Module 4

Introduction to DBMS

1 . Introduction to SQL :

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

SQL (Structured Query Language) is a standard programming language specifically designed for managing and manipulating relational databases.

Importance in database management:

- SQL allows users to **create**, **read**, **update**, and **delete** (CRUD) data in a database.
- It helps in managing **large amounts of structured data** efficiently.
- SQL provides a standardized way to interact with most relational database systems like MySQL, Oracle, SQL Server, and PostgreSQL.
- It ensures **data integrity**, **security**, and **consistency** when handling database operations.

---

2. Explain the difference between DBMS and RDBMS.

Feature	DBMS (Database Management System)	RDBMS (Relational Database Management System)
Data Storage	Stores data as files or in structures like JSON or XML	Stores data in **tables** with rows and columns
Data Relationship	No or limited support for relationships between data	Supports **relationships** using foreign keys
Normalization	Not strictly enforced	Enforces **normalization** to reduce redundancy

Feature	DBMS (Database Management System)	RDBMS (Relational Database Management System)
Example Systems	Microsoft Access, File System-based DB	MySQL, Oracle, SQL Server, PostgreSQL
Security	Less robust	Stronger security with user permissions and roles

---

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

SQL is the **primary interface** for working with relational databases. It helps in:

- **Defining** data structures using DDL (Data Definition Language) commands like CREATE, ALTER, DROP
- **Manipulating** data using DML (Data Manipulation Language) commands like INSERT, UPDATE, DELETE
- **Querying** data using SELECT statements to retrieve specific information
- **Controlling access** and **permissions** using DCL (Data Control Language)
- **Managing transactions** and ensuring data consistency using TCL (Transaction Control Language)

---

4. What are the key features of SQL?

- **Simple and Easy to Learn** – SQL uses English-like syntax.
- **Data Manipulation** – Allows inserting, updating, and deleting data easily.
- **Data Querying** – Powerful querying with filtering, sorting, and grouping.
- **Data Definition** – Supports table creation and modification.
- **Data Control** – Provides security and access control (e.g., GRANT, REVOKE).
- **Transaction Control** – Maintains data integrity using COMMIT, ROLLBACK, SAVEPOINT.
- **Standardized** – Supported by all major relational databases.

- **Support for Relationships** – Ensures integrity using **primary** and **foreign keys**.

2 . SQL Syntax :

1. What are the basic components of SQL syntax?

SQL syntax consists of commands and clauses used to interact with databases. The basic components include:

Component	Description
Keywords	Reserved words like SELECT, FROM, WHERE, INSERT, UPDATE, etc.
Identifiers	Names of database objects like tables, columns, databases (e.g., students, id)
Expressions	Used to calculate values or compare data (e.g., age > 18)
Clauses	Parts of SQL statements (e.g., WHERE, GROUP BY, ORDER BY)
Operators	Symbols for comparison or logic (=, >, <, AND, OR, LIKE)
Literals	Constant values like strings or numbers ('John', 100)
Semicolon ;	Marks the end of a statement in many SQL systems

---

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

sql

CopyEdit

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

Example:

sql

CopyEdit

```
SELECT name, age
FROM students
WHERE age > 18
ORDER BY age DESC;
```

---

3. Explain the role of clauses in SQL statements.

Clauses are used to filter, group, and sort data in SQL queries. Each clause serves a specific function:

Clause	Purpose
SELECT	Specifies the columns to retrieve
FROM	Specifies the table to fetch data from
WHERE	Filters rows based on conditions
GROUP BY	Groups rows that have the same values in specified columns
HAVING	Filters groups created by GROUP BY
ORDER BY	Sorts the result set (ascending or descending)

Example with all clauses:

sql

CopyEdit

```
SELECT department, COUNT(*)
FROM employees
WHERE salary > 30000
GROUP BY department
HAVING COUNT(*) > 5
```

ORDER BY COUNT(*) DESC;

3. SQL Constraints :

1. What are constraints in SQL?

Constraints in SQL are rules applied to columns in a table to enforce data integrity and ensure valid data is stored in the database.

They help prevent invalid, duplicate, or null values where they're not allowed.

Common Types of Constraints:

Constraint	Description
NOT NULL	Ensures that a column cannot have NULL values.
UNIQUE	Ensures that all values in a column are different.
PRIMARY KEY	Uniquely identifies each row in a table. Combines NOT NULL and UNIQUE.
FOREIGN KEY	Maintains referential integrity between two tables (connects them).
CHECK	Ensures that a column's values meet a specified condition.
DEFAULT	Sets a default value if no value is provided for the column.

