safety

Avoids updating or deleting the entire table unintentionally.

3. Helps maintain accuracy

Only relevant data is modified, keeping the rest intact.

4. Supports precise operations

You can target specific rows using conditions like:

Primary key

Range of values

Filters (e.g., age > 20)

Example showing danger without WHERE:

```
UPDATE Employees

SET salary = 0;           -- Updates ALL salaries → harmful!

DELETE FROM Employees;    -- Deletes ALL employees!
```

8. Data Query Language (DQL)

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

The **SELECT** statement is the most commonly used SQL command.
It is used to **retrieve (query) data** from one or more tables in a database.

Basic syntax:

SELECT `column1, column2, ...`

FROM `table_name;`

How SELECT is used:

To get specific columns

To get all columns using `*`

To filter rows using `WHERE`

To sort results using `ORDER BY`

To group data using `GROUP BY`

To combine tables using joins

Example:

SELECT `name, age`

FROM `Students;`

This retrieves the *name* and *age* columns from the *Students* table.

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

WHERE Clause

The **WHERE** clause is used to **filter rows** based on a condition.

Purpose:

Retrieve only the rows that meet specific criteria

Improve query accuracy and efficiency

Syntax:

SELECT *

```
FROM table_name
WHERE condition;
```

Example:

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

This returns only students **older than 18**.

ORDER BY Clause

The **ORDER BY** clause is used to **sort the result set**.

Purpose:

Sort data in **ascending (ASC)** or **descending (DESC)** order

Syntax:

```
SELECT *
FROM table_name
ORDER BY column_name ASC|DESC;
```

Example:

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

This sorts students by age from **highest to lowest**.

9. Data Control Language (DCL)

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

GRANT Command

The **GRANT** command is used to **give** specific privileges (permissions) to database users.

Purpose:

Allow a user to perform certain operations such as:

`SELECT`

`INSERT`

`UPDATE`

`DELETE`

`CREATE`

`DROP`

Example:

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

This gives *user1* the ability to read and insert data into the *Students* table.

REVOKE Command

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

Purpose:

Restrict access

Remove dangerous or unnecessary permissions

Example:

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

This removes *INSERT* permission from *user1*.

2. How do you manage privileges using these commands?

A. Granting Privileges

Syntax:

```
GRANT privilege1, privilege2
ON object_name
TO user_name;
```

Examples:

Grant SELECT to a user

```
GRANT SELECT ON Employees TO user2;
```

Grant all privileges

```
GRANT ALL PRIVILEGES ON Employees TO admin1;
```

B. Revoking Privileges

Syntax:

```
REVOKE privilege1, privilege2
ON object_name
FROM user_name;
```

Examples:

Remove UPDATE permission

```
REVOKE UPDATE ON Employees FROM user2;
```

Remove all permissions

```
REVOKE ALL PRIVILEGES ON Employees FROM admin1;
```

C. Managing User Access with GRANT and REVOKE

These commands allow you to:

1. Control what each user can do

Read data → `SELECT`

Insert data → `INSERT`

Modify data → `UPDATE`

Delete data → `DELETE`

2. Protect sensitive information

Only authorized users get access.

3. Prevent accidental or malicious changes

Restrict permissions for safety.

4. Provide custom access levels

For example:

Normal users → SELECT only

Data entry users → SELECT + INSERT

Admins → ALL PRIVILEGES

Summary Table

Command	**Purpose**
GRANT	Give privileges to users
REVOKE	Take away privileges from users

10. Transaction Control Language (TCL)

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

These commands belong to **TCL (Transaction Control Language)** and are used to manage transactions in a database.

COMMIT

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

Purpose:

Makes changes permanent

Releases locks on affected tables

Confirms DML operations like INSERT, UPDATE, DELETE

Example:

```
UPDATE Accounts SET balance = balance - 500 WHERE id = 1;
COMMIT;        -- Saves the changes permanently
```

ROLLBACK

The **ROLLBACK** command is used to **undo all changes** made during the current transaction.

Purpose:

Reverts data to the last committed state

Cancels unintentional or incorrect operations

Example:

```
DELETE FROM Students WHERE id = 10;
ROLLBACK;        -- Undoes the delete operation
```

2. Explain how transactions are managed in SQL databases

A **transaction** is a group of SQL operations executed as a single logical unit of work.

A transaction must follow the ACID properties:

A — Atomicity

A transaction is **all or nothing**.

