SQL & PL/SQL

Your First Step into Databases

Vaibhav Rajasha

Table of Contents:

11. Data Control Language (DCL)
   - GRANT, REVOKE

12. Joins
   - INNER, LEFT, RIGHT, FULL, CROSS, SELF JOIN

13. Set Operators
   - UNION, UNION ALL, INTERSECT, MINUS

14. Subqueries
   - Nested, Correlated, Multi-row
   - Subqueries with EXISTS, IN

15. SQL Functions
   - Aggregate, Scalar, Conversion,String, Numeric, Date,Conditional
   - Windows, System, JSON, XML
   - CASE, COALESCE, DECODE, NULLIF

16. Views and Temporary Tables
   - Types of Views
   - WITH CHECK OPTION

Part IV – Functions & Advanced SQL

17. Temporary Table (Advanced SQL concepts)
   - CTE and WITH Clause
   - Recursive CTE
   - PIVOT and UNPIVOT
   - MERGE Statement (UPSERT)
   - Regular Expressions (REGEXP)

18. Indexes
   - Normal and Composite Indexes

19. Optimization Concepts
   - Query Execution Plan
   - Indexing Tips
   - Performance Tuning Basics

20. Data Export and Import (Basics)
   - SPOOL, SQL*Loader, INSERT INTO Techniques

21. Real-Time SQL Use Cases
   - ETL Queries
   - Report Generation
   - Data Validation

22. OLTP vs OLAP

Part V – PL/SQL Programming

23. Introduction to PL/SQL
    - What is PL/SQL
    - SQL vs PL/SQL
    - Anonymous & Named Blocks
    - PL/SQL Syntax

24. Variables and Data Types
    - Constants, Scalar Types
    - Anchored Types (%TYPE, %ROWTYPE)
    - Constants and Scope

25. Control Structures
    - IF, CASE
    - LOOP, FOR, WHILE, Nested Loops
    - Operators

26. Procedures and Functions
    - IN, OUT, IN OUT Parameters
    - Subprograms
    - RETURN Values

27. Cursors
    - Implicit / Explicit
    - Cursor FOR Loop
    - Parameterized / REF Cursors

28. Exception Handling
    - Predefined Exceptions
    - User-defined Exceptions
    - RAISE / RAISE_APPLICATION_ERROR

29. Packages
    - Package Specification / Body
    - Public vs Private Members

30. Triggers
    - BEFORE / AFTER / INSTEAD OF
    - Row-Level / Statement-Level
    - Compound Triggers

31. Collections
    - RECORD, Nested Table, VARRAY
    - Associative Arrays

32. Dynamic SQL

- EXECUTE IMMEDIATE, DBMS_SQL

33. Bulk Operations
    - BULK COLLECT, FORALL, SAVE EXCEPTIONS

34. Advanced PL/SQL Topics
    - UTL_FILE, DBMS_SCHEDULER, DBMS_JOB
    - Autonomous Transactions, Sequences, Synonyms
    - Object Types

35. Debugging and Logging
    - DBMS_OUTPUT
    - SQL Developer Debugger
    - Logging Techniques

36. Real-Time PL/SQL Applications
    - Reusable Packages
    - Backend Jobs
    - Complex Business Logic
    - Project Case Studies

Additional Concepts

37. Data Modeling Concepts
    - Normalization
    - ER Diagrams

38. Performance Tuning Best Practices

39. Metadata Tables
    - USER_TABLES, ALL_OBJECTS, DBA Views

40. PL/SQL Code Reusability Best Practices

SQL

🔷 Introduction to Data :

🔹 What is Data?

- Data means raw facts or figures.
  - 💡 Examples*:* Names, phone numbers, sales records.
- In SQL/database terms, data refers to information stored in a structured format, usually in tables.
- A table is made up of:
  - Rows (also called records)
  - Columns (each holds a specific type of data)

💡 Example Table :

ID	Name	Age	Department
1	Raj	28	HR
2	Simran	30	IT

- Properly stored and organized data is valuable for analysis and decision-making.

🔹 Difference Between Data and Information :

- Data = Raw, unprocessed facts.
  - 💡 Example *:* 85, Raj, IT
- Information = Processed and meaningful data.
  - 💡 Example *:* Raj is 28 years old and works in the HR department.

🔷 What is a Database?

- A database is a place where data is stored in an organized way so it can be easily accessed, managed and updated.
- It helps reduce paperwork and store large amounts of data digitally.

💡 Example : A phone contact list or a student record system.

💻 Syntax to Create a Database :

```
CREATE DATABASE database_name;
```

💡 Example :

```
CREATE DATABASE StudentDB;
```

🔹 Why do we need a Database?

- To store data securely without paper files.
- To reduce duplication, errors, and data loss.
- To easily retrieve and analyze data whenever needed.

🔹 Types of Databases :

- Relational Database (RDBMS) :
  - Stores data in tables (rows and columns).
  - Uses SQL to manage data.
  - 💡 Example : MySQL, Oracle, PostgreSQL.
- NoSQL (Non-relational Database) :
  - Stores data in non-tabular formats: key-value pairs, documents, graphs.
  - Used for big data and real-time apps.
  - 💡 Example : MongoDB, Cassandra, Redis.
- Hierarchical Database :
  - Stores data in a tree-like structure (like folders and subfolders).
  - Used in old systems.
  - 💡 Example : IBM IMS.
- Network Database :
  - Similar to hierarchical, but allows multiple parent-child relationships.
  - More flexible but complex.
  - 💡 Example : Integrated Data Store (IDS), IDMS (Integrated Database Management System) by CA Technologies.
- Object-Oriented Database :
  - Stores data as objects, just like in object-oriented programming.
  - 💡 Example : db4o, ObjectDB.

🔹 Advantages of Using Databases :

- Stores large amounts of data easily
- Prevents duplicate data (reduces redundancy)
- Allows fast search, updates, and reporting
- Provides security and access control
- Supports multi-user access

🔸 Disadvantages of Databases :

- Requires software and hardware resources
- Needs skilled people to manage (DBA, developers)
- Complex to set up and design
- If not managed well, can cause data corruption or loss

❖ Key Database Terms (Must-Know) :

🔹 What is a Database Object?

- Any item created in the database is called a database object.

💡 Examples: Table, View, Index, Sequence, Procedure, etc.

🔹 What are Schema Objects?

- These are objects inside a schema (a user's area in the database).

💡 Example: Tables, Views, Indexes, Triggers, Synonyms, etc.

📌 *Think of a schema as a folder, and objects as files in that folder.*

🔹 Table :

- Stores data in rows and columns.
- Every table has a name and a set of columns with data types.

🔹 View :

- A virtual table based on a SELECT query.
- Doesn't store data, only displays it from other tables.

🔹 Index :

- Speeds up data search (like a book index).
- Improves performance while searching records.

🔹 Sequence :

- Generates a series of numbers, often used to create unique IDs (like auto-increment).

🔹 Synonym :

- A shortcut name for another object.
- Useful to simplify access to tables or views from other schemas.

🔹 Stored Procedure :

- A saved set of SQL commands.
- Helps to automate repeated tasks.

🔹 Trigger :

- A set of instructions that runs automatically when an event happens in a table (like insert, update, delete).

🔹 Cursor :

- Used to go through each row one by one.
- Mostly used in stored procedures when row-by-row operations are needed.

📌 *Note : These topics are explained in detail in the upcoming chapters*

What is DBMS (Database Management System)?

- DBMS is software used to store, manage, and retrieve data in a database.
- It makes sure the data is safe, consistent, and easily accessible.

Examples : MySQL, Oracle, SQL Server, PostgreSQL.

What is an RDBMS (Relational Database Management System)?

- RDBMS stores data in tables with rows and columns.
- Each table has a unique identifier (Primary Key).
- RDBMS supports relationships between tables (like links between student and marks tables).

Example : MySQL, Oracle, PostgreSQL.

What is SQL (Structured Query Language)?

- It is used to interact with databases – to insert, update, delete, or retrieve data.
- SQL is a standard language supported by all major Relational Database Management Systems (RDBMS) like Oracle, MySQL, SQL Server, PostgreSQL, etc.
- SQL is both human-readable and machine-readable, making it easy to learn and powerful to use.
- It works on structured data (organized in rows and columns using tables).

How does SQL help?

- You write SQL commands (queries) to talk to the database.
- SQL helps in :
    - Creating database objects like tables, views, indexes, procedures, etc.
    - Inserting new records into tables.
    - Fetching reports or data using SELECT queries.
    - Filtering, sorting, and grouping data to get meaningful insights.
    - Updating or deleting old/unwanted data.
    - Giving or restricting access using privileges and roles.
    - Supporting data integrity, consistency, and security in databases.

🔷 Schema in a Database

- A schema is like a folder in the database that contains related objects like tables, views, indexes, and procedures.
- It helps organize data and control access (who can see or use what).
- Each user in a database usually has their own default schema.

📌 Think of a schema as :

👉 A *library (database) and library section* (schema) that contains *books* (tables, views, etc.).

🔹 SQL Command to Create a Schema (in databases that support it like PostgreSQL,SQL Server, Oracle):

💻 Syntax :

```
CREATE SCHEMA schema_name;
```

- Then, you can create tables inside the schema like this :

```
CREATE TABLE schema_name.table_name (Col_name1  data_type,Col_name2 data_type,...n);
```

💡 Example :

```
CREATE SCHEMA hr;
CREATE TABLE hr.employees (id INT, name VARCHAR(50));
```

📌 *In MySQL, schemas and databases are the same thing.*

📌 *But in Oracle, SQL Server, PostgreSQL, schemas are like folders inside a database suppose database is the main container.*

🔷 SQL Variables

- Variables are temporary storage locations that store values while a program or query runs.
- Variables are NOT supported in standard SQL (like the basic SQL you use in MySQL, PostgreSQL, etc.) for general queries.
- SQL is a *declarative language*, mainly used for querying and modifying data — not for procedural logic like loops, conditions, or variables.
- Variables ARE supported — especially in procedural extensions like:
  - PL/SQL (Oracle)
  - T-SQL (SQL Server)
  - Stored Procedures / Functions in MySQL/PostgreSQL

📌 *In standard SQL, you write queries like:*

```
SELECT * FROM employees WHERE department_id = 10;
```

➡ No variable used here.

🔹 Why use variables?

- To hold intermediate values
- To reuse values in multiple statements
- To make procedures or functions dynamic

📌 *You use variables to store temporary data, control flow, loops, conditions, etc.*

🔹 What is a Bind Variable?

- A bind variable is a placeholder in an SQL query where you don't hard-code the value but instead pass the value later when the query is executed.
- Best Practice → Use meaningful names so it's easy to understand later.
- It does NOT have to match the column name, but should clearly describe the value.
- → :id, :param1, :userInput, etc.
- It helps in:
  - Preventing SQL Injection
  - Reusing the same SQL query with different values
  - Improving performance (by reducing parsing overhead)

💡 Example : Without Bind Variable (Bad Practice)

```
SELECT *
FROM employees
WHERE employee_id = 101;
```

- 👉 Here, 101 is hard-coded.
- Problem: Every time the value changes, the query needs to be parsed again → Performance hits.

💡 Example : With Bind Variable (Good Practice)

```
SELECT *
FROM employees
WHERE employee_id = :emp_id;
```

- 👉 :emp_id is a bind variable (placeholder).
- At runtime, you can pass different values (like 101, 102, 103, etc.) without changing the query itself.

💠 Why use :empid?

- Because in applications (like Oracle Forms, PL/SQL blocks, or other front-end programs) you don't hardcode the employee number.
- Instead, you pass it dynamically.
- For example, if you want to show department history for employee 101, you pass:
  :empid = 101
- If you want to show for employee 205, you pass:
  :empid = 205
- ✅ So the query becomes:

```
SELECT e.emp_no,
     d.dept_name,
     de.from_date,
     de.to_date
FROM   department d
JOIN   depet_no de  ON d.dept_no = de.dept_no
JOIN   employee e   ON e.emp_no = de.emp_no
WHERE  e.emp_no = :empid;
```

➡ When you run this in an application, the system asks you to enter the value for empid, or your program will supply it automatically.

🔷 SQL Data Types

- SQL Data Types define what kind of data a column can store, like numbers, text, or dates.
- They help in :
    - Data Accuracy (right type of data)
    - Efficient Storage (use minimum space)
    - Better Performance (faster queries)

🧠 Think of it like:

Data types are rules that say:
"This column will store numbers only" or "This column will store text only."

◇ SQL Data Types :

◇ 1. Numeric Data Types :

- Numeric data types are used to store numbers like whole numbers and decimals.
- They support math operations (+, –, ×, ÷,%), so they are useful in finance, science, and data analysis.

Data Type	Description	Supported Databases
INT / INTEGER	Stores whole numbers	Oracle, MySQL, SQL Server
NUMBER(p,s)	General-purpose numeric type with precision	Oracle
DECIMAL(p,s)	Exact decimal values (finance)	MySQL, SQL Server, Oracle
FLOAT	Approximate numeric values	MySQL, SQL Server, Oracle

◇ 2. Character and String Data Types :

- Character data types are used to store text or character-based data.
- The choice between fixed-length and variable-length data types depends on the nature of your data.

Data Type	Description	Supported Database
CHAR(n)	Fixed-length character string	Oracle, MySQL, SQL Server
VARCHAR(n)	Variable-length string	MySQL, SQL Server
VARCHAR2(n)	Oracle-specific variable-length string	Oracle
Text	Large text data	MySQL, SQL Server *(deprecated)*

◇ 3. Date and Time Data Type :

- SQL provides several data types for storing date and time information.
- They are essential for managing timestamps, events, and time-based queries. These are given in the below table.

Data Type	Description	Supported Databases
DATE	stores the data of date (year, month, day)	Oracle, MySQL, SQL Server
TIMESTAMP	Date & time with high precision	Oracle, MySQL, SQL Server
DATETIME	store both the data and time (year, month, day, hour, minute, second)	MySQL, SQL Server

◇ 4. Binary Data Types in SQL :

- Binary data types are used to store binary data such as images, videos, or other file types. These include:

Data Type	Description	Max Length
Binary	Fixed-length binary data.	8000 bytes
VarBinary	Variable-length binary data.	8000 bytes
Image	Stores binary data as images.	2,147,483,647 bytes

◇ 5. Boolean Data Type in SQL :

- The BOOLEAN data types are used to store logical values, typically TRUE or FALSE. It's commonly used for flag fields or binary conditions.

Data Type	Description	Supported Databases
BOOLEAN	Stores a logical value (TRUE/FALSE).	MySQL, **PL/SQL only**

💡 Example of Oracle SQL Data Types :

```
CREATE TABLE Employees(
   EmpID NUMBER,                -- Numeric data type (whole number)
   FirstName VARCHAR2(50),      -- Variable-length character string
   LastName VARCHAR2(50),       -- Variable-length character string
   Salary NUMBER(10, 2),        -- For precise numeric values like salary
   JoinDate DATE,               -- Date datatype (includes time by default in Oracle)
   IsActive CHAR(1)             -- Used for Boolean-like flag ('Y' or 'N')
);
```

🔷 SQL Operators

- SQL Operators are special symbols or keywords used to perform operations on data values in SQL queries.
- They are mostly used in WHERE, HAVING, and SELECT clauses to filter, compare, and manipulate data.

🔹 Types of SQL Operators :

- Arithmetic Operator
- Comparison Operator
- Logical Operator
- Set Operators
- Special Operators

◇ 1. Arithmetic Operators :

- Used to perform basic mathematical operations and return a numeric result.

Operator	Description	Example
+	Addition	SELECT 5 + 2 FROM DUAL; → 7
-	Subtraction	SELECT 10 - 4 FROM DUAL; → 6
*	Multiplication	SELECT 3 * 3 FROM DUAL; → 9
/	Division	SELECT 10 / 2 FROM DUAL; → 5
% *(Modulus)*	Remainder (some RDBMS only)	SELECT MOD(10, 3) FROM DUAL; → 1

◇ 2. Comparison Operators :

- Used to compare two values. Mostly used in WHERE clause.
- Returns a Boolean result (TRUE/FALSE) internally for each row.
- != and <> both mean "not equal". Use based on DBMS compatibility.

Operator	Description	Example
=	Equal to	WHERE salary = 50000
!= or <>	Not equal to	WHERE city != 'Pune'
>	Greater than	WHERE age > 25
<	Less than	WHERE salary < 60000
>=	Greater than or equal	WHERE marks >= 35
<=	Less than or equal	WHERE age <= 30

◇ 3. Logical Operators :

- Used to combine multiple conditions in the WHERE clause.

Operator	Description	Example
AND	Returns TRUE if all conditions are true	WHERE age > 25 AND city = 'Pune'
OR	Returns TRUE if at least one condition is true	WHERE dept = 'HR' OR dept = 'IT'
NOT	Reverses the condition	WHERE NOT city = 'Mumbai'

◇ 4. Set Operators :

- Used to combine results from two or more SELECT queries.
- These require both queries to have the same number of columns and compatible data types.

Operator	Description	Usage
UNION	Combines unique rows from both queries	SELECT city FROM A UNION SELECT city FROM B
UNION ALL	Includes duplicates too	SELECT city FROM A UNION ALL SELECT city FROM B
INTERSECT	Returns common rows	SELECT city FROM A INTERSECT SELECT city FROM B
MINUS	Returns rows from first query not in second	SELECT city FROM A MINUS SELECT city FROM B

◇ 5. Special Operators :

- These are helpful for advanced filtering in WHERE clauses.

Operator	Description	Example
BETWEEN	Checks within a range	WHERE age BETWEEN 18 AND 30
IN	Checks if value exists in list	WHERE city IN ('Pune', 'Mumbai')
NOT IN	Checks if value doesn't exist	WHERE dept NOT IN ('HR', 'IT')
LIKE	Pattern matching (with %, _)	WHERE name LIKE 'V%'
IS NULL	Checks for NULL	WHERE salary IS NULL
IS NOT NULL	Checks for non-NULL	WHERE city IS NOT NULL
EXISTS	Checks if subquery returns rows	WHERE EXISTS (SELECT * FROM dept WHERE id = 10)

Example :

```
SELECT
        employee_id,
        first_name || ' ' || last_name AS full_name,
        salary,
        salary * 12 AS annual_salary,
        Department_id
FROM employees
WHERE (salary BETWEEN 30000 AND 80000)
        AND department_id IN (10, 20, 30)
        AND last_name LIKE 'S%'
        AND manager_id IS NOT NULL
        AND salary + 5000 > 40000;
```

What is the DUAL Table in Oracle?

- The DUAL table is a special, built-in table in Oracle Database that is used to perform calculations or return a value without querying any real table.

Key Points About DUAL Table :

1. Single Row and Single Column :
   - The DUAL table has only one row and one column called DUMMY.
   - This column contains a single value: 'X'.

   Example :

```
SELECT * FROM DUAL;
-- Output: X
```

2. Used for Expressions and Functions :
   - It's commonly used when you want to evaluate an expression, function, or calculation without using any actual table.

   Example :

```
SELECT 10 + 5 FROM DUAL;
-- Output: 15
```

3. System Table :
   - DUAL is part of the data dictionary, created by Oracle.
   - You don't need to create or manage it. It's always available.

4. Use with Functions
   - You can test SQL functions easily using DUAL.

   Example :

```
SELECT SYSDATE FROM DUAL;
    -- Returns current system date

SELECT UPPER('vaibhav') FROM DUAL;
    -- Output: VAIBHAV
```

5. Use with Pseudo Columns :
   - You can also use pseudo-columns like SYSDATE, USER, ROWNUM with DUAL.

   Example :

```
SELECT USER FROM DUAL; -- Returns current username
```

6. Why Not Use Real Tables?
   - Using DUAL avoids unnecessary reads from real data tables when:
     - You just want to display a constant
     - You want to test a formula or function
     - You are writing a simple script for checking syntax or results

7. Other Databases :
   - DUAL is specific to Oracle.
   - In MySQL, you can write SELECT 5+2; without DUAL.
   - In SQL Server, use SELECT 5+2; as well.

8. Can You Insert or Modify DUAL?
   - No. You should not modify the DUAL table.
   - Oracle maintains it, and any change can break system behavior.

Examples Using DUAL :

Purpose	Query
Simple arithmetic	SELECT 100 * 2 FROM DUAL;
Current system date	SELECT SYSDATE FROM DUAL;
String manipulation	SELECT INITCAP('vaibhav') FROM DUAL;
Rounding numbers	SELECT ROUND(10.567, 2) FROM DUAL;
Concatenation	`SELECT 'Vaibhav'

Real-World Scenario :

- Imagine you're testing logic in a stored procedure or PL/SQL script. Before applying on real data :

Example :

```
-- Check discount formula
SELECT 1000 - (1000 * 10 / 100) AS DiscountedPrice FROM DUAL;
```

➞ 900

🔷 SQL COMMENTS:

🔶 What are SQL Comments?

- SQL Comments are non-executable statements used to explain and document your SQL code.
- They help make your code more readable and easier to understand — especially when revisiting later or sharing with others.

💻 Types of Comments in SQL:

- SQL supports two types of comments:

Type	Syntax	Description
◇ 1. Single-line comment	-- comment	Comment that spans one line
◇ 2. Multi-line comment	/* comment block */	Comment that can span multiple lines

💡 Examples:

Single-line Comment:

```
-- This query fetches all employees
SELECT * FROM employees;
```

Multi-line Comment:

```
/* This query fetches all employees
who have a salary more than 50000 */
SELECT * FROM employees
WHERE salary > 50000;
```

📝 Important Notes:

- Comments are ignored by the SQL engine — they do not affect query results.
- Use comments to explain:
  - Complex logic
  - Temporary conditions
  - Testing sections
- Over-commenting can make code messy. Use wisely!

🔷 SQL ALIASES :

🏵 What is an Alias?

- An alias is a temporary nickname given to a column or table in a SQL query to make the output more readable.
- It helps you:
  - Shorten long table/column names
  - Format output headers
  - Improve code readability

💻 Syntax of Column Alias

```
SELECT column_name AS alias_name
FROM table_name;
```

📝 You can also omit AS keyword:

```
SELECT column_name alias_name
FROM table_name;
```

💡 Example – Column Alias

```
SELECT first_name AS Name, salary AS Income
FROM employees;
```

📜 Output:

```
Name     | Income
---------|--------
John     | 50000
```

💻 Syntax of Table Alias

```
SELECT t.column_name
FROM table_name AS t;
```

📝 OR (without AS):

```
SELECT t.column_name
FROM table_name t;
```

💡 Example – Table Alias

```
SELECT e.first_name, d.department_name
FROM employees e
JOIN departments d ON e.department_id = d.department_id;
```

🔹 Here, e and d are aliases for employees and departments.

📝 Important Notes:

- Aliases are temporary — they exist only during that query execution.
- Aliases are helpful in:
  - Joins (when multiple tables used)
  - Aggregations (when using functions like COUNT, AVG)
  - Making column headers user-friendly
- You cannot use aliases in WHERE clause — because WHERE is evaluated before SELECT.

🔁 Summary Table:

Feature	Purpose	Used On	Syntax Style
Comments	Make code readable	Any SQL section	-- or /* ... */
Aliases	Rename temporarily (for clarity)	Columns, Tables	AS alias or just alias

🔷 SQL Constraints

- Constraints are rules applied on columns in a table to enforce data integrity (i.e., make sure only valid data is stored in the database).
- They automatically prevent invalid data from being inserted, updated, or deleted.
- Constraints can be added during table creation or later using ALTER TABLE.
- If constraint is violated during INSERT or UPDATE, the SQL query fails with an error.
- All constraints can be named for better management.
  CONSTRAINT unique_email UNIQUE (email)

💻 Syntax for View Existing Constraints :

```
SELECT constraint_name, constraint_type, table_name
FROM user_constraints
WHERE table_name = 'YOUR_TABLE_NAME';
```

👉 You only need to write the table name, but here are important things to remember:

- Table name should be in UPPERCASE
- Don't include schema name

💡 Example How to View Constraints :

```
-- For all constraints
SELECT * FROM user_constraints WHERE table_name = 'EMPLOYEES';

-- For columns with constraints
SELECT * FROM user_cons_columns WHERE table_name = 'EMPLOYEES';
```

🔹 Types of SQL Constraints :

Constraint	Description
NOT NULL	Ensures that a column cannot have a NULL value and It can only be applied at the column level, not at the table level.
UNIQUE	Ensures all values in a column are unique (no duplicates).
PRIMARY KEY	Uniquely identifies each row. Cannot be NULL + must be unique.
FOREIGN KEY	Connect two tables. Maintains referential integrity between tables.
CHECK	Limits the values based on a condition.
DEFAULT	Sets a default value if no value is given for that column.

◇ 1. NOT NULL Constraint :

- The NOT NULL constraint ensures that a column cannot have a NULL value.
- It guarantees that every row must contain a valid (non-empty) value in this column.

💡 Example :

```
CREATE TABLE employees(
    emp_id INT NOT NULL,
    name VARCHAR(50) NOT NULL
);
```

◇ 2. UNIQUE Constraint :

- Ensures values in the column are all different / Unique.
- Can be used at both column and table level.
- Multiple UNIQUE constraints allowed in a table.
- NULL is allowed once per column with UNIQUE.

💡 Example :

```
CREATE TABLE students(
    student_id INT UNIQUE,
    email VARCHAR(100) UNIQUE
);
```

◇ 3. PRIMARY KEY Constraint :

- Uniquely identifies each record in a table. Combines NOT NULL + UNIQUE.
- Only one PRIMARY KEY allowed per table.
- Can span multiple columns → called Composite Key.

💡 Example :

```
CREATE TABLE customers(
    cust_id INT PRIMARY KEY,
    name VARCHAR(100)
);
```

📝 You can also use composite keys :

💻 Syntax :

```
PRIMARY KEY (col1, col2)
```

💡 Composite PK Example :

```
CREATE TABLE project_assignment (
  emp_id INT,
  project_id INT,
  PRIMARY KEY (emp_id, project_id)
);
```

◇ 4. FOREIGN KEY Constraint :

- Links two tables together. The foreign key refers to the primary key of another table.
- Enforces referential integrity.
- You cannot insert a value unless it exists in the referenced table.
- You cannot delete a parent row if children exist, unless ON DELETE CASCADE is used.

💡 Example :

```
CREATE TABLE orders (
  order_id INT PRIMARY KEY,
  cust_id INT,
```

```
 FOREIGN KEY (cust_id) REFERENCES customers(cust_id)
);
```

📝 Prevents deleting a customer from customers if orders exist in orders.

💻 Basic FK Syntax :

```
FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
```

💡 Example :

```
FOREIGN KEY (cust_id) REFERENCES customers(cust_id)
ON DELETE CASCADE
```

📝 ON DELETE CASCADE → deletes child rows when parent is deleted.

📝 ON DELETE SET NULL → sets child FK to NULL when parent is deleted.

◇ 5. CHECK Constraint :

- Validates values using a condition.
- Used to limit column values based on logic.
- Can be applied at column or table level.

💡 Example :

```
CREATE TABLE products (
product_id INT PRIMARY KEY,
price DECIMAL(10, 2) CHECK (price > 0)
);
```

- Supports multiple conditions.

💡 Example :

```
CHECK (salary > 0 AND salary < 100000)
```

📌 Some databases (like older MySQL versions) may ignore CHECK constraints unless explicitly enabled.

◇ 6. DEFAULT Constraint :

- Sets a default value if none is provided.
- Automatically applies a value if the user does not provide one.

💡 Example :

```
CREATE TABLE accounts (
acc_id INT PRIMARY KEY,
status VARCHAR(10) DEFAULT 'Active'
);
```

🔁 Summary Table : CONSTRAINTS

Constraint	Description	Can be applied on	Default behavior
NOT NULL	The column must have a value. Can't be left empty.	Column only	Cannot insert or update with NULL
UNIQUE	Each value in the column (or set of columns) must be different.	Column/Table	Allows one NULL (Oracle)
PRIMARY KEY	Unique + Not Null. Only one per table. Used to uniquely identify rows.	Column/Table	Cannot contain NULL, always unique

FOREIGN KEY	Links one table's column to another table's primary key. Enforces referential integrity.	Column/Table	Rejects insert/update if reference not found
CHECK	Validates data against a logical condition.	Column/Table	Rejects rows failing the condition
DEFAULT	Assigns a default value when no value is provided.	Column only	Automatically inserts the default

---

🔷 SQL KEYS :

- SQL keys are used to uniquely identify rows, establish relationships, enforce uniqueness and to maintain data integrity

◇ 1. Primary Key :

- Main key to identify each record uniquely.
- Only one primary key per table.
- Cannot be NULL.
- Often used with AUTO_INCREMENT (MySQL) or SEQUENCE (Oracle).

💻 Syntax :

```
CREATE TABLE students (
  student_id NUMBER PRIMARY KEY,
  name VARCHAR2(100)
);
```

📝 In Oracle, you can also name it:

```
CONSTRAINT pk_students PRIMARY KEY(student_id)
```

◇ 2. Unique Key :

- Ensures all values in a column (or group of columns) are unique.
- Allows one NULL (Oracle).
- Multiple unique keys can exist in one table.

💻 Syntax :

```
CREATE TABLE employees (
  emp_id NUMBER PRIMARY KEY,
  email VARCHAR2(100) UNIQUE
);
```

◇ 3. Foreign Key :

- Used to link two tables together.
- Refers to the Primary key in another table.
- Prevents deletion/updates if linked data exists.

💻 Syntax :

```
CREATE TABLE departments (
  dept_id NUMBER PRIMARY KEY,
  dept_name VARCHAR2(50)
);

CREATE TABLE employees (
```

```
  emp_id NUMBER PRIMARY KEY,
  name VARCHAR2(100),
  dept_id NUMBER,
  CONSTRAINT fk_dept FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);
```

◇ 4. Composite Key :

- Primary Key made up of multiple columns.
- Ensures that the combination of values is unique.

💻 Syntax:

```
CREATE TABLE course_enrollments (
  student_id NUMBER,
  course_id NUMBER,
  enrollment_date DATE,
  PRIMARY KEY (student_id, course_id)
);
```

◇ 5. Candidate Key :

- A column (or group of columns) that can qualify as a Primary Key.
- A table can have many candidate keys.
- One of them is chosen as the Primary Key, others remain as alternate keys.
- All candidate keys are unique and not null.

💡 Example :

```
-- Assume both emp_id and email can uniquely identify employees
emp_id     -- candidate key (chosen as primary)
email      -- candidate key (becomes alternate)
```

📌 One candidate key is selected as primary, others become alternate keys.

◇ 6. Alternate Key :

- A candidate key that was not selected as the primary key.
- Still unique and NOT NULL.

💡 Example :

```
CREATE TABLE users (
  user_id NUMBER PRIMARY KEY,
  username VARCHAR2(50) UNIQUE  -- alternate key
);
```

◇ 7. Super Key :

- Any set of columns that uniquely identifies a row.
- Includes Primary Key, Composite Key, Candidate Keys, etc.
- May contain extra columns (not minimal).

💡 Example:

```
-- student_id is primary key
-- student_id + name = super key (but not minimal)
```

💡 Example of Keys and Constraints All Together :

```
CREATE TABLE employees (
  emp_id INT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
```

```
  email VARCHAR(100) UNIQUE,
  dept_id INT,
  salary DECIMAL(10,2) CHECK (salary > 0),
  status VARCHAR(10) DEFAULT 'Active',
  FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);
```

📝 Important Notes:

- Always define primary and foreign keys for data integrity.
- Use unique keys to enforce business rules (e.g., unique email).
- Foreign keys help maintain relationships between tables.
- Choose minimal columns for Primary Keys (avoid extra columns).
- Oracle allows one NULL in UNIQUE constraints.

🔁 Summary Table :

Key Type	Unique	NULL Allowed	Count per Table	Purpose
Primary Key	Yes	❌ No	One	Uniquely identifies a record
Unique Key	Yes	✅ One (Oracle)	Many	Enforce uniqueness
Foreign Key	No	✅ Yes	Many	Link to another table
Composite Key	Yes	❌ No	One	Unique combo of multiple columns
Candidate Key	Yes	❌ No	Many	Eligible for Primary Key
Alternate Key	Yes	❌ No	Many	Not chosen as Primary Key
Super Key	Yes	❌ No	Many	All keys that uniquely identify

🔷 DDL (Data Definition Language) in SQL :

🏵 What is DDL?

- DDL stands for Data Definition Language.
- It includes SQL commands that define, modify, or delete database objects such as tables, schemas, indexes, views, etc.

◇ DDL Commands :

DDL Command	Description
CREATE	Creates a new table or database object.
ALTER	Modifies an existing database object.
DROP	Deletes database objects permanently.
TRUNCATE	Removes all records from a table quickly (faster than DELETE).
RENAME	Renames a table or object.

◇ 1. CREATE TABLE :

- The CREATE command is used when you want to make something new in the database.
- This could be a table, a view, a sequence, a schema, etc.
- When you create a table, you define the structure — like column names, data types, and constraints.
- Once a table is created, it will exist until you delete it explicitly using DROP.

💻 Syntax :

```
CREATE TABLE table_name (
        column1 datatype,
        column2 datatype,
        …
);
```

💡 Example :

```
CREATE TABLE employees (
        emp_id NUMBER PRIMARY KEY,
        name VARCHAR2(50),
        salary NUMBER
);
```

📌 Note : Table is created with three columns. emp_id is marked as Primary Key.

◇ 2. ALTER TABLE :

- The ALTER command is used when you need to change the structure of an existing object like a table.
- You can add a new column, delete a column, rename a column, or change a column's data type.
- You can also use it to add or remove constraints.
- It's a powerful command used during design updates or changes to business needs.

💻 Syntax:

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

💡 Example:

```
ALTER TABLE employees
ADD department VARCHAR2(30);
```

📌 Note: Adds a new column department to the existing table.

◇ 3. DROP TABLE :

- The DROP command is used to completely delete a database object like a table, view, or sequence.
- Once something is dropped, it's gone permanently — along with all of its data.
- You should use it carefully because it cannot be rolled back (undone).

💻 Syntax:

```
DROP TABLE table_name;
```

💡 Example:

```
DROP TABLE employees;
```

📌 Note: Deletes the table and all of its data permanently.

◇ 4. TRUNCATE TABLE :

- The TRUNCATE command is used to quickly delete all data from a table, but keep the table structure intact.
- Unlike DELETE, it does not log each row deletion, so it's faster.
- However, it cannot be rolled back, and you can't use a WHERE clause with it.

💻 Syntax:

```
TRUNCATE TABLE table_name;
```

💡 Example:

```
TRUNCATE TABLE employees;
```

📌 Note: Removes all records from the table but does not delete the table.

📝 Tip: TRUNCATE is faster than DELETE and cannot be rolled back.

◇ 5. RENAME :

- The RENAME command is used to change the name of a table or other database object.
- It's useful if you want your naming to reflect updated business logic.
- The structure and data remain the same — only the object's name changes.

💻 Syntax :

```
RENAME old_table_name TO new_table_name;
```

💡 Example :

```
RENAME employees TO emp_details;
```

📌 Note: Renames the table from employees to emp_details.

🔷 DML (Data Manipulation Language) in SQL :

🏵 What is DML :

- DML stands for Data Manipulation Language.
- It is used to add, update, delete, and retrieve data from database tables.
- These commands do not change the structure of the table — they only work with the data inside.

◇ 1. INSERT – Add New Data

- This command is used when you want to add new records (rows) into a table.
- Think of it like filling out a new entry in a register — every detail is fresh and complete.
- You use it when new employees join, when you get a new order, or when new products are added.
- All required (NOT NULL) fields must be filled when inserting a new row.

💻 Syntax:

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

💡 Example:

```
INSERT INTO employees (emp_id, name, salary)
VALUES (101, 'Vaibhav', 55000);
```

📌 Note:

- All NOT NULL columns must be included in INSERT.
- You can insert multiple rows using INSERT ALL (Oracle) or bulk methods.

◇ 2. UPDATE – Modify Existing Data

- This command helps you modify existing data in your table.
- For example, if someone's address changes, or an employee gets a salary hike, you use UPDATE to reflect those changes.
- It's important to carefully choose which records to update, or you might change data in multiple rows by accident.
- Used when data is outdated or incorrect and needs fixing.

💻 Syntax:

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

💡 Example:

```
UPDATE employees
SET salary = 60000
WHERE emp_id = 101;
```

📌 Note: Always use a WHERE clause to avoid updating all records by mistake.

📝 Tip: Without WHERE, all rows will be updated.

◇ 3. DELETE – Remove Data

- This command is used to remove specific rows from a table.
- You might delete a customer record if they've unsubscribed or remove an order that was canceled.

- Like UPDATE, it should be used with conditions to avoid deleting everything by mistake.
- Unlike TRUNCATE (a DDL command), DELETE allows you to remove only specific records.

💻 Syntax:

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

💡 Example:

```
DELETE FROM employees
WHERE emp_id = 101;
```

📌 Note:

- Like UPDATE, ALWAYS use a WHERE clause.
- If WHERE is skipped, all rows will be deleted.

📝 Important:

- DELETE is slower than TRUNCATE because it logs individual row deletions and can be rolled back.

◇ 4. SELECT – Read Data

- This is the most commonly used command. It's used to retrieve data from a table — you're just reading the data, not changing it.
- It helps you answer questions like: "Who are the top 5 customers?", "What are the total sales this month?", or "Which products are low in stock?"
- Used to retrieve specific columns, filter rows, group results, and sort data.
- It's the foundation for reports, dashboards, and analysis.

💻 Syntax:

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

💡 Example:

```
SELECT name, salary
FROM employees
WHERE salary > 50000;
```

📌 Note:

- Use * to select all columns: SELECT * FROM employees;
- You can use WHERE, ORDER BY, GROUP BY, HAVING, etc., for advanced queries.

📝 Important Notes:

- DML commands affect the data, not the table structure.
- INSERT, UPDATE, and DELETE can be rolled back if something goes wrong (if not yet committed).
- SELECT is read-only and does not change data.
- These commands are often used in applications, reports, dashboards, and any system that interacts with a database.

🔁 Summary of DML Commands:

Command	Action	Can Be Rolled Back?
INSERT	Add new records	✅ Yes
UPDATE	Modify existing records	✅ Yes
DELETE	Remove specific records	✅ Yes
SELECT	Retrieve data (read-only)	❌ Not needed

🔷 DQL (Data Query Language) :

🏵 What is DQL?

- DQL stands for Data Query Language.
- It is used to fetch (retrieving) data from the database.
- It doesn't change the structure or content of the data
- They are read-only operations — they don't insert, update, or delete anything.

📌 Notes:

- It plays a major role in data analysis, reporting, and business intelligence.
- Other SQL command types (DDL, DML, DCL, TCL) may prepare or modify data, but DQL is used to understand and explore data.
- ✔ There is mainly one command in DQL: SELECT

◇ DQL Commands :

◇ 1. SELECT – Retrieve Data from Tables

- Used to get data from one or more tables in a database..
- You can select specific columns, all columns, or even calculated values.
- It supports a wide range of options like filtering, sorting, grouping, joining, and limiting data.
- It is a read-only command and doesn't modify data.

🔍 Real-Life Use Cases:

- Viewing all customers from a specific city.
- Getting a list of employees with salaries above a certain amount.
- Checking daily sales reports.
- Generating dashboards or analytics reports.
- Reading data for machine learning models or data analysis.

📌 Notes: Works closely with WHERE, GROUP BY, ORDER BY, HAVING, LIMIT.

◇ 1. Basic SELECT

💻 Syntax:

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

💡 Example:

```
SELECT first_name, last_name FROM employees;
```

📌 Notes:

- Use * to select all columns: SELECT * FROM employees;

◇ 2. DISTINCT – Remove Duplicates

- Used to return only unique values from a column.
- It removes all duplicate rows in the result set.

💻 Syntax:

```
SELECT DISTINCT column_name FROM table_name;
```

💡 Example:

```
SELECT DISTINCT department FROM employees;
```

📌 Notes:

- Useful when you want unique values from a column.
- Applies only to selected columns.

◇ 3. WHERE – Filter Rows

- Used to filter records based on a given condition.
- Only rows that satisfy the condition will be selected.