---

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

Feature	PRIMARY KEY	FOREIGN KEY
Purpose	Uniquely identifies each row in the same table	Links two tables by referencing the primary key of another table
Uniqueness	Must be unique and not null	Can have duplicate values (if allowed)

Feature	PRIMARY KEY	FOREIGN KEY
NULL values	Not allowed	NULL values are allowed
Defined on	The current table	Refers to a column in another table

Example:

sql

CopyEdit

```
-- PRIMARY KEY in students table
CREATE TABLE students (
   student_id INT PRIMARY KEY,
   name VARCHAR(50)
);

-- FOREIGN KEY in enrollments table
CREATE TABLE enrollments (
   enrollment_id INT PRIMARY KEY,
   student_id INT,
   FOREIGN KEY (student_id) REFERENCES students(student_id)
);
```

---

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

Constraint	Role
NOT NULL	Ensures that a column must always have a value; prevents inserting NULL into the column.
UNIQUE	Ensures that all the values in a column are different; allows only one NULL (depending on DBMS).

Example:

sql

CopyEdit

```
CREATE TABLE users (
   user_id INT NOT NULL UNIQUE,
   email VARCHAR(100) UNIQUE,
   name VARCHAR(50) NOT NULL
);
```

user_id must always have a value and must be unique.

email must be unique across users.

name must not be empty (NULL).

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

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

DDL (Data Definition Language) is a subset of SQL used to define, modify, and manage the structure of database objects like tables, schemas, and indexes.

Common DDL Commands:

Command	Purpose
CREATE	To create new database objects (tables, views, etc.)
ALTER	To modify the structure of an existing object
DROP	To delete database objects
TRUNCATE	To remove all data from a table (faster than DELETE)
RENAME	To rename a database object

---

2. Explain the CREATE command and its syntax.

The CREATE command is used to create new database objects, most commonly a table.

General Syntax to Create a Table:

sql

CopyEdit

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

Example:

sql

CopyEdit

```
CREATE TABLE employees (
   emp_id INT PRIMARY KEY,
   name VARCHAR(50) NOT NULL,
   salary DECIMAL(10, 2),
   department_id INT,
   UNIQUE (name)
);
```

- emp_id: Integer, unique and cannot be null (primary key)
- name: String up to 50 characters, cannot be null
- salary: Decimal value with up to 10 digits and 2 decimal places
- department_id: Integer, optional

---

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

Purpose of Data Types:

- Define what kind of data (e.g., INT, VARCHAR, DATE) can be stored in each column.
- Helps in memory optimization, validation, and accurate data processing.

Purpose of Constraints:

- Maintain data integrity by enforcing rules on data.
- Prevent invalid, duplicate, or incomplete entries.
- Ensure relationships between tables remain valid (using keys).

Feature	Benefit
Data Types	Avoids wrong data types (e.g., entering text in a numeric field)
NOT NULL	Ensures mandatory fields are filled
UNIQUE	Avoids duplicate entries
PRIMARY KEY	Uniquely identifies rows
FOREIGN KEY	Maintains relationships between tables
CHECK	Validates values using custom conditions
DEFAULT	Automatically fills default values if none provided

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.

Key uses of ALTER:

Add new columns

Modify existing columns

Drop (delete) columns

Rename columns or tables

Add or drop constraints

---

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

A. Add a new column

sql

CopyEdit

ALTER TABLE table_name

ADD column_name datatype [constraint];

Example:

sql

CopyEdit

ALTER TABLE employees

ADD birthdate DATE;

B. Modify an existing column

sql

CopyEdit

ALTER TABLE table_name

MODIFY column_name new_datatype [constraint]; -- (MySQL, Oracle)

-- or

ALTER TABLE table_name

ALTER COLUMN column_name TYPE new_datatype;   -- (PostgreSQL)

Example (MySQL):

sql

CopyEdit

ALTER TABLE employees

MODIFY salary DECIMAL(12,2);

Example (PostgreSQL):

sql

CopyEdit

ALTER TABLE employees

ALTER COLUMN salary TYPE DECIMAL(12,2);

C. Drop (delete) a column

sql

CopyEdit

ALTER TABLE table_name

DROP COLUMN column_name;

Example:

sql

CopyEdit

ALTER TABLE employees

DROP COLUMN birthdate;

6. DROP Command :

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

The **DROP** command is used to **permanently delete** a database object such as a **table, view, index, or even an entire database**.

Syntax for dropping a table:

sql

CopyEdit

```
DROP TABLE table_name;
```

Other uses:

- DROP DATABASE database_name;
- DROP VIEW view_name;
- DROP INDEX index_name;

Example:

sql

CopyEdit

```
DROP TABLE employees;
```

This command will completely delete the employees table and all its data.