If any part fails → entire transaction fails.

C — Consistency

Ensures the database moves from **one valid state to another valid state**.

Maintains rules, constraints, and relationships.

I — Isolation

Transactions should not affect each other while running simultaneously.

Prevents dirty reads, lost updates, etc.

D — Durability

After COMMIT, changes are **permanent**, even if the system crashes.

Transaction Management Steps

1. Start a Transaction

Automatically starts with the first DML command
(or manually with `BEGIN TRANSACTION` in some databases).

2. Perform Operations

Examples:

```
INSERT INTO Orders VALUES (...);

UPDATE Products SET stock = stock - 1;
```

3. Review the results

Check if everything is correct.

4. COMMIT or ROLLBACK

COMMIT → Save all changes

ROLLBACK → Undo all changes

Example of Transaction Management

```
BEGIN TRANSACTION;


UPDATE Accounts SET balance = balance - 1000 WHERE id = 1;

UPDATE Accounts SET balance = balance + 1000 WHERE id = 2;


IF no error:

    COMMIT;

ELSE:

    ROLLBACK;
```

This ensures money transfer is **safe and consistent**.

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?

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

Types of Joins:

1.INNER JOIN

Returns **only the matching rows** from both tables.

If there is no match, the row is **not included**.

Example result:

```
Table A ∩ Table B
```

2.LEFT JOIN (LEFT OUTER JOIN)

Returns **all rows from the left table**.

Matching rows from the right table are included.

If there's no match, NULL is shown for right table columns.

Example result:

```
All from LEFT + Matches from RIGHT
```

3. RIGHT JOIN (RIGHT OUTER JOIN)

Opposite of LEFT JOIN.

Returns **all rows from the right table**.

Matching rows from the left table are included.

If no match, NULL appears on the left side.

Example:

```
All from RIGHT + Matches from LEFT
```

4.FULL OUTER JOIN

Returns **all rows from both tables**.

If either side has no match, NULL is used.

Example:

```
All from A ∪ All from B (with NULLs where no match exists)
```

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

Joins combine data by matching values in columns from different tables.
For example, consider two tables:

Students

student_id	**name**
1	Alex
2	Ben

Marks

student_id	**subject**	**marks**
1	Math	90

student_id	subject	marks
1	Science	85

To combine them:

```
SELECT Students.name, Marks.subject, Marks.marks

FROM Students

INNER JOIN Marks

ON Students.student_id = Marks.student_id;
```

This gives:

name	subject	marks
Alex	Math	90
Alex	Science	85

Why we use joins?

To retrieve related information stored in separate tables.

To avoid redundancy (Normalization).

To perform complex queries on relational data.

12. SQL Group By

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

GROUP BY clause

The **GROUP BY** clause in SQL is used to group rows that have the same values in specified columns.
It is typically used **with aggregate functions** to perform calculations on each group.

Common aggregate functions:

`SUM()` – adds values

`COUNT()` – counts rows

`AVG()` – averages values

`MAX()` – maximum value

`MIN()` – minimum value

How GROUP BY works

You choose one or more columns to group by.
For each group, SQL applies aggregate functions.

Example

Suppose we have a table:

Sales

product	**amount**
Laptop	50000
Laptop	45000
Phone	20000
Phone	25000

Query:

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

FROM Sales

GROUP BY product;
```

Result:

product	**total_sales**
Laptop	95000
Phone	45000

SQL grouped rows by product and summed their amounts.

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

Feature	**GROUP BY**	**ORDER BY**
Purpose	Groups rows based on column values	Sorts rows based on column values
Used with	Aggregate functions (SUM, COUNT, etc.)	Any column or expression
Result	Returns one row per group	Returns all rows, sorted
Execution	Grouping happens before sorting	Sorting happens top of the final result

Example to show difference

```
SELECT department, COUNT(*) AS employees

FROM Employees

GROUP BY department

ORDER BY employees DESC;
```

GROUP BY creates groups (one per department).

ORDER BY sorts the groups based on the number of employees.

Simple explanation

GROUP BY = “Combine similar rows into groups.”

ORDER BY = “Sort the final output.”

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 **stored procedure** is a **precompiled set of SQL statements** that is stored and executed on the database server.

It can include:

SQL queries

Conditional logic (IF…ELSE)

Loops

Parameters (inputs/outputs)

You execute it using a single command:

```
EXEC procedure_name;
```

Difference from a Standard SQL Query

Standard SQL Query	**Stored Procedure**
A single SQL statement executed once.	A group of statements stored and executed together.
Written and run directly by the user at runtime.	Precompiled and saved in the database.
Cannot accept parameters (in most cases).	Can take input/output parameters.

Standard SQL Query	Stored Procedure
Used for simple data retrieval or updates.	Used for complex logic, repeated operations, automation.

Example Standard Query:

```
SELECT * FROM Customers WHERE country = 'India';
```

Example Stored Procedure:

```
CREATE PROCEDURE GetCustomersByCountry (@country VARCHAR(50))

AS

BEGIN

    SELECT * FROM Customers WHERE country = @country;

END;
```

2. Explain the advantages of using stored procedures.

1. Improved Performance

Stored procedures are **precompiled**, so the database does not recompile them every time.
This reduces execution time.

2. Better Security

Users can be given permission to **execute** a procedure without accessing underlying tables.

Protects data and structure.

3. Reusability

You can reuse the same procedure multiple times, avoiding repeated writing of code.

4. Reduced Network Traffic

Instead of sending multiple SQL queries from an application, you send **one call** (`EXEC ProcedureName`).

5. Easier Maintenance

Changes are made in one place (the procedure), not in multiple application files.

6. Supports Complex Logic

Stored procedures can include:

Loops

Conditions

Multiple SQL statements

Error handling

14. SQL View

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

View

A **view** is a **virtual table** in SQL.
It does **not store data physically**; instead, it displays data stored in real tables based on a query.

A view is created using:

```
CREATE VIEW view_name AS

SELECT ...

FROM ...

WHERE ...;
```

□ Difference Between View and Table

Feature	**Table**	**View**
Storage	Stores data physically	Does NOT store data; only stores query
Data source	Original data	Derived from tables
Updation	Can insert/update/delete	Usually read-only (some views can allow updates)
Performance	Faster because data is stored	Slower because query executes every time
Use-case	Main data storage	Security, abstraction, simplified queries

2. Explain the advantages of using views in SQL databases

1. Simplifies complex queries

If you frequently write a long SQL query, you can create a view to reduce complexity.

```
SELECT * FROM sales_report_view;
```

This hides the complex joins and calculations.

2. Provides security

Views let users see only specific columns or rows.

For example:

```
CREATE VIEW employee_public AS

SELECT name, position FROM employees;
```

Users accessing this view cannot see salaries or confidential data.

3. Ensures data consistency

If multiple queries use the same complex logic, a view ensures everyone gets **consistent results**.

4. Helps in abstraction

Views help hide the underlying table structure.
Even if tables change, the view names and structure can remain the same for applications.

5. Can improve performance (materialized views)

Though normal views don’t store data, **materialized views** store computed results and improve performance for large datasets.

(Note: These are supported in databases like Oracle, PostgreSQL, etc.)

6. Supports reusability

Once created, a view can be queried many times just like a table.

15. SQL Triggers

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

What is a Trigger?

A **trigger** is a special stored program in SQL that **automatically executes** when a specified event occurs on a table.

It is fired in response to:

`INSERT`

`UPDATE`

`DELETE`

Triggers help maintain data integrity, logging, validations, and automation.

✅Types of Triggers

1. BEFORE Trigger

Runs **before** an INSERT, UPDATE, or DELETE operation.

� **When used?**

To validate data before saving

To modify data before inserting/updating

Example: Prevent inserting negative salary values

2. AFTER Trigger

Runs **after** an INSERT, UPDATE, or DELETE operation.

� **When used?**

To create logs after data changes

To update related tables

Example: After a new order is placed, update the stock table

3. INSTEAD OF Trigger

Used on **views**, not tables.
Runs **instead of** the actual INSERT/UPDATE/DELETE operation.

� **When used?**

When a view is not directly updatable

To customize what happens when data is modified through a view

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

INSERT Trigger

Fires when a new row is inserted.

Used for:

Validating data before insert (BEFORE INSERT)

Creating audit logs (AFTER INSERT)

Updating related tables (AFTER INSERT)

Example:

When a new employee joins, insert a record into the `audit_log` table.

UPDATE Trigger

Fires when an existing row is modified.

Used for:

Tracking old vs new values

Preventing unauthorized changes

Updating last_modified timestamps

Example:

Save old salary and new salary in a history table.

DELETE Trigger

Fires when a row is deleted.

Used for:

Storing deleted data in an archive table (soft-delete)

Restricting delete operations (BEFORE DELETE)

Logging delete actions

Example:

Before deleting a customer, check if they have pending orders.

Summary Table

Trigger Type	**INSERT Trigger**	**UPDATE Trigger**	**DELETE Trigger**
When fires	On new row inserted	On row modified	On row removed
Used for	Logs, validation, defaults	Audit history, prevent changes	Archive, restrict deletion
Data available	NEW values	OLD and NEW values	OLD values

16. Introduction to PL/SQL

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

PL/SQL (Procedural Language/SQL) is Oracle’s procedural extension to SQL.
It adds **programming features** to SQL, allowing you to write full programs—not just queries.

PL/SQL supports:

Variables

Loops

Conditions

Functions & Procedures

Exception handling

Triggers

Packages

SQL alone is *declarative* (what to fetch),
but PL/SQL is *procedural* (how to perform operations).

How PL/SQL Extends SQL's Capabilities

1. Adds procedural features

You can use:

IF–ELSE

LOOP, WHILE, FOR

Variables & constants

Functions and stored procedures

SQL cannot do this by itself.

2. Allows grouping SQL statements

PL/SQL lets you execute **multiple SQL commands as a single block**, improving control and reducing redundancy.

3. Provides better error handling

Using **EXCEPTION** blocks, PL/SQL can catch and handle errors gracefully—something SQL alone cannot do.

4. Supports modular programming

You can create:

Functions

Procedures

Packages

This allows reuse and cleaner code management.

5. Allows triggers and automation

Database triggers rely on PL/SQL to automate actions on tables.

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

1.Improved Performance

PL/SQL sends an entire block of code to the database at once, reducing network traffic.

This **increases performance**, especially for loops and multi-step processes.

2. High Productivity

PL/SQL supports:

Reusable functions

Procedures

Packages

Modular code

This reduces development time.

3. Strong Error Handling

PL/SQL provides:

`EXCEPTION`

```
   WHEN NO_DATA_FOUND THEN ...
   WHEN OTHERS THEN ...
```

This makes applications more stable and reliable.

4. Tight Integration with SQL

PL/SQL is designed to work **seamlessly with SQL**, allowing SQL queries inside PL/SQL blocks without extra setup.

5. Supports OOP-like Concepts

Packages in PL/SQL allow:

Encapsulation

Abstraction

Data hiding

Similar to object-oriented programming.

6. Secure & Maintainable Code

Business logic stored in the database using PL/SQL:

Prevents SQL injection

Centralizes logic

Easier to update across applications

7. Portability

PL/SQL code runs on any Oracle database regardless of platform.

Summary Table

Feature	SQL	PL/SQL
Type	Declarative	Procedural
Loops, conditions	❌No	✅Yes
Error handling	❌No	✅Yes
Modular programming	❌No	✅Yes
Triggers, functions	❌No	✅Yes

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 allow programmers to **control the flow of execution** in a program. They add decision-making and repetition capabilities.

PL/SQL supports three types of control structures:

Conditional control → IF, ELSIF, CASE

Iterative control → LOOP, WHILE, FOR

Sequential control → GOTO, NULL

A. IF-THEN Control Structure

It is used to make **decisions** based on conditions.

⭐Syntax:

```
IF condition THEN
```

```
   statements;
END IF;
```

⭑Example:

```
IF salary < 30000 THEN
   DBMS_OUTPUT.PUT_LINE('Low salary');
END IF;
```

Variations:

IF-THEN-ELSE

IF-THEN-ELSIF-ELSE

B. LOOP Control Structure

Loops are used to **repeat** a block of statements multiple times.

PL/SQL supports 3 loops:

Simple LOOP

WHILE LOOP

FOR LOOP

1. Simple LOOP

Runs repeatedly until an **EXIT** condition is given.

Example:

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

2. WHILE LOOP

Runs as long as a condition is TRUE.

3. FOR LOOP

Runs a specific number of times automatically.

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

1. Enable decision-making

Using IF-THEN, programs can handle business rules:

Apply different discounts based on purchase amount

Validate data before inserting

Prevent invalid operations

2. Allow repetition

Using loops, PL/SQL can:

Process multiple rows

Perform calculations repeatedly

Generate reports

Insert/update data in batches

This is not possible with plain SQL.

3. Improve automation

Complex tasks like:

Monthly salary processing

Stock updates

Interest calculations

can be automated using loops and conditional logic.

4. Reduce complex SQL into simpler steps

Instead of writing one extremely complex query, PL/SQL allows breaking logic into steps using control structures:

First fetch data

Then check conditions

Then process or update accordingly

5. Increases flexibility

Business logic often changes.
Using control structures, PL/SQL programs can adapt easily without changing database tables.

Summary

Feature	**IF-THEN**	**LOOP**
Purpose	Decision-making	Repetition
Execution	Runs once if condition is TRUE	Runs until condition met
Example	Check salary, validate inputs	Process items, calculate totals

Feature	IF-THEN	LOOP

18. SQL Cursor

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

What is a Cursor?

A **cursor** in PL/SQL is a pointer to the result set of a SQL query.
It allows PL/SQL to **process rows one at a time**.

Whenever SQL returns multiple rows, PL/SQL needs a cursor to handle them.

A. Implicit Cursor

Created **automatically** by Oracle whenever a SQL statement is executed.

Used with:

`INSERT`

`UPDATE`

`DELETE`

`SELECT … INTO` (returns only one row)

Features:

No manual declaration

Automatically opened, fetched, and closed

Errors thrown if SELECT returns multiple or zero rows

Has attributes like:

`%FOUND`

```
%NOTFOUND

%ROWCOUNT

%ISOPEN
```

Example:

```
SELECT salary INTO v_sal FROM employees WHERE id = 101;
```

Oracle automatically manages the cursor.

B. Explicit Cursor

Created **manually** by the programmer for queries that return **multiple rows**.

Steps:

DECLARE cursor

OPEN cursor

FETCH rows

CLOSE cursor

Example:

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

```
END;
```

Difference Between Implicit and Explicit Cursors

Feature	**Implicit Cursor**	**Explicit Cursor**
Created by	Oracle automatically	Programmer manually
Purpose	Single-row operations	Multi-row operations
Control	No control over fetch	Full control (open, fetch, close)
Performance	Good for simple SQL	Good for complex logic
Cursor attributes	Yes	Yes
When used	SELECT INTO, insert/update/delete	SELECT returning multiple rows

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

1. Query returns multiple rows

`SELECT INTO` works only for **one row**.
For multi-row processing, explicit cursors are required.

2. You want full control over row-by-row processing

Explicit cursors allow:

Row-by-row fetch

Conditional processing

Skipping certain rows

Exiting based on custom conditions

3. You need to process each row individually

Examples:

Print employee records one by one

Calculate totals from each row

Apply different rules per row

4. You want to improve readability and maintainability

Custom named cursors make code easier to understand.

5. When performance tuning is required

Explicit cursors allow:

Bulk fetching

Fetching limited rows

Optimizing loops

19. Rollback and Commit Savepoint

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

What is a SAVEPOINT?

A **SAVEPOINT** is a marker inside a transaction that allows you to **partially roll back** the transaction to a specific point without affecting the complete transaction.

It creates a checkpoint.

Syntax:

SAVEPOINT `savepoint_name;`

How SAVEPOINT Works with ROLLBACK and COMMIT

ROLLBACK TO SAVEPOINT

`ROLLBACK TO savepoint_name` undoes only the changes made **after** the savepoint.

Example:

```
SAVEPOINT sp1;
/* Some SQL operations */
ROLLBACK  TO sp1;      -- undo operations after sp1
```

The transaction remains active after the rollback.

COMMIT and SAVEPOINT

When you execute a **COMMIT**:

All changes become permanent.

All savepoints are removed.

You cannot roll back to any previous savepoint after a commit.

Example:

```
SAVEPOINT sp1;
COMMIT;           -- removes sp1
ROLLBACK  TO sp1;     -- ✕ not allowed
```

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

1. When part of the transaction may fail

If one operation fails but others should still continue.

Example:
Updating multiple tables — if one fails, rollback just that part.

2. To test intermediate results

Developers can experiment with partial updates and roll back only the wrong section.

3. In large batch operations

While inserting or processing thousands of rows, savepoints help recover easily from small errors.

4. When you want finer control over transactions

You can roll back selectively instead of rolling back the entire transaction.

5. In financial or business operations

Example:

Deduct balance

Add reward points

Update transaction history

If reward calculation fails, you can roll back only that part.