💻 Syntax:

```
SELECT column1 FROM table_name WHERE condition;
```

💡 Example:

```
SELECT * FROM employees WHERE salary > 50000;
```

📌 Notes:

- Supports operators: =, >, <, >=, <=, !=, BETWEEN, LIKE, IN, IS NULL
- Use AND, OR, NOT for multiple conditions.

◇ 4. ORDER BY – Sort Results

- Used to sort the result set by one or more columns.
- Sorting can be in ascending (ASC) or descending (DESC) order.

💻 Syntax:

```
SELECT * FROM table_name ORDER BY column1 [ASC|DESC];
```

💡 Example:

```
SELECT * FROM employees ORDER BY salary DESC;
```

📌 Notes:

- The default is ASC (ascending).
- Can sort by multiple columns.

◇ 5. GROUP BY – Group Rows for Aggregation

- Used to group rows that have the same values in specified columns.
- Often used with aggregate functions like SUM(), COUNT(), AVG(), etc.

💻 Syntax:

```
SELECT column, AGG_FUNC(column2) FROM table_name GROUP BY column;
```

💡 Example:

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

📌 Notes:

- Used with aggregation functions like COUNT(), SUM(), AVG(), MAX(), MIN()
- All columns in SELECT must either be in GROUP BY or an aggregated result.

◇ 6. HAVING – Filter After Grouping

- Used to filter groups created by GROUP BY.
- It works like WHERE, but is applied after grouping.

💻 Syntax:

```
SELECT column, AGG_FUNC(col) FROM table GROUP BY column HAVING
condition;
```

💡 Example:

```
SELECT department, COUNT(*) FROM employees GROUP BY department
HAVING COUNT(*) > 5;
```

📌 Notes: WHERE filters before grouping. & HAVING filters after grouping.

◇ 7. LIMIT / TOP / ROWNUM

- Restrict Number of Rows
  - MySQL/PostgreSQL: LIMIT
  - SQL Server: TOP
  - Oracle: ROWNUM

💡 Example (MySQL):

```
SELECT * FROM employees LIMIT 5;
```

💡 Example (SQL Server):

```
SELECT TOP 3 * FROM employees;
```

💡 Example (Oracle):

```
SELECT * FROM employees WHERE ROWNUM <= 3;
```

◇ 8. JOINS – Combine Data from Multiple Tables

- Used to combine rows from two or more tables based on a related column.

💻 Syntax (INNER JOIN):

```
SELECT a.col1, b.col2 FROM table1 a
JOIN table2 b ON a.common_column = b.common_column;
```

💡 Example:

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

🔹 Types of Joins:

- INNER JOIN: Only matching rows
- LEFT JOIN: All rows from left, matched from right
- RIGHT JOIN: All from right, matched from left
- FULL JOIN: All rows from both tables
- SELF JOIN: Join table with itself
- CROSS JOIN: Cartesian product

🔍 Difference between WHERE vs HAVING Clause in SQL :

🧩 WHERE Clause	🧩 HAVING Clause
Filters rows before grouping	Filters groups after aggregation
Used with SELECT, UPDATE, DELETE	Used only with SELECT + GROUP BY
Cannot use aggregate functions (SUM(), etc.)	Can use aggregate functions like SUM(), AVG(), COUNT()
Applied before rows are grouped	Applied after GROUP BY has grouped rows
Works on individual records	Works on grouped/aggregated data
Used when no GROUP BY is present	Used when GROUP BY is present
Improves query performance (early filtering)	Typically used for reporting on aggregated results

✅ Simple Analogy:

- WHERE is like filtering raw vegetables before cooking.
- HAVING is like tasting the cooked dish and deciding whether to serve it.

💡 Example Showing Both:

```
-- Select departments with more than 5 employees and salary > 50000
SELECT department, COUNT(*) as emp_count
FROM employees
WHERE salary > 50000      -- Filters individual records
GROUP BY department
HAVING COUNT(*) > 5;      -- Filters grouped results
```

📝 Important Notes:

- SQL is case-insensitive, but it's a good habit to write keywords in UPPERCASE.
- Always use aliases for clarity when joining tables.
- Use GROUP BY with aggregate functions, not WHERE.
- WHERE filters individual rows, HAVING filters aggregated results.

🔁 Summary Table :

Clause	Purpose	Used With
SELECT	Retrieve data	All columns/tables
DISTINCT	Remove duplicate rows	After SELECT
WHERE	Filter raw data	Before grouping
ORDER BY	Sort the result set	At the end
GROUP BY	Group data for aggregation	With aggregate funcs
HAVING	Filter grouped data	After GROUP BY
LIMIT/TOP	Restrict number of rows returned	Last clause
JOIN	Combine rows from tables	ON condition

🔷 Transaction Control Language (TCL) :

🏵 What is TCL?

- TCL stands for Transaction Control Language.
- TCL commands are used to manage transactions in a database.
- A transaction is a group of operations (like INSERT, UPDATE, DELETE) that are executed as a single logical unit of work.
- They ensure data integrity and allow you to accept or discard changes made to the database during a transaction.

◇ TCL Commands :

◇ 1. COMMIT :

- Permanently saves all changes made during the current transaction to the database.
- Once you commit, you cannot undo those changes.

📝 When it's used:

- After a successful set of operations.
- To finalize a transaction so the changes are visible to other users.

📌 Think of COMMIT as saving your work in a Word file.

💻 Syntax:

```
COMMIT;
```

💡 Example:

```
INSERT INTO employees (id, name, salary)
VALUES (101, 'Vaibhav', 50000);
COMMIT;
```

✅ This will permanently save the inserted row in the employees table.

📝 Note: After COMMIT, changes are visible to other users and cannot be rolled back.

◇ 2. ROLLBACK :

- Cancels all changes made in the current transaction.
- Returns the database to the state it was in before the transaction began.

📝 When it's used:

- If something goes wrong (error, wrong data inserted, failed condition).
- To undo any uncommitted changes.

📌 Think of ROLLBACK as pressing 'Undo' or closing a file without saving.

💻 Syntax:

```
ROLLBACK;
```

💡 Example:

```
UPDATE employees
SET salary = 70000
WHERE id = 101;

ROLLBACK;
```

✅ This will undo the salary update for employee 101.

📝 Note: You can only ROLLBACK if you haven't committed yet.

◇ 3. SAVEPOINT :

- Creates a bookmark or checkpoint within a transaction.
- Allows partial rollback to that point instead of rolling back the entire transaction.

📝 When it's used:

- In complex transactions where you want to save progress in steps.
- Useful for selectively undoing certain parts of a transaction.

📌 Think of SAVEPOINT like checkpoints in a game — you can return there if something goes wrong later.

💻 Syntax:

```
SAVEPOINT savepoint_name;
```

💡 Example:

```
SAVEPOINT sp1;

INSERT INTO employees (id, name, salary)
VALUES (102, 'Rahul', 60000);

SAVEPOINT sp2;

UPDATE employees
SET salary = 65000
WHERE id = 102;

ROLLBACK TO sp1;
```

✅ This will undo everything after SAVEPOINT sp1, including the update and insert.

📝 Note: SAVEPOINTs help in partial rollbacks within a larger transaction.

◇ 4. SET TRANSACTION (less commonly used) :

- Configures properties of the current transaction.
- Can specify things like isolation level or read-only mode.

📝 When it's used:

- When you need fine control over how a transaction behaves (e.g., in multi-user environments).

📌 Think of it like setting rules for how your transaction should behave.

💻 Syntax:

```
SET TRANSACTION [READ ONLY | READ WRITE];
```

💡 Example:

```
SET TRANSACTION READ ONLY;
SELECT * FROM employees;
COMMIT;
```

✅ This transaction can only read data, not modify it.

📌 Note: SET TRANSACTION is advanced and rarely used in beginner-level projects. It's mainly useful in multi-user environments.

📝 Important Notes:

- TCL commands only work with DML commands (INSERT, UPDATE, DELETE) — not with DDL.
- If AUTO-COMMIT is ON (like in some tools), then COMMIT happens automatically after every command.
- You should use TCL to maintain data integrity and error recovery in your applications.

🔁 Summary of TCL Commands:

Command	Purpose
COMMIT	Save all changes permanently
ROLLBACK	Undo all changes made in the current session
SAVEPOINT	Create a restore point within a transaction
SET TRANSACTION	Set properties for the current transaction

🔷 What is DCL (Data Control Language)?

🔶 What is DCL ?

- DCL stands for Data Control Language.
- DCL commands are used to control access to data in the database.
- They help the database administrator or user with high privileges to grant or revoke rights (like SELECT, INSERT, UPDATE, etc.) on database objects (like tables, views, etc.).

🔹 DCL commands:

◇ 1. GRANT :

- GRANT command is used to give specific privileges to users.
- For example: allow someone to SELECT or INSERT into a table.

💻 Syntax:

```
GRANT privilege_name
ON object_name
TO user_name;
```

💡 Example:

```
GRANT SELECT, INSERT
ON employees
TO rahul;
```

✅ This means the user rahul is allowed to view & insert records into the employees table.

📌 Note:

- You can grant multiple privileges like SELECT, INSERT, UPDATE, DELETE, etc.
- Objects can be tables, views, procedures, etc.
- In Oracle, use WITH GRANT OPTION to allow the user to grant the same permission to others.

◇ 2. REVOKE :

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

💻 Syntax:

```
REVOKE privilege_name
ON object_name
FROM user_name;
```

💡 Example:

```
REVOKE INSERT
ON employees
FROM rahul;
```

✅ This removes the INSERT privilege from the user rahul. Now he can only SELECT (if that privilege remains).

📌 Note:

- If you revoke all privileges, the user cannot access the table anymore.
- Always revoke specific permissions when a user leaves a project or role.

📌 Real Life Analogy:

- GRANT is like giving keys to someone for a room.
- REVOKE is like taking the keys back when they no longer need access.

📝 Important Notes:

- GRANT and REVOKE apply to specific database objects like tables, views, procedures, etc.
- DCL ensures data security by limiting access to only authorized users.
- Only database administrators (DBAs) or users with sufficient privileges can use DCL commands.
- Be careful when granting UPDATE, DELETE access as it allows modifying or deleting data.

🔁 Summary Table:

Command	Action	Syntax Pattern
GRANT	Give privileges to a user	GRANT SELECT ON table TO user;
REVOKE	Take away privileges from user	REVOKE SELECT ON table FROM user;

🔷 SQL JOINS:

- A SQL JOIN is used to combine data from two or more tables based on a common column that links them.
- In real-world databases, data is split across multiple related tables (like employees and departments).
- JOINs allow you to fetch meaningful, combined information from these tables.

🏵 Types of SQL JOINS :

Join Type	Returns
🔹 INNER JOIN	Only matching rows from both tables
🔹 LEFT JOIN	All rows from left table + matching rows from right table
🔹 RIGHT JOIN	All rows from right table + matching from left table
🔹 FULL OUTER JOIN	All rows from both tables, matched where possible
🔹 CROSS JOIN	Cartesian product – every row from A with every row from B
🔹 SELF JOIN	Joining a table with itself

🔸 Sample Tables Used for Examples :

◇ Employees Table:

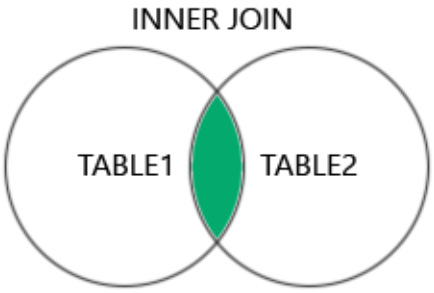
emp_id	emp_name	dept_id
1	John	101
2	Sara	102
3	Mike	103
4	Lily	NULL

◇ Departments Table:

dept_id	dept_name
101	HR
102	IT
104	Finance

◇ 1. INNER JOIN :

- It gives only the matching rows from both tables.
- If there's no match between the tables, the row is not included in the final result.
- Think of it as: *"Give me only the data where both tables have a valid match."*



💻 Syntax:

```
SELECT e.emp_name, d.dept_name
FROM employees e
INNER JOIN departments d
ON e.dept_id = d.dept_id;
```

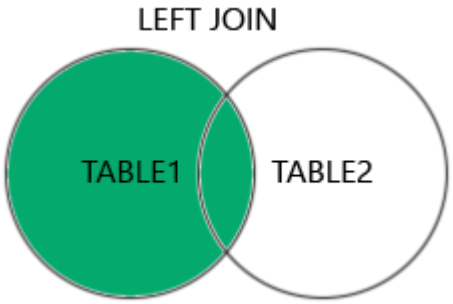
📜 Output:

```
emp_name | dept_name
-----------|-----------
John         | HR
Sara         | IT
```

📝 Note: Mike and Lily are excluded because no matching dept_id exists.

◇ 2. LEFT JOIN (LEFT OUTER JOIN):

- Returns all rows from the left table.
- Also returns the matching rows from the right table.
- If there is no match, the result still includes the left table row and shows NULL for the right table columns.
- Think of it as: *"Keep everything from the left, fill in the right when you can."*



💻 Syntax:

```
SELECT e.emp_name, d.dept_name
FROM employees e
LEFT JOIN departments d
ON e.dept_id = d.dept_id;
```

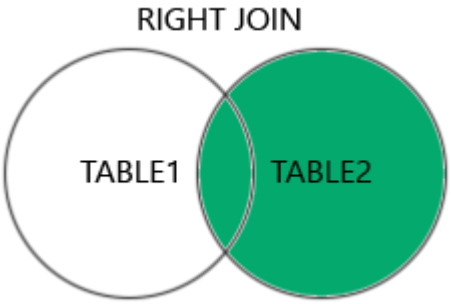
💡 Output:

```
Emp_name| dept_name
 ----------|-----------
John         |     HR
Sara         |     IT
Mike         |     NULL
Lily         |     NULL
```

📝 Note: Unmatched rows from the left side appear with NULL for right table columns.

◇ 3. RIGHT JOIN (RIGHT OUTER JOIN):

- The opposite of LEFT JOIN.
- Returns all rows from the right table.
- Also includes the matching rows from the left table.
- If there is no match, it still keeps the right table data and fills NULL for the left.



💻 Syntax:

```
SELECT e.emp_name, d.dept_name
FROM employees e
RIGHT JOIN departments d
ON e.dept_id = d.dept_id;
```

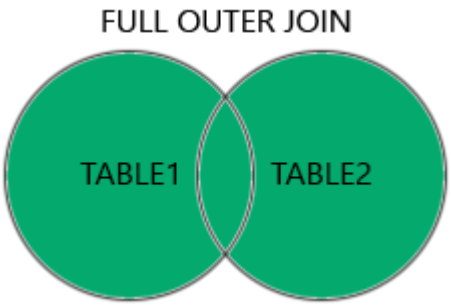
💡 Output:

```
emp_name | dept_name
------------|-----------
John          |     HR
Sara          |     IT
NULL         |     Finance
```

📝 Note: Returns unmatched rows from the right side with NULL from left table.

◇ 4. FULL OUTER JOIN:

- Combines the result of both LEFT and RIGHT JOIN.
- Returns all rows from both tables, with NULLs in places where no match is found.
- Think of it as: *"Show me everything from both tables, matched or not."*



💻 Syntax:

```
SELECT e.emp_name, d.dept_name
FROM employees e
FULL OUTER JOIN departments d
ON e.dept_id = d.dept_id;
```

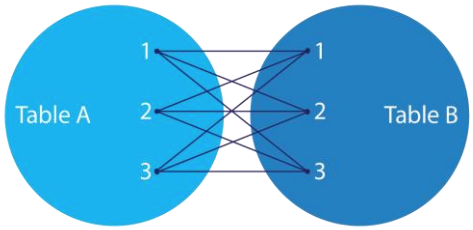
💡 Output:

```
emp_name | dept_name
------------|-----------
John        |    HR
Sara        |    IT
Mike        |    NULL
Lily        |    NULL
NULL        |    Finance
```

📝 Note: This is a combination of LEFT and RIGHT JOIN.

◇ 5. CROSS JOIN:

- Returns the cartesian product of both tables.
- That means every row of the first table is combined with every row of the second.
- Used rarely, and typically when we want all possible combinations.



💻 Syntax:

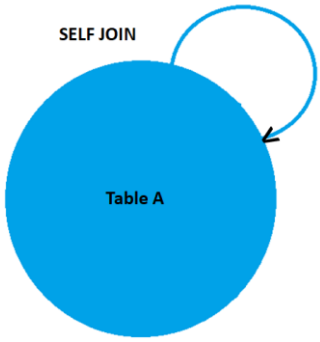
```
SELECT e.emp_name, d.dept_name
FROM employees e
CROSS JOIN departments d;
```

📝 Be careful! If employees have 4 rows and departments have 3, the result will have 4 x 3 = 12 rows.

◇ 6. SELF JOIN :

- A Self JOIN is a regular join where a table is joined with itself.

- It’s useful when the rows in the same table are related to each other, such as an employee reporting to a manager — both are part of the same table.



🔹 Key Points About Self Join :

- The same table is used twice, but with different aliases to differentiate them.
- It is often used to compare rows within the same table.
- Helps in building hierarchical relationships, such as:
  - Employees and their managers.
  - Categories and sub-categories.
  - Friends-of-friends in social networks.

📝 Example Scenarios Where Self Join is Useful:

- In an employee table, if each employee has a manager (also listed as an employee), you can use self join to display:
  - Employee Name → Manager Name.
- In a table of parts and subparts, you can link a part with its parent part using self join.

📌 Important Notes:

- Aliases are necessary in self joins to avoid confusion between column references.
- Self joins help uncover relationships within a single entity.
- It's logically similar to joining two different tables but actually joins different rows of the same table.

💻 Example:

```
SELECT A.emp_name AS Employee, B.emp_name AS Manager
FROM employees A
JOIN employees B ON A.manager_id = B.emp_id;
```

📝 Useful for hierarchical relationships like employees and managers.

📌 When to Use LEFT JOIN Over INNER JOIN?

- Use LEFT JOIN when:
  - You want to see all records from the left table, even if they don’t have a match in the right table.
  - You are okay with seeing NULLs for unmatched data from the right.
  - Example scenarios:
    - Showing customers and their orders (even if some customers didn’t order anything).
    - Listing employees and their departments (even if some employees are not assigned).

- Use INNER JOIN when:
    - You want only the matched data from both tables.
    - You are only interested in records that exist on both sides.

📝 Important Notes:

- Always use aliases when working with multiple tables – improves clarity.
- JOIN conditions should always be defined using ON, not WHERE, especially for OUTER joins.
- Performance tip: Use INNER JOIN for most use-cases unless nulls are needed.
- Use IS NULL to detect unmatched rows in OUTER joins.
- Not all databases support FULL OUTER JOIN directly – in such cases use:
  LEFT JOIN ...
  UNION
  RIGHT JOIN …

📊 Differences Between JOINs:

Feature	INNER JOIN	LEFT JOIN	RIGHT JOIN	FULL OUTER JOIN
Matching Rows	✅	✅	✅	✅
Unmatched Left Rows	❌	✅	❌	✅
Unmatched Right Rows	❌	❌	✅	✅
NULLs in result	❌	For unmatched right	For unmatched left	For both sides
Common Usage	To fetch only related records	When all records from left table are needed	When all right side data is needed	When complete data from both sides is needed

🔷 SQL SET OPERATORS :

- SQL Set Operators are used to combine the results of two or more SELECT queries.
- They work like set theory in mathematics — they merge, compare, or filter rows across two result sets.
- To use them:
  - Both queries must have the same number of columns.
  - The columns must have compatible data types.
  - The column order should match.

🔶 Types of SQL Set Operators:

- There are four main Set Operators in SQL:

Operator	Description
UNION	Combines results of two SELECTs, removes duplicates.
UNION ALL	Combines results of two SELECTs, keeps all duplicates.
INTERSECT	Returns only common records in both SELECTs.
MINUS (Oracle) / EXCEPT (SQL Server, PostgreSQL)	Returns records from the first SELECT that aren't in the second.

◇ 1. UNION:

- Combines result sets and removes duplicate rows.
- Used When: You want to merge data from two queries but remove duplicates.

💡 Example Use Cases:

- Combining customer data from two regions and showing only unique customers.
- Merging employees from two departments, excluding duplicates.

💻 Syntax:

```
SELECT column1, column2 FROM table1
UNION
SELECT column1, column2 FROM table2;
```

💡 Example:

🔹 Tables:

🔸 Employees_2023

Emp_ID	Name
101	Rahul
102	Priya

🔸 Employees_2024

Emp_ID	Name
102	Priya
103	Sameer

🔹 Example:

```
SELECT Emp_ID, Name FROM Employees_2023
UNION
SELECT Emp_ID, Name FROM Employees_2024;
```

🔸 Output:

Emp_ID	Name
101	Rahul
102	Priya
103	Sameer

✓ Removes duplicate entry 102 - Priya

📌 Important Notes:

- Duplicates are removed.
- Automatically sorts records in some DBs (like Oracle).
- Can be slower than UNION ALL due to the sorting process.

◇ 2. UNION ALL:

- Same as UNION, but does not remove duplicates.
- Used When: You want to retain duplicates for accurate record-keeping or counts.
- Faster than UNION since no duplicate-checking is done.
- Use when duplicates matter, like analytics and reporting.

💡 Example Use Cases:

- Merging multiple order records across months for total revenue tracking.
- Getting all attendance logs, even if users clock in/out multiple times.

📌 Notes: May need to use GROUP BY or DISTINCT manually if uniqueness is needed.

💻 Syntax:

```
SELECT column1, column2 FROM table1
UNION ALL
SELECT column1, column2 FROM table2;
```

💡 Example:

🔹 Same tables as above

```
SELECT Emp_ID, Name FROM Employees_2023
UNION ALL
SELECT Emp_ID, Name FROM Employees_2024;
```

🔸 Output:

Emp_ID	Name
101	Rahul

102	Priya
102	Priya
103	Sameer

✓ Shows all entries, including duplicate 102 - Priya

◇ 3. INTERSECT:

- Returns only the common rows between two SELECT statements.
- Used When: You need shared data points across sources.

💡 Example Use Cases:

- Find customers who are both buyers and newsletter subscribers.
- Identify students enrolled in both courses A and B.

📌 Notes:

- Duplicates are removed by default.
- Not supported in MySQL (workaround needed using JOIN + GROUP BY).
- Useful in comparative queries, e.g., common customers, shared access.

💻 Syntax:

```
SELECT column1, column2 FROM table1
INTERSECT
SELECT column1, column2 FROM table2;
```

💡 Example:

🔸 Customers_Mumbai

Cust_ID	Name
1	Anil
2	Ramesh

🔸 Customers_Delhi

Cust_ID	Name
2	Ramesh
3	Suresh

💡 Example:

```
SELECT Cust_ID, Name FROM Customers_Mumbai
INTERSECT
SELECT Cust_ID, Name FROM Customers_Delhi;
```

- Output:

Cust_ID	Name
2	Ramesh

✓ Only the common row is returned.

◇ 4. MINUS / EXCEPT:

- Returns rows from the first query that are not present in the second.
- Oracle uses MINUS, while SQL Server & PostgreSQL use EXCEPT.

💡 Example Use Cases:

- Find employees in HR who are not in the company email system.
- Show products listed in the catalog that have no sales.

📌 Important Notes:

- Order matters: A MINUS B ≠ B MINUS A.
- Removes duplicates automatically.
- MySQL doesn't support MINUS or EXCEPT natively (requires workaround).
- Great for finding unmatched data, e.g., not enrolled, not purchased.

💻 Syntax (Oracle):

```
SELECT column1, column2 FROM table1
MINUS
SELECT column1, column2 FROM table2;
```

💻 Syntax (SQL Server/PostgreSQL):

```
SELECT column1, column2 FROM table1
EXCEPT
SELECT column1, column2 FROM table2;
```

💡 Example:

- All_Students

Roll_No	Name
101	Riya
102	Amit
103	Soham

- Library_Members

Roll_No	Name
101	Riya
103	Soham

💡 Example:

```
SELECT Roll_No, Name FROM All_Students
MINUS
SELECT Roll_No, Name FROM Library_Members;
```

🔸 Output:

Roll_No	Name
102	Amit

✓ Only shows those not found in the second table.

📝 Important Notes:

🔹 All Set Operators:
- Must return the same number of columns.
- Same data type and sequence of columns.
- Column names are taken from the first SELECT statement.
- Results can be ordered using ORDER BY after the final SELECT block.

🔹 Preferred Practices:
- Use UNION when uniqueness is important.
- Use UNION ALL for performance-sensitive applications where duplicates are valid.
- Use INTERSECT and MINUS/EXCEPT for comparison-focused operations.

🔹 Common Mistakes:
- Mismatched column count or data types.
- Placing ORDER BY in each individual SELECT (should only be in the last SELECT).

🔁 Summary: SQL Set Operators:

Operator	Purpose	Removes Duplicates	Common Use Case
UNION	Combines results of two queries and removes duplicates	✅ Yes	Merging datasets where unique results are needed
UNION ALL	Combines results of two queries without removing duplicates	❌ No	Faster merging when all data is important
INTERSECT	Returns only common rows between two queries	✅ Yes	Finding shared data (e.g., common customers)
MINUS / EXCEPT	Returns rows in first query not found in the second (MINUS = Oracle, EXCEPT = SQL Server/PostgreSQL)	✅ Yes	Identifying unmatched or missing data

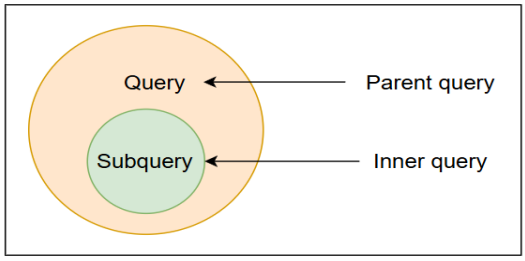
🔷 SQL Subqueries:

🏵 What is a Subquery?

- A subquery is a query that is written inside another query (usually inside the SELECT, FROM, or WHERE clause).
- It is also called a nested query or inner query.
- It helps break down complex problems by running a query inside another.

🔹 Why use Subqueries?

- To divide complex SQL logic into manageable parts.
- To use results from one query as input to another.
- To perform operations like filtering, comparison, or aggregation in one step.



🏵 Types of Subqueries:

◇ 1. Nested Subquery (Simple Subquery)

- A Nested Subquery, also called a Simple Subquery or Non-Correlated Subquery.
- Is a query inside another query
- It runs independently of the outer query and its result is passed to the main query.
- Common Use: Inside WHERE, IN, =, <, >, etc.

💻 Syntax Examples:

```
SELECT name FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

📌 Notes:

- Executes independently.
- Can return a single value or a list (in case of IN), a row, or multiple rows depending on the context.
- Can be placed in the SELECT, FROM, or WHERE clause.

◇ 2. Correlated Subquery:

- A Correlated Subquery is a subquery that depends on a value from the outer query.
- It runs once for each row processed by the outer query.

💻 Syntax Examples:

```
SELECT name FROM employees e1
WHERE salary > (SELECT AVG(salary)
                FROM employees e2
                WHERE e1.department_id = e2.department_id);
```

📌 Notes:

- Slower than nested queries due to multiple executions.— once for each row in the outer query.
- Useful when the inner query needs a value from the outer query row.

- Often used in WHERE EXISTS or WHERE clause but also with IN, =, <, etc.
- It cannot run independently because it uses columns from the outer query.

◇ 3. Multi-row Subquery
- A subquery that returns more than one row. You use it with operators like IN, ANY, ALL.
- You can use IN, ANY, ALL, or even a JOIN to handle multi-row results.
- If you're told "multi-row subquery," they usually mean:
  - A subquery intended to return multiple rows
  - Not a single-value result
  - And not one used with comparison operators like = (which fails with multiple rows)

💻 Syntax Examples:

```
SELECT employee_name
FROM employees
WHERE department_id IN (
   SELECT department_id
   FROM departments
   WHERE location = 'New York'
);
```

📌 Notes:
- Returns a list of values.
- Cannot be used with =, <, > unless combined with ANY or ALL.

◇ 4. Subquery with IN:
- Used when you want to filter the outer query using a list of results from inner query.
- Basic concept: A subquery that returns multiple values (more than one row) to the outer query.
- The outer query uses IN to check if a value exists in the list returned by the subquery.
- So technically, *every* subquery with IN is a multi-row subquery — as long as it can return multiple rows.

💻 Syntax Examples:

```
SELECT name
FROM employees
WHERE department_id IN (
   SELECT department_id FROM departments WHERE location = 'New York'
);
```

📌 Notes:
- Can be replaced by a JOIN in some cases (for performance)
- Safer with multiple results.

◇ 5. Subquery with EXISTS:
- Checks whether at least one row is returned by the subquery. Returns TRUE or FALSE.

💻 Syntax Examples:

```
SELECT d.department_name
FROM departments d
WHERE EXISTS (SELECT 1 FROM employees e WHERE e.department_id =
d.department_id);
```

📌 Notes:
- Great for performance with large data.

- Returns TRUE if a single row exists—doesn't retrieve data.
- Often used with correlated subqueries.

📝 Important Notes:

- You cannot use a subquery in the SELECT clause to return multiple values (only one).
- Subqueries can be used in:
  - SELECT clause
  - FROM clause
  - WHERE clause
- Correlated Subqueries are generally slower than Joins or nested subqueries due to row-by-row evaluation.
- Use EXISTS when you only need to check the presence of rows, not the actual values.
- Prefer JOINs if you want to fetch actual data from multiple tables instead of filtering based on it.
- If using ORDER BY in subqueries, you must use TOP, LIMIT, or ROWNUM depending on your SQL dialect.

🔁 Summary: Subqueries in SQL:

Type	Description	Common Use Case
Nested Subquery	A standalone query inside another query	Comparing values with an aggregate result
Correlated	Depends on outer query, runs for each row	Row-wise comparison within groups
Multi-row	Returns multiple values; uses IN, ANY, ALL	Filtering by a list of results
IN Subquery	Filters values found in the subquery result list	Check membership in another set
EXISTS Subquery	Checks if at least one row exists in the subquery result	Presence check, performance-friendly

❖ SELECT 1 FROM employees what does it means select 1?

```
SELECT 1 FROM employees;
```

- This query does not retrieve any column from the employees table.
- Instead, it simply returns the value 1 for each row in the employees table.
- So if your employees table has 5 rows, the result will be:

```
1
1
1
1
1
1
```

🔹 Why is it used like that?

- This pattern is often used when:
  - You don’t care about the actual data, just want to check existence or row count.
  - Used inside EXISTS clauses.

🔍 Example with EXISTS:

```
IF EXISTS (SELECT 1 FROM employees WHERE department_id = 10) THEN
  -- Do something
END IF;
```

Here, SELECT 1 is just a placeholder — it's used because Oracle requires a SELECT inside EXISTS, but we don't need actual data — we only care if at least one row exists.

🔷 SQL Functions:

- Functions in SQL are predefined programs that perform operations on data and return a result.
- They help you manipulate data, transform it, or make decisions based on it.

🏵 Types of SQL Functions:

▫ Aggregate Functions: SUM,AVG,COUNT,MIN,MAX

▫ Scalar Functions:

- String: CONCAT, SUBSTRING, LENGTH, TRIM, REPLACE,....etc
- Numeric: ABS, CEIL, FLOOR, ROUND, TRUNC, MOD, POWER,.....etc
- Date: SYSDATE, GETDATE, NOW, TO_DATE, LAST_DAY,.....etc
- Conversion: TO_CHAR, TO_DATE, CAST, CONVERT, TO_NUMBER
- Conditional: CASE, COALESCE, NULLIF, DECODE

▫ Analytical / Window Functions: ROW_NUMBER, RANK,DENSE_RANK,..etc

▫ System Functions: USER, SESSION_USER, DATABASE,VERSION,...etc

▫ JSON Functions: JSON_VALUE, JSON_QUERY, JSON_OBJECT, JSON_ARRAY

▫ XML Functions: EXTRACTVALUE, XMLTABLE, XMLAGG, FOR XML PATH

◇ 1. Aggregate Functions:

- Aggregate Functions perform operations on a group of rows and return a single value as a result.
- They are most often used with the GROUP BY clause but can also be used without it.

🔹 List of Aggregate Functions:

1. SUM(): Returns the total of a numeric column

📊 Table:

employee_id	name	salary
101	Vaibhav	50000
102	Rohan	60000
103	Priya	40000

💻 Syntax:

```
SELECT SUM(salary) FROM employees;
```

📄 Output:

```
SUM(salary)
------------
150000
```

2. AVG(): Returns the average of a numeric column

💻 Syntax:

```
SELECT AVG(salary) FROM employees;
```

📄 Output:

```
AVG(salary)
------------
50000
```

3. COUNT(): Returns the number of rows

💻 Syntax:

```
SELECT COUNT(*) FROM employees;
```

📄 Output:

```
COUNT(*)
---------
3
```

👉 You can also count specific values:

```
SELECT COUNT(salary) FROM employees;
```

4. MAX(): Finds highest value

💻 Syntax:

```
SELECT MAX(salary) FROM employees;
```

📄 Output:

```
MAX(salary)
------------
60000
```

5. MIN() : Finds lowest value

💻 Syntax:

```
SELECT MIN(salary) FROM employees;
```

📄 Output:

```
MIN(salary)
------------
40000
```

📝 Important Notes:

- All aggregate functions ignore NULLs by default.
- You can combine aggregate functions with GROUP BY to get summaries per category (e.g., per department).
- These functions do not modify data, only read and summarize it.

◇ 2. Scalar Functions:

- A Scalar Function is a function in SQL that returns a single value (a scalar) for each row in a result set.
- These functions operate on individual values, unlike aggregate functions (which operate on groups of rows).

🔹 Categories of Scalar Functions:

Function Type	Description
String Functions	Modify or return information about text values
Numeric Functions	Perform operations on numbers
Date Functions	Work with date and time values
Conversion Functions	Convert data types
Conditional Functions	Return values based on conditions

◇ 1. String Functions:

1. UPPER(): Converts all characters in a string to uppercase.
💻 Syntax: UPPER(string)
💡 Example: SELECT UPPER('vaibhav') FROM dual;
📜 Output: VAIBHAV

2. LOWER(): Converts all characters in a string to lowercase.
💻 Syntax: LOWER(string)
💡 Example: SELECT LOWER('SQL Expert') FROM dual;
📜 Output: sql expert

3. INITCAP(): Converts the first letter of each word in a string to uppercase.
💻 Syntax: INITCAP(string)
💡 Example: SELECT INITCAP('vaibhav dhakulkar') FROM dual;
📜 Output: Vaibhav Dhakulkar

4. LENGTH(): Returns the number of characters in a string.
💻 Syntax: LENGTH(string)
💡 Example: SELECT LENGTH('SQL') FROM dual;
📜 Output: 3

5. SUBSTR(): Extracts a substring from a string starting at a specific position for a given length.
💻 Syntax: SUBSTR(string, start_position, length)
💡 Example: SELECT SUBSTR('DATABASE', 1, 4) FROM dual;
📜 Output: DATA

6. INSTR(): Returns the position of the first occurrence of a substring in a string.
💻 Syntax: INSTR(string, substring)
💡 Example: SELECT INSTR('Hello World', 'o') FROM dual;
📜 Output: 5

7. TRIM(): Removes leading and trailing spaces (or specified characters) from a string.

💻 Syntax: TRIM(string)

💡 Example: SELECT TRIM(' Vaibhav ') FROM dual;

📜 Output: Vaibhav

8. LTRIM(): Removes spaces from the beginning (left side) of the string.

💻 Syntax: LTRIM(string)

💡 Example: SELECT LTRIM(' SQL') FROM dual;

📜 Output: SQL

9. RTRIM(): Removes spaces from the end (right side) of the string.

💻 Syntax: RTRIM(string)

💡 Example: SELECT RTRIM('SQL ') FROM dual;

📜 Output: SQL

10. REPLACE(): Replaces a part of the string with another substring.

💻 Syntax: REPLACE(string, search_string, replace_with)

💡 Example: SELECT REPLACE('SQL and PL/SQL', 'SQL', 'Oracle') FROM dual;

📜 Output: Oracle and PL/Oracle

11. CONCAT(): Concatenates (joins) two strings. Only works with 2 strings in Oracle.

💻 Syntax: CONCAT(string1, string2)

💡 Example: SELECT CONCAT('Data', 'Base') FROM dual;

📜 Output: DataBase

12. || (Concatenation Operator): Concatenates multiple strings. Preferred in Oracle over CONCAT() for more than 2 strings.

💻 Syntax: string1 || string2 || string3 ...

💡 Example: SELECT 'Vaibhav' || ' ' || 'Dhakulkar' FROM dual;

📜 Output: Vaibhav Dhakulkar

📝 Important Notes:

- UPPER, LOWER, INITCAP are useful for formatting names and text.
- LENGTH, INSTR, SUBSTR help in string analysis.
- || is the preferred method in Oracle for string concatenation when combining more than 2 strings.
- REPLACE is useful in cleaning and formatting text data.
- INSTR returns the position of first occurrence.
- Use SUBSTR() to extract values like first name, last name, etc.
- TRIM(), LTRIM(), and RTRIM() are useful for removing extra spaces before inserting or comparing strings.

🔁 Summary: String Functions

Function	Description	Supported In
CONCAT()	Joins two or more strings	✅ All
SUBSTR()	Extracts part of a string	❌ SQL Server
LEFT()	Left part of a string	❌ Oracle,
RIGHT()	Right part of a string	❌ Oracle
LENGTH()	Length of string	✅ All
INSTR()	Position of substring	❌ SQL Server
REPLACE()	Replace part of a string	✅ All
UPPER()	Convert to uppercase	✅ All
LOWER()	Convert to lowercase	✅ All
TRIM()	Removes leading/trailing spaces	✅ All

◇ 2. Numeric Functions:

1. ABS(): Returns the absolute (positive) value of a number.

💻 Syntax: ABS(number)

💡 Example: SELECT ABS(-42) FROM DUAL;

📜 Output: 42

2. CEIL(): Rounds a number *up* to the nearest integer.

💻 Syntax: CEIL(number)

💡 Example: SELECT CEIL(3.14) FROM DUAL;

📜 Output: 4

3. FLOOR(): Rounds a number *down* to the nearest integer.

💻 Syntax: FLOOR(number)

💡 Example: SELECT FLOOR(3.99) FROM DUAL;

📜 Output: 3

4. MOD(): Returns the remainder of division (modulus).

💻 Syntax: MOD(dividend, divisor)

💡 Example: SELECT MOD(10, 3) FROM DUAL;

📜 Output: 1

5. POWER(): Raises a number to the power of another number.

💻 Syntax: POWER(base, exponent)

💡 Example: SELECT POWER(2, 4) FROM DUAL;

📜 Output: 16

6. ROUND(): Rounds a number to a specified number of decimal places.

💻 Syntax: ROUND(number, decimal_places)

💡 Example: SELECT ROUND(3.4567, 2) FROM DUAL;

📜 Output: 3.46

7. TRUNC(): Truncates a number to a specified number of decimal places (no rounding).

💻 Syntax: TRUNC(number, decimal_places)

💡 Example: SELECT TRUNC(3.4567, 2) FROM DUAL;

📜 Output: 3.45

8. SIGN(): Returns -1, 0, or 1 based on whether the number is negative, zero, or positive.

💻 Syntax: SIGN(number)

💡 Example: SELECT SIGN(-20) FROM DUAL;

📜 Output: -1

9. SQRT(): Returns the square root of a number.

💻 Syntax: SQRT(number)

💡 Example: SELECT SQRT(16) FROM DUAL;

📜 Output: 4

10. EXP(): Returns *e* raised to the power of the specified number.

💻 Syntax: EXP(number)

💡 Example: SELECT EXP(1) FROM DUAL;

📜 Output: 2.7182818...

📝 Important Notes:

- These functions work on numeric data types (like NUMBER, INTEGER, FLOAT, etc.).
- DUAL is a special dummy table in Oracle used for quick function testing
- TRUNC() and ROUND() are often used when formatting currency or precision-sensitive data.

🔁 Summary: Numeric Functions

Function	Description	Supported In
ABS()	Absolute value	✅ All
CEIL() / CEILING()	Smallest integer ≥ number	✅ All
FLOOR()	Largest integer ≤ number	✅ All
MOD()	Remainder after division	❌ SQL Server
ROUND()	Round number to decimals	✅ All
TRUNC()	Truncate decimal part	❌ SQL Server, ❌ MySQL
POWER()	Raise number to a power	✅ All
SQRT()	Square root	✅ All
EXP()	Exponential value	✅ All
LOG() / LN()	Natural / base-10 logarithm	❌ SQL Server

◇ 3. Date Functions:

1. SYSDATE: Returns the current date and time of the database server.

💻 Syntax: SYSDATE
💡 Example: SELECT SYSDATE FROM DUAL;
📄 Output: 23-MAY-25 10:45:00 (your system date and time)

2. CURRENT_DATE: Returns the current date and time in the user's session time zone.

💻 Syntax: CURRENT_DATE
💡 Example: SELECT CURRENT_DATE FROM DUAL;
📄 Output: 23-MAY-25 10:45:00 (according to session time zone)

3. SYSTIMESTAMP: Returns the current date and time with the time zone of the system.

💻 Syntax: SYSTIMESTAMP
💡 Example: SELECT SYSTIMESTAMP FROM DUAL;
📄 Output: 23-MAY-25 10:45:00.123456 +05:30

4. ADD_MONTHS(): Adds or subtracts a number of months to/from a date.

💻 Syntax: ADD_MONTHS(date, number_of_months)
💡 Example: SELECT ADD_MONTHS(SYSDATE, 3) FROM DUAL;
📄 Output: 23-AUG-25

5. MONTHS_BETWEEN(): Returns the number of months between two dates.

💻 Syntax: MONTHS_BETWEEN(date1, date2)

💡 Example: SELECT MONTHS_BETWEEN('23-MAY-25', '23-JAN-25') FROM DUAL;
📜 Output: 4

6. NEXT_DAY(): Returns the next date of the specified weekday after the given date.
💻 Syntax: NEXT_DAY(date, 'DAY_NAME')
💡 Example: SELECT NEXT_DAY(SYSDATE, 'MONDAY') FROM DUAL;
📜 Output: 27-MAY-25 (next Monday)

7. LAST_DAY(): Returns the last day of the month of a given date.
💻 Syntax: LAST_DAY(date)
💡 Example: SELECT LAST_DAY(SYSDATE) FROM DUAL;
📜 Output: 31-MAY-25

8. ROUND(date): Rounds the date to the nearest day, month, or year, depending on the format.
💻 Syntax: ROUND(date, 'format')
💡 Example: SELECT ROUND(TO_DATE('23-MAY-25'), 'MONTH') FROM DUAL;
📜 Output: 01-JUN-25

9. TRUNC(date): Truncates the date to the start of a specific unit (day, month, year, etc.).
💻 Syntax: TRUNC(date, 'format')
💡 Example: SELECT TRUNC(TO_DATE('23-MAY-25'), 'MONTH') FROM DUAL;
📜 Output: 01-MAY-25

10. EXTRACT(): Extracts a specific part (YEAR, MONTH, DAY) from a date.
💻 Syntax: EXTRACT(YEAR FROM date)
💡 Example: SELECT EXTRACT(YEAR FROM SYSDATE) FROM DUAL;
📜 Output: 2025

📝 Important Notes:
- Dates are stored in Oracle in DD-MON-YY format by default.
- Use TO_DATE() to explicitly convert a string to a date format.
- You can do arithmetic on dates (e.g., SYSDATE - 7 gives last week's date).
- Oracle automatically handles leap years, months with 30/31 days, etc.

🔁 Summary: Date Functions

Function	Description	Supported In
SYSDATE	Current date and time	✅ Oracle
CURRENT_DATE	Current date	✅ All
NOW()	Current timestamp	❌ Oracle
ADD_MONTHS()	Add months to date	✅ Oracle
MONTHS_BETWEEN()	Months between two dates	✅ Oracle
NEXT_DAY()	Next weekday after given date	✅ Oracle
LAST_DAY()	Last day of month	✅ Oracle
EXTRACT()	Extract part of date (e.g., year)	✅ All
DATE_PART()	Extract part of date	✅ PostgreSQL
DATEDIFF()	Days between dates	❌ Oracle, ❌ PostgreSQL

◇ 4. Conversion Functions:

🔹 1. TO_CHAR(): Convert Date or Number to String & Mainly used in Oracle DB

💻 Syntax: TO_CHAR(value, 'format')

💡 Example:

```
SELECT TO_CHAR(SYSDATE, 'DD-Mon-YYYY') AS formatted_date FROM
DUAL;
```

📜 Output:

```
FORMATTED_DATE
----------------------
23-May-2025
```

🔹 2. TO_DATE() – Convert String to Date and Oracle-specific function

💻 Syntax: TO_DATE('string_date', 'format')

💡 Example:

```
SELECT TO_DATE('23-05-2025', 'DD-MM-YYYY') AS actual_date FROM DUAL;
```

📜 Output:

```
ACTUAL_DATE
-----------------
2025-05-23
```

🔹 3. TO_NUMBER() – Convert String to Number and Oracle-specific

💻 Syntax: TO_NUMBER('string_number')

💡 Example: SELECT TO_NUMBER('100') + 50 AS result FROM DUAL;

📜 Output:

```
RESULT
-------
150
```

🔹 4. CAST() – General Purpose Conversion and work in all DBMS

💻 Syntax: CAST(expression AS data_type)

💡 Example: SELECT CAST('2025-05-23' AS DATE) AS converted_date;

🗒️ Output: (in MySQL/PostgreSQL):

```
CONVERTED_DATE
——----------------------
2025-05-23
```

🔹 5. CONVERT() – Convert Between Data Types

⚠️ Mostly used in SQL Server

💻 Syntax: CONVERT(data_type, expression [, style])

💡 Example: SELECT CONVERT(VARCHAR, GETDATE(), 103) AS formatted_date;

🗒️ Output: (SQL Server):

```
FORMATTED_DATE
---------------
23/05/2025
```

📝 Important Notes:

- Use CAST() when writing portable SQL across different DBMSs.
- TO_CHAR, TO_DATE, TO_NUMBER are Oracle-specific but very powerful for formatting.
- CONVERT() is handy in SQL Server for date and string formatting.
- Always match format patterns (like 'YYYY-MM-DD') to input/output for correct results.
- If formats are mismatched, conversion may fail or return NULL.

🔁 Summary: SQL Conversion Functions

Function	Converts What	Commonly Used In
TO_CHAR()	Date/Number → String	✅ Oracle
TO_DATE()	String → Date	✅ Oracle
TO_NUMBER()	String → Number	✅ Oracle
CAST()	Any type → Any type	✅ All
CONVERT()	Expression → Data type	✅ SQL Server

◇ 5. Conditional Functions :

🔹 1. CASE :

- General Conditional Expression
- Works like IF-THEN-ELSE, lets you evaluate conditions and return specific
- Works in all DBMSs (Oracle, SQL Server, MySQL, PostgreSQL)

💻 Syntax:

```
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  ...
  ELSE default_result
END
```

💡 Example:

```
SELECT emp_name,
       salary,
       CASE
         WHEN salary >= 50000 THEN 'High'
         WHEN salary >= 30000 THEN 'Medium'
         ELSE 'Low'
       END AS salary_level
FROM employees;
```

📜 Output:

```
EMP_NAME   SALARY   SALARY_LEVEL
---------- ---------- ----------------
Raj              60000            High
Amit             35000   Medium
Ravi             20000            Low
```

🔹 2. COALESCE:

- Returns the first non-null value from the list.
- It checks each value in order and returns the first one that is not NULL.
- Works in all DBMSs

💻 Syntax: `COALESCE(value1, value2, ..., valueN)`

💡 Example:

```
SELECT emp_name,
       COALESCE(phone, email, 'No Contact') AS contact_info
FROM employees;
```

📜 Output:

```
EMP_NAME      CONTACT_INFO
-----------   ——------------------
Raj               9876543210
Amit              amit@mail.com
Ravi              No Contact
```

- 🔹 Returns the first non-null value among the list.

🔹 3. NULLIF:

- Returns NULL if two expressions are equal, else returns the first one.
- Works in all DBMSs

💻 Syntax: NULLIF(value1, value2)

💡 Example:

```
SELECT emp_name,
       NULLIF(dept_id, 10) AS updated_dept
FROM employees;
```

📜 Output:

```
EMP_NAME      UPDATED_DEPT
------------  ——----------------
Raj                 20
Amit                    NULL
Ravi                30
```

- If both values are equal, returns NULL, else returns value1.

🔹 4. DECODE:

- Oracle-Specific CASE Alternative
- Acts like a simple IF-THEN-ELSE, similar to CASE but Oracle-specific.
- ✅ Only in Oracle

💻 Syntax:

```
DECODE(expression, search1, result1, search2, result2, ..., default)
```

💡 Example:

```
SELECT emp_name,
      DECODE(dept_id,
            10, 'Finance',
            20, 'HR',
            30, 'IT',
            'Unknown') AS department
FROM employees;
```

📜 Output:

```
EMP_NAME      DEPARTMENT
------------  ——-------------
Raj                 HR
Amit                    Finance
Ravi                IT
```

- Works like CASE, but shorter syntax. Limited to equality checks.

🔹 5. NVL Function (Oracle Only)

- Support in Oracle only.
- Replaces NULL with a specified value.

💻 Syntax: NVL(expr1, expr2)

💡 Example:

```
SELECT NVL(commission_pct, 0) AS commission FROM employees;
```

📜 Output: If commission is NULL, it will display 0.

🔹 6. NVL2 Function (Oracle Only):

- Returns one value if the expression is NOT NULL, another if it is NULL.
- Support in Oracle only

💻 Syntax: NVL2(expr1, value_if_not_null, value_if_null)

💡 Example:

```
SELECT NVL2(commission_pct, 'Has Bonus', 'No Bonus') FROM employees;
```

📃 Output:

commission_pct	result
NULL	No Bonus
0.3	Has Bonus

📝 Important Notes:

- Use CASE for complex, multi-condition logic.
- Use COALESCE to replace NULLs with fallback values.
- Use NULLIF to suppress equal values by converting to NULL.
- Use DECODE for simpler conditions in Oracle, but prefer CASE for cross-platform compatibility.
- All are Scalar Functions — return one value per row.

🔁 Summary: Conditional Functions

Function	Purpose	Supported In DBMS
CASE	IF-ELSE style logic	✅ All
COALESCE	Return first non-NULL value	✅ All
NULLIF	Return NULL if both values are equal	✅ All
DECODE	Oracle-specific conditional replacement	✅ Oracle Only
NVL	Replace NULL with value	✅ Oracle Only
NVL2	Two results based on NULL or NOT NULL	✅ Oracle Only

◇ 3. Window (Analytical) Functions:

💠 What are Window Functions?

- Windows functions in Oracle SQL are powerful tools.
- perform calculations across a set of rows related to the current rows without collapsing the result. Set into a single value like aggregate functions do.
- Unlike aggregate functions, they do not collapse rows into a single result.
- Useful for ranking, running totals, moving averages, etc.

🔹 What is a Window?

- A window is a defined set of rows that the function can access to perform its calculation.
- Specified using the OVER() clause.

🔹 PARTITION BY Clause:
- Divides the result set into groups (partitions).
- Each partition is treated independently by the window function.
- If we don’t use Partition By the entire result set is treated as a single partition

🔹 ORDER BY Clause (Inside OVER())
- Determines the order in which rows are processed within each partition.
- Essential for functions like RANK(), LEAD(), LAG(), etc.

🔹 There is no official division of the SQL window functions into categories but high level we can divide into 3 types

💻 Syntax:

```
function_name (expression) OVER (
  [PARTITION BY column]
  [ORDER BY column]
  [ROWS BETWEEN ...]
)
```

🔸 Let’s go one-by-one with examples, syntax, and outputs:
We'll use this sample sales table:

EMP_ID	EMP_NAME	DEPT	SALARY	DOJ
101	Aakash	HR	50000	2022-01-15
102	Neha	IT	60000	2021-06-10
103	Amit	HR	50000	2020-03-20
104	Riya	IT	75000	2022-11-25
105	Raj	SALES	62000	2019-09-01
106	Priya	SALES	62000	2020-01-10
107	Arjun	IT	60000	2023-04-03
108	Sneha	HR	80000	2021-02-17
109	Vinay	IT	60000	2022-07-19
110	Kiran	SALES	55000	2022-08-30

◇ Aggregate Functions:

- Windows aggregate functions are the set of functions that allows you to perform calculations across a set of rows that are related to the current row.
- Unlike regular aggregate functions which group data and return a single result per group, windows functions can return multiple results within a partition and ordered data set

🔍 What's the difference between normal aggregate functions and window aggregate functions?

- Normal aggregates (like GROUP BY) collapse rows into 1 per group.
- Window aggregates return a result per row while calculating across a logical group.

1. SUM()
   - Sums values within a window.
   - Returns the total (sum) of values over the window of rows.
   - Example: Total salary for each department.

   💡 Example:

```
SELECT
EMP_ID, EMP_NAME, DEPT, SALARY,
SUM(SALARY) OVER (PARTITION BY DEPT) AS DEPT_TOTAL
FROM EMPLOYEE;
```

2. AVG()
   - Calculates the average value within a window.
   - Useful for comparing each row with department average.

   💡 Example:

```
SELECT
EMP_ID, EMP_NAME, DEPT, SALARY,
AVG(SALARY) OVER (PARTITION BY DEPT) AS DEPT_AVG
FROM EMPLOYEE;
```

3. MIN()
   - The MIN() function returns the minimum value from a specified column within the window of rows defined by the partition by and order by clauses.
   - Used to find minimum marks, dates, etc., within a group.

   💡 Example:

```
SELECT
EMP_ID, EMP_NAME, DEPT, SALARY,
MIN(SALARY) OVER (PARTITION BY DEPT) AS DEPT_MIN
FROM EMPLOYEE;
```

4. MAX()
   - The MAX()function returns the minimum value from a specified column within the window of rows defined by the partition by and order by clauses.
   - Helps find max salary, score, or date in a group.

   💡 Example:

```
SELECT
EMP_ID, EMP_NAME, DEPT, SALARY,
MAX(SALARY) OVER (PARTITION BY DEPT) AS DEPT_MAX
FROM EMPLOYEE;
```

5. COUNT()
   - The COUNT() window function counts the number of rows within the window of rows

   💡 Example:

```
SELECT
EMP_ID,  EMP_NAME,  DEPT,  SALARY,
COUNT(*) OVER (PARTITION BY DEPT) AS DEPT_COUNT
FROM EMPLOYEE;
```

Output:

EMP_ID	EMP_NAME	DEPT	SALARY	DEPT_SUM	DEPT_AVG	DEPT_MIN	DEPT_MAX	DEPT_COUNT
101	Aakash	HR	50000	180000	60000	50000	80000	3
103	Amit	HR	50000	180000	60000	50000	80000	3
108	Sneha	HR	80000	180000	60000	50000	80000	3
102	Neha	IT	60000	255000	63750	60000	75000	4
104	Riya	IT	75000	255000	63750	60000	75000	4
107	Arjun	IT	60000	255000	63750	60000	75000	4
109	Vinay	IT	60000	255000	63750	60000	75000	4
105	Raj	SALES	62000	179000	59666.67	55000	62000	3
106	Priya	SALES	62000	179000	59666.67	55000	62000	3
110	Kiran	SALES	55000	179000	59666.67	55000	62000	3

◇ Ranking Functions:

- Window Ranking Functions are a type of analytic (window) function in SQL that assigns a rank or position to each row within a partition of the result set.
- Operate within a specified partition (group of rows) defined by PARTITION BY.
- Order rows using ORDER BY inside the OVER() clause.
- Unlike aggregate functions, they don't collapse rows — they return one row per input row with an additional calculated value (rank/position).

📦 Use Cases:

- Ranking employees by salary within each department.
- Finding top-N performers.
- Dividing data into quartiles or percentiles.
- Detecting ties in values and assigning ranks accordingly.

1. ROW_NUMBER() –

- Assigns a unique sequential number to each row in a partition of a result set, starting with 1 for the first row in each partition.
- Use: To get unique row IDs, top-N rows per group.

💡 Example:

```
SELECT
        EMP_ID, EMP_NAME, DEPT, SALARY,
        ROW_NUMBER() OVER (PARTITION BY DEPT ORDER BY SALARY
        DESC) AS ROW_NUM
FROM EMPLOYEE;
```

2. RANK() –

- Assigns a rank to each row within a partition of a result set.
- Rows with equal values (Duplicate) receive the same rank, and the next rank after a tie is incremented by the number of tied rows.
- Use: Ranking with gaps.
- As the name suggests the rank function assigns rank to all the rows within every partition rank is assigned such that rank 1 given to the first row and rows having the same values are assigned the same rank.
- For the next rank after two same rank values one rank value will be skipped. For instance if two rows share rank1 the next row gets rank 3 not 2

💡 Example:

```
SELECT
        EMP_ID, EMP_NAME, DEPT, SALARY,
        RANK() OVER (PARTITION BY DEPT ORDER BY SALARY DESC) AS
        RANKING
FROM EMPLOYEE;
```

3. DENSE_RANK() –

- Like RANK(), but no gaps after duplicates.
- This function is similar to the RANK() function, but it handles ties differently when assigning ranks.
- The key difference is that ‘DENSE_RANK()’ does not leave gaps in the ranking sequence when there are ties. Instead it continues with the next consecutive rank.
- Use: Compact ranking.

💡 Example:

```
SELECT
    EMP_ID, EMP_NAME, DEPT, SALARY,
    DENSE_RANK() OVER (PARTITION BY DEPT ORDER BY SALARY
    DESC) AS DENSE_RANKING
FROM EMPLOYEE;
```

4. NTILE(n) –

- Divides the rows in an ordered partition into 'N' groups as evenly as possible and assigns a unique group member to each row.
- Number of Tiles: Specify how many tiles (groups) you want the rows to be divided into
- Distribution: The function divides the rows in each partition into the specified number of tiles. If the number of rows does not divide evenly some tiles will have one more row than others.
- Use: Bucketizing for quartiles, deciles, etc.

💡 Example:

```
SELECT
    EMP_ID, EMP_NAME, DEPT, SALARY,
    NTILE(2) OVER (PARTITION BY DEPT ORDER BY SALARY DESC)
    AS NTILE_2
FROM EMPLOYEE;
```

5. PERCENT_RANK()

- Calculates the relative rank (0 to 1) of each row within a partition.
- Use: To find percentile-like values.

💡 Example:

```
SELECT
    EMP_ID, EMP_NAME, DEPT, SALARY,
    ROUND(PERCENT_RANK() OVER (PARTITION BY DEPT ORDER BY
    SALARY DESC), 2) AS PERCENT_RANKING
FROM EMPLOYEE;
```

📜 Output:

EMP_ID	EMP_NAME	DEPT	SALARY	ROW_NUM	RANKING	DENSE_RANKING	NTILE_2	PERCENT RANKING
108	Sneha	HR	80000	1	1	1	1	0.00
101	Aakash	HR	50000	2	2	2	2	0.5
103	Amit	HR	50000	3	2	2	2	0.5
104	Riya	IT	75000	1	1	1	1	0.00
102	Neha	IT	60000	2	2	2	1	0.33
107	Arjun	IT	60000	3	2	2	2	0.33
109	Vinay	IT	60000	4	2	2	2	0.33
105	Raj	SALES	62000	1	1	1	1	0.00
106	Priya	SALES	62000	2	1	1	1	0.00

110	Kiran	SALES	55000	3	3	2	2	0.66

◇ Value Functions:

- Window Value Functions return a value from another row in the same result set, relative to the current row — without collapsing the data.
- They are very useful for comparing rows (e.g., previous, next, first, last) within a partition.

1.LEAD()

- Returns the value from the next row in the window.
- Useful for comparing the current row with the next one.

💡 Example:

```
SELECT
    EMP_ID, EMP_NAME, DEPT, SALARY,
    LEAD(SALARY) OVER (PARTITION BY DEPT ORDER BY SALARY)
    AS NEXT_SALARY
FROM employees;
```

2.LAG()

- Returns the value from the previous row.
- Used for difference calculations like sales vs. previous month.

💡 Example:

```
SELECT
    EMP_ID, EMP_NAME, DEPT, SALARY,
    LAG(SALARY) OVER (PARTITION BY DEPT ORDER BY SALARY)
    AS PREV_SALARY
FROM employees;
```

3.FIRST_VALUE()

- Returns the first value in the window frame.
- Helps track the starting value of a group.

💡 Example:

```
SELECT
    EMP_ID, EMP_NAME, DEPT, SALARY,
    FIRST_VALUE(SALARY) OVER (PARTITION BY DEPT ORDER BY
    SALARY) AS FIRST_SALARY
FROM employees;
```

4.LAST_VALUE()

- Returns the last value in the window frame.
- Tracks the ending value of a partition.

💡 Example:

```
SELECT
    EMP_ID, EMP_NAME, DEPT, SALARY,
    LAST_VALUE(SALARY) OVER (PARTITION BY DEPT ORDER BY
SALARY
```

```
                ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED
                FOLLOWING) AS LAST_SALARY
        FROM employees;
```

5.NTH_VALUE()

- Returns the nth value (e.g., 2nd, 3rd...) in the partition/order of the result set.
- If n = 2, it gives the second value in the frame.

💡 Example:

```
SELECT
        EMP_ID, EMP_NAME, DEPT, SALARY,
        NTH_VALUE(SALARY, 2) OVER (PARTITION BY DEPT ORDER BY SALARY
         ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED
        FOLLOWING) AS SECOND_SALARY
FROM employees;
```

📜 Output:

EMP_ID	EMP_NAME	DEPT	SALARY	PREV_SALARY	NEXT_SALARY	FIRST_SALARY	LAST_SALARY	SECOND_SALARY
103	Amit	HR	50000	null	50000	50000	80000	50000
101	Aakash	HR	50000	50000	80000	50000	80000	50000
108	Sneha	HR	80000	50000	null	50000	80000	50000
102	Neha	IT	60000	null	60000	60000	75000	60000
107	Arjun	IT	60000	60000	60000	60000	75000	60000
109	Vinay	IT	60000	60000	75000	60000	75000	60000
104	Riya	IT	75000	60000	null	60000	75000	60000
110	Kiran	SALES	55000	null	62000	55000	62000	62000
105	Raj	SALES	62000	55000	62000	55000	62000	62000
106	Priya	SALES	62000	62000	null	55000	62000	62000

📝 Important Notes:

- PARTITION BY: Like GROUP BY inside a window. It splits the data into partitions to apply the function.
- ORDER BY: Sorts rows within each partition.
- ROWS BETWEEN: Specifies the range of rows in a frame.
- Window functions do not reduce rows, unlike GROUP BY. They add an extra column with calculated values.

🔁 Summary: Windows Functions

Function	Description
SUM()	Running total of values over a partition
AVG()	Running average of values over a partition
MIN()	Minimum value in the current window or partition
MAX()	Maximum value in the current window or partition
COUNT()	Counts rows or non-null values over a partition
ROW_NUMBER()	Unique sequential number for each row per partition
RANK()	Rank with gaps
DENSE_RANK()	Rank without gaps
NTILE(n)	Distributes rows into n buckets
PERCENT_RANK()	Percentile rank of the row in the partition
LEAD()	Next row value
LAG()	Previous row value
FIRST_VALUE()	First value in partition
LAST_VALUE()	Last value in partition
NTH_VALUE()	Nth row's value in the partition

◇ 4. System Functions:

- System functions in SQL are built-in scalar functions used to retrieve:
  - System-level details (like current user, DB name, server)
  - Session-related info
  - Metadata about database objects
- They do not need table data and often work without FROM clauses.

🏷️ Common System Functions (with Support):

🔹 1. CURRENT_USER

- Returns: The database username that is executing the current SQL command.
- Useful for: Tracking who is executing which SQL logic in multi-user environments.

💡 Example: SELECT CURRENT_USER;

🧾 Output: 'VAIBHAV_USER'

🔹 2. SESSION_USER

- Returns: The name of the user who opened the session, which may differ from the user who is executing the query (CURRENT_USER).
- Useful for: Understanding session context vs. current query user.

💡 Example: SELECT SESSION_USER;

🧾 Output: 'ADMIN'

🔹 3. SYSTEM_USER
- Returns: The system login name of the user running the SQL session (e.g., Windows or OS-level login).
- Useful for: Security auditing and OS-level user tracking.

💡 Example: SELECT SYSTEM_USER;

📜 Output: 'DESKTOP-VAIBHAV\Vaibhav'

🔹 4. CURRENT_DATE
- Returns: The current system date (only the date part).
- Useful for: Logging, date comparisons, filtering current-day records.

💡 Example: SELECT CURRENT_DATE;

📜 Output: 2025-05-23

🔹 5. CURRENT_TIME
- Returns: The current system time (only time part).
- Useful for: Time-based operations like logging or session tracking.

💡 Example: SELECT CURRENT_TIME;

📜 Output: 15:07:58

🔹 6. CURRENT_TIMESTAMP
- Returns: The current system date and time (full timestamp).
- Useful for: Creating timestamps in audit tables or logging.

💡 Example: SELECT CURRENT_TIMESTAMP;

📜 Output: 2025-05-23 15:07:58.123

🔹 7. SYSDATE() *(Oracle/MySQL specific)*
- Returns: The system's current date and time.
- Useful for: Same use cases as CURRENT_TIMESTAMP, but behavior differs slightly across DBMS.

💡 Example:
```
SELECT SYSDATE FROM DUAL;  -- Oracle
-- or
SELECT SYSDATE();          -- MySQL
```

📜 Output: 2025-05-23 15:08:30

🔹 8. DB_NAME() *(SQL Server only)*
- Returns: The name of the current database.
- Useful for: Dynamic database name retrieval in scripts.

💡 Example: SELECT DB_NAME();

📜 Output: 'StudentDB'

🔹 9. DATABASE() *(MySQL/PostgreSQL)*
- Returns: Name of the currently selected database.
- Useful for: Knowing which DB you're working in, especially in dynamic environments.

💡 Example: SELECT DATABASE();

📜 Output: 'hr_management'

🔹 10. VERSION()
- Returns: The version of the current SQL engine.
- Useful for: Debugging or compatibility checks across environments.

💡 Example: SELECT VERSION();
📜 Output: 'PostgreSQL 15.2 (Ubuntu 20.04)'

🔹 11. HOST_NAME() *(SQL Server only)*
- Returns: The host machine name where the SQL Server is running.
- Useful for: Knowing where your DB engine is hosted.

💡 Example: SELECT HOST_NAME();
📜 Output: 'DESKTOP-VAIBHAV'

🔹 12. @@SERVERNAME *(SQL Server only)*
- Returns: Name of the SQL Server instance.
- Useful for: Tracking server names in multi-server environments.

💡 Example: SELECT @@SERVERNAME;
📜 Output: 'SQLPROD01'

📝 Important Notes:
- System functions help you interact with user, session, server, time, and environment details.
- No table is needed for these queries.
- Useful in auditing, logging, and debugging and understanding system behavior.
- SQL Server provides a lot of system functions using @@.

🔁 Summary : System Functions

Function	Purpose	DBMS Supported
CURRENT_USER	User running the query	All
SESSION_USER	Session creator	PostgreSQL, SQL Server
SYSTEM_USER	OS/network login	SQL Server, PostgreSQL
CURRENT_DATE	System date	All
CURRENT_TIME	Current time	All
CURRENT_TIMESTAMP	Date + time	All
SYSDATE()	Oracle/MySQL system date	Oracle, MySQL
DATABASE()	Name of the current DB	MySQL, PostgreSQL
DB_NAME()	Name of current DB	SQL Server
VERSION()	DB engine version	All

HOST_NAME()	Host machine name	SQL Server
@@SERVERNAME	Name of SQL server	SQL Server

◇ 5. XML Functions:

🔷 What Are XML Functions in SQL?

🔹 XML functions allow you to:

- Parse XML data.
- Extract values or nodes using XPath.
- Convert SQL data to XML format.
- Manipulate, transform, or store XML as structured data inside relational databases.

Use Case Example:
A company stores employee info in a column of type XML. You want to extract the <name> tag or split the <skills> into rows — XML functions help you do this.

🔸 XML Functions in Oracle SQL:

- Oracle treats XML as a first-class data type using XMLTYPE.

🔹 1. XMLTYPE():

- Converts a string (VARCHAR, CLOB) to XML format so it can be queried using XML functions.

💡 Example:

```
SELECT XMLTYPE('<emp><name>Vaibhav</name></emp>') AS xml_col FROM dual;
```

📌 *Why it's useful:* You can now use XPath queries on it.

🔹 2. EXTRACT() / EXTRACTVALUE(): Extracts a part of the XML using XPath.

💡 Example:

```
SELECT EXTRACT(XMLTYPE('<emp><name>Vaibhav</name></emp>'),
'/emp/name/text()') AS emp_name FROM dual;
```

📜 Output: Vaibhav

📌 *Use it when:* You want a specific value (like a tag's text).

🔹 3. XMLTABLE():

- Converts XML data into a virtual table so that you can SELECT rows/columns.

💡 Example:

```
SELECT *
FROM XMLTABLE('/employees/employee'
  PASSING
XMLTYPE('<employees><employee><id>1</id></employee></employees>')
  COLUMNS emp_id NUMBER PATH 'id');
```

📜 Output:

```
Emp_id
——------
1
```

📌 *Use it when:* You want to treat XML data as a regular table and extract multiple rows.

🔹 4. XMLSERIALIZE():

- Converts XMLTYPE back into VARCHAR or CLOB to view as plain text.

💡 Example:

```
SELECT XMLSERIALIZE(DOCUMENT XMLTYPE('<a>1</a>') AS VARCHAR2(50))
AS plain_xml FROM dual;
```

📜 Output: <a>1

🔸 XML Functions in SQL Server:

- SQL Server stores XML in columns of type XML.

🔹 1. .value():

- Extract a single scalar value using XPath.

💡 Example:

```
SELECT xml_data.value('(/employee/name)[1]', 'VARCHAR(100)') AS emp_name
FROM employees;
```

📜 Output: Vaibhav

📌 *Use it when:* You want just one tag's value.

🔹 2. .query(): Returns an XML fragment.

💡 Example:

```
SELECT xml_data.query('/employee/name') AS name_xml
FROM employees;
```

📜 Output: <name>Vaibhav</name>

🔹 3. .nodes(): Break XML into multiple rows (nodes).

💡 Example:

```
SELECT x.value('(name)[1]', 'VARCHAR(50)') AS emp_name
FROM employees
CROSS APPLY xml_data.nodes('/employees/employee') AS T(x);
```

📜 Output:

```
Emp_name
——---------
Vaibhav
Akash
```

📌 *Use it when:* You want to normalize XML into rows.

🔹 4. FOR XML: Converts SQL query results into an XML structure.

💡 Example: SELECT emp_id, emp_name FROM employees FOR XML AUTO;

📜 Output:

```
<employees emp_id="1" emp_name="Vaibhav" />
<employees emp_id="2" emp_name="Akash" />
```

📌 *Use it when:* You want to export data in XML format.

🔸 XML Functions in PostgreSQL:

- PostgreSQL uses built-in XML data type and XPath queries.

🔹 1. xpath(): Returns an array of values matching an XPath query.

💡 Example:

```
SELECT xpath('/employee/name/text()', xml
'<employee><name>Vaibhav</name></employee>');
```

🧾 Output: {Vaibhav}

📌 *Use it when:* You want XPath results from an XML column.

📝 Important Notes:

- XML is mostly used in Oracle, SQL Server, and PostgreSQL.
- MySQL does not support advanced XML functions — use JSON instead.
- XML is great for interoperability, especially in web APIs and metadata.
- Learn XPath to query XML more powerfully.
- XML data can be complex to process — it's often better stored as relational if possible.

🔁 Summary : XML Functions

Function	Oracle	SQL Server	PostgreSQL	Description
XMLTYPE()	✅	❌	❌	Converts to XML type
EXTRACT()	✅	❌	❌	Extracts XML node using XPath
XMLTABLE()	✅	❌	❌	Converts XML to rows
XMLSERIALIZE()	✅	❌	❌	Converts XML to string
.value()	❌	✅	❌	Gets a single value from XML
.query()	❌	✅	❌	Returns XML node
.nodes()	❌	✅	❌	Converts XML array into rows
FOR XML	❌	✅	❌	Converts result set to XML
xpath()	❌	❌	✅	Gets values using XPath in array

🔷 VIEWS:

🔹 What is a View in SQL?

- A View is a virtual table in SQL that is created by writing a SELECT query and saving it with a name.
- It doesn't store data physically — it just shows the result of the SELECT query when you use it.

🔹 Real-life Analogy:

- Think of a view like a lens or a filter placed over your data.
- You can choose what data to display, how to format it, and even hide sensitive information — all without touching the original tables.

💻 Syntax

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

🏵 TYPES OF VIEWS (with Multi-line Explanations)

1. Simple View
2. Complex View
3. Materialized View
4. Updatable View
5. Non-Updatable View

◇ 1. Simple View

- A simple view is based on only one table.
- It does not contain any of the following: GROUP BY, JOIN, DISTINCT, subqueries, or aggregate functions.
- It is often used to limit access to specific columns.
- Can be updated easily if it includes only base columns.

💡 Example

```
CREATE VIEW vw_EmployeeNames AS
SELECT Emp_ID, Emp_Name FROM Employees;
```

◇ 2. Complex View

- A complex view is based on multiple tables and can include JOINs, GROUP BY, aggregate functions, and more.
- It is often used to combine data from different tables or summarize data.
- Usually not updatable, especially when it includes groupings or aggregates.

💡 Example

```
CREATE VIEW vw_DeptEmployeeCount AS
SELECT Dept_ID, COUNT(*) AS EmployeeCount
FROM Employees
GROUP BY Dept_ID;
```

◇ 3. Materialized View

- A materialized view is different from a normal view.
- It physically stores the query result, which improves performance for heavy aggregations.
- Data inside a materialized view does not automatically update. It must be refreshed manually or on a schedule.

- Mostly used in data warehouses for reporting and analytics.

💡 Example (Oracle/PostgreSQL):

```
CREATE MATERIALIZED VIEW mv_EmployeeSummary AS
SELECT Dept_ID, COUNT(*) AS EmpCount
FROM Employees
GROUP BY Dept_ID;
```

🔹 🛠 To refresh:

```
REFRESH MATERIALIZED VIEW mv_EmployeeSummary;
```

◇ 4. Updatable View

- An updatable view is one that allows INSERT, UPDATE, and DELETE operations.
- Usually based on a single base table and contains no aggregates or joins.
- Columns in the view should map directly to columns in the table.

💡 Example:

```
CREATE VIEW vw_Salaries AS
SELECT Emp_ID, Emp_Salary FROM Employees;
```

You can then run:

```
UPDATE vw_Salaries SET Emp_Salary = 80000 WHERE Emp_ID = 101;
```

✔ This will update the base table.

◇ 5. Non-Updatable View

- A non-updatable view does not support DML operations.
- This includes views that:
  - Use GROUP BY, DISTINCT, JOINs
  - Include aggregate functions like SUM(), AVG()
- Such views are useful for read-only reports or dashboards.

💡 Example: (non-updatable):

```
CREATE VIEW vw_DeptAvgSalary AS
SELECT Dept_ID, AVG(Emp_Salary) AS AvgSalary
FROM Employees GROUP BY Dept_ID;
```

Trying to update this view will result in an error.

🔹 🔍 Additional Concepts

🔹 🖍 Modify a View

```
CREATE OR REPLACE VIEW vw_EmployeeNames AS
SELECT Emp_ID, Emp_Name FROM Employees;
```

🔹 🗑 Drop a View

```
DROP VIEW vw_EmployeeNames;
```

🔹 View Metadata (See View Definition)

🔹 SQL Server:

```
sp_helptext 'vw_EmployeeNames';
```

🔹 MySQL:

```
SHOW CREATE VIEW vw_EmployeeNames;
```

🔸 Oracle:

```
SELECT TEXT FROM USER_VIEWS WHERE VIEW_NAME =
'VW_EMPLOYEENAMES';
```

📝 Important Notes:

- Views help organize and simplify your SQL queries.
- They improve security by showing only selected columns.
- Views don't store data → they use base table data every time you query them.
- Use Materialized Views to improve performance for large data sets.
- Always check if a view is updatable before trying DML operations.
- Best practice: Use views to isolate business logic and protect data.

🔁 Summary : Views in SQL

View Type	Description
Simple View	Based on one table, no aggregates or joins. Often updatable and simple.
Complex View	Based on multiple tables, often includes joins or aggregates. Read-only.
Materialized View	Stores actual data. Used for performance optimization. Needs to refresh.
Updatable View	Allows INSERT/UPDATE/DELETE. Based on one table, no aggregates.
Non-Updatable View	Read-only. Includes joins, groups, functions. Can't modify the base table.

🔷 Temporary Tables

🔹 What is a Temporary Table?

- A Temporary Table is a special kind of table that stores data temporarily during a session or transaction.
- It behaves just like a normal table, but it gets deleted automatically once the session ends or the transaction completes.
- These tables are useful in scenarios like:
  - Storing intermediate results
  - Complex calculations
  - Simplifying stored procedures
  - Reducing redundant calculations in large queries

🔶 Types of Temporary Tables:

🔸 1. Local Temporary Table (SQL SERVER)

- It is specific to the current user session.
- Other users or sessions cannot access it.
- The table is automatically dropped once the session ends or connection is closed.
- In SQL Server, it starts with a single #.

💻 Syntax (SQL Server):

```
CREATE TABLE #LocalTemp (
   ID INT,
   Name VARCHAR(50)
);
```

💡 Example:

```
INSERT INTO #LocalTemp VALUES (1, 'Vaibhav'), (2, 'Amit');

SELECT * FROM #LocalTemp;
```

📜 Output:

ID	Name
1	Vaibhav
2	Amit

🔸 2. Global Temporary Table (SQL SERVER)

- It is accessible to all sessions/users.
- The table is automatically deleted only when all sessions that use it are closed.
- In SQL Server, it starts with double ##.

💻 Syntax (SQL Server):

```
CREATE TABLE ##GlobalTemp (
   ID INT,
   Status VARCHAR(20)
);
```

💡 Example:

```
INSERT INTO ##GlobalTemp VALUES (1, 'Active'), (2, 'Inactive');

SELECT * FROM ##GlobalTemp;
```

📃 Output:

ID	Status
1	Active
2	Inactive

📌 This table will be accessible from other connections too.

◇ 3. Temporary Table using TEMP or TEMPORARY Keyword (MySQL, PostgreSQL, Oracle)

- Works across many databases.
- Automatically dropped when session ends (similar to local temp).
- Use CREATE TEMP TABLE or CREATE TEMPORARY TABLE.

💻 Syntax (MySQL/PostgreSQL):

```
CREATE TEMPORARY TABLE TempOrders (
   OrderID INT,
   OrderAmount DECIMAL(10,2)
);
```

💡 Example:

```
INSERT INTO TempOrders VALUES (101, 500.00), (102, 750.50);

SELECT * FROM TempOrders;
```

📃 Output:

OrderID	OrderAmount
101	500.00
102	750.50

◇ 4. ON COMMIT DELETE ROWS / PRESERVE ROWS (PostgreSQL Specific)

- Used in PostgreSQL for better control over when rows are deleted from a temporary table.
- You can choose whether data should persist or be deleted after commit.

✅ Options:

- ON COMMIT DELETE ROWS → clears data after each commit.
- ON COMMIT PRESERVE ROWS → keeps data until session ends.

💡 Example:

```
CREATE TEMP TABLE temp_sales (
   sale_id INT,
   amount DECIMAL
) ON COMMIT DELETE ROWS;
```

◇ 5. Common Table Expressions (CTE) – Temporary Result Set

- Not an actual table, but acts like a temporary table during the execution of a single query.
- It is like a named subquery that improves readability and modularity of complex SQL queries.

💻 Syntax