---

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

Dropping a table has **serious and irreversible consequences**:

Consequence	Description
Permanent Deletion	The table and **all its data** are permanently removed.
Loss of Constraints	All **indexes, constraints, keys**, and triggers associated with the table are also deleted.
Cannot Rollback (usually)	In most systems, once a table is dropped, it **cannot be recovered** unless you have a backup.
Dependent Objects Fail	Any **views, procedures, or queries** that depended on the dropped table will stop working.

7. Data Manipulation Language (DML) :

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

These are **Data Manipulation Language (DML)** commands used to **work with data** inside tables.

INSERT – Add new records into a table.

Syntax:

sql

CopyEdit

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

Example:

sql

CopyEdit

```
INSERT INTO students (id, name, age)
VALUES (101, 'Amit', 20);
```

UPDATE – Modify existing records in a table.

Syntax:

sql

CopyEdit

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

Example:

sql

CopyEdit

```
UPDATE students
SET age = 21
WHERE id = 101;
```

DELETE – Remove one or more records from a table.

Syntax:

```
sql
CopyEdit
DELETE FROM table_name
[WHERE condition];
```

Example:

```
sql
CopyEdit
DELETE FROM students
WHERE id = 101;
```

---

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

The **WHERE clause** is **critical** in UPDATE and DELETE operations because it **controls which rows** are affected.

Without WHERE	With WHERE
Updates or deletes all rows in the table — often by mistake!	**Targets only specific rows** that match the condition
Can result in **loss of all data** in a table	Helps in **safe and precise** data manipulation
Irreversible without a backup	Minimizes risk of data loss

Example of a dangerous mistake:

sql

CopyEdit

```
DELETE FROM students;
-- Deletes ALL records from the table!
```

Safer version using WHERE:

sql

CopyEdit

```
DELETE FROM students
WHERE age < 18;
-- Deletes only students under 18
```

8. Data Query Language (DQL) :

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

The SELECT statement is used in SQL (Structured Query Language) to **retrieve data from a database**. It allows you to specify **which columns** and **which rows** you want from one or more database tables.

Basic Syntax:

sql

CopyEdit

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

Example:

sql

CopyEdit

SELECT name, age

FROM students;

This query gets the name and age columns from the students table.

To select all columns:

sql

CopyEdit

SELECT * FROM students;

---

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

WHERE Clause:

The WHERE clause is used to **filter** records. It retrieves **only the rows** that meet a certain condition.

Syntax:

sql

CopyEdit

SELECT column1, column2

FROM table_name

WHERE condition;

Example:

sql

CopyEdit

SELECT name, age

FROM students

WHERE age > 18;

This returns names and ages of students who are older than 18.

ORDER BY Clause:

The ORDER BY clause is used to **sort** the result set **in ascending or descending order**.

Syntax:

sql

CopyEdit

SELECT column1, column2

FROM table_name

ORDER BY column1 [ASC|DESC];

Example:

sql

CopyEdit

SELECT name, age

FROM students

ORDER BY age DESC;

This lists students sorted by age, from highest to lowest.

Combining Both:

You can use both WHERE and ORDER BY in a single query.

Example:

sql

CopyEdit

SELECT name, age

FROM students

WHERE age > 18

ORDER BY name ASC;

This gets students older than 18 and sorts their names in alphabetical order.

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 permissions** (also called privileges) to users on database objects like tables, views, or procedures.

🔹 **Common privileges:**

- SELECT – Read data
- INSERT – Add data
- UPDATE – Modify data
- DELETE – Remove data

👉 **Example:**

sql

CopyEdit

GRANT SELECT, INSERT ON students TO user1;

This gives user1 permission to **view and add data** in the students table.

✅ **REVOKE Command:**

The REVOKE command is used to **take back** permissions that were previously granted to a user.

👉 **Example:**

sql

CopyEdit

REVOKE INSERT ON students FROM user1;

This removes the INSERT permission from user1 on the students table. Now they can no longer add new data there.

---

2. How do you manage privileges using these commands?

You can use GRANT and REVOKE to control **what users are allowed or not allowed to do** in your database.

Example Scenario:

Suppose you're a database admin, and you have a table called employees. You want:

- user1 to only read data.
- user2 to be able to read and update data.
- Later, you want to remove update access from user2.

Step 1 – Grant permissions:

sql

CopyEdit

GRANT SELECT ON employees TO user1;

GRANT SELECT, UPDATE ON employees TO user2;

Step 2 – Revoke permissions:

sql

CopyEdit

REVOKE UPDATE ON employees FROM user2;

10. Transaction Control Language (TCL) :

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

COMMIT:

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

Example:

sql

CopyEdit

BEGIN;

UPDATE accounts SET balance = balance - 500 WHERE id = 1;

UPDATE accounts SET balance = balance + 500 WHERE id = 2;

COMMIT;

Once COMMIT is executed, the changes to the accounts table are **saved permanently**.

ROLLBACK:

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

Example:

sql

CopyEdit

BEGIN;

UPDATE accounts SET balance = balance - 500 WHERE id = 1;

UPDATE accounts SET balance = balance + 500 WHERE id = 2;

ROLLBACK;

Here, the money transfer operation is **cancelled**, and the database goes back to its previous state.

---

2. How are transactions managed in SQL databases?

Definition:

A **transaction** is a sequence of one or more SQL operations treated as a **single unit of work**. It must either **complete fully** or **not happen at all**.

Properties of Transactions – ACID:

Property	Description
A – Atomicity	All operations succeed or none do. If one fails, everything rolls back.
C – Consistency	Database remains in a valid state before and after the transaction.
I – Isolation	Transactions do not interfere with each other.
D – Durability	Once committed, changes are permanent even in case of a crash.

Transaction Management Commands:

Command	Description
BEGIN or START TRANSACTION	Starts a new transaction
COMMIT	Saves the changes permanently
ROLLBACK	Cancels changes made in the transaction

Example – Full Transaction:

sql

CopyEdit

```
BEGIN;

INSERT INTO orders (id, customer, total) VALUES (101, 'John', 250);
UPDATE inventory SET stock = stock - 1 WHERE product_id = 55;
```

-- If no error:

COMMIT;

-- If there's an issue:

-- ROLLBACK;

11. SQL Joins :

1. What is the concept of JOIN in SQL?

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** and a **primary key**).

For example, if you have:

- A customers table with customer info
- An orders table with order details

You can join them to get **customer names with their order info**.

Types of SQL Joins and Their Differences:

1. INNER JOIN

- Returns **only the matching rows** from both tables.
- If there's no match, the row is **not included**.

Example:

sql

CopyEdit

SELECT customers.name, orders.order_date

FROM customers

INNER JOIN orders ON customers.id = orders.customer_id;

Result: Only customers who placed orders will be shown.

---

2. LEFT JOIN (or LEFT OUTER JOIN)

- Returns **all rows from the left table** and the **matched rows** from the right table.
- If there’s no match, you'll get NULL for right table columns.

Example:

sql

CopyEdit

SELECT customers.name, orders.order_date

FROM customers

LEFT JOIN orders ON customers.id = orders.customer_id;

Result: All customers will be shown, even if they did not place any orders (with NULL for order_date).

---

3. RIGHT JOIN (or RIGHT OUTER JOIN)

- Opposite of LEFT JOIN.
- Returns **all rows from the right table**, and **matched rows** from the left.
- If no match, NULL is shown for left table columns.

Example:

sql

CopyEdit

SELECT customers.name, orders.order_date

FROM customers

RIGHT JOIN orders ON customers.id = orders.customer_id;

Result: All orders will be shown, even if there's no matching customer (e.g., data inconsistency).

---

4. FULL OUTER JOIN

- Returns **all rows** from both tables.
- If no match is found on either side, NULL is used.

Example:

sql

CopyEdit

SELECT customers.name, orders.order_date

FROM customers

FULL OUTER JOIN orders ON customers.id = orders.customer_id;

Result: Combines results of LEFT and RIGHT JOIN. Shows all customers and all orders, matched or not.

Summary Table:

Type	Includes from Left	Includes from Right	NULLs Where?
INNER JOIN	Only if matched	Only if matched	No
LEFT JOIN	All rows	Only if matched	Right side if unmatched
RIGHT JOIN	Only if matched	All rows	Left side if unmatched
FULL JOIN	All rows	All rows	Both sides if unmatched

---

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

Joins allow us to **merge data** from related tables so that we can:

- Report across tables
- Analyze linked data
- Reduce duplication by normalizing the database

Example Scenario:

Two tables:

- employees(id, name, department_id)
- departments(id, dept_name)

SQL to get employee names and their department names:

sql

CopyEdit

SELECT employees.name, departments.dept_name

FROM employees

INNER JOIN departments

ON employees.department_id = departments.id;

This combines rows from both tables where department_id matches id.