```
WITH TempCTE AS (
   SELECT Emp_ID, Emp_Salary FROM Employees WHERE Dept_ID = 2
)
SELECT * FROM TempCTE WHERE Emp_Salary > 60000;
```

🔧 Modify Temp Table Structure

- You can modify temp tables using standard ALTER TABLE commands if needed.
  ALTER TABLE #LocalTemp ADD Email VARCHAR(100);

🔹🗑️ Dropping Temporary Tables

- Temp tables are usually auto-dropped, but you can drop them manually too:
  DROP TABLE #LocalTemp;

📝 Important Notes:

- Indexes, constraints, and primary keys can be used on temporary tables.
- Temporary tables are often used in stored procedures.
- You cannot use temp tables between sessions unless it's a global temp table.
- In PostgreSQL/MySQL, avoid naming temp tables the same as base tables — they can shadow them.
- Don't forget to drop temp tables manually in long-running apps to save memory.

🔁 Summary : Regular Expression

Type	Description
Local Temp Table	Exists only during the current session. Named with #.
Global Temp Table	Visible to all sessions. Deleted when all sessions close. Named with ##.
TEMPORARY Table (MySQL)	Temporary table, auto-deleted at session end. Uses TEMP keyword.
ON COMMIT Options	PostgreSQL-specific control of row persistence in TEMP tables.
CTE	Temporary result set used only during query execution.

🔷 CTE (Common Table Expression)

- A Common Table Expression (CTE) is a temporary result set defined within the execution scope of a single SELECT, INSERT, UPDATE, or DELETE statement.
- It is like a named subquery that improves readability and modularity of complex SQL queries.
- CTEs are defined using the WITH keyword followed by a query block, which you can reference just like a table in the main SQL query.

🔸 Does CTE get stored in database?

❌ No.

- CTE does not store data (like a table).
- CTE does not store query definition (like a view).
- It's just like a temporary alias for a subquery.
- Once the query finishes execution → the CTE disappears

🔸 Key Features of CTE:

1. Improves Readability
   - Breaks down large queries into logical blocks.
   - Makes SQL cleaner and easier to understand.
2. Reusable in the Same Query
   - Once defined, a CTE can be used multiple times within the main query.
3. Supports Recursion
   - Allows writing recursive queries like hierarchical structures (e.g., org charts, folder paths).
4. Scoped to a Single Query
   - CTEs exist only during the execution of the query.
   - Not stored permanently like a view or table.

💻 Syntax:

```
WITH cte_name AS (
   SELECT ...
)
SELECT * FROM cte_name;
```

💡 Example:

```
WITH high_salary AS (
   SELECT employee_id, salary FROM employees WHERE salary > 50000
)
SELECT * FROM high_salary;
```

📄 Output: Returns employees with salaries above 50,000.

🔹 WITH Clause:

- The WITH clause is used to define a CTE.
- It acts like a temporary table that can be reused in your query.
- It simplifies complex SQL queries by breaking them into simpler pieces.
- The WITH clause is not separate from CTE—it's the keyword used to define a CTE.

📝 Important Notes:

- Exists only during query execution.
- Can be recursive (for hierarchical data).

◇ PIVOT

- The PIVOT operator turns row data into columns.
- It is helpful for generating cross-tab reports.
- It helps to summarize and transform data for better reporting.
- You use PIVOT when you want to turn category values (like months, years, or status) into separate columns.

🧠 Think of it like:

- You're converting data from vertical (rows) into horizontal (columns).

💻 Syntax (Oracle):

```
SELECT * FROM (
  SELECT department, year, sales FROM sales_data
)
PIVOT (
  SUM(sales) FOR year IN (2022, 2023)
);
```

💡 Example:

Department	2022	2023
HR	5000	6000
IT	7000	9000

📝 Important Notes:

- Only supported in Oracle and SQL Server.
- Helps in reporting formats.

◇ UNPIVOT

- The UNPIVOT operator turns columns into rows.
- It is the reverse of PIVOT.
- It converts columns into rows — useful for normalizing wide tables.

🧠 Think of it like:

- You're taking horizontal data (columns) and turning it vertical (rows).

💻 Syntax (Oracle):

```
SELECT * FROM sales_data
UNPIVOT (
  sales FOR year IN (sales_2022 AS '2022', sales_2023 AS '2023')
);
```

💡 Example:

Department	Year	Sales
HR	2022	5000
HR	2023	6000

📝 Important Notes:

- Very useful for data normalization or ETL jobs.
- All these functions (except UNPIVOT and PIVOT) are supported in Oracle, SQL Server, PostgreSQL, and MySQL 8+.
- Use analytical functions with OVER() clauses.
- LEAD() and LAG() are non-aggregate functions used with windowing.
- Recursive CTEs are supported in Oracle, SQL Server, PostgreSQL but not in MySQL (until version 8+).
- Pivoting is DBMS-specific—standard SQL doesn't support it natively.

🔁 Summary :

Feature	Description	Supported In
CTE / WITH	Temporary result set used in query	Oracle, SQL Server, PostgreSQL, MySQL 8+
PIVOT	Converts rows to columns	Oracle, SQL Server
UNPIVOT	Converts columns to rows	Oracle, SQL Server

◇ Recursive CTE in SQL

- A Recursive CTE is a CTE (Common Table Expression) that calls itself to return multiple levels of data.
- A Recursive CTE is a powerful SQL feature that allows a query to refer to itself.
- It's especially useful when working with hierarchical or tree-structured data (e.g., employees reporting to managers, category trees, etc.).

🔹 Structure:

A recursive CTE has two parts:

- Anchor Member: The base result (starting point)
- Recursive Member: The part that references the CTE itself and builds on the result

💻 Syntax:

```
WITH RECURSIVE cte_name (column_list) AS (
   -- Anchor member
   SELECT ...
   FROM ...
   WHERE ...
   UNION ALL
   -- Recursive member
   SELECT ...
   FROM table_name
   JOIN cte_name ON ...
)
SELECT * FROM cte_name;
```

💡 Example: Employee Hierarchy

Let's say you have this employee table:

```
CREATE TABLE employees (
  emp_id INT,
  emp_name VARCHAR(50),
  manager_id INT
);
INSERT INTO employees VALUES
(1, 'CEO', NULL),
(2, 'Manager A', 1),
(3, 'Manager B', 1),
(4, 'Dev A1', 2),
(5, 'Dev A2', 2),
(6, 'Dev B1', 3);
```

🔁 Goal: Find the hierarchical structure of employees starting from the CEO.

💻 Recursive CTE Query:

💡 Example:

```
WITH RECURSIVE emp_hierarchy AS (
   -- Anchor member (CEO has no manager)
   SELECT emp_id, emp_name, manager_id, 1 AS level
   FROM employees
   WHERE manager_id IS NULL
```

```
    UNION ALL

    -- Recursive member: find employees reporting to previous result
    SELECT e.emp_id, e.emp_name, e.manager_id, h.level + 1
    FROM employees e
    JOIN emp_hierarchy h ON e.manager_id = h.emp_id
)
SELECT * FROM emp_hierarchy;
```

📜 Output:

emp_id	emp_name	manager_id	level
1	CEO	NULL	1
2	Manager A	1	2
3	Manager B	1	2
4	Dev A1	2	3
5	Dev A2	2	3
6	Dev B1	3	3

📝 Important Notes:

- WITH RECURSIVE is supported in PostgreSQL, MySQL 8+, SQL Server (using different style), and Oracle 11g+
- Use UNION ALL (not UNION) to avoid performance issues unless duplicates are a concern
- You must ensure the recursion terminates — else it causes infinite loop errors

🔸 Use Cases:

- Employee-manager hierarchy
- Bill of materials
- Category-subcategory tree
- Folder-directory structure
- Graphs (paths in networks)

🔁 Summary : Recursive CTE

Feature	Description
Anchor member	Base query (non-recursive part)
Recursive member	Refers back to the CTE to continue building result
Use case	Tree/hierarchy traversal
Limitation	Needs proper termination condition (avoid infinite loop)
SQL support	PostgreSQL ✅, MySQL 8+ ✅, SQL Server ✅ (CTE style), Oracle ✅

🔷 MERGE Statement (also called UPSERT)

🔹 What is MERGE?

- The MERGE statement is used to perform INSERT, UPDATE, or DELETE operations in a single SQL command, based on whether matching rows exist or not.
- It's also known as UPSERT = UPDATE + INSERT.
- Common in data warehousing, ETL, data sync, and slowly changing dimensions (SCDs).

🔹 When to Use MERGE?

- You have a target table and a source table/data
- If a match is found → UPDATE existing row
- If no match is found → INSERT new row
- (Optional) If match is found and some condition → DELETE

💻 Syntax

```
MERGE INTO target_table t
USING source_table s
ON (t.id = s.id)
WHEN MATCHED THEN
   UPDATE SET t.col1 = s.col1, t.col2 = s.col2
WHEN NOT MATCHED THEN
   INSERT (id, col1, col2)
   VALUES (s.id, s.col1, s.col2);
```

💡 Example

🎯 Scenario: We have two tables:

📊 employees_target

emp_id	name	salary
101	Amit	30000
102	Vaibhav	40000

📊 employees_source

emp_id	name	salary
102	Vaibhav R	45000
103	Rahul	35000

💡 Example:

```
MERGE INTO employees_target tgt
USING employees_source src
ON (tgt.emp_id = src.emp_id)
WHEN MATCHED THEN
   UPDATE SET tgt.name = src.name, tgt.salary = src.salary
WHEN NOT MATCHED THEN
   INSERT (emp_id, name, salary)
   VALUES (src.emp_id, src.name, src.salary);
```

📜 Output:

📊 employees_target becomes:

emp_id	name	salary
101	Amit	30000
102	Vaibhav R	45000
103	Rahul	35000

📝 Important Notes:

Feature	MERGE Behavior
Multiple operations	Yes – INSERT, UPDATE, DELETE in one go
Atomicity	Fully atomic – succeeds or fails together
Conditions	Can be added to both matched & unmatched
Performance	Better than writing separate queries

📌 Notes:

- The ON clause is mandatory and defines how to match rows.
- WHEN MATCHED can have a condition like AND salary <> src.salary.
- WHEN NOT MATCHED only happens if no row is matched in the target.
- MERGE can update, insert, and even delete if conditions are added.
- Oracle, SQL Server, DB2 support MERGE.
- ✅ MySQL and PostgreSQL use INSERT ... ON DUPLICATE KEY UPDATE or INSERT ... ON CONFLICT instead.

🔁 Summary : MERGE Statement

Concept	Description
MERGE	Combines INSERT, UPDATE, DELETE based on matching
Use Case	Data sync, ETL jobs, deduplication
Synonyms	UPSERT
Supported DBs	✅ Oracle, SQL Server, DB2; ❌ MySQL/PostgreSQL (use alternative syntax)

UPSERT in PostgreSQL & MySQL

PostgreSQL - INSERT ... ON CONFLICT

- PostgreSQL uses the ON CONFLICT clause to handle duplicates.

💻 Syntax:

```
INSERT INTO table_name (col1, col2, col3)
VALUES (val1, val2, val3)
ON CONFLICT (conflict_column)
DO UPDATE
SET col2 = EXCLUDED.col2,
   col3 = EXCLUDED.col3;
```

💡 Example:

Assume we have this table in PostgreSQL:

```
CREATE TABLE employees (
  emp_id INT PRIMARY KEY,
  name VARCHAR(50),
  salary INT
);
```

Then run:

```
INSERT INTO employees (emp_id, name, salary)
VALUES (102, 'Vaibhav R', 45000)
ON CONFLICT (emp_id)
DO UPDATE
SET name = EXCLUDED.name,
   salary = EXCLUDED.salary;
```

📜 Output:

emp_id	name	salary
101	Amit	30000
102	Vaibhav R	45000
103	Rahul	35000

MySQL — INSERT ... ON DUPLICATE KEY UPDATE

- MySQL uses ON DUPLICATE KEY UPDATE to achieve the same functionality.

💻 Syntax:

```
INSERT INTO table_name (col1, col2, col3)
VALUES (val1, val2, val3)
ON DUPLICATE KEY UPDATE
col2 = VALUES(col2),
col3 = VALUES(col3);
```

💡 Example:

Assume this MySQL table:

```
CREATE TABLE employees (
  emp_id INT PRIMARY KEY,
  name VARCHAR(50),
  salary INT
);
```

Then run:

```
INSERT INTO employees (emp_id, name, salary)
VALUES (102, 'Vaibhav R', 45000)
ON DUPLICATE KEY UPDATE
name = VALUES(name),
salary = VALUES(salary);
```

Output:
Same output as before — updates existing or inserts new based on PRIMARY KEY.

Important Notes:

Feature	PostgreSQL	MySQL
Clause	ON CONFLICT	ON DUPLICATE KEY UPDATE
Conflict Detection	Any UNIQUE or PRIMARY KEY	Only PRIMARY KEY or UNIQUE
Access to incoming values	Uses EXCLUDED.col_name	Uses VALUES(col_name) (deprecated, now use VALUES() in older versions)
Flexibility in conflict keys	Yes (you can choose conflict columns)	No (only keys defined in schema)

Summary : Upsert

DBMS	UPSERT Syntax Equivalent
Oracle	MERGE INTO ...
SQL Server	MERGE INTO ...
PostgreSQL	INSERT ... ON CONFLICT ... DO UPDATE
MySQL	INSERT ... ON DUPLICATE KEY UPDATE

🔷 Regular Expressions in SQL (REGEXP)

🔹 What is a Regular Expression (Regex)?

- Regular Expression (Regex) is a sequence of characters that defines a search pattern, often used for pattern matching, searching, and string manipulation.
- help you perform advanced string matching beyond simple LIKE or INSTR()

💻 Syntax (General for MySQL & Oracle)

```
SELECT column
FROM table
WHERE column REGEXP 'pattern';
```

🔹 Common REGEXP Operators:

Pattern	Meaning	Example
^	Start of string	^A (starts with A)
$	End of string	end$ (ends with "end")
.	Any single character	b.d matches "bad", "bed", etc.
*	Zero or more of the preceding character	ab*c matches "ac", "abc", "abbc"
+	One or more of the preceding character	ab+c matches "abc", "abbc", but not "ac"
?	Zero or one of the preceding character	ab?c matches "abc" or "ac"
`	`	OR operator
[]	Match any character inside brackets	[aeiou] (vowels)
[^]	Not any character inside brackets	[^0-9] (not a digit)
{n}	Exactly n occurrences	a{3} matches "aaa"
{n,}	n or more occurrences	a{2,}
{n,m}	Between n and m occurrences	a{2,4}

🔹 Examples (MySQL / Oracle)

1. Match Names Starting with "A"

💡 Example:

```
SELECT name
FROM employees
WHERE name REGEXP '^A';
```

📜 Output: Names like "Alice", "Andrew", etc.

2. Names Ending with "y"

💡 Example:

```
SELECT name
FROM employees
WHERE name REGEXP 'y$';
```

📜 Output: Names like "Mary", "Anthony"

3. Names Containing Digits

💡 Example:

```
SELECT name
FROM employees
WHERE name REGEXP '[0-9]';
```

📜 Output: "John123", "Emp007"

4. Names That Do Not Contain Vowels

💡 Example:

```
SELECT name
FROM employees
WHERE name REGEXP '^[^aeiouAEIOU]+$';
```

📜 Output: "Rrr", "Bcd"

5. Names with At Least Two Consecutive "a"

💡 Example:

```
SELECT name
FROM employees
WHERE name REGEXP 'a{2,}';
```

📜 Output: "Aarav", "Saaab"

🔹 Oracle Specific REGEXP Functions

🔸 REGEXP_LIKE(): Checks if string matches a pattern (boolean filter)

💡 Example:

```
SELECT emp_name
FROM employees
WHERE REGEXP_LIKE(emp_name, '^A');
```

🔸 REGEXP_INSTR(): Returns position of pattern in string

💡 Example: SELECT REGEXP_INSTR('abc123xyz', '[0-9]+') AS position FROM dual;

📜 Output: 4 (position of first number)

🔸 REGEXP_SUBSTR(): Returns substring matching a pattern

💡 Example: SELECT REGEXP_SUBSTR('abc123xyz', '[0-9]+') AS number FROM dual;

📜 Output: 123

🔸 REGEXP_REPLACE(): Replaces substring matching pattern with something else

💡 Example: SELECT REGEXP_REPLACE('abc123xyz', '[0-9]', '#') AS masked FROM dual;

📜 Output: abc###xyz

📝 Important Notes:

- Use REGEXP carefully, as it's powerful but slower than LIKE.
- Useful for data validation, cleaning, text filtering, and complex searches.
- MySQL and Oracle have better REGEXP support than SQL Server (which is more limited).
- SQL Server uses LIKE, CHARINDEX, or CLR integration for regex-like behavior.

🔁 Summary : Regular Expression

Feature	Description
REGEXP_LIKE	Boolean test for regex pattern
REGEXP_INSTR	Position of regex match in string
REGEXP_SUBSTR	Extract substring based on regex
REGEXP_REPLACE	Replace substring using regex
Common use cases	Validation, formatting, pattern matching
SQL support	Oracle ✅, MySQL ✅, PostgreSQL ✅, SQL Server ❌ (limited)

🔷 Indexes

🔹 What is an Index in SQL?

- An index in SQL is a performance-tuning method used to speed up the retrieval of data from a database table.
- It works like an index in a book — instead of searching each page (row), the index lets you jump directly to the page (row) containing the data.
- Indexes are especially useful for large tables and are automatically used by the SQL engine when you run queries.

🔹 Why Use Indexes?

- To speed up SELECT queries and WHERE clause searches.
- To improve JOIN performance.
- Helps with ORDER BY, GROUP BY, and DISTINCT operations.

🔹 How Index Works?

- Indexes are created on one or more columns.
- Behind the scenes, most databases use a B-Tree or Hash structure for indexing.
- When you query the table, the index allows faster lookup compared to scanning every row.

🏵 Types of Indexes:

1. Single-Column Index
2. Composite Index (Multi-Column Index)
3. Unique Index
4. Clustered Index (❌Oracle)
5. Non-Clustered Index
6. Full-Text Index (MySQL, SQL Server)
7. Bitmap Index (Oracle, Data Warehouses)
8. Filtered Index (SQL Server, ❌Oracle)

◇ 1. Single-Column Index

- Indexes built on a single column of a table.

- Best for queries filtering or sorting by that specific column.
- Simple and fast to create.

💻 Syntax:

```
CREATE INDEX idx_employee_name ON Employees(Name);
```

💡 Example:

```
SELECT * FROM Employees WHERE Name = 'Vaibhav';
```

◇ 2. Composite Index (Multi-Column Index)

- An index on two or more columns.
- Improves performance for queries using multiple columns in WHERE or JOIN.
- The order of columns in the index matters.

💻 Syntax:

```
CREATE INDEX idx_emp_dept ON Employees(DepartmentID, Name);
```

💡 Example:

```
SELECT * FROM Employees WHERE DepartmentID = 3 AND Name = 'Vaibhav';
```

📌 Note: This index is effective only if DepartmentID is included in WHERE clause (left-to-right rule).

◇ 3. Unique Index

- Ensures that all values in the indexed column(s) are unique.
- Automatically created when you define a PRIMARY KEY or UNIQUE constraint.

💻 Syntax:

```
CREATE UNIQUE INDEX idx_unique_email ON Employees(Email);
```

💡 Example:

```
INSERT INTO Employees (Email) VALUES ('vaibhav@example.com'); -- Allowed
INSERT INTO Employees (Email) VALUES ('vaibhav@example.com'); -- Error
```

◇ 4. Clustered Index

- The actual data is stored in the order of the index.
- Each table can have only one clustered index because data can be sorted only one way.
- Automatically created with the PRIMARY KEY in many databases.

💻 Syntax (SQL Server):

```
CREATE CLUSTERED INDEX idx_emp_id ON Employees(EmployeeID);
```

💡 Example:

```
SELECT * FROM Employees ORDER BY EmployeeID; -- Fast
```

◇ 5. Non-Clustered Index

- The index contains pointers to the actual table rows.
- Can be multiple non-clustered indexes on a table.
- Faster than scanning the whole table but slower than clustered index for range queries.

💻 Syntax

```
CREATE NONCLUSTERED INDEX idx_emp_name ON Employees(Name);
```

💡 Example:

```
SELECT * FROM Employees WHERE Name = 'Vaibhav';
```

◇ 6. Full-Text Index (MySQL, SQL Server)

- Used for searching large blocks of text like blogs, articles, descriptions.
- Enables MATCH ... AGAINST or CONTAINS style queries.

💻 Syntax:

```
CREATE FULLTEXT INDEX idx_description ON Products(Description);
```

💡 Example:

```
SELECT * FROM Products WHERE MATCH(Description) AGAINST('wireless');
```

◇ 7. Bitmap Index (Oracle, Data Warehouses)

- Used for columns with low cardinality (few distinct values, like Gender, Yes/No).
- Very efficient for data warehouses and analytical queries.

💡 Example: (Oracle):

```
CREATE BITMAP INDEX idx_gender ON Employees(Gender);
```

◇ 8. Filtered Index (SQL Server)

- An index with a WHERE condition, so it only indexes a subset of rows.
- Useful when most values in a column are NULL or you query a specific value frequently.

💻 Syntax:

```
CREATE INDEX idx_active_emps ON Employees(Status) WHERE Status = 'Active';
```

💡 Example:

```
SELECT * FROM Employees WHERE Status = 'Active'; -- Fast using filtered index
```

🔍 View Existing Indexes:

🔸 SQL Server:

```
EXEC sp_helpindex 'Employees';
```

🔸 PostgreSQL:

```
SELECT * FROM pg_indexes WHERE tablename = 'employees';
```

🔸 MySQL:

```
SHOW INDEXES FROM Employees;
```

🔹🗑️ Dropping Indexes

💻 Syntax:

```
DROP INDEX idx_employee_name; -- SQL Server, Oracle
DROP INDEX idx_employee_name ON Employees; -- MySQL, PostgreSQL
```

📝 Important Notes:

- Indexes increase read performance but can slow down insert/update/delete operations because indexes also need to be updated.
- Use indexes only when needed — don't over-index.
- For large-scale OLAP systems, bitmap indexes and filtered indexes are more efficient.
- Always check query execution plans to ensure indexes are being used.
- Use Composite Indexes carefully: follow the leftmost prefix rule.

🔁 Summary : SQL Index

Index Type	Description
Single-Column	Index on one column. Improves filtering/sorting.
Composite	Multi-column index. Useful for multi-condition queries.
Unique	Prevents duplicate values. Enforces data integrity.
Clustered	Data is physically stored in order. Only one per table.
Non-Clustered	Pointers to actual rows. Multiple allowed.
Full-Text	Optimized for text searches.
Bitmap	Efficient for few distinct values (analytics).
Filtered	Indexes a subset of rows. Good for NULLs or common values.

🔷 SQL Optimization Concepts

🔹 🔍 1. Query Execution Plan

🔸 What is a Query Execution Plan?

- A query execution plan (QEP) is like a map that tells you how your SQL query is executed internally by the database.
- It shows which indexes are used, which joins are performed, in what order, and how many rows are processed.
- It helps you understand why a query is slow and how to make it faster.

🔸 Why is it Important?

- It helps identify bottlenecks (e.g., full table scans).
- You can fine-tune queries for better performance.
- It shows cost estimates for each operation.

🔸 How to View the Execution Plan?

🔹 1. SQL Server:

```
-- Show estimated plan
SET SHOWPLAN_ALL ON;
GO
SELECT * FROM Employees WHERE Name = 'Vaibhav';
GO
SET SHOWPLAN_ALL OFF;
```

OR (easier in SSMS):

```
-- Click on "Display Estimated Execution Plan" (Ctrl + L)
```

🔹 2. PostgreSQL:

```
EXPLAIN SELECT * FROM Employees WHERE Name = 'Vaibhav';
```

🔹 3. MySQL:

```
EXPLAIN SELECT * FROM Employees WHERE Name = 'Vaibhav';
```

✅ Sample Output (simplified):

id	select_type	table	type	key	rows	Extra
1	SIMPLE	Employees	index	idx_emp_name	10	Using index

- key: the index used.
- type: shows how the table was accessed (e.g., ALL, index, ref).
- rows: number of rows processed (lower is better).

🔹 ⚙️ 2. Indexing Tips for Optimization

🔸 How to Use Indexes Wisely

1. Create indexes on columns used in:
    - WHERE, JOIN, ORDER BY, GROUP BY.
2. Use composite indexes when you query using multiple columns.
3. Avoid indexing:
    - Columns with too many NULLs or updates.
    - Columns that are rarely used in filters.

💡 Example : Slow vs Optimized Query

❌ Without index (slow):

```
SELECT * FROM Employees WHERE Email = 'v@example.com';
```

✅ Create index:

```
CREATE INDEX idx_email ON Employees(Email);
```

✅ With index (fast):

```
SELECT * FROM Employees WHERE Email = 'v@example.com';
```

🔍 Execution plan will now show idx_email is used.

🔹 🚦 3. Performance Tuning Basics

🔸 What is Performance Tuning?

- It means improving speed and efficiency of your SQL queries.
- You tune your queries, indexes, or schema to reduce resource usage (CPU, RAM, Disk I/O).

🔸 🔧 Basic Tuning Techniques:

A. SELECT Only Required Columns

❌ Bad:

```
SELECT * FROM Employees;
```

✅ Good:

```
SELECT Name, Email FROM Employees;
```

Reason: SELECT * fetches all columns even if not needed.

B. Use WHERE Clause to Filter Early

❌ Bad:

```
SELECT Name FROM Employees;
-- Retrieves entire table then filters in app
```

✅ Good:

```
SELECT Name FROM Employees WHERE Department = 'IT';
```

C. Avoid Functions on Indexed Columns

❌ Index won't work:

```
SELECT * FROM Employees WHERE UPPER(Name) = 'VAIBHAV';
```

✅ Let index help:

```
SELECT * FROM Employees WHERE Name = 'Vaibhav';
```

D. Use JOINS Smartly

- Use appropriate JOINs (INNER, LEFT, etc.)
- Always join on indexed keys

E. Use LIMIT or TOP if You Need Few Rows

💡 Example:

```
SELECT TOP 10 * FROM Employees ORDER BY Salary DESC;
-- SQL Server
SELECT * FROM Employees ORDER BY Salary DESC LIMIT 10;
– MySQL/PostgreSQL
```

🔹 📘 Execution Plan Types (Common Terms Explained)

Term	Meaning
Table Scan	Reads every row (slowest)
Index Scan	Reads every row in index (better, but not ideal)
Index Seek	Jumps directly to matching row using index (fastest)
Nested Loops Join	Good for small data sets
Hash Join	Efficient for large unsorted data
Merge Join	Best for pre-sorted data sets

📝 Important Notes:

- Always analyze query plans to tune performance.
- Create indexes based on actual query patterns.
- Test and compare query speed with and without indexes.
- Use tools like SQL Server Profiler, PostgreSQL EXPLAIN ANALYZE, or MySQL Workbench Visual EXPLAIN for deeper insights.

🔁 Summary : Key Optimization Concept

Concept	Description
Query Execution Plan	Shows how your query runs internally (like a GPS map)
Indexing	Improves speed for read-heavy queries, use wisely
Selective Columns	Use only needed columns, avoid SELECT *
Function-Free WHERE	Don’t use functions on columns in WHERE clause
Proper Joins	Use indexed keys and choose right type of JOIN
Use LIMIT/TOP	Always fetch only what you need

Oracle SQL: Optimization Concepts

🔹 🔍 1. Query Execution Plan in Oracle

🔸 What is It in Oracle?

- In Oracle, the execution plan shows the steps Oracle takes to execute a SQL statement, including:
  - Access paths (table scan vs index scan),
  - Join methods,
  - Sort operations, etc.

🔸 💉 How to View It in Oracle?

Option 1: EXPLAIN PLAN

```
EXPLAIN PLAN FOR
SELECT * FROM Employees WHERE Name = 'Vaibhav';

SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
```

Option 2: AUTOTRACE in SQL*Plus or SQL Developer

```
SET AUTOTRACE ON;
SELECT * FROM Employees WHERE Name = 'Vaibhav';
```

Output includes:

- The execution plan
- Number of consistent gets and physical reads (I/O stats)
- Indexes used (if any)

🔹 ⚙️ 2. Indexing in Oracle SQL

- Oracle supports many types of indexes. The most common are:

🔸 B-Tree Index (Default)

- Fast for equality and range queries.
- Used in WHERE, JOIN, ORDER BY.

```
CREATE INDEX idx_name ON Employees(Name);
```

🔸 Bitmap Index

- Ideal for low-cardinality columns (e.g., gender, region).
- Not suitable for frequently updated columns.

```
CREATE BITMAP INDEX idx_gender ON Employees(Gender);
```

🔸 Function-Based Index

- Indexes on expressions or functions.

```
CREATE INDEX idx_upper_name ON Employees(UPPER(Name));
```

And then query:

```
SELECT * FROM Employees WHERE UPPER(Name) = 'VAIBHAV';
```

🔸 Composite Index

- Multi-column index.

```
CREATE INDEX idx_dept_salary ON Employees(Department, Salary);
```

📌 Oracle uses the leftmost column(s) for index lookups.

🔹 🚦 3. Performance Tuning in Oracle SQL

📌 Key Tips:

1. Use Bind Variables to Avoid Hard Parsing:
   ```
   SELECT * FROM Employees WHERE Name = :name;
   ```
   - Helps Oracle reuse execution plans.
2. Avoid Functions on Indexed Columns (unless using function-based index)
3. Use Hints (Advanced Tuning)
   Oracle provides hints to guide the optimizer:
   SELECT /*+ INDEX(Employees idx_name) */ * FROM Employees WHERE Name = 'Vaibhav';
4. Use Statistics and Gather Regularly
   Oracle relies on table/index statistics for optimization.
   EXEC DBMS_STATS.GATHER_TABLE_STATS('HR', 'EMPLOYEES');
5. Check Cost, Cardinality, Bytes in Plan Output
   SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
   Look at:
   - Cost: Estimated effort to run the query.
   - Cardinality: Estimated number of rows returned.

🔹 Oracle-Specific Tools for Tuning

Tool	Purpose
EXPLAIN PLAN	Shows estimated plan
AUTOTRACE	Shows actual statistics
DBMS_XPLAN	Better formatting of plan
SQL Tuning Advisor	Offers tuning suggestions in Oracle Enterprise
AWR / ADDM Reports	Historical performance diagnostics

🔁 Summary :

Concept	Oracle-Specific Feature or Note
Execution Plan	Use EXPLAIN PLAN + DBMS_XPLAN.DISPLAY
Index Types	B-Tree (default), Bitmap (for few distinct values), Function-based
Optimizer Use	Oracle Cost-Based Optimizer (CBO) chooses best plan using stats
Statistics Gathering	Crucial for performance — use DBMS_STATS regularly
Bind Variables	Improve plan reuse and parsing efficiency
SQL Hints	Manually influence execution plan if needed

🔷 📦 Understanding Data Export and Import

- Exporting Data: Extracting data from a database and saving it in a file format (like .sql or .csv) for backup, migration, or analysis.CodeLucky
- Importing Data: Loading data from external files into a database.

🔶 🐬 MySQL

🔹 Exporting Data

1. Using mysqldump:
   - Exports the entire database or specific tables.
     DBACLASS+9Kimis.org - We make games reality+9Coefficient+9
     mysqldump -u username -p database_name > backup.sql

2. Exporting Specific Tables:
   mysqldump -u username -p database_name table1 table2 > tables_backup.sql

3. Exporting Data to CSV:
   ```
   SELECT * FROM table_name
   INTO OUTFILE '/path/to/file.csv'
   FIELDS TERMINATED BY ','
   ENCLOSED BY '"'
   LINES TERMINATED BY '\n';
   ```

🔹 Importing Data

1. Importing SQL Dump:
   mysql -u username -p database_name < backup.sql
2. Importing CSV Data:
   ```
   LOAD DATA INFILE '/path/to/file.csv'
   INTO TABLE table_name
   FIELDS TERMINATED BY ','
   ENCLOSED BY '"'
   LINES TERMINATED BY '\n';
   ```

🔶 🔴 Oracle SQL

🔹 Exporting Data

1. Using Data Pump (expdp):
   expdp username/password DIRECTORY=dir_name DUMPFILE=export.dmp LOGFILE=export.log FULL=Y

2. Exporting Specific Schema:
   expdp username/password SCHEMAS=schema_name DIRECTORY=dir_name DUMPFILE=schema_export.dmp LOGFILE=schema_export.log

3. Exporting Table Data to CSV:
   ```
   SPOOL /path/to/file.csv
   SELECT * FROM table_name;
   SPOOL OFF;
   ```

- Importing Data

    1. Using Data Pump (impdp):
        ```
        impdp username/password DIRECTORY=dir_name DUMPFILE=export.dmp
        LOGFILE=import.log FULL=Y
        ```

    2. Importing Specific Schema:
        ```
        impdp username/password SCHEMAS=schema_name DIRECTORY=dir_name
        DUMPFILE=schema_export.dmp LOGFILE=schema_import.log
        ```

    3. Importing Data Using SQL*Loader:
        Control File (control.ctl):
        ```
        LOAD DATA
        INFILE '/path/to/file.csv'
        INTO TABLE table_name
        FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
        TRAILING NULLCOLS
        (column1, column2, column3)
        ```
    Command:
        ```
        sqlldr username/password control=control.ctl log=loader.log
        ```

PostgreSQL

- Exporting Data
    1. Using pg_dump:
        ```
        pg_dump -U username -W -F c -b -v -f backup.dump database_name
        ```
    2. Exporting Table Data to CSV:
        ```
        COPY table_name TO '/path/to/file.csv' DELIMITER ',' CSV HEADER;
        ```

- Importing Data
    1. Restoring from Dump:
        ```
        pg_restore -U username -W -d database_name -v backup.dump
        ```
    2. Importing CSV Data:
        ```
        COPY table_name FROM '/path/to/file.csv' DELIMITER ',' CSV HEADER;
        ```

SQL Server

- Exporting Data

    1. Using SQL Server Management Studio (SSMS):
        - Navigate to Tasks > Export Data.
        - Follow the wizard to export data to desired format (e.g., CSV, Excel).

    2. Using bcp Utility:

        ```
        bcp database_name.schema_name.table_name out "C:\path\to\file.csv" -c -t, -S
        server_name -U username -P password
        ```

- Importing Data

    1. Using SSMS:
        - Navigate to Tasks > Import Data.

- Follow the wizard to import data from various sources.

2. Using bcp Utility:

```
bcp database_name.schema_name.table_name in "C:\path\to\file.csv" -c -t, -S
server_name -U username -P password
```

3. Using BULK INSERT:

```
BULK INSERT schema_name.table_name
FROM 'C:\path\to\file.csv'
WITH (
  FIELDTERMINATOR = ',',
  ROWTERMINATOR = '\n',
  FIRSTROW = 2
);
```

🔷 Real-time Use Cases

1. ETL Data Queries
2. Report Generation
3. Data Validation

🔹 1. ETL DATA QUERIES

- ETL stands for:
  - Extract → Get data from source tables.
  - Transform → Modify or clean data.
  - Load → Insert into target tables (often for reporting or analytics).

- 🧠 Real-Life Scenario:
  - Let's say you work in a bank.
  - Customer and Transaction data are stored in raw tables.
  - You want to prepare a summary table of customer spending per month.

✔ Step 1: Extract

- Extract data from source.

```
-- Common for all SQL engines
SELECT * FROM transactions;
```

✔ Step 2: Transform

- Use aggregation, filters, joins, etc.

```
-- Monthly total amount spent by customer
SELECT
   customer_id,
   DATE_FORMAT(transaction_date, '%Y-%m') AS month,
   SUM(amount) AS total_spent
FROM transactions
GROUP BY customer_id, DATE_FORMAT(transaction_date, '%Y-%m');
```

🧮 Differences across SQLs:

SQL Engine	Syntax Difference
MySQL	DATE_FORMAT(transaction_date, '%Y-%m')
PostgreSQL	TO_CHAR(transaction_date, 'YYYY-MM')
SQL Server	FORMAT(transaction_date, 'yyyy-MM')
Oracle SQL	TO_CHAR(transaction_date, 'YYYY-MM')

✔ Step 3: Load

- Insert into the summary table.