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 is used in SQL to **group rows that have the same values** in one or more columns. It is most commonly used with **aggregate functions** like:

- SUM() – total value
- AVG() – average
- COUNT() – number of rows
- MAX() – highest value
- MIN() – lowest value

Example:

Table: sales

product	quantity
Apple	10
Banana	5
Apple	8
Banana	12

Query:

```
sql
CopyEdit
SELECT product, SUM(quantity) AS total_quantity
FROM sales
GROUP BY product;
```

Output:

product	total_quantity
Apple	18
Banana	17

This shows the **total quantity** sold for each product.

Rules:

- Every column in SELECT (except aggregate functions) **must be in the GROUP BY clause**.
- Use HAVING to filter **groups**, not WHERE.

---

2. What is the difference between GROUP BY and ORDER BY?

Feature	GROUP BY	ORDER BY
Purpose	Groups rows by common values	Sorts the result set
Used with	Aggregate functions (SUM, AVG, etc.)	Any query
Output	One row per group	All rows, just sorted
Clause order	Appears **before** ORDER BY	Appears **at the end** of query

---

Example using both:

sql

CopyEdit

```
SELECT product, SUM(quantity) AS total_quantity
FROM sales
GROUP BY product
ORDER BY total_quantity DESC;
```

- GROUP BY product → Groups sales by product
- ORDER BY total_quantity DESC → Sorts the grouped results from highest to lowest total

---

Summary:

Clause	Groups or Sorts?	Works With Aggregates?	Filters Groups?
GROUP BY	Groups rows	Yes	Yes (with HAVING)
ORDER BY	Sorts result	No	No

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** (like queries, logic, loops, etc.) stored in the database. It can be **executed multiple times** by calling its name.

Think of it like a **function in programming**, but for SQL operations.

Syntax (MySQL-style):

sql

CopyEdit

```
DELIMITER //

CREATE PROCEDURE GetEmployeeByDept(IN dept_id INT)
BEGIN
  SELECT * FROM employees WHERE department_id = dept_id;
END //

DELIMITER ;
```

You can then call it like:

sql

CopyEdit

```
CALL GetEmployeeByDept(2);
```

Difference from Standard SQL Query:

Feature	Standard SQL Query	Stored Procedure
Written where?	In SQL editor or app directly	Saved in the database
Reusable?	No – written each time	Yes – just call it
Supports logic/loops?	No	Yes (IF, LOOP, etc.)
Performance	Compiled every time	Precompiled – faster on repeated runs
Input/Output parameters	Not supported	Supported

---

2. Advantages of Using Stored Procedures:

1. Reusability:

Write once, use many times—no need to rewrite complex logic again and again.

2. Performance:

Stored procedures are **precompiled**, so they run faster than repeated ad hoc queries.

3. Security:

You can **control access** by granting permission to execute the procedure **without giving direct access** to underlying tables.

4. Maintainability:

If you need to change business logic, just update the procedure—not the application code.

5. Reduced Network Traffic:

You send a single CALL command instead of multiple SQL statements over the network.

6. Modularity:

Helps break down big operations into manageable, logical blocks—just like functions in a program.

Example Use Case:

If your app needs to:

- Add a new order
- Reduce inventory
- Log the transaction

You can write a stored procedure like:

sql

CopyEdit

```
CREATE PROCEDURE ProcessOrder(IN product_id INT, IN qty INT)
BEGIN
  INSERT INTO orders(product_id, quantity) VALUES(product_id, qty);
  UPDATE products SET stock = stock - qty WHERE id = product_id;
  INSERT INTO order_log(product_id, status) VALUES(product_id, 'Placed');
END;
```

Call it with:

sql

CopyEdit

```
CALL ProcessOrder(101, 3);
```

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’s based on the **result of a SQL query**, and it doesn’t store actual data—it just **displays data from existing tables**.

Syntax:

sql

CopyEdit

```
CREATE VIEW view_name AS
SELECT column1, column2
FROM table_name
WHERE condition;
```

Example:

Suppose you have a table called employees:

id	name	salary
1	Alice	50000
2	Bob	70000

Now create a view:

sql

CopyEdit

```
CREATE VIEW high_salary AS
SELECT name, salary FROM employees WHERE salary > 60000;
```

Now if you run:

sql

CopyEdit

```
SELECT * FROM high_salary;
```

You'll get only the employees with salary above 60,000.

Difference between a View and a Table:

Feature	Table	View
Stores Data?	Yes	No (virtual – stores query only)
Can Have Indexes?	Yes	No (depends on database system)
Updates Allowed?	Yes	Sometimes (only for simple views)
Performance Impact	Fast (data stored)	Depends on underlying tables

---

2. Advantages of Using Views in SQL Databases

1. Simplifies Complex Queries

Views can wrap complex joins or calculations into a single virtual table.