```
INSERT INTO monthly_summary (customer_id, month, total_spent)
SELECT
   customer_id,
   TO_CHAR(transaction_date, 'YYYY-MM') AS month,
   SUM(amount)
```

```
FROM transactions
GROUP BY customer_id, TO_CHAR(transaction_date, 'YYYY-MM');
```

📌 Tip: In real projects, ETL tools like Informatica, Airflow, or SSIS automate these steps.

🔹 2. REPORT GENERATION

- Purpose: To prepare reports for management, clients, or dashboards.

🧠 Real-Life Scenario:

  - You want to create a report of top 5 customers by spending for this month.

- MySQL, PostgreSQL

```
SELECT customer_id, SUM(amount) AS total_spent
FROM transactions
WHERE MONTH(transaction_date) = MONTH(CURDATE())
GROUP BY customer_id
ORDER BY total_spent DESC
LIMIT 5;
```

- SQL Server:

```
SELECT TOP 5 customer_id, SUM(amount) AS total_spent
FROM transactions
WHERE MONTH(transaction_date) = MONTH(GETDATE())
GROUP BY customer_id
ORDER BY total_spent DESC;
```

- Oracle SQL:

```
SELECT customer_id, SUM(amount) AS total_spent
FROM transactions
WHERE EXTRACT(MONTH FROM transaction_date) = EXTRACT(MONTH
FROM SYSDATE)
GROUP BY customer_id
ORDER BY total_spent DESC
FETCH FIRST 5 ROWS ONLY;
```

🔹 3. DATA VALIDATION

- Purpose: To check data quality, duplicates, nulls, business rule failures, etc.

🧠 Real-Life Scenario:

You want to ensure that:

- No null values exist in amount.
- Each transaction has a valid customer.

A. Check for NULLs

```
SELECT * FROM transactions WHERE amount IS NULL;
```

B. Check Invalid Foreign Keys

```
-- Transactions where customer_id does NOT exist in customer table
SELECT *
FROM transactions t
LEFT JOIN customers c ON t.customer_id = c.customer_id
WHERE c.customer_id IS NULL;
```

C. Duplicate Records

```
-- Finding duplicate transaction_ids
SELECT transaction_id, COUNT(*)
FROM transactions
GROUP BY transaction_id
HAVING COUNT(*) > 1;
```

📃 Final Output Example (Assume transactions table)

customer_id	month	total_spent
101	2025-06	12,500.00
102	2025-06	11,400.00
103	2025-06	10,900.00

🧾 Summary : Syntax Differences

Action	MySQL	PostgreSQL	SQL Server	Oracle SQL
Date format	DATE_FORMAT()	TO_CHAR()	FORMAT()	TO_CHAR()
Limit rows	LIMIT 5	LIMIT 5	TOP 5	FETCH FIRST 5 ROWS ONLY
Current date	CURDATE()	CURRENT_DATE	GETDATE()	SYSDATE

Summary :

Database	Export Tool/Command	Import Tool/Command	Notes
MySQL	mysqldump, SELECT INTO OUTFILE	mysql, LOAD DATA INFILE	Simple and widely used
Oracle SQL	expdp, SPOOL	impdp, SQL*Loader	Powerful with advanced options
PostgreSQL	pg_dump, COPY TO	pg_restore, COPY FROM	Flexible and efficient
SQL Server	SSMS Export Wizard, bcp	SSMS Import Wizard, bcp, BULK INSERT	Integrated tools with GUI support

🔷 OLTP (Online Transaction Processing) vs OLAP (Online Analytical Processing)

Feature	OLTP	OLAP
1. Purpose	Handles day-to-day operations (insert, update, delete)	Helps analyze data for decision making
2. Users	End users like cashiers, clerks, customers	Managers, analysts, business users
3. Operations Type	Read + Write (Insert, Update, Del)	Mostly Read (Query, Analyze)
4. Speed	Very fast for small transactions	Slower (but optimized for complex queries)
5. Data Volume	Handles small amounts of data per transaction	Works with large volumes of historical data
6. Examples	ATM withdrawal, online shopping, ticket booking	Sales analysis, profit comparison, forecasting
7. Data Structure	Highly normalized (many small, related tables)	Denormalized (fewer, bigger tables like star schema)
8. Query Complexity	Simple, fast queries	Complex queries with aggregations (SUM, AVG, etc.)
9. Backup Frequency	Regular & frequent (daily or even hourly)	Less frequent (weekly or monthly)
10. Example Systems	Banking, e-commerce, railway reservation systems	Business Intelligence tools, Data Warehouses
11. Data Type	Current, real-time data	Historical data

🧠 Quick Example to Understand:

- OLTP = Swiggy/Zomato order system 🍔
  - You place an order → DB updates order table with your payment & food details.
  - Needs to be fast & accurate for thousands of customers at the same time.

- OLAP = Swiggy Business Dashboard 📊
  - Management checks: *"What was the total revenue last month?"*, *"Which city orders the most pizzas?"*.
  - Needs historical + summarized data for decisions.

PL/SQL

🔷 Basic Introduction to PL/SQL :

🏵 What is PL/SQL?

- PL/SQL (Procedural Language/Structured Query Language) is Oracle's extension to SQL, used to write code blocks that include programming constructs like variables, loops, conditions, and exception handling.
- It is Oracle Corporation's procedural extension to SQL.
- 🧠 Think of PL/SQL as a way to make SQL “smart” — it lets you add logic and flow control to your database tasks.

🔹 2. What Makes PL/SQL Special?

- SQL is declarative: you just ask "what to do?" (e.g., SELECT, INSERT).
- PL/SQL is procedural: you also control "how to do it" (e.g., using IF, LOOP, etc.).
- 💡 PL/SQL = SQL + Programming Logic

🔹 Why do we use PL/SQL?

Reason	Explanation
✅ Procedural Logic	You can write IF-ELSE, loops, and conditions.
✅ Performance	You send one PL/SQL block, not multiple SQL queries -- faster and efficient.
✅ Reusability	You can create procedures and functions to reuse code.
✅ Security	You can hide complex logic inside procedures or packages.
✅ Error Handling	You can handle runtime errors using EXCEPTION blocks.

🔹 Components of PL/SQL

Part	Description
1. Block	Group of statements executed together.
2. Variables	Temporary memory to store values.
3. SQL Statements	You can use SELECT, INSERT, etc.
4. Control Statements	IF-ELSE, LOOP, FOR, WHILE etc.
5. Exception Handling	Manages runtime errors gracefully.

🔹 🧠 Real-Life Scenario

- Problem: You want to check if a customer's bill is greater than ₹1000 and give them a discount.
- In SQL, you'd need manual checking for each row.
- In PL/SQL, you can write:

```
DECLARE
  bill_amount NUMBER := 1200;
BEGIN
  IF bill_amount > 1000 THEN
    DBMS_OUTPUT.PUT_LINE('Apply 10% Discount');
  ELSE
    DBMS_OUTPUT.PUT_LINE('No Discount');
  END IF;
END;
```

📜 Output:

```
Apply 10% Discount
```

📝 Important Notes: When to use PL/SQL over SQL?

Use Case	SQL	PL/SQL
Run one-time query	✅	❌
Apply business rules	❌	✅
Execute multiple statements	❌	✅
Handle logic + error handling	❌	✅
Reusable logic (functions/procs)	❌	✅

🔷 Difference between SQL vs PL/SQL

🔸 Feature	✅ SQL	✅ PL/SQL
1. Full Form	Structured Query Language	Procedural Language / Structured Query Language
2. Type	Declarative language	Procedural programming language
3. Purpose	Used to query and manipulate data	Used to write full programs with logic and control
4. Executes	One statement at a time	A block of code (can have multiple SQL statements)
5. Can it contain logic?	❌ No logic like IF, LOOP, etc.	✅ Yes – has IF, FOR, WHILE, CASE, etc.
6. Code Structure	Simple statements like SELECT, INSERT	Structured block: DECLARE - BEGIN - EXCEPTION - END
7. Use of Variables	❌ Not possible	✅ Yes, variables can be declared and used
8. Reusability	❌ Every time you must write queries again	✅ Write once, reuse as Procedures, Functions, or Packages
9. Performance	Less efficient for complex operations	More efficient as it reduces multiple database calls
10. Error Handling	❌ Limited (only SQL error codes)	✅ Full error handling with EXCEPTION blocks
11. Interaction With DB	Directly interacts with data	Can call SQL within its block to interact with data
12. Used For	CRUD operations – SELECT, UPDATE, etc.	Automating logic – validations, calculations, loops
13. Example	SELECT * FROM EMP;	See example below for full PL/SQL block
14. Can call each other?	SQL cannot call PL/SQL	PL/SQL can run SQL inside it
15. Best For	Simple queries and data fetching	Complex programs, business logic, automation

◇ What is a PL/SQL Block?

- PL/SQL code is always written inside a block structure. A block is like a program with a beginning, logic, and end.
- 📌 Use SET SERVEROUTPUT ON; in SQL*Plus, Oracle SQL Developer, or LiveSQL to see output.

- Basic Structure:

```
DECLARE
  -- variable declarations (optional)
BEGIN
  -- executable statements
EXCEPTION
  -- error handling (optional)
END;
```

◈ 1. Anonymous Block in PL/SQL:

- It is a temporary block of PL/SQL code.
- It does not have a name.
- It cannot be stored in the database for reuse.
- Mostly used for quick testing, debugging, and ad-hoc operations.

💻 Syntax

```
DECLARE
  -- Declare variables (optional)
BEGIN
  -- Write logic here
EXCEPTION
  -- Error handling (optional)
END;
```

💡 Example

```
DECLARE
  name VARCHAR2(20) := 'Vaibhav';
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello ' || name || ', Welcome to PL/SQL!');
END;
```

📜 Output:

```
Hello Vaibhav, Welcome to PL/SQL!
```

📝 Important Notes:

1. It is not stored in the Oracle database.
2. It's like a scratchpad for running PL/SQL code quickly.
3. Can include variables, control structures, and SQL queries.
4. Ideal for testing small logic or debugging.
5. Every time you want to run it, you need to retype it.

🧠 Real-Life Analogy:

- Like writing a reminder on a sticky note — it's useful now, but not saved permanently.

◈ 2. Named Block in PL/SQL:

- It is a reusable PL/SQL block.
- It has a name and can be stored permanently in the Oracle database.
- Can be called/executed again and again.

- Mainly used for modular, large applications.

🔸 Types of Named Blocks:

1. Procedures
2. Functions
3. Packages
4. Triggers

💻 Syntax of a Named Procedure:

```
CREATE OR REPLACE PROCEDURE greet_user IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello from a Named Block!');
END greet_user;
```

- To execute it:

```
BEGIN
  greet_user;
END;
```

📜 Output:

```
Hello from a Named Block!
```

📝 Important Notes:

1. Named blocks are stored in the database.
2. They are reusable, and callable by applications or users.
3. They are modular, which improves code organization and performance.
4. Can have parameters, logic, loops, and exception handling.
5. Best suited for complex logic, batch jobs, or application back-end code.

🧠 Real-Life Analogy:

- Like saving a Microsoft Word template – you can reuse it again and again without typing it from scratch.

📝 Tips for Remembering:

🔹 Anonymous Block:

- Think "A" for Ad-hoc
- Good for temporary tasks, learning, or debugging

🔹 Named Block:

- Think "N" for Name + Reusability
- Use when logic needs to be used again or shared with others

🔁 Summary : Anonymous vs Named Block

Feature	Anonymous Block	Named Block
Has a Name?	❌ No name	✅ Has a name
Stored in DB?	❌ No (temporary)	✅ Yes (permanent)
Reusable?	❌ Not reusable	✅ Reusable
Called by Programs?	❌ No	✅ Yes (via EXEC or BEGIN block)
Used For	Quick test, learning, debugging	Application logic, reports, stored tasks
Example Use Case	Show welcome message	Calculate tax, send report, update salaries

🔷 PL/SQL Variables

🔹 What is a Variable in PL/SQL?

- A variable is a named memory location that stores a value during the execution of a PL/SQL block.
- You can use it to store, change, pass, or return data.

🔹 Declaring a Variable

💻 Syntax

```
variable_name data_type [:= initial_value];
```

💡 Example

```
DECLARE
  message VARCHAR2(30) := 'Good Morning Vaibhav!';
BEGIN
  DBMS_OUTPUT.PUT_LINE(message);
END;
```

📜 Output

```
Good Morning Vaibhav!
```

🔶 Variable Scope in PL/SQL

- Scope refers to the region or block of the PL/SQL program where a variable can be accessed or is visible.
- Variables can be local to a block or global to multiple blocks, depending on where they are declared.

🔹 Types of Scope

🔸 1. Local Scope (Inner Block)

- Variables are declared inside a block.
- Only accessible within that block.
- Other blocks cannot use it.

💻 Syntax:

```
DECLARE
  emp_name VARCHAR2(50) := 'Vaibhav';
BEGIN
  DECLARE
    salary NUMBER := 50000;  -- Local to inner block
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Salary: ' || salary);
  END;

  -- DBMS_OUTPUT.PUT_LINE(salary); ❌ Error: salary not visible here
  DBMS_OUTPUT.PUT_LINE('Employee: ' || emp_name);  -- ☑ Accessible
END;
```

📃 Output:

```
Salary: 50000
Employee: Vaibhav
```

📌 Note: salary is not accessible outside the inner block.

🔸 2. Global Scope (Outer Block)

- Variables are declared in the outer block.
- It is accessible inside all inner blocks.

💻 Syntax:

```
DECLARE
  dept_name VARCHAR2(50) := 'Sales';  -- Global
BEGIN
  DECLARE
    location VARCHAR2(30) := 'Mumbai';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Department: ' || dept_name);  -- ☑ Accessible
    DBMS_OUTPUT.PUT_LINE('Location: ' || location);     -- ☑ Accessible
  END;
END;
```

📃 Output:

```
Department: Sales
Location: Mumbai
```

📌 Note: dept_name is accessible inside inner block — it has global scope within the entire BEGIN...END.

🔸 3. Shadowing (Variable Overriding)

- The inner block variable can hide a variable from the outer block with the same name.
- This is called variable shadowing.

💻 Syntax:

```
DECLARE
  name VARCHAR2(20) := 'Outer Vaibhav';
BEGIN
  DECLARE
    name VARCHAR2(20) := 'Inner Vaibhav';  -- Shadows outer 'name'
  BEGIN
```

```
    DBMS_OUTPUT.PUT_LINE('Name: ' || name);  -- Inner variable used
  END;
END;
```

🧾 Output:

```
Name: Inner Vaibhav
```

📌 Note: Inner block's variable overrides the outer block's variable temporarily.

🧠 Real-Life Scenario:

- In a billing system, you declare a total_amount variable in the main block (global), and declare a discount in a sub-block only when an offer is available (local).
- This helps manage which variables are used where, and prevents accidental misuse.

📝 Important Notes:

- Always use unique variable names to avoid confusion.
- Variables declared in inner blocks are not visible outside that block.
- PL/SQL does not allow global variables across programs unless you use packages.

🔁 Summary

Scope Type	Declared In	Accessible In	Shadowing Allowed?
Local	Inside block/procedure	Only within that block	✅ Yes
Global	Outer block	Inner blocks of same block	✅ Yes
Package Scope	Package declaration	Across procedures/functions	✅ Yes

Constants

- A constant is a variable whose value cannot change once it is declared.
- Constants are useful when you want to store fixed values like tax rates, discount percentages, or status codes that should not be altered during program execution.
- Must use the CONSTANT keyword.
- Must be initialized at the time of declaration.
- Cannot assign a new value to a constant once defined.
- Improves code readability, maintainability, and avoids accidental changes.

Syntax:

```
DECLARE
  pi CONSTANT NUMBER := 3.14159;
  company_name CONSTANT VARCHAR2(50) := 'OpenAI';
BEGIN
  DBMS_OUTPUT.PUT_LINE('Company: ' || company_name);
  DBMS_OUTPUT.PUT_LINE('Pi value: ' || pi);
END;
```

Output:

```
Company: OpenAI
Pi value: 3.14159
```

- Constants can be anchored using %TYPE:

```
DECLARE
  emp_id employees.employee_id%TYPE := 101;
  max_id CONSTANT employees.employee_id%TYPE := 999;
```

Real-Life Scenario:

- You're building a payroll system. The maximum allowed leave per year is fixed to 20.
  You can declare it as:
  `MAX_LEAVE CONSTANT NUMBER := 20;`

Summary

Feature	Description
Keyword Used	CONSTANT
Value Change Allowed?	❌ No
Initialization Mandatory?	✅ Yes
Use Case	Fixed values like tax rate, limits, etc.
Anchor Support	✅ %TYPE can be used

🔷 PL/SQL Data Types

📊 PL/SQL Data Types Table:

🔷 Main Category	🔹 Sub Category	🔸 Data Type
1. Scalar	Numeric	NUMBER, BINARY_INTEGER, PLS_INTEGER, SIMPLE_INTEGER, DEC, DECIMAL, FLOAT, INTEGER, INT, NUMERIC, SMALLINT, REAL, DOUBLE PRECISION
	Character	CHAR, VARCHAR2, VARCHAR, NCHAR, NVARCHAR2, LONG
	Date/Time	DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL YEAR TO MONTH, INTERVAL DAY TO SECOND
	Boolean	BOOLEAN
2. Composite	Record	RECORD
	Collection	TABLE, VARRAY, NESTED TABLE, ASSOCIATIVE ARRAY (INDEX BY table)
3. LOB	Binary/CLOB	BLOB, CLOB, NCLOB, BFILE
4. Reference	Cursor/Object Reference	REF CURSOR (Weak, Strong), REF Object
5. User-Defined	Object/Collection	OBJECT TYPE, COLLECTION TYPE

🏵 1. SCALAR DATA TYPES

- Scalar types hold a single value — unlike composite types (like RECORDs).
- Classified into 3 major categories:

◇ a) Numeric Data Types:

- Used for storing numbers: integers, decimals, etc.

◇ 1. NUMBER

- Most common numeric data type.

💻 Syntax:

```
salary NUMBER(7,2); -- 7 digits total, 2 after decimal
```

💡 Example:

```
DECLARE
  salary NUMBER(7,2) := 12345.67;
```

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Salary: ' || salary);
END;
```

📃 Output:

```
Salary: 12345.67
```

📌 Note:

- Can store very large or very small numbers.
- Precision and scale are optional.

◇ 2. BINARY_INTEGER (Deprecated)

- Used for integer arithmetic.
- Now replaced by PLS_INTEGER (better performance).

◇ 3. PLS_INTEGER

- Native to PL/SQL, faster than NUMBER for integers.

💻 Syntax:

```
count_ PLS_INTEGER := 100;
```

💡 Example:

```
DECLARE
  counter PLS_INTEGER := 5;
BEGIN
  counter := counter * 2;
  DBMS_OUTPUT.PUT_LINE('Double: ' || counter);
END;
```

📃 Output:

```
Double: 10
```

📌 Note: No scale/decimal allowed. Good for loops and counters.

◇ 4. SIMPLE_INTEGER

- Faster and more memory efficient than PLS_INTEGER

✔️ Steps:

- No NULLs (NULLs are not allowed).
- Overflow checks are skipped for performance.

💡 Example:

```
DECLARE
  val SIMPLE_INTEGER := 10;
BEGIN
  val := val + 5;
  DBMS_OUTPUT.PUT_LINE(val);
END;
```

📃 Output:

```
15
```

📌 Note: Ideal for performance-critical applications.

◇ 5. Synonym Types (Same as NUMBER)

- These are subtypes of NUMBER for readability:
  - DEC, DECIMAL
  - FLOAT
  - INTEGER, INT
  - NUMERIC
  - SMALLINT
  - REAL
  - DOUBLE PRECISION

💡 Example:

```
DECLARE
  age INT := 27;
  marks FLOAT := 88.7;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Age: ' || age || ', Marks: ' || marks);
END;
```

📜 Output:

```
Age: 27, Marks: 88.7
```

◇ b) Character Data Types

- Used to store text/strings.

◇ 1. CHAR

- Fixed-length strings.
- Right-padded with spaces.

💡 Example:

```
DECLARE
  code CHAR(5) := 'AB';
BEGIN
  DBMS_OUTPUT.PUT_LINE('Code:' || code || '.');
END;
```

📜 Output:

```
Code:AB   .
```

📌 Note: Use only if fixed size is required (like gender = 'M').

◇ 2. VARCHAR2 (Recommended)

- Variable-length string.
- Max size up to 32,767 bytes in PL/SQL (4000 in SQL).

💡 Example:

```
DECLARE
  name VARCHAR2(20) := 'Vaibhav';
BEGIN
  DBMS_OUTPUT.PUT_LINE('Name: ' || name);
END;
```

📜 Output:

Name: Vaibhav

◇ 3. VARCHAR
- ANSI-compatible version of VARCHAR2.
- Treated the same in Oracle but should be avoided.
- ❌ Not recommended for future-proof PL/SQL code.

◇ 4. NCHAR, NVARCHAR2
- Used for Unicode character storage (multi-language).

💡 Example:
```
DECLARE
  country NCHAR(5) := 'भारत';
BEGIN
  DBMS_OUTPUT.PUT_LINE(country);
END;
```

📃 Output:
```
भारत
```

📌 Note: Required when building apps in multiple languages.

◇ 5. LONG
- Legacy type. Stores long strings (up to 2 GB).
  ❌ Avoid using — deprecated. Use CLOB instead.

◇ c) Date/Time Data Types
- Used to work with dates and time.

◇ 1. DATE
- Stores date and time up to seconds.

💡 Example:
```
DECLARE
  today DATE := SYSDATE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Today: ' || today);
END;
```

📃 Output:
```
Today: 03-JUN-25 10:30:20
```

📌 Note: Always includes time part.

◇ 2. TIMESTAMP
- More precise than DATE.
- Stores fractional seconds.

💡 Example:
```
DECLARE
```

```
  ts TIMESTAMP := SYSTIMESTAMP;
BEGIN
  DBMS_OUTPUT.PUT_LINE(ts);
END;
```

📜 Output:

```
03-JUN-25 10.30.20.123456
```

◇ 3. TIMESTAMP WITH TIME ZONE

- Includes timezone information.

💡 Example:

```
DECLARE
  tz TIMESTAMP WITH TIME ZONE := SYSTIMESTAMP;
BEGIN
  DBMS_OUTPUT.PUT_LINE(tz);
END;
```

📜 Output:

```
03-JUN-25 10.30.20.123456 +05:30
```

◇ 4. TIMESTAMP WITH LOCAL TIME ZONE

- Stores time in the server's local timezone.

◇ 5. INTERVAL YEAR TO MONTH

- Stores a span of months/years (e.g., age, subscription)

💡 Example:

```
DECLARE
  i1 INTERVAL YEAR(2) TO MONTH := INTERVAL '1-6' YEAR TO MONTH;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Interval: ' || i1);
END;
```

📜 Output:

```
Interval: +01-06
```

◇ 5. INTERVAL DAY TO SECOND

- Stores span in days, hours, minutes, seconds

💡 Example:

```
DECLARE
  i2 INTERVAL DAY TO SECOND := INTERVAL '2 04:05:06' DAY TO
SECOND;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Interval: ' || i2);
END;
```

📜 Output:

```
Interval: +02 04:05:06.000000
```

◇ d) BOOLEAN

- The BOOLEAN data type represents logical values — TRUE, FALSE, and NULL.
- It's used mainly for control structures, such as IF, LOOP, or logical decisions in procedures and functions.
- BOOLEAN is only available in PL/SQL – ❌ not supported in SQL statements.
- It is not stored in database tables.
- You cannot select or insert a BOOLEAN column directly in SQL.
- It's mostly used in procedural code and logic flows.

💻 Syntax:

```
DECLARE
  is_valid BOOLEAN := TRUE;
BEGIN
  IF is_valid THEN
    DBMS_OUTPUT.PUT_LINE('The condition is TRUE');
  ELSE
    DBMS_OUTPUT.PUT_LINE('The condition is FALSE');
  END IF;
END;
```

📜 Output:

```
The condition is TRUE
```

💡 Example:

```
DECLARE
  flag1 BOOLEAN := TRUE;
  flag2 BOOLEAN := FALSE;
  flag3 BOOLEAN := NULL;
BEGIN
  IF flag1 THEN
    DBMS_OUTPUT.PUT_LINE('flag1 is TRUE');
  END IF;

  IF flag2 THEN
    DBMS_OUTPUT.PUT_LINE('flag2 is TRUE');
  ELSE
    DBMS_OUTPUT.PUT_LINE('flag2 is FALSE');
  END IF;

  IF flag3 IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('flag3 is NULL');
  END IF;
END;
```

📜 Output:

```
flag1 is TRUE
flag2 is FALSE
flag3 is NULL
```

📌 Note:

- Comparing with = TRUE or = FALSE is not necessary.
  ✅ Just writing IF my_flag THEN ... is enough.

🧠 Real-Life Scenario:

- Let's say you're building an HR system:
  - Store employee name in VARCHAR2
  - Store salary in NUMBER(8,2)
  - Store date of joining in DATE
  - Store age difference using INTERVAL YEAR TO MONTH

📝 Important Notes:

- Always prefer VARCHAR2 over CHAR.
- Use PLS_INTEGER or SIMPLE_INTEGER for performance.
- For date/time: DATE is enough for most cases unless high precision or timezone is required.
- Avoid legacy types like LONG, BINARY_INTEGER.

🔁 Summary

Category	Types	Key Usage
Numeric	NUMBER, PLS_INTEGER, SIMPLE_INTEGER, FLOAT, INT, REAL, etc.	For storing integers and decimals
Character	CHAR, VARCHAR2, VARCHAR, NCHAR, NVARCHAR2, LONG	For storing text
Date/Time	DATE, TIMESTAMP, INTERVALs	Storing time/date values and time spans

🔶 2. Composite Data Types in PL/SQL

- Composite types group multiple values (unlike scalar types which store single values).
- There are 2 major categories:
  Records – Like a row with multiple fields
  Collections – Like arrays (single or multi values)

◇ 1. RECORD (like a row structure)

- RECORD stores multiple fields of different data types in one variable (like a table row).

🔹 Ways to declare RECORD:

a) Explicit Definition
b) %ROWTYPE (based on a table)
c) %TYPE (based on columns)

◇ a) Explicit Definition:

- When you manually declare a variable by specifying its data type, it is called an explicit definition.
- You *explicitly write* the data type such as NUMBER, VARCHAR2, DATE, etc., instead of deriving it from a table column or another variable.
- You control the data type and size.
- Commonly used when you don't want to tie the variable to a table column.
- Opposite of anchored declaration (%TYPE, %ROWTYPE).

💻 Syntax:

```
DECLARE
  TYPE emp_record_type IS RECORD (
    emp_id   NUMBER,
```

```
  emp_name VARCHAR2(50),
  salary   NUMBER(8,2)
 );
 emp_rec emp_record_type;
BEGIN
 emp_rec.emp_id := 101;
 emp_rec.emp_name := 'Vaibhav';
 emp_rec.salary := 55000;
 DBMS_OUTPUT.PUT_LINE(emp_rec.emp_name || ' earns ' || emp_rec.salary);
END;
```

🧾 Output:

```
Vaibhav earns 55000
```

You've defined:

- A custom record type emp_record_type ➜ ✅
- You explicitly specified the data types (NUMBER, VARCHAR2(50), etc.) ➜ ✅
- Then created a variable emp_rec of that type ➜ ✅
- Assigned values manually ➜ ✅
- Used DBMS_OUTPUT.PUT_LINE to show the output ➜ ✅

🧠 Real-Life Scenario:

- Suppose you're building a mini calculator or a data entry screen, and you want some working variables like total, name, status, etc.
- These don't need to be tied to any table – just define them explicitly.

📝 Important Notes:

- Use explicit definition when:
  - You want temporary storage variables.
  - No direct dependency on table columns.
- Don't confuse it with default initialization (like: v_count NUMBER := 0;).
- Explicit definition makes your code independent from table changes.
- But you must manually update data types if business rules change (e.g., size of name becomes 100 chars).

◇ b) Using %ROWTYPE:

- It is used to declare a variable that can store an entire row of a table.
- %ROWTYPE is used to declare a record that can hold an entire row of a table or a cursor with the same column structure and data types.
- It saves you from declaring each column's data type individually.
- Automatically adapts to table structure changes.
- Can be used with tables or cursors.
- All fields from the table are accessible using dot notation (e.g., record.column_name).

💻 Syntax:

```
DECLARE
 emp_record employees%ROWTYPE;
BEGIN
 SELECT * INTO emp_record FROM employees WHERE emp_id = 101;
END;
```

📌 Now emp_record can access all columns like:

```
Emp_record.name
Emp_record.salary
```

Emp_record.department_id

📝 Important Notes:

- Dynamically matches the structure of a table row.
- Auto-updates if table structure changes.
- If the table structure changes (e.g., column added), %ROWTYPE automatically adjusts — no need to update your code manually.
- The record holds all columns, even if you don't use them.
- %TYPE is not just for tables — it can refer to variables too.
- If table structure changes, no need to manually update variable types.

🧠 Real-Life Scenario:

- You're creating a report or logic that processes rows from a table like employees, orders, etc.
- Instead of defining every column separately, just use %ROWTYPE — quick, efficient, and clean!

◇ c) Using %TYPE

- It is used to declare a variable with the same data type as a column in a table.
- %TYPE is used to declare a variable in PL/SQL that automatically takes the data type of an existing table column or another variable.
- Keep your variable's data type in sync with the table column.
- Reduces manual effort to change variable types when the table changes.
- Works with both table columns and previously declared variables.
- Prevents data type mismatch errors in long-term use.

💻 Syntax:

```
DECLARE
  TYPE dept_rec_type IS RECORD (
    dept_id departments.department_id%TYPE,
    dept_name departments.department_name%TYPE
  );
  dept_rec dept_rec_type;
BEGIN
  dept_rec.dept_id := 10;
  dept_rec.dept_name := 'HR';
  DBMS_OUTPUT.PUT_LINE(dept_rec.dept_name);
END;
```

🧠 Real-Life Scenario:

- You're developing a payroll system and referencing employee.salary in many places.
- If the column data type changes (e.g., NUMBER(10,2) ➜ NUMBER(12,2)),
- all %TYPE variables will adapt automatically. ✅

📝 Important Notes:

- Records cannot be directly compared or assigned to each other.
- You can nest records inside other records.

🧠 Real-Life Scenario:

- You want to hold full employee info temporarily during logic processing — use a RECORD instead of many separate variables.

🧾 Difference Between %ROWTYPE vs %TYPE

Feature	%TYPE	%ROWTYPE
Used for	Single column/variable	Entire row from table/cursor
Columns it holds	One	All columns in a row
Syntax	var column_name%TYPE	var table_name%ROWTYPE
Flexibility	Moderate	High – adapts to full row structure

◇ 2. COLLECTIONS (Array-like Structures)

- Collections are groups of elements (same data type) stored in one variable.

✔ Three types of collections:

◇ a) INDEX-BY TABLE (Associative Array)

- Oldest, simplest array — indexed by number or string.
- An INDEX-BY TABLE (now known as Associative Array) is a key-value pair collection where:
  - The key can be a number or a string.
  - The value is of any scalar, record, or even object type.
  - Unlike other collections, it has no fixed size and is sparse (can have gaps).
- Indexed using either PLS_INTEGER or VARCHAR2.
- Best for look-up operations or temporary data storage.

✔ Steps to Use:

1. Declare a TYPE using TABLE OF with INDEX BY.
2. Declare a variable of that TYPE.
3. Use index (number or string) to assign values.
4. Access using the same index.

💻 Syntax:

```
DECLARE
  TYPE emp_table IS TABLE OF VARCHAR2(100) INDEX BY PLS_INTEGER;
  emp_names emp_table;
BEGIN
  emp_names(1) := 'Vaibhav';
  emp_names(2) := 'Amit';
  DBMS_OUTPUT.PUT_LINE(emp_names(1)); -- prints Vaibhav
END;
```

📜 Output:

```
Vaibhav
```

📌 Note:

- Dynamic size
- Sparse (not all indexes need to be filled)
- No need to initialize

🧠 Real-Life Scenario:

- Suppose you are calculating daily sales per store in memory, and want to store:
  'Pune' ➜ ₹50,000, 'Mumbai' ➜ ₹80,000
  No need for a table — just use an associative array with string index.

📝 Important Notes:
- No need to initialize size.
- Cannot be stored in DB or used in SQL queries.
- Ideal for temporary logic, like loops, lookups, caching values.
- Can be passed to procedures/functions.

◇ b) NESTED TABLE
- A Nested Table is a collection type in PL/SQL that can:
- Store multiple elements like an array (but with no upper bound).
- Be stored in the database as a column.
- Be sparse (can have gaps between elements).
- Be queried and modified using SQL.
- It behaves like a normal array in PL/SQL, but you can store it in a DB column.
- Allows non-contiguous elements (you can delete elements and leave gaps).
- More flexible than VARRAYs.
- Needs to be initialized before usage.

💻 Syntax:

```
DECLARE
  TYPE num_table IS TABLE OF NUMBER;
  nums num_table := num_table(10, 20, 30);
BEGIN
  FOR i IN 1..nums.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(nums(i));
  END LOOP;
END;
```

📃 Output:

```
10
20
30
```

📌 Note:
- Needs initialization.
- Can use EXTEND, DELETE, etc.
- Index starts from 1.

🧠 Real-Life Scenario:
- Imagine storing multiple phone numbers or email addresses for a customer in a single row of a table.
- A nested table allows you to do this without creating a separate table and joining.

📝 Important Notes:
- Nested tables are unbounded.
- Elements can be modified, deleted, or extended.
- They are ideal for storing sets of data like orders, scores, emails, etc.
- Must use TABLE() keyword to query from SQL.

◇ c) VARRAY (Variable-Size Array):

- A VARRAY (Variable-Size Array) is a collection type in PL/SQL used to store a fixed number of ordered elements of the same type.
- VARRAYs have a maximum size (upper limit defined).
- They are dense (no gaps allowed between elements).
- Useful for small, ordered, fixed-size data.
- Elements are stored in contiguous memory (no gaps allowed).
- Can be used in both PL/SQL and SQL.
- Ideal for small lists that don't change much.
- Must declare the size during type creation.
- Can be stored as a column in a table.
- Maintains order of elements.
- Deleting individual elements not allowed in VARRAY.

💡 VARRAY Methods:

Method	Description
.COUNT	Number of elements
.FIRST	Index of first element
.LAST	Index of last element
.EXTEND(n)	Adds n empty elements
.DELETE	Not allowed in VARRAY

🧠 Real-Life Scenario:

- You want to store monthly sales data for 6 months per product.
- Since the size is known and fixed, a VARRAY(6) would be a perfect fit.

💻 Syntax:

```
DECLARE
  TYPE salary_array IS VARRAY(5) OF NUMBER;
  sal salary_array := salary_array(20000, 30000, 40000);
BEGIN
  FOR i IN 1..sal.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(sal(i));
  END LOOP;
END;
```

📄 Output:

```
20000
30000
40000
```

📌 Note:

- Size is fixed at declaration.
- Good for when the limit is known (e.g., 12 months).

🧠 Real-Life Scenario:

- RECORD: Temporary storage of a full employee or order row.
- Associative Array: Mapping employee ID → name or salaries.
- Nested Table: Holding a list of dynamically added items like project names.
- VARRAY: Store fixed monthly sales data (12 months).

📝 Important Notes:

- Always initialize Nested Table and VARRAY before use.
- Use .COUNT, .DELETE, .EXTEND, .EXISTS for collection operations.
- Associative Arrays are very useful in procedural logic.

🗃️ Difference Between Associative Array vs Nested Table VARRAY

Feature / Type	Associative Array (INDEX-BY TABLE)	Nested Table	VARRAY
Index Type	PLS_INTEGER / VARCHAR2 (string or number)	Sequential (1..N)	Sequential (1..N)
Fixed Size?	❌ No (dynamic)	❌ No (dynamic)	✅ Yes (must specify size at creation)
Sparse (Gaps Allowed)?	✅ Yes	✅ Yes	❌ No (always dense)
Can Store in DB?	❌ No	✅ Yes	✅ Yes
Used in SQL Queries?	❌ No	✅ Yes	✅ Yes
Maintains Order?	❌ No (order not guaranteed)	✅ Yes	✅ Yes
Performance	✅ Fast (in-memory)	⚠️ Moderate	✅ Fast for small data
Deletion Allowed?	✅ Yes (any index)	✅ Yes (DELETE or TRIM)	❌ No
Initialization Required?	✅ Yes (by assigning values directly)	✅ Yes	✅ Yes
Can be used in PL/SQL?	✅ Yes	✅ Yes	✅ Yes

🔁 Summary:

Composite Type	Purpose	Example Use
RECORD	Row-like, multiple columns/fields	Store full employee row
Assoc. Array	Dynamic key-value pair storage	ID to Name mapping
Nested Table	Dynamic list (storable in DB)	Multiple items per customer
VARRAY	Fixed-size ordered list	Monthly salary/sales values

🔷 3. LOB (Large Object)

- LOBs are used to store very large data like text, images, files, audio, video, etc.

◇ 1. BLOB (Binary Large Object)

- Used for storing binary data like:
  - Images (JPG, PNG)
  - Audio files (MP3)
  - PDFs, documents

💻 Syntax:

```
CREATE TABLE images (
  id NUMBER,
  photo BLOB
);
```

📌 Note:

- Stored inside the database.
- No character encoding applied.
- Maximum size: up to 4 GB.

🧠 Real-Life Scenario:

- Storing scanned documents, resumes, or product images.

◇ 2. CLOB (Character Large Object)

- 🔸 Used for storing large text data like:
  - Articles
  - Books
  - Long descriptions

💻 Syntax:

```
CREATE TABLE articles (
  id NUMBER,
  content CLOB
);
```

💡 Example:

```
DECLARE
  my_clob CLOB;
BEGIN
  SELECT content INTO my_clob FROM articles WHERE id = 1;
  DBMS_OUTPUT.PUT_LINE(DBMS_LOB.SUBSTR(my_clob, 100));
END;
```

🗒️ Output:

(First 100 characters of the content)

📌 Note:

- Supports up to 4 GB of text.
- Can use DBMS_LOB package for operations.

◇ 3. NCLOB (National Character LOB)

- Same as CLOB, but stores Unicode text.

💻 Syntax:

```
CREATE TABLE translations (
  id NUMBER,
  content NCLOB
);
```

📌 Note:

- Used when storing multilingual content (e.g., Hindi, Japanese).
- Supports global applications.

◇ 4. BFILE (Binary File)

- Used for storing external binary files.
- Key Difference: The data is not stored in the database, only a pointer to an OS file.

💻 Syntax:

```
CREATE TABLE audio_files (
  id NUMBER,
  song BFILE
);
```

💡 Example:

```
DECLARE
  music BFILE;
BEGIN
  SELECT song INTO music FROM audio_files WHERE id = 1;
  DBMS_OUTPUT.PUT_LINE(DBMS_LOB.GETLENGTH(music));
END;
```

📌 Note:

- Read-only from PL/SQL.
- File must be in a directory object registered in Oracle.

✔ Steps to use:

1. Create a DIRECTORY object.
   CREATE OR REPLACE DIRECTORY my_files AS '/home/oracle/audio';

2. Grant access:
   GRANT READ ON DIRECTORY my_files TO your_user;

Difference Between LOB Types

Feature	BLOB	CLOB	NCLOB	BFILE
Data Type	Binary	Text	Unicode	Binary
Stored In DB	✅ Yes	✅ Yes	✅ Yes	❌ No (external)
Max Size	~4 GB	~4 GB	~4 GB	~4 GB (pointer)
Read/Write	✅ Yes	✅ Yes	✅ Yes	❌ Read-only
Use Case	Images	Articles	Unicode Docs	External Media

Important Notes:

- Use DBMS_LOB package to manipulate LOBs (read/write/append/delete).
- Always use temporary LOBs when doing large LOB operations in PL/SQL.
- LOBs are usually not fetched entirely for performance — use chunks.

Real-Life Scenarios:

- Resume Upload Module → use BLOB
- Product Description in E-commerce → use CLOB
- Multilingual Blog Posts → use NCLOB
- Referencing Audio/Video files from server → use BFILE

Summary

Type	Used For	Stored In DB?	Notes
BLOB	Images, PDFs, Binary data	✅ Yes	Binary content
CLOB	Long Text, Documents	✅ Yes	Character content
NCLOB	Unicode content	✅ Yes	For multilingual apps
BFILE	External Files	❌ No	Pointer to server-side file

🔷 4. Reference Types in PL/SQL

- PL/SQL Reference Types are pointers to complex data structures like cursors and objects.
- Categories of Reference Types:
  1. REF CURSOR – Pointer to a result set (Dynamic SQL)
  2. REF OBJECT – Pointer to an object instance (like a row object)

◇ 1. REF CURSOR (Cursor Variables)

- A REF CURSOR does not store data, it points to a query result set — dynamic and flexible.
- Types of REF CURSOR

🔹 a) Strong REF CURSOR

- Has fixed return structure (data type)

💻 Syntax:

```
DECLARE
  TYPE emp_cursor_type IS REF CURSOR RETURN employees%ROWTYPE;
  emp_cursor emp_cursor_type;
  emp_rec employees%ROWTYPE;
BEGIN
  OPEN emp_cursor FOR SELECT * FROM employees WHERE department_id =
10;
  LOOP
    FETCH emp_cursor INTO emp_rec;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(emp_rec.first_name || ' earns ' || emp_rec.salary);
  END LOOP;
  CLOSE emp_cursor;
END;
```

📃 Output:

```
John earns 50000
Ravi earns 60000
…
```

📌 Note:

- Return type is fixed to employees%ROWTYPE
- Useful in static query scenarios

🔹 b) Weak REF CURSOR

- No return type check, more flexible

💻 Syntax:

```
DECLARE
  TYPE any_cursor IS REF CURSOR;
  emp_cursor any_cursor;
  emp_name employees.first_name%TYPE;
BEGIN
  OPEN emp_cursor FOR SELECT first_name FROM employees;
  LOOP
    FETCH emp_cursor INTO emp_name;
    EXIT WHEN emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(emp_name);
  END LOOP;
```

```
  CLOSE emp_cursor;
END;
```

📜 Output:

```
John
Ravi
…
```

📝 Important Notes:

- No return type specified
- Allows more dynamic SQL
- REF CURSORs are used to return data from procedures/functions to front-end apps (like Java, .NET, Python).
- Also widely used in report generation and pagination logic.

🧠 Real-Life Scenario:

- A procedure needs to return different columns based on the user role.
- You can use a weak REF CURSOR to return various result sets dynamically.

◇ 2. REF Object (Object References)

- These are pointers to object instances in an object-relational model (advanced usage).
- Steps to Use REF OBJECT:

✔ Step 1: Create Object Type

```
CREATE OR REPLACE TYPE emp_obj AS OBJECT (
  emp_id NUMBER,
  emp_name VARCHAR2(50)
);
```

✔ Step 2: Create Object Table

```
CREATE TABLE emp_table OF emp_obj;
```

✔ Step 3: Insert Data

```
INSERT INTO emp_table VALUES (101, 'Vaibhav');
```

✔ Step 4: Use REF to point to object

```
DECLARE
  emp_ref REF emp_obj;
  emp_instance emp_obj;
BEGIN
  SELECT REF(e) INTO emp_ref FROM emp_table e WHERE emp_id = 101;
  FETCH DEREF(emp_ref) INTO emp_instance;
  DBMS_OUTPUT.PUT_LINE(emp_instance.emp_name);
END;
```

📜 Output:

```
Vaibhav
```

📌 Note:

- REF is the pointer
- DEREF() is used to get the object's values

📝 Important Notes:

- REF CURSORs are much more common than REF OBJECTs.
- Use weak cursors when you don't know the structure in advance.

- Use strong cursors for better compile-time checks.
- REF OBJECT is useful in object-relational models (rare in general business apps).

Difference Between Strong & Weak REF CURSOR

Feature	Strong REF CURSOR	Weak REF CURSOR
Return Type	Fixed (e.g., ROWTYPE)	Not defined
Flexibility	Low	High
Compile-Time Check	✅ Yes	❌ No
Use Case	Static queries	Dynamic SQL, joins

Summary:

Reference Type	Description	Used For
REF CURSOR	Pointer to a result set	Dynamic queries, procedure return
Strong CURSOR	Fixed structure (compile-time check)	Fixed table queries
Weak CURSOR	No structure (more flexible)	Dynamic SQL, varying results
REF OBJECT	Pointer to object instance in DB	Advanced, object-relational use

5. User-Defined Types (UDTs) in PL/SQL

- PL/SQL allows you to create your own data types using:

1. OBJECT TYPE – Custom structure like a class in OOP
2. COLLECTION TYPE – Custom lists/arrays like arrays, sets

1. OBJECT TYPE

- Object Type is a user-defined composite type that groups related variables (attributes) and subprograms (methods), like a class in object-oriented programming.

Syntax:

```
CREATE OR REPLACE TYPE emp_type AS OBJECT (
  emp_id NUMBER,
  emp_name VARCHAR2(50),
  emp_salary NUMBER
);
```

✔ Steps to Use Object Type:
1. Create the object type
2. Create a table OF that object (optional)
3. Declare and use it in PL/SQL

💡 Example: Create and Use Object Type

```
DECLARE
  emp_obj emp_type;
BEGIN
  emp_obj := emp_type(101, 'Vaibhav', 55000);
  DBMS_OUTPUT.PUT_LINE('Name: ' || emp_obj.emp_name || ', Salary: ' ||
emp_obj.emp_salary);
END;
```

📃 Output:

```
Name: Vaibhav, Salary: 55000
```

📌 Note:

- You can define methods (functions/procedures) inside object types as well.
- Useful in OOP-based design in PL/SQL or when interacting with object-relational DB tables.

🧠 Real-Life Scenario:

- You're building an HR system where each employee has a structure (ID, name, dept, etc.). Instead of using multiple variables, define a reusable emp_type object.

📝 Important Notes:

- Object Types help modularize and reuse code like OOP.
- Collections are powerful for loops, bulk processing, and in-memory operations.
- Nested Tables and VARRAYs can be stored in DB tables, but Associative Arrays are only for PL/SQL blocks.

🔁 Summary:

Type	Use For	Stored in DB?	Dynamic?
OBJECT TYPE	Create reusable data structures	✅ Yes	✅ Yes
VARRAY	Small fixed-size lists	✅ Yes	❌ No
Nested Table	Unbounded list (SQL + PL/SQL)	✅ Yes	✅ Yes
Associative Array	Key-value pairs (PL/SQL only)	❌ No	✅ Yes

🔷 Control Structures in PL/SQL

- Control structures in PL/SQL allow us to control the flow of execution in our programs.
- Just like the steering wheel in a car 🛞.
- They help you make decisions, repeat actions, and navigate conditions dynamically.

🏵 Three Main Categories of Control Structures

◇ 1. Conditional Control – IF, CASE

- Used to execute certain parts of code based on specific conditions (TRUE/FALSE).
- Helps in decision-making inside the PL/SQL block.
- You can choose between multiple actions depending on variable values or expressions.
- 💡 Examples include: IF, IF-ELSE, ELSIF, and CASE.

◇ 1. IF Statement

- Used to execute code based on one or more conditions.
- Helps you make decisions like grading, salary calculation, or validations.
- You can use IF, ELSIF, and ELSE for multi-branch decisions.

💻 Syntax:

```
IF condition THEN
  -- statements
ELSIF another_condition THEN
  -- statements
ELSE
  -- statements
END IF;
```

💡 Example:

```
DECLARE
 marks NUMBER := 75;
BEGIN
 IF marks >= 90 THEN
  DBMS_OUTPUT.PUT_LINE('Grade: A');
 ELSIF marks >= 75 THEN
  DBMS_OUTPUT.PUT_LINE('Grade: B');
 ELSE
  DBMS_OUTPUT.PUT_LINE('Grade: C');
 END IF;
END;
```

📃 Output:

```
Grade: B
```

◇ 2. CASE Statement

- Handling multiple conditions more clearly than nested IFs
- Provides a cleaner alternative to multiple IF...ELSIF conditions.
- Used when checking the value of a single variable against many options.

💻 Syntax:

```
CASE variable
 WHEN value1 THEN statements
 WHEN value2 THEN statements
```

```
  ELSE statements
END CASE;
```

💡 Example:

```
DECLARE
  marks NUMBER := 75;
BEGIN
  CASE
    WHEN marks >= 90 THEN
      DBMS_OUTPUT.PUT_LINE('Grade: A');
    WHEN marks >= 75 THEN
      DBMS_OUTPUT.PUT_LINE('Grade: B');
    ELSE
      DBMS_OUTPUT.PUT_LINE('Grade: C');
  END CASE;
END;
```

📃 Output:

```
Very Good
```

◇ 2. Iterative Control – LOOP, WHILE, FOR, Nested Loops

- Used to repeat a set of statements multiple times until a condition is met or a counter ends.
- Allows you to repeat a block of code multiple times, either until a condition is met or a count ends.
- Mainly used for loops and repetition tasks like traversing rows, calculating sums, etc.
- Prevents code duplication and enhances readability during repetitive logic.
- 💡 Examples include: LOOP, WHILE, FOR, and Nested Loops.

◇ 1. LOOP Statement (Simple Loop)

- Repeats a set of statements indefinitely until manually exited.
- Must include an EXIT condition, otherwise becomes an infinite loop.

💻 Syntax:

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

💡 Example:

```
DECLARE
  i NUMBER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('Value: ' || i);
    i := i + 1;
    EXIT WHEN i > 3;
  END LOOP;
END;
```

📃 Output:

```
Value: 1
Value: 2
```

```
Value: 3
```

◇ 2. WHILE Loop

- Repeats statements as long as a condition is TRUE.
- Condition is checked before entering the loop.

💻 Syntax:

```
WHILE condition LOOP
  -- statements
END LOOP;
```

💡 Example:

```
DECLARE
 i NUMBER := 1;
BEGIN
 WHILE i <= 3 LOOP
  DBMS_OUTPUT.PUT_LINE('WHILE Value: ' || i);
  i := i + 1;
 END LOOP;
END;
```

🗒️ Output:

```
WHILE Value: 1
WHILE Value: 2
WHILE Value: 3
```

◇ 3. FOR Loop

- Looping a known number of times.
- Executes code for a fixed number of iterations.
- Ideal for known ranges.
- The loop variable is auto-incremented and scope-limited to the loop.

💻 Syntax:

```
FOR counter IN [REVERSE] start..end LOOP
  -- statements
END LOOP;
```

💡 Example:

```
BEGIN
 FOR i IN 1..3 LOOP
  DBMS_OUTPUT.PUT_LINE('FOR Loop: ' || i);
 END LOOP;
END;
```

🗒️ Output:

```
FOR Loop: 1
FOR Loop: 2
FOR Loop: 3
```

◇ 4. Nested Loops

- Looping inside another loop (common in matrix, table operations)
- One loop inside another loop.

- Used for matrix-like structures or when comparing multiple sets of data.

💡 Example:

```
BEGIN
  FOR i IN 1..2 LOOP
    FOR j IN 1..2 LOOP
      DBMS_OUTPUT.PUT_LINE('i=' || i || ', j=' || j);
    END LOOP;
  END LOOP;
END;
```

📜 Output:

```
i=1, j=1
i=1, j=2
i=2, j=1
i=2, j=2
```

◇ 3. Sequential Control – GOTO, NULL, EXIT, etc.

- Controls the linear flow or sequence of execution within a block.
- Does not involve decision-making or loops, but alters how control moves (e.g., skip, exit, jump).
- Used to terminate, jump to labels, or perform no action at all.
- 💡 Examples include: GOTO, NULL, EXIT, RETURN.

◇ 1. GOTO

- Used to transfer control to a labeled part of the PL/SQL block.
- Helps you jump from one part of the code to another.
- Can make code hard to read, so should be used carefully.
- GOTO should be avoided in structured programming unless absolutely necessary.

💻 Syntax:

```
<<label_name>>
-- statements
GOTO label_name;
```

💡 Example:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Before GOTO');
  GOTO skip;
  DBMS_OUTPUT.PUT_LINE('This line will be skipped');

  <<skip>>
  DBMS_OUTPUT.PUT_LINE('After GOTO');
END;
```

📜 Output:

```
Before GOTO
After GOTO
```

📌 Note: The DBMS_OUTPUT.PUT_LINE('This line will be skipped') is not executed due to the jump.

◇ 2. NULL

- A placeholder that does nothing.
- Often used in IF statements where no action is needed for certain cases.

💡 Example:

```
BEGIN
  IF 1 = 2 THEN
    NULL;
  ELSE
    DBMS_OUTPUT.PUT_LINE('Condition is false');
  END IF;
END;
```

📜 Output:

```
Condition is false
```

◇ 3. EXIT

- Used to exit a loop prematurely based on a condition.
- Can be used with or without a label.

💡 Example:

```
DECLARE
  i NUMBER := 1;
BEGIN
  LOOP
    EXIT WHEN i > 3;
    DBMS_OUTPUT.PUT_LINE('EXIT Example: ' || i);
    i := i + 1;
  END LOOP;
END;
```

📜 Output:

```
EXIT Example: 1
EXIT Example: 2
EXIT Example: 3
```

◇ 4. RETURN

- Used to exit from a PL/SQL subprogram (procedure or function).
- Can be placed anywhere inside the block.
- In a function, it's also used to return a value to the caller.

💻 Syntax (for Procedure):

```
RETURN;
```

💻 Syntax (for Function)

```
RETURN value;
```

💡 Example (Procedure):

```
CREATE OR REPLACE PROCEDURE greet IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello Vaibhav!');
  RETURN;
  DBMS_OUTPUT.PUT_LINE('This will not be printed');
```

```
END;
```

📜 Output:

```
Hello Vaibhav!
```

💡 Example (Function):

```
CREATE OR REPLACE FUNCTION square(n NUMBER)
RETURN NUMBER IS
BEGIN
  RETURN n * n;
END;
```

💡 Calling:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Square is: ' || square(5));
END;
```

📜 Output:

```
Square is: 25
```

📝 Important Notes:

- In procedures, RETURN simply exits the block.
- In functions, RETURN also sends back a value.
- Only one RETURN is required in a function, but multiple can be used for different conditions.

💡 Example:

```
BEGIN
  NULL;  -- placeholder that does nothing
  DBMS_OUTPUT.PUT_LINE('Executed NULL without error');
END;
```

📝 Important Notes:

- Use EXIT to break any loop.
- IF and CASE can't be used interchangeably always. CASE is better when checking values of a single variable.
- Avoid infinite LOOP without EXIT – it causes runtime hangs.

🔁 Summary Table

Control Structure	Used For	Exit Type
IF	Conditional logic	No exit needed
CASE	Multi-branch conditional	No exit needed
LOOP	Infinite loop until EXIT condition	Manual EXIT
WHILE	Loop while condition is true	Auto-check
FOR	Known range of iteration	Auto exit
Nested Loops	Matrix/table traversal	Manual per loop
NULL	Placeholder	—
EXIT	Exit from a loop	Immediate exit

🔷 Operators in PL/SQL

- Operators are symbols or keywords used to perform operations on variables, constants, or expressions.
- These operations can be arithmetic, comparison, logical, string manipulation, and more.

◇ 1. Arithmetic Operators

- These are used to perform basic mathematical operations on numeric data types.
- Arithmetic operators return numeric results and follow mathematical precedence rules.

🔹 Operators:

- `+` → Adds two numbers
- `-` → Subtracts one number from another
- `*` → Multiplies two numbers
- `/` → Divides one number by another
- `**` → Raises a number to the power of another
- `MOD(a, b)` → Returns the remainder of `a / b`

💻 Syntax:

```
result := a + b;
```

💡 Example:

```
DECLARE
  a NUMBER := 10;
  b NUMBER := 3;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Addition: ' || (a + b));
  DBMS_OUTPUT.PUT_LINE('Power: ' || (a ** b));
  DBMS_OUTPUT.PUT_LINE('Modulus: ' || MOD(a, b));
END;
```

📄 Output:

```
Addition: 13
Power: 1000
Modulus: 1
```

◇ 2. Relational (Comparison) Operators

- Used to compare values and return a Boolean result: TRUE, FALSE, or NULL.
- These are often used inside IF statements and loops to make decisions.

🔹 Operators:

- `=` → Equal to
- `!=` or `<>` → Not equal to
- `>` → Greater than
- `<` → Less than
- `>=` → Greater than or equal to
- `<=` → Less than or equal to

💻 Syntax:

```
IF a < b THEN
  -- do something
END IF;
```

💡 Example:

```
DECLARE
```

```
  x NUMBER := 15;
  y NUMBER := 20;
BEGIN
  IF x < y THEN
    DBMS_OUTPUT.PUT_LINE('x is less than y');
  END IF;
END;
```

📜 Output:

```
x is less than y
```

◇ 3. Logical Operators

- Used to combine multiple Boolean expressions.
- Mainly used in control structures like IF, CASE, LOOP, etc.

🔹 Operators:

- AND → TRUE if both conditions are true
- OR → TRUE if any one condition is true
- NOT → Reverses the logical result (TRUE becomes FALSE)

💻 Syntax:

```
IF condition1 AND condition2 THEN
  -- logic
END IF;
```

💡 Example:

```
DECLARE
  age NUMBER := 28;
  salary NUMBER := 45000;
BEGIN
  IF age > 25 AND salary > 40000 THEN
    DBMS_OUTPUT.PUT_LINE('Eligible for loan');
  END IF;
END;
```

📜 Output:

```
Eligible for loan
```

◇ 4. String Operator

- PL/SQL provides || as a concatenation operator to join strings.
- Useful in reporting, logging, or printing full names, messages, etc.

💻 Syntax:

```
full_name := first_name || ' ' || last_name;
```

💡 Example:

```
DECLARE
  fname VARCHAR2(20) := 'Vaibhav';
  lname VARCHAR2(20) := 'Dhakulkar';
BEGIN
  DBMS_OUTPUT.PUT_LINE('Full Name: ' || fname || ' ' || lname);
END;
```

📜 Output:

```
Full Name: Vaibhav Dhakulkar
```

◇ 5. Assignment Operator

- Used to assign a value to a variable.
- This operator does not return a value — it simply sets a value.

🔹 Operator:

- := → Assigns the value on the right to the variable on the left

💻 Syntax:

```
variable_name := expression;
```

💡 Example:

```
DECLARE
  marks NUMBER;
BEGIN
  marks := 90;
  DBMS_OUTPUT.PUT_LINE('Marks: ' || marks);
END;
```

📜 Output: Marks: 90

◇ 6. Membership & Range Operators

- These operators check if a value belongs to a list or a range.
- Used in filtering and decision-making in IF/CASE structures.

🔹 Operators:

- IN → Checks if a value is in a list of values
- BETWEEN a AND b → Checks if a value lies between a and b (inclusive)

💻 Syntax:

```
IF grade IN ('A', 'B', 'C') THEN ...
IF salary BETWEEN 20000 AND 50000 THEN …
```

💡 Example:

```
DECLARE
  grade CHAR := 'B';
  salary NUMBER := 30000;
BEGIN
  IF grade IN ('A', 'B', 'C') THEN
    DBMS_OUTPUT.PUT_LINE('Valid Grade');
  END IF;

  IF salary BETWEEN 25000 AND 50000 THEN
    DBMS_OUTPUT.PUT_LINE('Middle Range Salary');
  END IF;
END;
```

📜 Output:

```
Valid Grade
Middle Range Salary
```

🧠 Real-Life Use Case:

- In a banking app, you could use:
  - BETWEEN to filter accounts by balance range.
  - AND to check customer eligibility.
  - || to prepare names and messages for reports
  - := to assign limits and thresholds.

🔁 Summary Table

🔸 Operator Type	🔹 Used For	💡 Example
Arithmetic	Math operations	a + b, MOD(a, b)
Relational	Value comparison	a = b, a <> b
Logical	Logical condition evaluation	a > 10 AND b < 50
String	Joining string values	First_name || Last_name
Assignment	Set values to variables	x := 100
Membership / Range	Value in list or range	IN, BETWEEN

🔷 Subprogram in PL/SQL

🔹 What is a Subprogram?

- A subprogram is a named PL/SQL block that can be called and reused to perform a specific task.
- It helps to modularize code and avoid repetition.

🔹 Types of Subprograms:

1. Procedure – performs an action (does not have to return a value)
2. Function – performs a calculation and returns a value

🔶 Procedure In PL/SQL

- A procedure is a named block of PL/SQL code that performs one or more actions, but it does not return a value directly.
- It can accept input (IN), output (OUT), or both (IN OUT) parameters.
- You can call procedures to perform tasks repeatedly (e.g., insert data, generate reports, update records).

💻 Syntax: Create Procedure

```
CREATE OR REPLACE PROCEDURE procedure_name (
  param1 IN datatype,
  param2 OUT datatype
)
IS
  -- local variables
BEGIN
  -- executable statements
END;
```

💡 Example:

```
CREATE OR REPLACE PROCEDURE greet_employee (
  emp_name IN VARCHAR2
)
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello, ' || emp_name || '!');
END;
```

💻 Call it:

```
BEGIN
  greet_employee('Vaibhav');
END;
```

📜 Output:

```
Hello, Vaibhav!
```

🧠 Real-Life Scenario:

- You can create a procedure to:
  - Insert daily attendance
  - Generate monthly report
  - Log error details to an error table

🔷 Functions In PL/SQL

- A Function in PL/SQL is a named block of code that performs a specific task and returns a single value
- A function is similar to a procedure, but it must return a single value.
- It can accept input parameters, perform calculations or logic, and must return a value using the RETURN keyword.
- It is often used in expressions and SQL SELECT statements, unlike procedures.

💻 Syntax: Create Function

```
CREATE OR REPLACE FUNCTION function_name (
  param1 IN datatype
)
RETURN return_datatype
IS
  -- local variables
BEGIN
  -- executable statements
  RETURN value;
END;
```

💡 Example:

```
CREATE OR REPLACE FUNCTION get_bonus (
  salary IN NUMBER
)
RETURN NUMBER
IS
  bonus NUMBER;
BEGIN
  bonus := salary * 0.10;
  RETURN bonus;
END;
```

💻 Call it:

```
DECLARE
  b NUMBER;
BEGIN
  b := get_bonus(50000);
  DBMS_OUTPUT.PUT_LINE('Bonus is: ' || b);
END;
```

📄 Output:

```
Bonus is: 5000
```

🧠 Real-Life Scenario:

- You can create functions to:
  - Calculate tax based on salary
  - Return department name for a department ID
  - Validate input formats (e.g., phone number)

🧾 Difference Between Procedure and Function

Feature	Procedure	Function
Return Value?	❌ No	✅ Yes (must return a value)
Use in SQL Statement?	❌ No	✅ Yes (can be used inside SQL)
Call with BEGIN?	✅ Yes	✅ Yes
Use in SELECT?	❌ Not Allowed	✅ Allowed
Return with RETURN?	Optional (for exit)	Mandatory to return a value
Example Use	Insert, Update, Delete operations	Calculations, Validations, Lookups

📝 Important Notes:

- You can use parameters as:
  - IN — for input
  - OUT — for output
  - IN OUT — for both input and output
- Use DBMS_OUTPUT.PUT_LINE() to display output during debugging
- Procedures and functions can be stored in the database and called by other programs
- Modular Code: Both procedures and functions help in creating reusable, organized logic

🔁 Summary:

- Procedure → Performs a task but doesn't return value directly
- Function → Performs a task and returns a single value
- Use procedures for actions, like inserting data
- Use functions for calculations, like returning bonus or tax

PL/SQL Parameters: IN, OUT, IN OUT

1. IN Parameter (Default)

- Used to pass a value *into* a procedure or function.
- Acts like a read-only variable inside the subprogram.
- You cannot change its value inside the procedure — trying to do so gives an error.
- It's the default mode.

Syntax:

```
PROCEDURE greet_user(name IN VARCHAR2)
```

Example:

```
CREATE OR REPLACE PROCEDURE greet_user(name IN VARCHAR2) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello, ' || name || '!');
END;

BEGIN
  greet_user('Vaibhav');
END;
```

Output:

```
Hello, Vaibhav!
```

Note: name is just used — not modified.

2. OUT Parameter

- Used to return a value *out of* a procedure.
- The calling program sends nothing — the procedure fills the variable.
- Cannot be read *before* assigning a value inside the procedure.

Syntax:

```
PROCEDURE square(num IN NUMBER, result OUT NUMBER)
```

Example:

```
CREATE OR REPLACE PROCEDURE square(num IN NUMBER, result OUT
NUMBER) IS
BEGIN
  result := num * num;
END;

DECLARE
  output NUMBER;
BEGIN
  square(5, output);
  DBMS_OUTPUT.PUT_LINE('Square is: ' || output);
END;
```

Output:

```
Square is: 25
```

Note: result value is *sent back* to the calling block.

◇ 3. IN OUT Parameter

- Acts as both input and output.
- Value is passed in, used and modified, then passed back.
- Useful when you want to update the passed value.

💻 Syntax:

```
PROCEDURE update_salary(salary IN OUT NUMBER)
```

💡 Example:

```
CREATE OR REPLACE PROCEDURE update_salary(salary IN OUT NUMBER)
IS
BEGIN
  salary := salary + 1000;
END;

DECLARE
  emp_salary NUMBER := 15000;
BEGIN
  update_salary(emp_salary);
  DBMS_OUTPUT.PUT_LINE('Updated Salary: ' || emp_salary);
END;
```

📜 Output:

```
Updated Salary: 16000
```

🧠 Real-Life Scenario: Pass an employee's current salary, give hike, and return the updated amount.

◇ RETURN in Function

- Unlike procedures, functions return a value directly using RETURN.
- A function always has a return data type defined.
- The returned value can be used in SELECTs, assignments, etc.

💻 Syntax:

```
FUNCTION function_name(param1 IN type) RETURN return_type
```

💡 Example:

```
CREATE OR REPLACE FUNCTION get_area(radius IN NUMBER)
RETURN NUMBER IS
BEGIN
  RETURN 3.14 * radius * radius;
END;

DECLARE
  area NUMBER;
BEGIN
  area := get_area(5);
  DBMS_OUTPUT.PUT_LINE('Circle Area: ' || area);
END;
```

📜 Output:

```
Circle Area: 78.5
```

📌 Note: You cannot use OUT or IN OUT in function parameter list if you're calling it from SQL.

📝 Important Notes:

- Always use IN when you just want to pass input data
- Use OUT to return results
- Use IN OUT when value needs to be updated inside and returned
- Functions must return exactly one value via RETURN

🗃️ Difference Between Procedure vs Function (Summary)

Feature	Procedure	Function
Return value	❌ Uses OUT parameter	✅ Uses RETURN keyword
Usage in SQL	❌ Not allowed directly	✅ Allowed
Purpose	Perform an action	Perform and return a value
Parameters	IN, OUT, IN OUT	Mostly IN
Syntax	PROCEDURE p_name(...)	FUNCTION f_name(...) RETURN type

🔷 What is a Cursor in PL/SQL?

- When an SQL statement is processed, oracle creates a memory area known as context area.
- A cursor is a pointer to this context area. It contains all information needed for processing the statements.
- In PL/SQL the context area is controlled by the cursor. A cursor contains information on a select statement & the rows of data accessed by it.
- A cursor is used to refer to a program to fetch and process the rows returned by the SQL statement one at a time.
- A cursor is a memory pointer in PL/SQL that helps in retrieving and processing query results row by row.
- In PL/SQL, when you run a SELECT query that returns multiple rows, you cannot directly assign all rows to variables — instead, you use cursors to process each row individually.
- A cursor acts like a bridge between a SQL query and PL/SQL code to read multiple records one at a time.
- A cursor is like a pointer or handle that helps you fetch and process multiple rows returned by a query one row at a time.

🧠 Think of it like:
A cursor is a for-loop for SQL results — it lets you go through each row of a query result step by step.

🧠 Why Do We Need a Cursor?
In PL/SQL, a normal SELECT INTO can only handle one row.
But if your query returns multiple rows, you need a cursor to handle them one at a time.

🔄 Cursor Life Cycle (Explicit & Parameterized)

- ✔ 1. Declare — define the SELECT query
- ✔ 2. Open — start executing the query
- ✔ 3. Fetch — retrieve one row at a time
- ✔ 4. Close — release memory and lock

❖ Types of Cursors

◇ 1. Implicit Cursor

- Created automatically by Oracle when you perform a DML (INSERT, UPDATE, DELETE) or a SELECT INTO query.
- You do not declare it manually — Oracle handles everything for you in the background.
- Use attributes like SQL%ROWCOUNT, SQL%FOUND, SQL%NOTFOUND, and SQL%ISOPEN to track status.

🔹 Cursor Attributes (for both types)

- %FOUND → Returns TRUE if a row was fetched
- %NOTFOUND → Returns TRUE if no row was fetched
- %ROWCOUNT → Number of rows fetched so far
- %ISOPEN → Returns TRUE if the cursor is still open

💡 Example: SELECT salary INTO v_sal FROM employees WHERE emp_id = 100;

📌 Use when: You expect only one row or performing DML operations.

💻 Syntax:

```
SELECT column INTO variable FROM table WHERE condition;
```

💡 Example:

```
DECLARE
  v_name employees.first_name%TYPE;
BEGIN
  SELECT first_name INTO v_name
  FROM employees
  WHERE employee_id = 100;

  DBMS_OUTPUT.PUT_LINE('Name: ' || v_name);
END;
```

📃 Output:

```
Name: Steven
```

📌 Note: Use SQL%ROWCOUNT, SQL%FOUND, SQL%NOTFOUND, and SQL%ISOPEN for status.

◇ 2. Explicit Cursor

- You manually declare and manage the cursor to fetch multiple rows from a SELECT query.
- Gives full control over the row-fetching process.
- You need to perform 4 steps:
  1. DECLARE
  2. OPEN
  3. FETCH
  4. CLOSE

📌 Use when: Query returns multiple rows, and you want to manually handle data row-by-row.

💻 Syntax:

```
DECLARE
  CURSOR emp_cur IS
    SELECT first_name, salary FROM employees WHERE department_id = 10;
  v_name employees.first_name%TYPE;
  v_salary employees.salary%TYPE;
BEGIN
  OPEN emp_cur;
  LOOP
    FETCH emp_cur INTO v_name, v_salary;
    EXIT WHEN emp_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(v_name || ' earns ' || v_salary);
  END LOOP;
  CLOSE emp_cur;
END;
```

📜 Output:

```
Neena earns 12000
Lex earns 10000
…
```

📌 Note: Always close the cursor to free resources.

◇ 3. Cursor FOR Loop

- A simplified version of an explicit cursor.
  PL/SQL automatically handles OPEN, FETCH, and CLOSE steps.
- Each row is fetched into a record variable inside the loop.
- Makes the code cleaner and less error-prone.
- No need to declare extra variables for fetched values — the loop variable acts as a record.

📌 Use when: You just need to iterate through records and don't need complex logic between fetch and close.

💻 Syntax:

```
FOR record_var IN cursor_name LOOP
  -- use record_var.column_name
END LOOP;
```

💡 Example:

```
DECLARE
  CURSOR emp_cur IS
    SELECT first_name, salary FROM employees WHERE department_id = 20;
BEGIN
  FOR emp IN emp_cur LOOP
    DBMS_OUTPUT.PUT_LINE(emp.first_name || ' earns ' || emp.salary);
  END LOOP;
END;
```

📜 Output:

```
Michael earns 13000
Pat earns 6000
…
```

📌 Note: emp acts as a row-type record variable automatically.

◇ 4. Parameterized Cursor

- A cursor that accepts input parameters (like functions) to make queries dynamic and reusable.
- You can pass different values each time the cursor is opened.
- Helps in filtering data without rewriting the query.

📌 Use when: You want to run the same cursor logic with different WHERE conditions dynamically.

💡 Example: Cursor with department ID or job title as parameter.

💻 Syntax:

```
CURSOR cursor_name(parameter datatype) IS
  SELECT ... WHERE column = parameter;
```

💡 Example:

```
DECLARE
  CURSOR emp_cur(dept_id NUMBER) IS
    SELECT first_name, salary FROM employees WHERE department_id = dept_id;
BEGIN
  FOR emp IN emp_cur(30) LOOP
    DBMS_OUTPUT.PUT_LINE(emp.first_name || ' earns ' || emp.salary);
  END LOOP;
END;
```

📜 Output:

```
David earns 7500
John earns 7200
…
```

📌 Note: Acts like a function with input arguments.

◇ 5. REF Cursor (Cursor Variables)

- A cursor variable that can point to different result sets at runtime.
- Unlike normal cursors, REF Cursors are dynamic — the structure of the result set can vary.
- Often used to return query results from procedures/functions to other PL/SQL blocks or front-end apps.

🔹 Two types:

- Strong REF Cursor → Fixed return structure
- Weak REF Cursor → Flexible (any structure)

📌 Use when:

- Writing generic procedures that return varying result sets.
- Returning rows to host programs like Java, .NET, or Python.

💻 Syntax (Weak REF CURSOR):

```
DECLARE
  TYPE ref_type IS REF CURSOR;
  emp_ref ref_type;
  v_name employees.first_name%TYPE;
BEGIN
```

```
  OPEN emp_ref FOR SELECT first_name FROM employees WHERE
department_id = 40;
  LOOP
    FETCH emp_ref INTO v_name;
    EXIT WHEN emp_ref%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Employee: ' || v_name);
  END LOOP;
  CLOSE emp_ref;
END;
```

🗒️ Output:

```
Employee: Rose
Employee: Jack
…
```

📌 Note:

- REF CURSORs can be passed to procedures/functions.
- Ideal for dynamic applications.

🧮 Difference Between Cursors

Cursor Type	Need to Declare	Suitable for	💡 Example Use
Implicit	❌	Single row	SELECT INTO
Explicit	✅	Multiple rows	Fetch employees manually
Cursor FOR Loop	✅ (simplified)	Multiple rows	Clean looping
Parameterized	✅	Dynamic WHERE	Employees by dept ID
REF Cursor	✅ (advanced)	Dynamic SQL	Procedure/function passing

🧠 Real-Life Scenarios:

Cursor Type	Used In Scenario
Implicit Cursor	When updating employee salaries in a single command
Explicit Cursor	Reading employee records and applying custom logic row-by-row
Cursor FOR Loop	Generating monthly salary slips automatically
Parameterized Cursor	Creating reports for different departments by passing department_id dynamically
REF Cursor	Passing dynamic result sets from PL/SQL to Java front-end for data visualization

📝 Important Notes:

- Always handle %NOTFOUND and %FOUND to avoid infinite loops.
- Cursors consume memory. Always CLOSE explicitly declared ones.
- Prefer Cursor FOR Loop for cleaner, readable code.
- Use REF cursors in advanced procedures/packages.

🔁 Summary: Cursor

- Cursors help fetch multiple rows from a query one at a time
- You use:
  - Implicit Cursors → For automatic handling of single-row or DML queries
  - Explicit Cursors → When you need full control over row-by-row logic
  - FOR Loop Cursors → When you want cleaner code and don't need manual handling
  - Parameterized Cursors → When you need dynamic query filtering
  - REF Cursors → When working with dynamic SQL and interacting with applications.

🔷 What is Exception Handling in PL/SQL?

- An exception is an error or unexpected situation that occurs during the execution of a PL/SQL block.
- Oracle provides mechanisms to catch and handle these exceptions gracefully without crashing the program.
- Exception handling is done inside the EXCEPTION section of a PL/SQL block.

💻 Syntax of Exception Handling:

```
DECLARE
  -- variable declarations
BEGIN
  -- executable statements
EXCEPTION
  WHEN exception_name THEN
    -- handle the exception
END;
```

🔶 Types of Exceptions in PL/SQL

◇ 1. Predefined Exceptions

- These are internally defined by Oracle.
- You can directly use them in your WHEN clause — no need to declare them.
- They're automatically triggered when a standard runtime error occurs.

🔹 Common Predefined Exceptions:

Exception Name	Triggered When...
NO_DATA_FOUND	A SELECT INTO returns no rows.
TOO_MANY_ROWS	A SELECT INTO returns more than one row.
ZERO_DIVIDE	Division by zero is attempted.
INVALID_NUMBER	Conversion from string to number fails.
VALUE_ERROR	Assignment fails due to incompatible values or sizes.

💡 Example: NO_DATA_FOUND

```
DECLARE
  v_name VARCHAR2(50);
BEGIN
  SELECT name INTO v_name FROM employees WHERE emp_id = 9999;
  DBMS_OUTPUT.PUT_LINE('Name: ' || v_name);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Employee not found.');
END;
```

📜 Output:

```
Employee not found.
```

◇ 2. User-Defined Exceptions

- When Oracle doesn't provide a specific exception for your logic, you can create your own named exceptions.
- You can define your own custom exceptions using the EXCEPTION keyword.
- Use RAISE to trigger them manually.

💻 Syntax:

```
DECLARE
  e_invalid_salary EXCEPTION;  -- define
  salary NUMBER := -5000;
BEGIN
  IF salary < 0 THEN
    RAISE e_invalid_salary;  -- raise
  END IF;
EXCEPTION
  WHEN e_invalid_salary THEN
    DBMS_OUTPUT.PUT_LINE('Invalid salary!');
END;
```

📜 Output:

```
Invalid salary!
```

🧠 Real-Life Scenario:

- If a user tries to enter a negative quantity or price, raise a custom error before inserting into the DB.

◇ 3. Using RAISE Statement

- RAISE is used to manually throw an exception.
- You can raise either:
  - A predefined exception
  - A user-defined exception
- Use inside IF conditions or logic checks to exit early.
- Often used after a validation check or business logic failure.

💡 Example:

```
DECLARE
  e_negative_salary EXCEPTION;
  salary NUMBER := -2000;
BEGIN
  IF salary < 0 THEN
```

```
    RAISE e_negative_salary;
  END IF;
EXCEPTION
  WHEN e_negative_salary THEN
    DBMS_OUTPUT.PUT_LINE('Salary cannot be negative!');
END;
```

📌 Note: Always use RAISE when your business rule fails — e.g., "Employee age should be > 18".

◇ 4. Using RAISE_APPLICATION_ERROR

- This is a built-in procedure of the DBMS_STANDARD package.
- This is used to raise custom, application-specific errors with error codes and messages.
- You can return an Oracle error between -20000 and -20999.
- Mostly used in:
  - Stored Procedures
  - Business Logic validation
  - Logging and auditing

💡 Example:

```
BEGIN
  RAISE_APPLICATION_ERROR(-20001, 'Something went wrong in your logic!');
END;
```

📜 Output:

```
ORA-20001: Something went wrong in your logic!
```

🔸 IMP POINT: RAISE_APPLICATION_ERROR can be used in stored procedures to inform calling programs (Java, Python, etc.) about the failure.

🧠 Real-Life Use:

- When you want the application calling your PL/SQL procedure to know why it failed (like invalid stock code, unauthorized access, etc.)

◇ 5. Extra: WHEN OTHERS

- It is a catch-all exception — it catches any unhandled exception.
- Use it wisely — don't silently suppress real errors.

💻 Syntax: Using WHEN OTHERS

```
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Divide by zero error!');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Some unknown error occurred.');
```

💡 Example:

```
BEGIN
  NULL; -- simulate unknown error
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An unknown error occurred.');
END;
```

🧠 Real-Life Scenarios:

Scenario	Use
Employee not found in database	Use NO_DATA_FOUND
Invalid bonus calculation (like < 0)	Raise a user-defined exception
Division or average salary error	Catch ZERO_DIVIDE
Complex business rule violation	Use RAISE_APPLICATION_ERROR

📊 Difference Table:

Feature	Predefined Exception	User-defined Exception	RAISE	RAISE_APPLICATION_ERROR
Defined by Oracle?	✅ Yes	❌ No (you define)	❌	❌
Need to Declare?	❌ No	✅ Yes	✅	❌
Raises Exception?	✅ Automatically	✅ Manually (with RAISE)	✅	✅
Custom Message Support	❌ No	❌ (but with logic)	❌	✅ Yes
Used in Business Logic	⚠️ Sometimes	✅ Frequently	✅	✅

📝 Important Notes:

- Always handle exceptions to prevent program crashes.
- You can have multiple WHEN clauses for different exceptions.
- Use OTHERS clause to catch any unhandled exception.
- Always write exception handlers to avoid Oracle throwing unhandled errors
- Don't suppress with WHEN OTHERS — always log or handle
- Use logging tables to store exceptions during failures
- Use RAISE_APPLICATION_ERROR in production systems for clear communication

🔁 Summary:

- Exception handling in PL/SQL helps in writing error-resilient code
- Types:
  - Predefined → Oracle built-in
  - User-defined → Defined and raised manually
  - RAISE → To trigger exceptions
  - RAISE_APPLICATION_ERROR → Custom message with error number
- Use exception blocks to log errors, continue the process, or notify users properly.

🔷 PL/SQL Packages

- A package is a schema object that groups related procedures, functions, variables, constants, cursors, and exceptions into a single unit.
- A PL/SQL Package is a collection of related subprograms and other elements grouped together.
- It has two parts:
  - Specification (Spec) – declares what's available publicly.
  - Body – contains the actual implementation.

- Think of it like a class in Java: the spec is like a class interface and the body is the implementation.

🔹 Package Specification (Header)

- Declares procedures, functions, cursors, variables, constants that are public (visible outside).
- No logic is written here — only declarations.

💻 Syntax:

```
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE display_emp(emp_id IN NUMBER);
  FUNCTION get_salary(emp_id IN NUMBER) RETURN NUMBER;
END emp_pkg;
```

🔹 Package Body

- Contains the implementation of what is declared in the spec.
- You can also define private procedures/functions here that are not accessible outside.

💻 Syntax:

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS

  -- Public Procedure
  PROCEDURE display_emp(emp_id IN NUMBER) IS
    v_name VARCHAR2(50);
  BEGIN
    SELECT ename INTO v_name FROM emp WHERE empno = emp_id;
    DBMS_OUTPUT.PUT_LINE('Employee Name: ' || v_name);
  END;

  -- Public Function
  FUNCTION get_salary(emp_id IN NUMBER) RETURN NUMBER IS
    v_sal NUMBER;
  BEGIN
    SELECT sal INTO v_sal FROM emp WHERE empno = emp_id;
    RETURN v_sal;
  END;

  -- Private Procedure (Not in spec)
  PROCEDURE internal_log IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Log: Internal Use Only');
  END;

END emp_pkg;
```

📜 Output Example (Calling Package Elements):

```
BEGIN
  emp_pkg.display_emp(7369);
  DBMS_OUTPUT.PUT_LINE('Salary: ' || emp_pkg.get_salary(7369));
END;
```

📜 Output:

```
Employee Name: SMITH
Salary: 800
```

🧾 Public vs Private Procedures in Package

Feature	Public Procedure	Private Procedure
Declared In	Package Specification	Package Body Only
Accessible Outside?	✅ Yes	❌ No
Use Case	Used by external code (SQL/Apps)	Helper subroutines inside package
Example Name	display_emp, get_salary	internal_log

📌 Note: If a procedure is not declared in the spec, it's private.

🧠 Real-Life Scenario:

Let's say a company has:

- display_emp (Public): For showing employee names.
- get_salary (Public): For returning salaries.
- internal_log (Private): For logging internally, no one outside needs it.

They wrap it all into a package so that:

- Everything related to employee logic stays together.
- Security is maintained (no one can call internal_log directly).

📝 Important Notes:

- You can initialize global variables in the package body.
- Packages improve performance by loading all logic into memory on first call.
- You can overload procedures/functions in packages (same name, different parameters).
- Can also define package-level constants, cursors, and exceptions.

🔁 Summary:

- A PL/SQL Package = Spec (what) + Body (how)
- Spec declares public stuff
- Body writes the logic and can include private subprograms
- Used for modularity, reusability, performance, and security

🔷 What is a Trigger in PL/SQL?

- A trigger is a stored PL/SQL block that automatically executes or fires when a specific event (INSERT, UPDATE, DELETE) occurs on a table or view.
- Triggers are invoked by the oracle engine automatically whenever a specified event occurs.
- Triggers are stored into the database and invoked repeatedly when specific conditions match.
- It helps automate tasks.
- Cannot be manually called like procedures.
- Associated with a table or a view.
- Can be defined to fire before or after the DML operation.
- In a trigger, :NEW refers to the new values of the row (inserted or updated).
- Similarly, :OLD refers to the old values of the row (before update /delete).

🔹 Trigger Structure (Syntax)

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF}
{INSERT | UPDATE | DELETE}
ON table_name
[FOR EACH ROW]
[WHEN (condition)]
DECLARE
  -- Variable declarations
BEGIN
  -- Trigger logic
END;
/
```

🏵 Types of Triggers:

1. BEFORE Trigger
2. AFTER Trigger
3. INSTEAD OF Trigger (Only for Views)
4. Row-Level Trigger
5. Statement-Level Trigger
6. Compound Triggers (Oracle 11g+)

◇ 1. BEFORE Trigger

- Fires before the DML operation.
- Mainly used for validation or modification of data before committing to the table.