Example: You can query SELECT * FROM sales_summary instead of writing a long join every time.

---

2. Enhances Security

Views can **restrict access** to certain columns or rows.

Example: A view may expose only name and email from a users table—hiding sensitive info like password or salary.

---

3. Provides Logical Independence

If the structure of the base table changes, the view can be **updated without changing the application code**.

---

4. Reusability and Consistency

Views act as reusable SQL templates for repeated analysis or reporting.

- **5. Readability**

Makes queries easier to read and understand, especially for reporting or business users.

Summary Table:

Feature	Table	View
Data Storage	Physically stores data	Does not store data (virtual table)
Definition	Created with CREATE TABLE	Created with CREATE VIEW
Speed	Fast (data is stored)	Slower if based on complex queries
Purpose	Data storage and updates	Data presentation and filtering

15. SQL Triggers :

1. What is a trigger in SQL?

Trigger:

A **trigger** is a **special stored procedure** that **automatically runs** (is "triggered") when certain **events** happen on a table or view—like an INSERT, UPDATE, or DELETE.

- It's **automatic**—you don't need to call it manually.
- Used for enforcing business rules, data validation, or logging changes.

Basic Syntax (MySQL-style):

sql

CopyEdit

CREATE TRIGGER trigger_name

AFTER INSERT ON table_name

FOR EACH ROW

```
BEGIN
   -- SQL statements
END;
```

Types of Triggers in SQL:

Based on timing:

Type	Description
BEFORE Trigger	Executes **before** the triggering event (e.g., before an insert/update/delete happens). Useful for validation or modification of data.
AFTER Trigger	Executes **after** the triggering event. Useful for logging or audit trails.

Based on event:

Event Type	Triggered When...
INSERT	A new row is added to a table
UPDATE	A row is modified
DELETE	A row is removed

Example: Logging insert to another table

sql

CopyEdit

```
CREATE TRIGGER log_insert
AFTER INSERT ON orders
FOR EACH ROW
BEGIN
```

```
  INSERT INTO order_log(order_id, action)
  VALUES (NEW.id, 'Inserted');
END;
```

This creates a log entry **every time a new order is added**.

---

2. Difference between INSERT, UPDATE, and DELETE triggers

Trigger Type	When It Fires	Common Uses	Access Variables
INSERT	When a new row is added	Logging new data, auto-filling fields	NEW.column_name
UPDATE	When data in a row is changed	Audit changes, enforce business rules	OLD.column_name, NEW.column_name
DELETE	When a row is removed	Archive or log deleted records	OLD.column_name

---

Example for Each:

INSERT Trigger:

```
sql
CopyEdit
CREATE TRIGGER after_insert_employee
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
  INSERT INTO logs(action) VALUES ('New employee added');
END;
```

UPDATE Trigger:

sql

CopyEdit

```
CREATE TRIGGER after_update_salary
AFTER UPDATE ON employees
FOR EACH ROW
BEGIN
  INSERT INTO salary_log(emp_id, old_salary, new_salary)
  VALUES (OLD.id, OLD.salary, NEW.salary);
END;
```

DELETE Trigger:

sql

CopyEdit

```
CREATE TRIGGER after_delete_employee
AFTER DELETE ON employees
FOR EACH ROW
BEGIN
  INSERT INTO logs(action) VALUES ('Employee deleted');
END;
```

Summary:

Trigger Type	When Used	Useful For
INSERT	New record added	Logging, auto-calculations
UPDATE	Record modified	Audit logs, tracking changes

Trigger Type	When Used	Useful For
DELETE	Record removed	Backups, logging deleted data

16. Introduction to PL/SQL :

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

PL/SQL stands for:

Procedural Language extensions to SQL

It is Oracle's **procedural programming language** built on top of SQL. It allows you to write **programming logic (like IF, loops, variables)** within SQL-based programs.

Key Features:

- Combines **SQL with procedural features** (like IF, WHILE, FOR, FUNCTIONS, PROCEDURES)
- Supports **variables, conditions, loops, and error handling**
- Used to create **stored procedures, functions, triggers, packages**

How it extends SQL:

SQL Can Do	PL/SQL Adds
Simple queries & updates	Variables and constants
DDL/DML statements	IF-ELSE, loops (FOR/WHILE)
Transactions	Exception handling (TRY-CATCH style)
Single statements	Group multiple statements in blocks

Example:

SQL-only approach:

sql

CopyEdit

```
UPDATE employees SET salary = salary + 1000 WHERE id = 101;
```

PL/SQL version with logic:

sql

CopyEdit

```
DECLARE
  emp_id NUMBER := 101;
BEGIN
  UPDATE employees SET salary = salary + 1000 WHERE id = emp_id;
  DBMS_OUTPUT.PUT_LINE('Salary updated');
END;
```

---

2. Benefits of Using PL/SQL

1. Procedural Capabilities

- Adds programming logic: IF, LOOP, CASE, etc.
- Helps automate and control data operations more flexibly.