💡 Example: Validate salary before insert

```
CREATE OR REPLACE TRIGGER trg_before_emp
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
  IF :NEW.salary < 10000 THEN
    RAISE_APPLICATION_ERROR(-20001, 'Salary must be at least 10,000');
  END IF;
END;
/
```

🧻 Output

If someone tries to insert an employee with salary 5000, Oracle will throw an error:

```
ORA-20001: Salary must be at least 10,000
```

🧠 Real-Life Scenario:

🔹 Used in e-commerce platforms for data validation:

- When a customer places an order, a BEFORE INSERT trigger checks if the product stock is available.
- If stock is zero, the trigger prevents the insert into the orders table.

◇ 2. AFTER Trigger

- Fires after the DML operation.
- Used for logging, auditing, notifications, etc.

💡 Example: Log employee inserts

```
CREATE OR REPLACE TRIGGER trg_after_emp
AFTER INSERT ON employees
FOR EACH ROW
BEGIN
  INSERT INTO emp_audit(emp_id, action_time)
  VALUES (:NEW.emp_id, SYSDATE);
END;
/
```

📜 Output

Every new insert in employees logs the action in emp_audit.

🧠 Real-Life Scenario:

🔹 Used in banking systems to log transactions:

- After a money transfer (UPDATE on balance), an AFTER UPDATE trigger logs the old and new balances into a transaction log table for audit trail and regulatory compliance.

◇ 3. INSTEAD OF Trigger (Only for Views)

- Used when you want to perform actions on a view (which normally doesn't support DML).
- Define how the view should handle DML.

💡 Example: Insert into view emp_view that joins multiple tables

```
CREATE OR REPLACE TRIGGER trg_instead_view
INSTEAD OF INSERT ON emp_view
FOR EACH ROW
BEGIN
  INSERT INTO employees(emp_id, name) VALUES (:NEW.emp_id, :NEW.name);
  INSERT INTO departments(dept_id, dept_name) VALUES (:NEW.dept_id,
:NEW.dept_name);
END;
/
```

📜 Output

Allows inserting into a complex view by inserting into base tables.

🧠 Real-Life Scenario:

🔹 Used in HR reporting dashboards:

- The front-end reports are built using views that combine employee and department data. An INSTEAD OF trigger is used to allow inserts into these views so that both employee and department records are updated in their respective base tables.

◇ 4. Row-Level Trigger

- Defined using FOR EACH ROW.
- Executes once for each row affected.
- Access old/new values using :OLD, :NEW.

💡 Example: Audit salary changes row-by-row

```
CREATE OR REPLACE TRIGGER trg_salary_audit
AFTER UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
  INSERT INTO salary_audit(emp_id, old_sal, new_sal, mod_date)
  VALUES (:OLD.emp_id, :OLD.salary, :NEW.salary, SYSDATE);
END;
/
```

📜 Output:

Each salary update will log the old and new values.

🧠 Real-Life Scenario:

🔹 Used in inventory systems:

- Every time a product's quantity is updated in the inventory table, a ROW-LEVEL AFTER UPDATE trigger records the quantity change in an inventory audit table with user details and timestamp.

◇ 5. Statement-Level Trigger

- Default type (if FOR EACH ROW is not specified).
- Executes once per statement, not per row.

💡 Example: Log the number of rows deleted

```
CREATE OR REPLACE TRIGGER trg_del_log
AFTER DELETE ON employees
BEGIN
  INSERT INTO log_table(msg, log_time)
  VALUES ('Employees deleted', SYSDATE);
END;
/
```

📜 Output

Deletes 10 rows → trigger runs once, logs one message.

📝 Important Notes:

- Row-level → Runs row by row → uses :NEW and :OLD.
- Statement-level → Runs once per SQL statement → cannot use row values.

🧠 Real-Life Scenario:

🔹 Used in access monitoring systems:

- When someone deletes a record from a secured table (e.g., user data), a STATEMENT-LEVEL AFTER DELETE trigger sends an alert email to the admin team without caring about which rows were deleted.

◇ 6. Compound Triggers (Oracle 11g+)

- Useful when you want multiple timing points (before, after) in one trigger.
- Avoids multiple trigger creation and avoids mutating table error.

💡 Example: Track rows inserted & log count

```
CREATE OR REPLACE TRIGGER trg_compound_example
FOR INSERT ON employees
COMPOUND TRIGGER

  -- Declare section
  row_count NUMBER := 0;

  BEFORE STATEMENT IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Starting insert...');
  END BEFORE STATEMENT;

  AFTER EACH ROW IS
  BEGIN
    row_count := row_count + 1;
  END AFTER EACH ROW;

  AFTER STATEMENT IS
  BEGIN
    INSERT INTO insert_log(msg, row_count, log_time)
    VALUES ('Inserted into employees', row_count, SYSDATE);
  END AFTER STATEMENT;

END trg_compound_example;
/
```

📜 Output:

If 5 employees are inserted, it logs row_count = 5 after the entire statement.

🧠 Real-Life Scenario:

🔹 Used in bulk data operations (e.g., data migration tools):

- When migrating thousands of customer records, a COMPOUND TRIGGER is used to track total inserted rows and log the result only once at the end of the operation — improving performance and avoiding mutating errors.

📝 Important Notes on Triggers

- Triggers run automatically — you cannot call them manually.
- Triggers should not contain complex business logic; use procedures/functions for that.
- Avoid DML on the same table inside a row-level trigger (mutating table error).
- Use compound triggers to handle bulk processing efficiently.
- Always test triggers thoroughly as they silently run in the background.
- Avoid having too many triggers on one table — it may slow down DML operations.
- Don't create triggers just for fun — always have a valid use case.
- :OLD and :NEW are available only in row-level triggers.
- INSTEAD OF triggers are the only way to perform DML on complex views.
- Make sure to document triggers — they're often hard to trace in large systems.

Summary

Type	Timing	Scope	Use Case
BEFORE	Before DML	Row/Stmt	Validation, Data Transformation
AFTER	After DML	Row/Stmt	Auditing, Logging
INSTEAD OF	Replaces DML	Row Only	DML on Views
Row-Level	Each Row	Needs FOR EACH ROW	Fine-grained Control
Statement-Level	Entire DML	Default	Summary, Count, Control
Compound	Multi-timing	Row & Stmt	Combine logic, avoid mutating error

PL/SQL COLLECTIONS & RECORDS

- Collections are PL/SQL structures used to store multiple values (like arrays in other languages).
- They help in bulk processing, looping through values, and temporary storage during PL/SQL operations.

Collection Methods

- All PL/SQL collections support some common built-in methods that help you handle elements:

Method	Purpose	Supported By
COUNT	Returns total number of elements	All
EXTEND	Adds empty elements	Nested Table, VARRAY
DELETE	Removes elements	Nested Table, Assoc. Array
EXISTS(n)	Checks if element at index n exists	Assoc. Array
FIRST/LAST	Gives lowest/highest index value	All
PRIOR/NEXT	Used for iteration	All
TRIM	Removes elements from end	Nested Table, VARRAY

💡 Example:

```
names.DELETE(2); -- deletes 2nd element
IF names.EXISTS(2) THEN …
```

📝 Important Notes:

- Not all methods are available for all collection types.
- Always check .EXISTS() before accessing index in associative arrays.

🔷 Collection Types

◇ 1. RECORD

- A PL/SQL Record is a group of related data items (like a row in a table) of different data types.
- You can use it to store a single row of data from a table or cursor.
- Record = Row; Collection = Column Group

💻 Syntax:

```
DECLARE
  TYPE emp_rec_type IS RECORD (
    emp_id    NUMBER,
    emp_name  VARCHAR2(50),
    salary    NUMBER
  );
  emp_rec emp_rec_type;
BEGIN
  emp_rec.emp_id := 101;
  emp_rec.emp_name := 'Vaibhav';
  emp_rec.salary := 60000;

  DBMS_OUTPUT.PUT_LINE('ID: ' || emp_rec.emp_id || ', Name: ' ||
emp_rec.emp_name || ', Salary: ' || emp_rec.salary);
END;
/
```

💡 Example: Storing a single employee's details

🧾 Output:

```
ID: 101, Name: Vaibhav, Salary: 60000
```

🧠 Real-Life Scenario:

- Used in procedures/functions to hold one row of data fetched from the employees table for further processing.

📝 Important Notes:

- Record fields can be of different data types.
- Can be based on table%ROWTYPE to automatically match table structure.

◇ 2. NESTED TABLE

- A nested table is an unordered set of homogeneous elements (same type), just like a regular table inside PL/SQL.
- Can grow dynamically and be stored in database columns.

💻 Syntax:

```
DECLARE
```

```
  TYPE name_table IS TABLE OF VARCHAR2(100);
  names name_table := name_table('Vaibhav', 'Ajay', 'Ravi');
BEGIN
  names.EXTEND;
  names(4) := 'Karan';

  FOR i IN 1..names.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('Name: ' || names(i));
  END LOOP;
END;
/
```

💡 Example: Storing and printing multiple names

🗒️ Output:

```
Name: Vaibhav
Name: Ajay
Name: Ravi
Name: Karan
```

🧠 Real-Life Scenario:

- Used in student registration systems to store multiple subjects selected by each student in a single row.

📝 Important Notes:

- Supports methods like EXTEND, DELETE, COUNT.
- Needs constructor syntax to initialize (:= type_name(...)).
- Can be used in SQL if stored in database columns.

◇ 3. VARRAY (Variable-size Array)

- A VARRAY is a bounded collection — meaning it has a fixed maximum size.
- Best when the maximum number of elements is known in advance.

💻 Syntax:

```
DECLARE
  TYPE score_array IS VARRAY(5) OF NUMBER;
  scores score_array := score_array(78, 85, 92);
BEGIN
  FOR i IN 1..scores.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('Score: ' || scores(i));
  END LOOP;
END;
/
```

💡 Example: Storing exam scores (maximum 5)

🗒️ Output:

```
Score: 78
Score: 85
Score: 92
```

🧠 Real-Life Scenario:

- Used in quiz apps or test systems to store maximum 5 test scores of each student.

📝 Important Notes:

- Size must be declared while defining the type.
- Can be stored in tables.

- Does not support the DELETE method.

◇ 4. ASSOCIATIVE ARRAY (INDEX-BY Table)

- Also called index-by tables, these are key-value pairs, where keys can be numbers or strings.
- Useful when you need dynamic access via key and don't care about order.

💻 Syntax:

```
DECLARE
  TYPE emp_table_type IS TABLE OF VARCHAR2(50) INDEX BY
BINARY_INTEGER;
  emp_names emp_table_type;
BEGIN
  emp_names(101) := 'Vaibhav';
  emp_names(102) := 'Ajay';
  emp_names(103) := 'Ravi';

  FOR i IN 101..103 LOOP
    DBMS_OUTPUT.PUT_LINE('EmpID: ' || i || ', Name: ' || emp_names(i));
  END LOOP;
END;
/
```

💡 Example: Associating employee IDs with names

🗄️ Output:

```
EmpID: 101, Name: Vaibhav
EmpID: 102, Name: Ajay
EmpID: 103, Name: Ravi
```

🧠 Real-Life Scenario:

- Used in billing systems to map product IDs to names/prices without using joins or lookup tables.

📝 Important Notes:

- Cannot be stored in DB tables.
- Extremely fast and efficient in memory.
- Supports dynamic key types (like strings too).

◇ 5. Nested Collections (Multi-level)

- A collection inside another collection — advanced concept but helpful in scenarios like hierarchical data (e.g., departments → employees).

💻 Syntax:

```
TYPE emp_list IS TABLE OF VARCHAR2(30);
TYPE dept_emp_table IS TABLE OF emp_list;
```

🧠 Real-Life Scenario: A department stores a list of employees in each row as a nested list.

📝 Important Notes:

- Not commonly used in beginner projects.
- Needs special handling using loops and constructors.

Difference Between All Collection Types

Feature	RECORD	NESTED TABLE	VARRAY	ASSOCIATIVE ARRAY
Heterogeneous types	✅	❌	❌	❌
Fixed Size	❌	❌	✅	❌
Can be stored in DB	❌	✅	✅	❌
Indexed by	Field name	Integer	Integer	Integer/String
Supports methods	❌	✅ (EXTEND, DELETE)	✅ (EXTEND)	✅ (DELETE, EXISTS)

Important Notes:

- Use %TYPE and %ROWTYPE to make collections more flexible and maintainable.
- Always initialize collections before using.
- Collections are stored in memory, so use wisely in large data operations.
- Prefer BULK COLLECT + FORALL when performance matters.

Summary

Type	Use Case	Key Feature
RECORD	Store one row of different fields	Like a table row
NESTED TABLE	Dynamic array	Can grow/shrink, used in tables
VARRAY	Fixed size array	Predictable size, SQL support
ASSOCIATIVE ARRAY	Key-value structure	Memory-only, super fast
%ROWTYPE	Auto match table structure	Dynamic and reusable

🔷 DYNAMIC SQL

- Dynamic SQL lets you build and run SQL statements at runtime.
- It's useful when the structure of SQL (table/column names, conditions) is not known until execution.
- Use dynamic SQL when static SQL won't work (e.g., dynamic table names, dynamic DML, etc.)

🧠 Real-Life Scenario:

- In a multi-client system, dynamically creating or updating tables like client1_users, client2_users, etc.

◇ 1. EXECUTE IMMEDIATE

- Most commonly used for dynamic SQL in PL/SQL.
- Supports DML, DDL, and anonymous PL/SQL blocks.
- It allows you to run a SQL statement dynamically at runtime.
- Commonly used when SQL structure (like table name or condition) changes based on logic or user input.
- It supports DML (INSERT, UPDATE, DELETE), DDL (CREATE, DROP), and PL/SQL blocks.
- It's a simple and faster alternative to the DBMS_SQL package for most use cases.

💻 Basic Syntax:

```
EXECUTE IMMEDIATE 'SQL statement';
```

💡 Example – Create Table Dynamically:

```
DECLARE
  table_name VARCHAR2(30) := 'temp_users';
BEGIN
  EXECUTE IMMEDIATE 'CREATE TABLE ' || table_name || '(id NUMBER, name
VARCHAR2(50))';
  DBMS_OUTPUT.PUT_LINE('Table created dynamically!');
END;
/
```

📜 Output:

```
Table created dynamically!
```

🧠 Real-Life Scenario:

- Useful in automation scripts where table names are passed dynamically.

◇ 2. Using BIND VARIABLES with EXECUTE IMMEDIATE

- Prevents SQL injection and improves performance by reusing parsed statements.
- Bind variables are placeholders (:1, :2, etc.) that let you pass values safely into your dynamic SQL.
- They prevent SQL injection and allow Oracle to reuse parsed SQL for performance.
- Values are passed using the USING clause after the SQL statement.
- You can bind IN, OUT, and IN OUT variables, but only by position, not by name.

💻 Syntax with USING:

```
EXECUTE IMMEDIATE 'UPDATE employees SET salary = :1 WHERE emp_id =
:2' USING new_salary, emp_id_val;
```

💡 Example:

```
DECLARE
  new_salary NUMBER := 5000;
  emp_id NUMBER := 101;
BEGIN
  EXECUTE IMMEDIATE
   'UPDATE employees SET salary = :1 WHERE emp_id = :2'
   USING new_salary, emp_id;
  DBMS_OUTPUT.PUT_LINE('Salary updated dynamically.');
END;
/
```

📜 Output:
Salary updated dynamically.

📌 Note: Only positional binds (:1, :2) are supported.

◇ 3. RETURNING INTO Clause

- Used to fetch values from DML statements dynamically.
- It allows you to capture output from a DML statement (INSERT, UPDATE, DELETE) during dynamic execution.
- You use RETURNING INTO to get values like inserted IDs or deleted column values immediately.
- Only works if the SQL returns exactly one row.
- This is supported in EXECUTE IMMEDIATE but not in DBMS_SQL.

💻 Syntax:

```
EXECUTE IMMEDIATE
 'DELETE FROM employees WHERE emp_id = :1 RETURNING name INTO :2'
 USING emp_id_val RETURNING INTO emp_name;
```

📝 Important Notes:

- Only works for single-row operations.
- RETURNING INTO is not supported in DBMS_SQL.
-

◇ 4. EXECUTE IMMEDIATE with INTO Clause (for SELECT)

- Used when you dynamically select a value into a variable.
- Used to dynamically run a SELECT query and fetch its result into a variable.
- It only works when the SELECT returns one and only one row.
- You can combine it with USING if the query has placeholders.
- Great for dynamic counts, max/min, single lookups, etc.

💻 Syntax:

```
EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM employees' INTO v_count;
```

📜 Output:

```
Total count: 87
```

📝 Important Notes:

- Works only when the query returns exactly one row.

◇ 5. EXECUTE IMMEDIATE for Anonymous PL/SQL Block

- You can build dynamic PL/SQL blocks and run them on the fly.
- You can dynamically build and run entire PL/SQL logic blocks using EXECUTE IMMEDIATE.
- Useful when the logic changes based on input or needs to be stored externally (e.g., logs, templates).
- Helps in executing multiple statements or logic from strings.

💻 Example:

```
BEGIN
  EXECUTE IMMEDIATE '
    BEGIN
      DBMS_OUTPUT.PUT_LINE("Hello from dynamic block");
    END;';
END;
/
```

📜 Output:

```
Hello from dynamic block
```

◇ 6. SQL Injection & Prevention

- SQL Injection happens when user input is directly concatenated into a dynamic SQL string, allowing malicious SQL to run.
- It's a major security threat if not handled.
- Prevention: Always use bind variables, input validation, and Oracle's DBMS_ASSERT functions.
- Bind variables ensure user input is treated as data, not code.

❌ Dangerous:

```
EXECUTE IMMEDIATE 'DELETE FROM employees WHERE name = '' || user_input || '';
```

✅ Safe:

```
EXECUTE IMMEDIATE 'DELETE FROM employees WHERE name = :1' USING user_input;
```

📌 Note: Use DBMS_ASSERT.simple_sql_name() to validate object names.

📝 Important Notes:

- Always avoid concatenating raw input.
- Bind parameters protect against attacks.

◇ 7. DBMS_SQL Package

- Advanced method when you need flexibility and metadata control.
- A lower-level API for fully dynamic SQL execution, offering more control and flexibility than EXECUTE IMMEDIATE.
- Useful when you don't know the number or names of columns beforehand.
- Supports describe columns, bind by name, fetch rows, get metadata dynamically.
- More verbose and complex to use, but powerful in advanced use cases like report builders or ETL tools.

💻 Syntax Example:

```
DECLARE
  cur_id    INTEGER;
  rows      NUMBER;
BEGIN
  cur_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(cur_id, 'DELETE FROM employees WHERE department_id
= 90', DBMS_SQL.NATIVE);
  rows := DBMS_SQL.EXECUTE(cur_id);
  DBMS_SQL.CLOSE_CURSOR(cur_id);
  DBMS_OUTPUT.PUT_LINE(rows || ' rows deleted');
END;
/
```

📜 Output:

```
3 rows deleted
```

🧠 Real-Life Scenario:

- Used in dynamic reporting tools where column names are not known until runtime.

📝 Important Notes:

- More verbose than EXECUTE IMMEDIATE.
- Supports dynamic column parsing, binding, and fetching.
- Use EXECUTE IMMEDIATE for 90% of use cases.
- Use DBMS_SQL only when:
- You don't know the number/type of columns
- You want to fetch metadata dynamically
    - Avoid string concatenation — it's slow and insecure.
    - Use BULK COLLECT + FORALL with dynamic SQL for better performance.

🧮 Difference Between EXECUTE IMMEDIATE vs DBMS_SQL

Feature	EXECUTE IMMEDIATE	DBMS_SQL
Simplicity	✅ Easy	❌ Complex
Positional Bind Support	✅ Yes	✅ Yes
Named Bind Support	❌ No	✅ Yes
Metadata Control	❌ No	✅ Yes (describe columns)
Performance	✅ Faster	❌ Slightly slower
Use Case	DML, DDL, PL/SQL blocks	Dynamic queries/columns

🧠 Real-Life Scenario Summary:

Use Case	Dynamic SQL Role
Multi-client database architecture	Create/alter tables dynamically
Data migration utilities	Load data into dynamic structures
Dynamic reporting systems	Fetch unknown columns/tables
Admin tools or ETL framework	Execute stored SQL scripts dynamically

🔁 Summary

Concept	Description
EXECUTE IMMEDIATE	For simple DDL, DML, SELECT, anonymous blocks
USING Clause	For binding variables to dynamic SQL
RETURNING INTO	To fetch values from DML queries dynamically
INTO Clause	For fetching single-row SELECT results
Dynamic PL/SQL Blocks	Run runtime PL/SQL logic dynamically
SQL Injection Prevention	Always use bind variables or DBMS_ASSERT
DBMS_SQL	Advanced option for unknown SQL at runtime

🔷 Bulk Operations

- designed to boost performance when dealing with large volumes of data.
- All four are directly part of or related to PL/SQL Bulk Operations.
- They are widely used in ETL, reporting, automation scripts, data migration, and high-volume batch processing.

◇ 1. BULK COLLECT

- BULK COLLECT is used to fetch multiple rows from a query into a collection variable (like VARRAY, Nested Table, Associative Array) in one go.
- BULK COLLECT is a PL/SQL feature used to fetch multiple rows from a SELECT statement into a collection (like Nested Table, VARRAY, or Associative Array) in a single context switch between SQL and PL/SQL engines.
- Normally, each row fetched requires communication between SQL and PL/SQL, which slows down performance for large datasets.
- With BULK COLLECT, all the rows are fetched at once, reducing the overhead and significantly improving performance for read-heavy operations.
- It's especially useful when processing large volumes of data like logs, orders, or transaction records.
- Collections populated using BULK COLLECT can be processed using loops, FORALL, or further business logic.

- Reduces the number of times PL/SQL must fetch rows one-by-one → improves performance significantly.
- ➡️ Bulk Data Fetching : Used to fetch multiple rows from the database into a collection in one go.

💻 Syntax:

```
SELECT column1, column2
BULK COLLECT INTO collection1, collection2
FROM table_name
WHERE condition;
```

💡 Example:

```
DECLARE
  TYPE name_tab IS TABLE OF employees.ename%TYPE;
  TYPE sal_tab IS TABLE OF employees.sal%TYPE;
  v_names name_tab;
  v_sals sal_tab;
BEGIN
  SELECT ename, sal
  BULK COLLECT INTO v_names, v_sals
  FROM employees
  WHERE department_id = 10;

  FOR i IN v_names.FIRST .. v_names.LAST LOOP
    DBMS_OUTPUT.PUT_LINE(v_names(i) || ' earns ' || v_sals(i));
  END LOOP;
END;
/
```

📜 Output:

```
SMITH earns 3000
ADAMS earns 2500
```

🧠 Real-Life Scenario:

- Used in report generation, data migration, or when fetching large datasets like 1000+ records in one query.

📝 Important Notes:

- Can be used with SELECT INTO, FETCH INTO, or RETURNING INTO.
- Supports LIMIT clause for controlling memory usage in loops.

◇ 2. FORALL

- FORALL is used to perform DML operations (INSERT, UPDATE, DELETE) on collection elements in bulk, instead of looping row-by-row.
- FORALL is used for bulk DML operations (INSERT, UPDATE, DELETE) based on data stored in a PL/SQL collection.
- Instead of executing a DML operation row by row using a loop, FORALL sends all the operations to the SQL engine in a single call, drastically reducing processing time.
- It is designed for high-speed write-heavy processing — like batch updates, mass insertions, or deletions.
- Unlike FOR loops, FORALL does not support SELECT operations — it is strictly for modifying data.

- It works best with index-by tables and nested tables, and each row is referenced by its index in the collection.
- Improves performance drastically by reducing context switches between PL/SQL and SQL engines
- ➞ Bulk DML Processing: Used to perform DML (INSERT, UPDATE, DELETE) on multiple rows using a collection..

💻 Syntax:

```
FORALL index IN collection.FIRST .. collection.LAST
  DML_statement_using_collection(index);
```

💡 Example:

```
DECLARE
  TYPE id_array IS TABLE OF employees.emp_id%TYPE;
  v_ids id_array := id_array(101, 102, 103);
BEGIN
  FORALL i IN v_ids.FIRST .. v_ids.LAST
    DELETE FROM employees WHERE emp_id = v_ids(i);

  DBMS_OUTPUT.PUT_LINE('Bulk delete done.');
END;
/
```

📜 Output:

```
Bulk delete done.
```

🧠 Real-Life Scenario:

- Used in ETL scripts, bulk data cleanup, or mass updates/inserts, e.g., marking users inactive in bulk.

📝 Important Notes:

- Works only with index-by or nested tables (not VARRAY).
- Can't use SELECT in FORALL; only DML.
- Use SAVE EXCEPTIONS to continue on error.

◇ 3. SAVE EXCEPTIONS

- Used with FORALL to handle errors gracefully during bulk DML. If one DML fails, the others still proceed.
- When using FORALL, if a single row operation fails (e.g., constraint violation), the entire operation stops unless you use SAVE EXCEPTIONS.
- SAVE EXCEPTIONS allows PL/SQL to continue processing the remaining rows even if one or more fail.
- After execution, the details of the failed rows (error index and error code) can be accessed using SQL%BULK_EXCEPTIONS collection.
- This approach is extremely valuable in production scenarios where skipping a few bad rows is better than halting the whole job.
- It ensures fault-tolerant, resilient bulk operations and is heavily used in real-time ETL and automation pipelines.
- Allows identifying which rows failed while continuing execution for valid rows.
- ➞ Error Handling in FORALL: Used with FORALL to catch and continue on errors during bulk DML, instead of stopping execution.

💻 Syntax:

```
FORALL i IN collection.FIRST .. collection.LAST SAVE EXCEPTIONS DML_statement;
```

🔹 Then handle the error:

```
EXCEPTION
  WHEN OTHERS THEN
    FOR i IN 1 .. SQL%BULK_EXCEPTIONS.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE('Error on row ' ||
SQL%BULK_EXCEPTIONS(i).ERROR_INDEX);
      DBMS_OUTPUT.PUT_LINE(SQLERRM(-
SQL%BULK_EXCEPTIONS(i).ERROR_CODE));
    END LOOP;
```

💡 Example:

```
DECLARE
  TYPE emp_ids IS TABLE OF employees.emp_id%TYPE;
  v_ids emp_ids := emp_ids(101, 102, 999);  -- 999 doesn't exist
BEGIN
  FORALL i IN v_ids.FIRST .. v_ids.LAST SAVE EXCEPTIONS
    DELETE FROM employees WHERE emp_id = v_ids(i);

  DBMS_OUTPUT.PUT_LINE('Bulk delete attempted.');
EXCEPTION
  WHEN OTHERS THEN
    FOR i IN 1 .. SQL%BULK_EXCEPTIONS.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE('Failed at index: ' ||
SQL%BULK_EXCEPTIONS(i).ERROR_INDEX);
    END LOOP;
END;
/
```

🧾 Output:

```
Bulk delete attempted.
Failed at index: 3
```

🧠 Real-Life Scenario:

- Useful in financial systems or record processing where skipping errors is better than failing the entire batch (e.g., retrying later).

📝 Important Notes:

- Works only with FORALL.
- Use SQL%BULK_EXCEPTIONS to analyze failed rows.

◇ 4. LIMIT Clause with BULK COLLECT

- Used to control how many rows you fetch at once in a loop, reducing memory overload.
- The LIMIT clause is used when fetching large amounts of data in chunks using a cursor + BULK COLLECT.
- It restricts the number of rows fetched into memory during each iteration, helping manage memory consumption effectively.
- Without LIMIT, fetching thousands of rows in one go can cause memory overflow and slow performance.
- Typically used in loops: fetch a batch of rows, process them, then fetch the next batch until all data is consumed.

- Very useful for processing millions of rows in manageable segments, ensuring both performance and resource optimization.
- ➡️ Controlled Bulk Fetching: Used to fetch data in chunks when processing very large data sets, for memory efficiency.

💡 Example:

```
LOOP
  FETCH c1 BULK COLLECT INTO v_names LIMIT 100;
  EXIT WHEN v_names.COUNT = 0;
  -- Process 100 records at a time
END LOOP;
```

📝 Important Notes:
- Especially useful when fetching large datasets in chunks.
- Prevents memory overflow and improves performance.

🧮 Difference Between: BULK COLLECT vs FORALL

Feature	BULK COLLECT	FORALL
Use	Fetching rows from SELECT	Executing DML (INSERT/UPDATE/DELETE)
Works With	SELECT, FETCH	INSERT, UPDATE, DELETE
Performance Gain	On fetching	On DML
Error Handling	Via EXCEPTION block	Use SAVE EXCEPTIONS for partial fails

📝 Important Notes:
- Always use collection types with bulk operations.
- Combine BULK COLLECT + FORALL for best performance.
- Use %ROWTYPE or RECORD if fetching full rows.
- Avoid using BULK COLLECT without LIMIT for very large datasets.
- Use DBMS_OUTPUT.PUT_LINE or logs to track processing.

🧠 Real-Life Scenario:

Real-Life Task	Bulk Operation Used
Archiving old records	BULK COLLECT (fetch) + FORALL (insert/delete)
Mass update of pricing in e-commerce	FORALL with SAVE EXCEPTIONS
Generating monthly reports in HR	BULK COLLECT with LIMIT clause
Syncing large tables in ETL pipelines	BULK COLLECT + FORALL

🔁 Summary

Concept	Purpose	Use Case
BULK COLLECT	Fetch multiple rows into collection in one go	Report generation, large fetch
FORALL	Perform bulk DML using collection values	Mass delete/update/insert
SAVE EXCEPTIONS	Allow some DML failures without halting all	Resilient operations with logs
LIMIT Clause	Restrict rows fetched per iteration	Avoid memory overflow in loops

🔷 Advanced Concepts & Utilities

🏵 UTL_FILE (File Handling in PL/SQL)

- UTL_FILE is a tool provided by Oracle that lets your PL/SQL program read from or write to text files (like .txt files).
- These files are stored on the database server (not your personal computer).
- You can use this when you want to save data from your table into a file, or read data from a file into your program.
- UTL_FILE is an Oracle PL/SQL package used to read from and write to operating system text files from within the database.
- It allows interaction with files located on the server's file system, making it useful for exporting logs, audit data, or reports.
- Requires DIRECTORY object and write/read permissions.

🔸 Why do we need it?

- Suppose you want to export a list of all employee names to a file for HR.
- Or maybe save all errors into a log file during a batch job.
- UTL_FILE lets you do that directly from the PL/SQL code without needing any external script.

💻 Syntax:

```
DECLARE
  file_handler UTL_FILE.FILE_TYPE;  -- Like opening a file in notepad
BEGIN
  -- Open the file named 'output.txt' for writing ('W')
  file_handler := UTL_FILE.FOPEN('MY_DIR', 'output.txt', 'W');

  -- Write one line into that file
  UTL_FILE.PUT_LINE(file_handler, 'Hello from PL/SQL!');

  -- Close the file (like saving and closing notepad)
  UTL_FILE.FCLOSE(file_handler);
END;
```

💡 Example:

- You have a table of employees and want to write all names to a file:
  📌 This helps when HR asks for a quick export in a simple .txt file.

🗄 Output:

- A new file called output.txt will be created inside a special folder on the server.

- That file will contain:
  Hello from PL/SQL!

🧠 Real-Life Scenario:
- Error Logs: Save all failed records or error messages in a text file after a data load.
- Audit Report: Write every login record or transaction into a file for auditing purposes.
- Daily Report: Send a .txt file with daily sales to the sales team from the backend.

📝 Important Notes:
- You can't access your local laptop files — the file is created on the Oracle database server.
- You must ask your DBA (Database Admin) to create a folder (called DIRECTORY) and give you access to it.
- Without this permission, your program will show an error like:
  ORA-29283: Invalid file operation.

📌 In Short:
- UTL_FILE is like a notepad for PL/SQL — you can open a file, write lines, and save it — all using PL/SQL code! 📝

🔷 DBMS_SCHEDULER (Advanced Job Scheduling)
- It is a tool in Oracle used to automatically run programs at a fixed time or repeatedly (like every hour, day, week, etc.).
- You can think of it like an alarm clock for your PL/SQL code — you set a time, and it will execute your code automatically.
- Used for creating, managing, and monitoring scheduled jobs within the database.
- Replaces the older DBMS_JOB with more advanced features like logging, repeat intervals, dependencies, etc.

🔸 Why do we need it?
- Suppose you have a cleanup job that must run every night at 11 PM.
- Or you want to send daily sales reports to your team.

📌 This tool helps automate those jobs without manually running them.

💻 Syntax:

```
BEGIN
  DBMS_SCHEDULER.CREATE_JOB (
    job_name        => 'test_job',
    job_type        => 'PLSQL_BLOCK',
    job_action      => 'BEGIN DBMS_OUTPUT.PUT_LINE("Hello World"); END;',
    start_date      => SYSTIMESTAMP,
    repeat_interval => 'FREQ=DAILY; BYHOUR=9; BYMINUTE=0', -- Every day at
9 AM
    enabled         => TRUE
  );
END;
```

💡 Example:
- You want to print "Hello World" every day at 9 AM.
- This job will start at the given time and run automatically.

📜 Output:

- The above code runs every day at 9:00 AM and prints:
  Hello World
- You can replace this with any logic like inserting data, generating reports, etc.

🧠 Real-Life Scenario:

- Auto-send email reminders daily
- Generate a monthly billing report
- Clean up temporary files or data from tables weekly

📝 Important Notes:

- Supports external jobs (e.g., OS-level scripts).
- Can be monitored via ALL_SCHEDULER_JOBS.
- It is more powerful than DBMS_JOB (which is older).
- You can monitor, pause, or stop jobs easily using this tool.
- Uses standard scheduling format like:
  FREQ=HOURLY, FREQ=DAILY, FREQ=WEEKLY etc.

🏵 DBMS_JOB (Basic Job Scheduler - Legacy)

- It is an older method used to schedule jobs in Oracle.
- It's simple and good for small or one-time jobs.
- Works inside the database, and is useful for beginners or simple tasks.
- Older packages used to schedule and run background jobs like procedures or anonymous blocks.
- Simpler but less powerful than DBMS_SCHEDULER.

🔸 Why do we need it?

- You want to schedule a stored procedure to run every hour.
- You want a small job to clean a table daily — DBMS_JOB can do that quickly.

💻 Syntax:

```
DECLARE
  job_no NUMBER;
BEGIN
  DBMS_JOB.SUBMIT(job_no,
    'BEGIN DBMS_OUTPUT.PUT_LINE("Hello from DBMS_JOB"); END;',
    SYSDATE,
    'SYSDATE + 1/24' -- every 1 hour
  );
END;
```

💡 Example:

- Print a message every 1 hour using a scheduled job.

📜 Output:

Hello from DBMS_JOB
Repeated every hour automatically.

🧠 Real-Life Scenario:

- Run a cleanup job to delete logs daily
- Update data in a table every night at midnight

📝 Important Notes:

- Still supported, but DBMS_SCHEDULER is preferred.
- DBMS_JOB is still available, but Oracle recommends using DBMS_SCHEDULER now.
- It's easy to use but less flexible (can't do external scripts, for example).

❖ AUTONOMOUS TRANSACTIONS

- A normal transaction in PL/SQL (like inserting or updating data) can only commit or rollback once when your block ends.
- But with Autonomous Transaction, you can commit/rollback immediately — even while the main code is still running.
- Used when you need to commit/rollback a transaction independently from the main transaction block.
- Helpful in logging, auditing, or error-tracking without affecting the main logic.

🔸 Why do we need it?

- Imagine your code fails and you want to log an error in a table, but the full block is rolling back.
- You still want the error log to stay saved — so you use Autonomous Transaction to insert it separately and commit it immediately.

💻 Syntax:

```
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION; -- Special keyword
BEGIN
  INSERT INTO error_logs VALUES ('Something went wrong');
  COMMIT;  -- This will happen separately from the main block
END;
```

💡 Example:

- Inside an exception block, log error messages without affecting main logic.

📜 Output:

The log will be inserted even if your main transaction fails.

🧠 Real-Life Scenario:

- Logging error messages during failures
- Save log or audit information even if your main task fails
- Insert a record for tracking access, regardless of main logic

📝 Important Notes:

- Must use COMMIT or ROLLBACK inside.
- Commonly used in exception sections.
- You must always use COMMIT or ROLLBACK inside the autonomous block.
- Works well for logs, tracking, audit, or recording exceptions.
- Declared using:
  PRAGMA AUTONOMOUS_TRANSACTION;

❖ SEQUENCES

- A sequence is a special object in Oracle that automatically generates numbers (like 1, 2, 3, …).
- You mostly use it when you need a unique value each time — like for primary keys.
- Objects used to generate unique numbers, often for primary keys.
- Can be used in triggers, INSERTs, and programmatically.

🔸 Why do we need it?

- To give each new row in a table a unique ID.
- Avoids manual work or duplicates.

💻 Syntax:

```
CREATE SEQUENCE emp_seq
START WITH 1
INCREMENT BY 1;
```

Use it like:

```
INSERT INTO employees (id, name) VALUES (emp_seq.NEXTVAL, 'Vaibhav');
```

💡 Example:

Adding employee IDs automatically:

ID: 1 → Vaibhav

ID: 2 → Rahul

📜 Output:

A new ID is created automatically each time you insert.

🧠 Real-Life Scenario:

- Auto-generating Order IDs
- Assigning unique Order Numbers
- Generating Invoice IDs
- Tracking user registrations

📝 Important Notes:

- Use CURRVAL only after NEXTVAL has been used in session.
- Use NEXTVAL to get the next number.
- Use CURRVAL to get the current value used in the session.
- Sequences can also jump values (due to caching).

🔷 SYNONYMS

- Alias for a database object (table, view, sequence, procedure).
- Makes it easier to refer to an object without needing the schema name.
- A synonym is like a nickname or shortcut name for a table, view, or object.
- It makes code shorter and helps you avoid writing full object names.

🔸 Why do we need it?

- To simplify code when table names are long
- To access objects from other schemas easily

💻 Syntax:

```
CREATE SYNONYM emp FOR hr.employees;
```

Now you can just:

```
SELECT * FROM emp;
```

💡 Example:

Instead of:

```
SELECT * FROM hr_department_data_view_2025;
```

Use:

```
SELECT * FROM dept_data;
```

📜 Output:

Same data, shorter code.

🧠 Real-Life Scenario:

- Shortening access to cross-schema tables
- Developers access tables from other users
- Simplify code for views or stored procedures
- Centralize access to common database objects

📝 Important Notes:

- Can be public or private.
- You can create public or private synonyms
- Synonyms don't store data — they just refer to other objects

🔸 Alias

- Temporary name given to a column or table in a SQL query.
- Exists only during that query execution.