---

2. Performance Optimization

- **Reduces network traffic**: Multiple SQL statements can be sent to the server in a single PL/SQL block.
- **Precompiled**: Stored procedures/functions execute faster.

---

3. Modularity

- You can break complex logic into **procedures, functions, and packages**.
- Promotes clean and maintainable code.

---

4. Exception Handling

- Built-in support to **handle errors gracefully** using EXCEPTION blocks.
- Example:

sql

CopyEdit

```
BEGIN
  -- risky operation
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No record found');
END;
```

---

5. Tight Integration with SQL

- You can use SELECT, INSERT, UPDATE, and DELETE directly inside a PL/SQL block.
- Smooth access to database objects.

---

6. Reusability and Maintainability

- Code can be stored as reusable **stored procedures** and **packages**.
- Makes long-term maintenance easier.

Summary Table:

Feature	**SQL Only**	**PL/SQL Adds**
Queries & Transactions	Yes	Yes
Logic & Control Flow	No	Yes (IF, LOOP, CASE, etc.)
Error Handling	Limited	Yes (EXCEPTION blocks)

Feature	SQL Only	PL/SQL Adds
Modularity	No	Yes (Procedures, Functions)

17. PL/SQL Control Structures :

1. What are control structures in PL/SQL?

Control structures are used to:

- **Control the flow** of execution in a PL/SQL block
- Add **decision-making**, **repetition**, and **branching**

PL/SQL supports **three types** of control structures:

Type	Example Keywords
Conditional	IF, ELSE, ELSIF
Iterative (Loops)	LOOP, WHILE, FOR
Sequential (Branching)	GOTO, EXIT, RETURN

IF-THEN control structure (Conditional)

Used for **decision making** based on conditions.

Syntax:

sql

CopyEdit

```
IF condition THEN
  -- statements;
END IF;
```

With ELSE / ELSIF:

sql

CopyEdit

```
IF condition1 THEN
  -- statements;
ELSIF condition2 THEN
  -- other statements;
ELSE
  -- default statements;
END IF;
```

Example:

sql

CopyEdit

```
DECLARE
  salary NUMBER := 30000;
BEGIN
  IF salary > 50000 THEN
    DBMS_OUTPUT.PUT_LINE('High Salary');
  ELSIF salary BETWEEN 30000 AND 50000 THEN
    DBMS_OUTPUT.PUT_LINE('Medium Salary');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Low Salary');
  END IF;
END;
```

LOOP control structures (Iterative)

Used for **repeating a block** of statements multiple times.

---

1. BASIC LOOP

Runs **infinitely** unless you explicitly use EXIT.

sql

CopyEdit

```
LOOP
  -- statements
  EXIT WHEN condition;
END LOOP;
```

Example:

sql

CopyEdit

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

---

2. WHILE LOOP

Executes **as long as the condition is TRUE**.

sql

CopyEdit

```
WHILE condition LOOP
  -- statements
```

```
END LOOP;
```

---

3. FOR LOOP

Simplest loop with **automatic counter**.

```
sql
CopyEdit
FOR i IN 1..5 LOOP
  DBMS_OUTPUT.PUT_LINE(i);
END LOOP;
```

---

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

Adds logic and flexibility:

Control structures allow you to:

- Make decisions (IF) based on business rules
- Repeat operations (LOOP) until conditions are met
- Dynamically process rows, logs, validations, etc.

Example Use Case – Complex Business Logic:

```
sql
CopyEdit
DECLARE
  total NUMBER := 0;
BEGIN
  FOR i IN 1..10 LOOP
    IF MOD(i, 2) = 0 THEN
      total := total + i;
```

```
    END IF;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('Sum of even numbers from 1 to 10: ' || total);
END;
```

This kind of logic isn't possible with plain SQL alone.

Summary Table:

Structure Type	Keyword(s)	Use Case
Conditional	IF, ELSE	Make decisions
Iterative	LOOP, FOR, WHILE	Repeat tasks (like counters, validations)
Branching	GOTO, EXIT	Jump or exit specific parts

18. SQL Cursors :

1. What is a cursor in PL/SQL?

Cursor:

A **cursor** is a pointer to the **context area** that stores the result set of a query. It allows **row-by-row** processing in PL/SQL when the query returns **multiple rows**.

There are **two types** of cursors:

- **Implicit Cursor**
- **Explicit Cursor**

Implicit Cursor

- Created **automatically by Oracle** for **single-row queries** (like INSERT, UPDATE, DELETE, or simple SELECT INTO)

- You **don’t declare or open** it—it’s managed internally.
- Useful for **simple operations**

Example:

sql

CopyEdit

```
DECLARE
  emp_name employees.name%TYPE;
BEGIN
  SELECT name INTO emp_name FROM employees WHERE id = 101;
  DBMS_OUTPUT.PUT_LINE('Employee: ' || emp_name);
END;
```

Oracle internally creates and manages an **implicit cursor** here.

Explicit Cursor

- Created **manually** for queries that **return more than one row**
- Gives **more control**: You must declare, open, fetch, and close it.

Syntax:

sql

CopyEdit

```
DECLARE
  CURSOR emp_cursor IS SELECT name FROM employees;
  emp_name employees.name%TYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO emp_name;
```

```
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(emp_name);
  END LOOP;
  CLOSE emp_cursor;
END;
```

Difference between Implicit and Explicit Cursors:

Feature	**Implicit Cursor**	**Explicit Cursor**
Created by	Oracle (automatically)	Manually by programmer
Control	No control (automatic)	Full control over opening, fetching
For single/multiple rows	Single row or DML	Multiple rows
Declaration needed?	No	Yes
Performance	Faster for simple tasks	Useful for row-by-row logic

---

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

Use an explicit cursor when:

- Your query returns **more than one row**
- You need to **loop through** each row and perform custom logic
- You want to **monitor** cursor attributes (like %ROWCOUNT, %FOUND, %NOTFOUND)
- You're performing **complex processing per row**

Example Use Case:

Imagine a scenario where you need to give a **bonus** to every employee whose salary is below 50,000.

sql

CopyEdit

```
DECLARE
  CURSOR emp_cursor IS
    SELECT id, salary FROM employees WHERE salary < 50000;

  v_id employees.id%TYPE;
  v_salary employees.salary%TYPE;
BEGIN
  OPEN emp_cursor;
  LOOP
    FETCH emp_cursor INTO v_id, v_salary;
    EXIT WHEN emp_cursor%NOTFOUND;

    UPDATE employees SET salary = salary + 500 WHERE id = v_id;
  END LOOP;
  CLOSE emp_cursor;
END;
```

19. Rollback and Commit Savepoint :

1. What is the concept of SAVEPOINT in transaction management?

SAVEPOINT:

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

Think of it like setting a checkpoint in your work—if something goes wrong, you can go back to that checkpoint instead of starting from the beginning.

Syntax:

sql

CopyEdit

SAVEPOINT savepoint_name;

You can then roll back to it:

sql

CopyEdit

ROLLBACK TO savepoint_name;

How ROLLBACK and COMMIT interact with SAVEPOINT:

ROLLBACK TO savepoint_name:

- Undoes all changes **after** that savepoint
- Keeps changes **before** the savepoint

COMMIT:

- **Ends the transaction** and makes **all changes permanent**
- **Removes all savepoints**—you can't roll back after commit

ROLLBACK (without TO):

- Rolls back the **entire transaction**, ignoring any savepoints

---

Example:

sql

CopyEdit

BEGIN;

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

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

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

-- Oops! Something wrong with Charlie's record:
ROLLBACK TO sp2;

COMMIT;
```

Result:

- Only Alice and Bob are inserted
- Charlie’s insertion is rolled back

---

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

Situations where SAVEPOINT is helpful:

Use Case	Explanation
Complex transactions	When a transaction involves multiple steps, and you may want to undo only part of it if something goes wrong
Partial error handling	If one part of your logic fails, you can rollback to the last known good state
Testing small sections	Helps in debugging parts of the transaction without losing everything

Use Case	Explanation
Nested operations	Useful inside PL/SQL blocks or loops where each iteration might need a separate rollback point

---

Real-World Example:

Imagine a banking transaction:

sql

CopyEdit

```
BEGIN;

-- Debit from sender
UPDATE accounts SET balance = balance - 1000 WHERE account_id = 1;
SAVEPOINT debit_done;

-- Credit to receiver
UPDATE accounts SET balance = balance + 1000 WHERE account_id = 2;

-- Check: what if receiver's account doesn't exist?
-- ROLLBACK TO debit_done;

COMMIT;
```