🔸 Synonym

- A permanent alternate name for a database object (table, view, sequence, etc.).
- Stored in the database dictionary.

🔷 OBJECT TYPES (User-Defined Types)

- They let you create your own custom data type — a combination of variables (attributes) and actions (methods).
- Think of it like defining your own "employee" object with name, ID, and a display method.
- Used to define custom complex data structures in the database.
- Enables OOP-style programming (Encapsulation, Methods).

🔸 Why do we need it?

- To model real-world entities like Employee, Customer, Product.
- Useful in object-oriented PL/SQL programs.

💻 Syntax:

```
CREATE OR REPLACE TYPE emp_type AS OBJECT (
  emp_id   NUMBER,
  emp_name VARCHAR2(100),
  MEMBER PROCEDURE show_info
);
```

Then create a body:

```
CREATE OR REPLACE TYPE BODY emp_type AS
  MEMBER PROCEDURE show_info IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('ID: ' || emp_id || ', Name: ' || emp_name);
  END;
END;
```

💡 Example:

Creating an object for an employee and showing info:

```
DECLARE
  e emp_type := emp_type(101, 'Vaibhav');
BEGIN
  e.show_info;
END;
```

📄 Output:

ID: 101, Name: Vaibhav

🧠 Real-Life Scenario:

- Passing complex data to/from procedures
- Represent customers, products, vehicles with properties
- Use object tables for storing structured data
- Enable reuse and better design of complex systems

📝 Important Notes:

- Can be used in tables or collections.
- Object types are used in Object-Relational databases
- Can include methods like in OOP (Object-Oriented Programming)

🏵 PERFORMANCE TUNING (PL/SQL & SQL)

- It is the process of improving the speed and efficiency of your PL/SQL or SQL queries and procedures.
- Helps in reducing load time, improving user experience, and avoiding server overload.
- Process of optimizing database operations to ensure maximum performance and efficiency.
- Involves identifying slow SQLs, optimizing queries, reducing context switches, etc.

🔸 Why do we need it?

- Slow queries affect app performance.
- Helps in reducing cost and improving scalability.

🔹 Common Tuning Areas:

- SQL Query Tuning:
  - Use EXPLAIN PLAN to analyze slow queries
  - Create proper INDEXES
  - Avoid unnecessary DISTINCT, GROUP BY, ORDER BY
- PL/SQL Tuning:
  - Use BULK COLLECT, FORALL to handle large data
  - Avoid unnecessary context switches
  - Use PRAGMA UDF for faster functions in SQL

🔹 Key Techniques:

- Use BULK COLLECT / FORALL
- Use INDEXES wisely
- Avoid unnecessary loops
- Use proper joins and WHERE clauses
- Analyze execution plans

💻 Example of Tuning:

Before:

```
FOR i IN (SELECT * FROM employees) LOOP
  -- process row
END LOOP;
```

After:

```
SELECT * BULK COLLECT INTO emp_list FROM employees;
FOR i IN emp_list.FIRST .. emp_list.LAST LOOP
  -- process row
END LOOP;
```

🧠 Real-Life Scenario:

- Optimizing ETL jobs, monthly reports
- You notice a report taking 10 minutes — tuning brings it to 10 seconds
- A slow procedure during month-end slows everything — tuning improves performance

📝 Important Notes:

- Use EXPLAIN PLAN, DBMS_PROFILER, and AWR reports
- Avoid implicit cursors inside loops
- Always test after tuning
- Use tools like SQL Developer, Toad, or AWR Reports
- Indexes, proper joins, and good logic are the key

🔁 SUMMARY:

Topic	Use Case	Tool/Package
UTL_FILE	File reading/writing	UTL_FILE
DBMS_SCHEDULER	Job scheduling with advanced features	DBMS_SCHEDULER
DBMS_JOB	Legacy job scheduling	DBMS_JOB
AUTONOMOUS TRANSACTION	Independent logging or commits	PRAGMA + COMMIT
SEQUENCES	Unique key generation	Sequence object
SYNONYMS	Object aliasing	CREATE SYNONYM
OBJECT TYPES	OOP style custom types	CREATE TYPE
PERFORMANCE TUNING	Improving overall DB performance	EXPLAIN PLAN, AWR, etc.

🔷 Debugging with Messages

🏵 DBMS_OUTPUT – Basic Debugging with Messages

◇ Debugging in PL/SQL

- Debugging helps you find and fix issues in your PL/SQL code (procedures, functions, triggers, etc.). It's like investigating what's going wrong step-by-step.

🔹 What Exactly Is It?

- DBMS_OUTPUT is a built-in Oracle debugging utility.
- It lets you print out messages, values of variables, or processing checkpoints to the output screen.
- Very useful during development to check how your code behaves at each step

🔸 IMP POINT: It doesn't interrupt your code like errors. It silently shows messages — you must enable it in tools like SQL Developer or SQL*Plus.

💻 Syntax:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('This is a debug message');
END;
```

You can also display values:

```
DECLARE
  salary NUMBER := 40000;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Salary is: ' || salary);
END;
```

💡 Example:

Let's debug a bonus calculation:

```
DECLARE
  emp_id NUMBER := 101;
  bonus  NUMBER;
BEGIN
  bonus := 2000;
  DBMS_OUTPUT.PUT_LINE('Bonus assigned to emp ' || emp_id || ' is ' || bonus);
END;
```

📜 Output:

```
Bonus assigned to emp 101 is 2000
```

🧠 Real-Life Scenario:

- You're calculating discounts for customers and you want to track the applied discount
- You're troubleshooting a PL/SQL trigger and want to see which condition passed or failed

📝 Important Notes:

- Always use SET SERVEROUTPUT ON in SQL*Plus or enable it in SQL Developer
- Doesn't work well in live systems — not visible to end users
- Good for development/testing only

🏵 SQL Developer Debugger – Advanced, Visual Debugging

🔹 What Is It?

- A graphical debugger tool inside Oracle SQL Developer
- Helps you pause and inspect PL/SQL code line-by-line
- You can view variable values, change them, and simulate logic flows

🔹 Why Is It Powerful?

- Unlike DBMS_OUTPUT, it doesn't just show text.
- You can:
  - Set breakpoints
  - Watch variables as they change
  - Step into loops, conditionals, nested procedures

✔ Steps to Use:

1. Open your PL/SQL procedure
2. Right-click > Debug
3. Add breakpoints (click left of line numbers)
4. Run the debugger
5. Use tabs like "Watches", "Stack", "Variables" to observe live changes

💡 Example:

Let's debug a procedure:

```
CREATE OR REPLACE PROCEDURE calc_bonus(emp_id NUMBER) IS
  salary NUMBER := 50000;
  bonus  NUMBER;
BEGIN
  IF salary > 30000 THEN
    bonus := salary * 0.1;
  ELSE
    bonus := salary * 0.05;
  END IF;
  DBMS_OUTPUT.PUT_LINE('Bonus: ' || bonus);
END;
```

➡ You set breakpoints on IF and DBMS_OUTPUT lines

➡ You run the debugger and watch how bonus is calculated

🧠 Real-Life Scenario:

- You get a bug in salary update procedure, but no errors shown
- You debug and realize the bonus line never runs because salary logic was skipped

📝 Important Notes:

- You may need DEBUG CONNECT SESSION privilege
- Best used for testing before deployment
- Not usable in live systems — only for developers

Logging (Custom Debug Logs) – Production-Level Debugging

What Is Logging?

- Instead of printing on screen, you insert debug messages into a custom table.
- These messages can be retrieved later and help in understanding what went wrong.

Why Use Logging?

- DBMS_OUTPUT doesn't work in production
- Logging keeps permanent records
- You can include timestamps, error messages, usernames, inputs

Syntax:

Step 1. – Create a table:

```
CREATE TABLE debug_log (
  log_time TIMESTAMP,
  module_name VARCHAR2(100),
  message     VARCHAR2(4000)
);
```

Step 2. – Insert logs:

```
INSERT INTO debug_log VALUES (SYSTIMESTAMP, 'Add_Employee', 'Insert started');
```

Step 3. – Review logs later:

```
SELECT * FROM debug_log WHERE module_name = 'Add_Employee';
```

Example:

```
DECLARE
  emp_name VARCHAR2(100) := 'Vaibhav';
BEGIN
  INSERT INTO debug_log VALUES (SYSTIMESTAMP, 'Emp_Insert', 'Started insert for ' || emp_name);
  INSERT INTO employees(name) VALUES (emp_name);
  INSERT INTO debug_log VALUES (SYSTIMESTAMP, 'Emp_Insert', 'Insert completed');
END;
```

Output:

log_time	module_name	message
08-JUN-2025 10:12AM	Emp_Insert	Started insert for Vaibhav
08-JUN-2025 10:12AM	Emp_Insert	Insert completed

Important Notes:

- Add indexes to log tables for performance
- Archive or delete old logs to save space
- Can combine with EXCEPTION WHEN OTHERS THEN block for error logs

🧠 Real-Life Scenario:

- You want to track why data is failing during night batch jobs
- Logs tell you exactly where failure happened

📝 Important Notes:

- Use DBMS_OUTPUT when you're developing or practicing
- Use the Debugger for professional-level tracing and fixes
- Use Logging when code is live or in UAT/prod, and you can't see the output directly

🔁 Summary Table:

Technique	What It Does	When To Use
DBMS_OUTPUT	Prints text to screen	Simple testing & variable checks
SQL Debugger	Step-by-step debugging	Full control over execution flow
Logging	Saves messages to a table	For production-level, trackable debugging

◇ 1. EXCEPTION HANDLING (Error Trapping)

- Exception handling is a key PL/SQL feature that helps manage runtime errors in a controlled way, allowing the program to handle errors gracefully without abrupt termination.
- This ensures smoother user experience and better program stability.

🔹 You can trap specific exceptions like:

- NO_DATA_FOUND
- TOO_MANY_ROWS
- ZERO_DIVIDE
- or a generic one like WHEN OTHERS THEN

💻 Syntax:

```
BEGIN
  -- Code that might raise exception
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No record found.');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;
```

💡 Example:

Handling a division by zero error using exceptions.

📜 Output:

Displays a user-friendly message instead of abrupt termination.

🧠 Real-Life Scenario:

- When a product price is divided by quantity and quantity is zero.

📝 Important Notes:

- Use specific exceptions when possible; use WHEN OTHERS carefully.

◇ 2. SQLERRM & SQLCODE (Error Message & Code)

- These are built-in functions used inside exception blocks to retrieve the error message and error number respectively.

💻 Syntax:

```
BEGIN
  -- Risky operation
        EXCEPTION
          WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('Error Code: ' || SQLCODE);
            DBMS_OUTPUT.PUT_LINE('Error Message: ' || SQLERRM);
        END;
```

🧠 Real-Life Scenario:

- Logging these values in an audit/error log table.

📝 Important Notes:

- SQLERRM gives detailed error message
- SQLCODE returns negative value for error

◇ 3. AUTONOMOUS TRANSACTIONS – For Independent Logging

- Allows you to perform independent DML operations like logging errors or messages, even if the main transaction fails or rolls back.

💻 Syntax:

```
        CREATE OR REPLACE PROCEDURE log_error(p_msg VARCHAR2) IS
          PRAGMA AUTONOMOUS_TRANSACTION;
        BEGIN
          INSERT INTO error_log (log_time, message) VALUES (SYSDATE, p_msg);
          COMMIT;
        END;
```

💡 Example:

Use this inside your EXCEPTION block to make sure logs are saved even if the main transaction is rolled back.

📜 Output:

Data gets inserted into the error_log table even if outer block fails.

🧠 Real-Life Scenario:

- Logging failed payment transactions.

📝 Important Notes:

- Requires an explicit COMMIT
- Can't read or write from parent transaction's context

◇ 4. TRACING with DBMS_TRACE

- Oracle-supplied package that traces line-by-line execution of PL/SQL code to help you detect the exact flow and where the issue happens.

💻 Syntax:

```
EXEC DBMS_TRACE.SET_PLSQL_TRACE(DBMS_TRACE.TRACE_ALL_CALLS);
```

🧠 Real-Life Scenario:

- Debugging a complex stored procedure to track flow and logic.

📝 Important Notes:

- Needs configuration
- Mostly used in dev/test environments

◇ 5. DBMS_DEBUG / DBMS_DEBUG_JDWP

- Packages that allow debugging of stored procedures from remote tools (e.g., via SQL Developer or other IDEs).

💡 Example:

Setting breakpoints, stepping through code, inspecting variables.

🧠 Real-Life Scenario:

- Used by teams where developers debug code from IDEs without direct database console access.

📝 Important Notes:

- Requires extra setup
- More relevant in team and enterprise environments

◇ 6. Logging to Files using UTL_FILE

- Instead of logging to tables, logs can be written to text files using the UTL_FILE package.

💻 Syntax:

```
DECLARE
  file_handler UTL_FILE.FILE_TYPE;
BEGIN
  file_handler := UTL_FILE.FOPEN('LOG_DIR', 'debug.log', 'A');
  UTL_FILE.PUT_LINE(file_handler, 'Error at step 1');
  UTL_FILE.FCLOSE(file_handler);
END;
```

🧠 Real-Life Scenario:

- Used in scheduled jobs or ETL processes to write logs for each run.

📝 Important Notes:

- Directory must be created and granted to the user
- Handle file I/O exceptions

◇ 7. Custom Logging Procedures

- Creating a reusable procedure like log_step() to insert logs with procedure name, step name, and timestamp.

💻 Syntax:

```
PROCEDURE log_step(p_proc VARCHAR2, p_step VARCHAR2, p_msg
VARCHAR2) IS
```

```
BEGIN
  INSERT INTO app_log VALUES (SYSDATE, p_proc, p_step, p_msg);
  COMMIT;
END;
```

🧠 Real-Life Scenario:
- Used across all procedures to maintain standard logs.

📝 Important Notes:
- Helps in auditing and debugging
- Pair with autonomous transactions for safety

◇ 8. SHOW ERRORS Command
- Command to view compilation errors of procedures, packages, functions, etc.

💻 Syntax:
```
SHOW ERRORS PROCEDURE my_procedure;
```

🧠 Real-Life Scenario:
- Fixing compile-time errors in procedures/packages quickly.

📝 Important Notes:
- Available in SQL*Plus, SQL Developer, and other tools.

◇ 9. DBMS_PROFILER – Code Performance Analyzer
- Helps to measure how much time each line of PL/SQL code takes to execute.

🧠 Real-Life Scenario:
- Useful for identifying slow-performing queries or blocks in procedures.

📝 Important Notes:
- Used more in the optimization phase than debugging, but helps identify performance issues.

◇ 10. PLSQL_WARNINGS
- Enabling this setting during the session allows you to see compile-time warnings.

💻 Syntax:
```
ALTER SESSION SET PLSQL_WARNINGS = 'ENABLE:ALL';
```

🧠 Real-Life Scenario:
- Detecting dead code, unhandled exceptions, or performance risks at compile-time.

📝 Important Notes:
- Helps to proactively catch bugs.

🔁 Summary:
- PL/SQL provides a wide range of debugging and error-tracking tools, from simple output to full-on trace and profiling systems.
- As you build more complex applications, combining these techniques gives better insight into what your code is doing and why something went wrong.

🔷 PL/SQL Real-Time Applications

◇ 1. Reusable Packages

🔹 What Are Packages?

- Packages in PL/SQL are collections of related procedures, functions, variables, and cursors grouped together in two parts:
  - Specification: Declaration (just names, no logic)
  - Body: Definition (actual logic inside)
- They help in modular programming, so you don't need to write the same code again and again. Also, it hides private logic from users and improves performance by loading once into memory.

💡 Example:

```
-- PACKAGE SPEC
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE insert_emp(p_id NUMBER, p_name VARCHAR2);
  FUNCTION get_emp_count RETURN NUMBER;
END emp_pkg;
/
-- PACKAGE BODY
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  PROCEDURE insert_emp(p_id NUMBER, p_name VARCHAR2) IS
  BEGIN
    INSERT INTO employees VALUES(p_id, p_name);
  END;

  FUNCTION get_emp_count RETURN NUMBER IS
    v_count NUMBER;
  BEGIN
    SELECT COUNT(*) INTO v_count FROM employees;
    RETURN v_count;
  END;
END emp_pkg;
/
```

🔹 How to Use:

```
BEGIN
  emp_pkg.insert_emp(101, 'Vaibhav');
  DBMS_OUTPUT.PUT_LINE('Employee Count: ' || emp_pkg.get_emp_count);
END;
```

📃 Output:

```
Employee Count: 11
```

🧠 Real-Life Scenario:

- Imagine you're building an HR software where employee insert and count functions are used in many places. Instead of repeating the same logic, you write it once in a package and reuse it across the system.

◇ 2. PL/SQL Jobs using DBMS_SCHEDULER

🔹 What Are They?

- Jobs are background tasks that run automatically at a fixed time — just like setting an alarm.
- These are helpful for doing tasks like taking backup, sending emails, generating reports without human action.
- In PL/SQL, we use DBMS_SCHEDULER to schedule these jobs.

💡 Example:

```
BEGIN
  DBMS_SCHEDULER.create_job (
    job_name        => 'daily_report_job',
    job_type        => 'PLSQL_BLOCK',
    job_action      => 'BEGIN generate_report(); END;',
    start_date      => SYSTIMESTAMP,
    repeat_interval => 'FREQ=DAILY; BYHOUR=23;',
    enabled         => TRUE
  );
END;
/
```

🧠 Real-Life Scenario:

- Let's say you have an online shopping app. You want to send a sales report to your team every night at 11 PM. You write a report procedure and schedule it using DBMS_SCHEDULER.

◇ 3. Backend Processes (with Triggers or Procedures)

🔹 What Are Backend Processes?

- Backend processes are tasks that happen behind the scenes when users interact with your application.
- They run automatically based on some event, like data insert/update.
- Example: Stock update after placing an order.

🔹 They are written using triggers or procedures.

💡 Example: Trigger to update stock

```
CREATE OR REPLACE TRIGGER trg_stock_update
AFTER INSERT ON orders
FOR EACH ROW
BEGIN
  UPDATE stock
  SET quantity = quantity - :NEW.qty
  WHERE product_id = :NEW.product_id;

  INSERT INTO order_log(order_id, status)
  VALUES(:NEW.order_id, 'Processed');
END;
/
```

🧠 Real-Life Scenario:

- In an e-commerce site like Flipkart, when someone buys a product, the stock gets updated automatically in the database. This is done using backend logic like this trigger.

◇ 4. Complex Data Logic in PL/SQL

🔹 What Is Complex Logic?

- PL/SQL allows writing decision-making or business rules using IF-ELSE, CASE, loops, etc.
- You can use it to calculate tax, discount, commission, salary, EMI, etc.

💡 Example: Tax Calculation Function

```
CREATE OR REPLACE FUNCTION calculate_tax(salary NUMBER) RETURN
NUMBER IS
  tax NUMBER;
BEGIN
  IF salary <= 250000 THEN
    tax := 0;
  ELSIF salary <= 500000 THEN
    tax := salary * 0.05;
  ELSE
    tax := salary * 0.1;
  END IF;
  RETURN tax;
END;
/
```

🔸 Usage:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Tax: ' || calculate_tax(300000));
END;
```

📜 Output:

```
Tax: 15000
```

🧠 Real-Life Scenario:

- In a payroll system, this function will be used for every employee to calculate the tax based on their salary automatically before generating payslips.

◇ 5. Real-Time Case Study: Insurance Premium Calculation

- Let's build a small PL/SQL project example for Insurance Domain.
- Objective: Calculate premium based on vehicle age and risk type (HIGH/LOW) and store it in a table.

💡 Example:

Tables:

```
CREATE TABLE vehicles (
  id NUMBER,
  owner_name VARCHAR2(100),
  vehicle_age NUMBER,
  risk_category VARCHAR2(10)
);

CREATE TABLE premiums (
  id NUMBER,
  vehicle_id NUMBER,
  premium_amount NUMBER
);
```

💡 Example:

Procedure:

```
CREATE OR REPLACE PROCEDURE calculate_premium IS
  CURSOR c1 IS SELECT id, vehicle_age, risk_category FROM vehicles;
  premium NUMBER;
BEGIN
  FOR rec IN c1 LOOP
    IF rec.vehicle_age < 3 THEN
      premium := 5000;
    ELSIF rec.vehicle_age BETWEEN 3 AND 10 THEN
      premium := 7000;
    ELSE
      premium := 10000;
    END IF;

    IF rec.risk_category = 'HIGH' THEN
      premium := premium + 3000;
    END IF;

    INSERT INTO premiums VALUES(premiums_seq.NEXTVAL, rec.id, premium);
  END LOOP;
END;
/
```

🧠 Real-Life Scenario:

- This logic can be used in a real insurance system like PolicyBazaar or HDFC Ergo, where customers' vehicle details are entered and premiums are calculated dynamically.

📝 Important Notes:

- PL/SQL is not just about writing queries — it's about building real business logic.
- Reusable packages save time and avoid duplicate code.
- Scheduler jobs help automate daily tasks like backups or reports.
- Backend logic keeps the system intelligent and responsive.
- Case studies help combine everything you've learned into a working system.

🔁 Summary:

Concept	Description	Real-World Usage
Packages	Group functions/procedures together	Used across modules (reusable code)
Scheduler Jobs	Auto run tasks like reports	Nightly backups, auto emails
Backend Processes	Auto run when data changes	Stock update, logging
Complex Logic	Decision-making logic	Tax, salary, discounts
Project Case Study	End-to-end PL/SQL usage	Insurance premium calc system

⭐ Additional Concepts You Might Encounter (already included indirectly):

🔷 Data Modeling Concepts:

🔹 What is Data Modeling?

- Data Modeling is the process of designing how data will be stored, related, and accessed in a database.
- It's like creating a blueprint of your database.
- In Simple Words:
  - Just like a civil engineer draws a map of a house, a data modeler draws a map of how tables, relationships, and data should be designed in a system.

🎯 Purpose:

- Define structure of data (tables, columns, relationships)
- Remove redundancy
- Improve performance
- Help business understand data flow

📐 Types of Data Models:

Type	Description
Conceptual	High-level model, for business understanding (ER Diagram)
Logical	More detailed, defines attributes, primary/foreign keys
Physical	Actual database structure with data types, constraints, etc.

🔹 Normalization

Normalization is the process of organizing data in a database to:

- Reduce data redundancy
- Ensure data integrity
- Break large tables into smaller related tables

🔹 Why Normalize?

📊 Let's say you have this table:

EMP_ID	EMP_NAME	DEPT_NAME	DEPT_LOCATION
101	Raj	HR	Mumbai
102	Neha	HR	Mumbai
103	Amit	IT	Pune

Here, "HR" and "Mumbai" are repeated — this causes data redundancy.

🔹 Types (Forms) of Normalization:

1. 1NF (First Normal Form)
2. 2NF (Second Normal Form)
3. 3NF (Third Normal Form)
4. BCNF (Boyce-Codd Normal Form)

🔸 1NF (First Normal Form)

Rules:

- Atomic values only (no multivalued columns)
- Unique column names
- Rows are uniquely identified

📄 Example:

EMP_ID	EMP_NAME	PHONE_NUMBERS
101	Raj	9999999999,88888888

🔴 This is not 1NF. It contains multi-valued fields.

🔵 Convert to 1NF:

EMP_ID	EMP_NAME	PHONE_NUMBER
101	Raj	9999999999
101	Raj	8888888888

🔸 2NF (Second Normal Form)

Rules:

- Should be in 1NF
- No partial dependency (applies to composite keys)

💡 Example:

EMP_ID	PROJECT_ID	EMP_NAME	PROJECT_NAME

Here, EMP_NAME depends only on EMP_ID, not full key (EMP_ID + PROJECT_ID) — so break it:

🔹 Convert to 2NF:

- EMPLOYEE Table → (EMP_ID, EMP_NAME)
- PROJECT Table → (PROJECT_ID, PROJECT_NAME)
- EMP_PROJECT → (EMP_ID, PROJECT_ID)

🔸 3NF (Third Normal Form)
Rules:

- Should be in 2NF
- No transitive dependency (non-key column depending on another non-key)

💡 Example:

```
EMP_ID          EMP_NAME        DEPT_ID         DEPT_NAME
```

DEPT_NAME depends on DEPT_ID, not directly on EMP_ID.

🔹 Convert to:

- EMPLOYEE → (EMP_ID, EMP_NAME, DEPT_ID)
- DEPARTMENT → (DEPT_ID, DEPT_NAME)

🔸 BCNF (Boyce-Codd Normal Form)

- Advanced version of 3NF
- Every determinant must be a candidate key

Use when tables have more than one candidate key and 3NF still has redundancy

🔁 Summary Table for Normalization

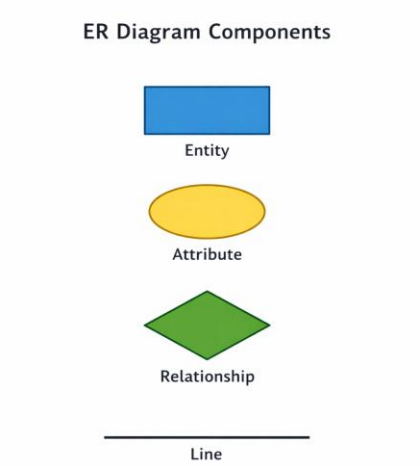
Form	Goal	Rule	Example Fix
1NF	Atomic Columns	No multi-values	Split multi-phone rows
2NF	Remove Partial Dependency	Full primary key dependency	Break composite key tables
3NF	Remove Transitive Dep.	Non-key depending on other non-key	Move dept to new table
BCNF	Candidate Key Clarity	Every determinant is a key	Handle unusual key structures

ER Diagrams (Entity-Relationship Diagrams)

What is an ER Diagram?

- An ER (Entity-Relationship) Diagram is a graphical representation of:
  - Entities (tables)
  - Attributes (columns)
  - Relationships (keys & foreign keys)

It shows how tables are related in a database.



Basic Components:

Symbol	Meaning
Rectangle	Entity (like TABLE)
Ellipse	Attribute (like COLUMN)
Diamond	Relationship (like "works in")
Line	Connects entity to relationship

Example Case: Employee–Department

```
[EMPLOYEE] --------- (Works_In) -------- [DEPARTMENT]
|                                |
|-- EMP_ID (PK)                  |        -- DEPT_ID (PK)
|-- NAME                         |        -- DEPT_NAME
|-- DEPT_ID (FK)                 |        -- LOCATION
```

This ER diagram represents:

- Employee belongs to a department
- DEPT_ID in EMPLOYEE is a foreign key to DEPARTMENT

Cardinality:

- One to One (1:1) – e.g., each employee has one laptop
- One to Many (1:N) – e.g., one department has many employees
- Many to Many (M:N) – e.g., students and courses

Primary Key, Foreign Key in ER

- Primary Key (PK) – Uniquely identifies a record (like EMP_ID)
- Foreign Key (FK) – Points to a PK in another table (like DEPT_ID in employee)

📌 Use arrows or FK notation in ER diagrams to represent links

🔷 Real-Time Scenario of Data Modeling

🔹 Suppose you're building a School Management System:

- Entities:
  - Student
  - Course
  - Teacher
  - Department
- Relationships:
  - A student enrolls in many courses (M:N)
  - A teacher teaches many courses (1:N)
  - Each course belongs to one department (1:N)
- So you design:
  - STUDENT (SID, Name, Age)
  - COURSE (CID, Title, Dept_ID)
  - TEACHER (TID, Name)
  - DEPARTMENT (Dept_ID, Dept_Name)
  - ENROLLMENT (SID, CID) → handles many-to-many

🔁 Summary

Concept	Description
Data Modeling	Blueprint of your database structure
Normalization	Breaks data into smaller tables to reduce redundancy
1NF	Remove multivalued columns
2NF	Remove partial dependency
3NF	Remove transitive dependency
ER Diagram	Visual map of tables and relationships
PK & FK	Connect tables with unique references

🔷 Performance Tuning in PL/SQL and SQL

- Learn how to make your queries and PL/SQL code run faster, avoid unnecessary processing, and optimize database usage
- This is very important in real-world projects like banking, insurance, and ecommerce apps.

🔶 What is Performance Tuning?

- Performance tuning is the process of making your SQL queries or PL/SQL code work faster and use fewer resources like CPU, memory, and disk I/O.
- A poorly written query can take minutes or hours, while an optimized query does the same work in seconds.
- In real-time applications, slow queries affect customers, dashboards, reports, and even crash systems.
- In Simple Words:
    - Performance tuning means making your SQL/PLSQL code faster and lighter.
    - Just like a car needs tuning for better mileage, your database queries need tuning to avoid slow performance.
    - It saves time, cost, and improves user experience in real-time systems like mobile apps, dashboards, or financial systems.

🔸 Major Areas of PL/SQL and SQL Performance Tuning

Area	Description
Indexes	Help speed up WHERE conditions
Execution Plan	Shows how Oracle executes your query
Bulk Processing	Avoids row-by-row (slow) operations
Bind Variables	Prevent parsing and reusing queries
Avoiding Cursors or Loops when possible	Use set-based logic
Limiting SELECT	Avoid SELECT * and unnecessary joins
Proper Exception Handling	Prevent errors and improve control

◇ 1. Use Indexes Properly

🔹 What is an Index?

- It’s like an address book. If you need to search for Vaibhav in a phonebook, it's quicker to jump to "V" than checking every page.

💻 Syntax to Create Index:

```
CREATE INDEX idx_emp_name ON employees(emp_name);
```

💡 Example:

```
-- Before index
SELECT * FROM employees WHERE emp_name = 'Vaibhav';
-- After the index is created on emp_name, it will be much faster.
```

📌 Real-time Use: Search features in apps (e.g., employee lookup, order status check) use indexes for fast results.

◇ 2. Check and Understand Execution Plan

🔹 What is Execution Plan?

- It shows how Oracle executes your SQL — whether it uses full table scan, index scan, nested loops etc.

🔹 How to check:

```
EXPLAIN PLAN FOR
SELECT * FROM employees WHERE emp_id = 100;

SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
```

📌 Use this in performance tuning to identify bottlenecks.

◇ 3. Avoid Using Loops for Row-by-Row Processing

🔹 Problem:

```
-- BAD PRACTICE
FOR rec IN (SELECT * FROM employees) LOOP
  UPDATE employees SET salary = salary * 1.1 WHERE emp_id = rec.emp_id;
END LOOP;
```

This is very slow when thousands of records are involved.

🔹 Better Option (SET-BASED Operation):

```
-- GOOD PRACTICE
UPDATE employees SET salary = salary * 1.1;
```

📌 Real-time: Use this in salary hikes, tax updates, mass record changes.

◇ 4. Use BULK COLLECT and FORALL

🔹 Problem with Normal Loop:

```
FOR rec IN (SELECT * FROM emp) LOOP
  INSERT INTO emp_log VALUES (rec.emp_id, rec.name);
END LOOP;
```

It inserts one row at a time — very slow.

🔹 Optimized Version:

```
DECLARE
  TYPE emp_table IS TABLE OF emp%ROWTYPE;
  l_data emp_table;
BEGIN
  SELECT * BULK COLLECT INTO l_data FROM emp;

  FORALL i IN 1 .. l_data.COUNT
    INSERT INTO emp_log VALUES (l_data(i).emp_id, l_data(i).name);
END;
```

📌 This is 100x faster for inserting/updating/deleting large records.

◇ 5. Use Bind Variables

🔹 Problem:

```
-- BAD: Hard-coded value forces query parsing every time
SELECT * FROM employees WHERE emp_id = 101;
```

🔹 Better:

```
-- GOOD: Bind variable (used in apps or PL/SQL blocks)
SELECT * FROM employees WHERE emp_id = :emp_id;
```

📌 Saves parsing time and memory in apps like Oracle Forms, APEX, Java apps.

◇ 6. Avoid SELECT * — Always Use Required Columns

🔹 Problem:

```
SELECT * FROM employees;
```

This fetches unnecessary columns, wastes time and memory.

🔹 Better:

```
SELECT emp_id, emp_name FROM employees;
```

📌 Especially useful in reports, dashboards where speed matters.

◇ 7. Avoid Too Many Joins/Subqueries

🔹 Problem:

```
SELECT * FROM emp WHERE dept_id IN (SELECT dept_id FROM dept WHERE
location = 'Pune');
```

🔹 Better (using JOIN):

```
SELECT e.emp_id, e.name
FROM emp e
JOIN dept d ON e.dept_id = d.dept_id
WHERE d.location = 'Pune';
```

📌 Joins are faster and better optimized by the Oracle optimizer.

◇ 8. Analyze and Gather Statistics

- Oracle uses table statistics to decide execution plans.

```
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS('HR', 'EMPLOYEES');
END;
```

📌 Use it after major updates/inserts to keep queries fast.

◇ 9. Use Parallel Query for Large Data

```
SELECT /*+ parallel(4) */ * FROM big_sales_data;
```

📌 Divides the work into 4 threads, speeds up large data processing.

🧠 Real-World Scenario: Tuning a Report Query

🔹 Problem:

Monthly report query is taking 10 minutes. You observe:

- It uses SELECT *
- It scans full 10 million row table
- Joins are written with subqueries

🔹 Fix:

- Add index on filtering column (e.g. sale_date)
- Replace SELECT * with only needed columns
- Convert subqueries to JOINs
- Gather stats after index creation

🔁 Summary: Performance Tuning

Technique	What it does	When to use
Indexes	Speeds up search/filter	WHERE clause on big table
Execution Plan	Shows how query runs	Analyze slow queries
BULK COLLECT/FORALL	Fast batch processing	INSERT/UPDATE many rows
Bind Variables	Avoid re-parsing	Dynamic SQL in apps
Avoid SELECT*	Fetch only needed data	Reports, dashboards
Set-based Logic	Avoid row-by-row	Mass updates or reports
Join Optimization	Better than subqueries	Large table joins
Stats Gathering	Helps optimizer	After data changes
Parallel Query	Speeds up large queries	Millions of rows

🔷 PL/SQL Metadata Tables

- Metadata tables (also called data dictionary views) are system-generated views in Oracle that help you explore the structure and status of your own objects — like tables, columns, procedures, triggers, views, etc.
- Think of it like:

  📘 *"Instead of opening SQL Developer and right-clicking on a table, just run a query on these views and get all the hidden details in seconds!"*

◇ 1. USER_TABLES

🔹 Purpose:

- Shows the list of all tables owned by the current user.
- It doesn't show columns, only table-level details.

🔹 Key Columns:

- TABLE_NAME: Name of your table
- TABLESPACE_NAME: Where the table is stored
- NUM_ROWS: Number of rows (based on statistics)
- LAST_ANALYZED: When stats were last collected

💡 Example:

```
SELECT table_name, tablespace_name, num_rows, last_analyzed
FROM USER_TABLES;
```

📜 Output:

TABLE_NAME	TABLESPACE_NAME	NUM_ROWS	LAST_ANALYZED
EMPLOYEES	USERS	2000	17-JUN-2025 10:30am
DEPARTMENTS	USERS	12	16-JUN-2025 09:00am

🧠 Real-Time Use Case:

- You're building a report on employee data, and need to confirm if the table is created, where it's stored, and how large it is.

◇ 2. USER_TAB_COLUMNS

🔹 Purpose:

- Shows the structure of your tables — i.e., column names, data types, and nullability.

🔹 Key Columns:

- TABLE_NAME
- COLUMN_NAME
- DATA_TYPE
- DATA_LENGTH
- NULLABLE

💡 Example:

```
SELECT table_name, column_name, data_type, data_length, nullable
FROM USER_TAB_COLUMNS
WHERE table_name = 'EMPLOYEES';
```

📜 Output:

TABLE_NAME	COLUMN_NAME	DATA_TYPE	DATA_LENGTH	NULLABLE
EMPLOYEES	EMP_ID	NUMBER	22	N
EMPLOYEES	NAME	VARCHAR2	50	Y
EMPLOYEES	SALARY	NUMBER	22	Y

🧠 Real-Time Use Case:

- You are writing a PL/SQL procedure to insert data, and want to check which columns are NOT NULL or what data types they are.

◇ 3. ALL_OBJECTS

🔹 Purpose:

- Gives info about all objects (tables, views, procedures, packages, triggers, etc.) that you can access — both yours and those granted by others.

🔹 Key Columns:

- OBJECT_NAME: Name of the object
- OBJECT_TYPE: TABLE / PROCEDURE / PACKAGE / TRIGGER / VIEW
- STATUS: VALID / INVALID
- CREATED, LAST_DDL_TIME: Created or modified date

💡 Example:

```
SELECT object_name, object_type, status, created
FROM ALL_OBJECTS
WHERE object_type IN ('PROCEDURE', 'PACKAGE');
```

📜 Output:

OBJECT_NAME	OBJECT_TYPE	STATUS	CREATED
EMP_PKG	PACKAGE	VALID	16-JUN-2025 10:30
CALC_BONUS	PROCEDURE	INVALID	15-JUN-2025 12:00

🧠 Real-Time Use Case:

- You want to compile only invalid packages, or see when your procedure was last updated.

◇ 4. USER_VIEWS

🔹 Purpose:

- Lists all views created by the current user.
- It even contains the entire SQL code used to build the view.

🔹 Key Columns:

- VIEW_NAME: Name of the view
- TEXT: The actual SQL that defines the view
- TEXT_LENGTH: Length of the view SQL

💡 Example:

```
SELECT view_name, text_length
FROM USER_VIEWS;
```

To view the SQL behind a specific view:

```
SELECT text
FROM USER_VIEWS
WHERE view_name = 'EMP_SAL_VIEW';
```

📜 Output (text):

```
SELECT e.emp_id, e.name, d.dept_name
FROM employees e JOIN departments d ON e.dept_id = d.dept_id;
```

🧠 Real-Time Use Case:

- When debugging or modifying dashboards or reporting views, you want to extract and improve existing view queries.

◇ 5. USER_PROCEDURES & USER_SOURCE

- USER_PROCEDURES: Lists all your PL/SQL procedures, functions, and packages.
- USER_SOURCE: Gives source code (line-by-line) of those objects.

💡 Example 1: Get List of PL/SQL Units

```
SELECT object_name, procedure_name, object_type, status
FROM USER_PROCEDURES;
```

OBJECT_NAME	PROCEDURE_NAME	OBJECT_TYPE	STATUS
EMP_PKG	CALC_SALARY	PACKAGE BODY	VALID

💡 Example 2: View Code of Procedure

```
SELECT line, text
FROM USER_SOURCE
WHERE name = 'EMP_PKG'
ORDER BY line
```

LINE	TEXT
1	PROCEDURE calc_salary IS
2	BEGIN
3	-- logic to calculate salary
4	END;

🧠 Real-Time Use Case:

- You want to download, edit, or version control your procedures and packages, especially before a release or code migration.

◇ 6. USER_TRIGGERS

🔹 Purpose:

- Lists all triggers created by the user, and what events or tables they are associated with.

🔹 Key Columns:

- TRIGGER_NAME
- TABLE_NAME
- TRIGGERING_EVENT (INSERT / UPDATE / DELETE)
- STATUS
- DESCRIPTION

💡 Example:

```
SELECT trigger_name, table_name, triggering_event, status
FROM USER_TRIGGERS;
```

TRIGGER_NAME	TABLE_NAME	TRIGGERING_EVENT	STATUS
TRG_LOG_ORD	ORDERS	INSERT	ENABLED

🧠 Real-Time Use Case:

- You are debugging an audit trigger, or checking why insert is failing — and you want to confirm if any trigger is fired on that table.

◇ 7. USER_SEQUENCES

🔹 Purpose:

- Lists all sequences (auto-increment counters) created by you.

🔹 Key Columns:

- SEQUENCE_NAME
- MIN_VALUE, MAX_VALUE
- INCREMENT_BY
- LAST_NUMBER (next value to be generated)

💡 Example:

```
SELECT sequence_name, last_number, increment_by
FROM USER_SEQUENCES;
```

SEQUENCE_NAME	LAST_NUMBER	INCREMENT_BY
EMP_SEQ	101	1

🧠 Real-Time Use Case:

- You are assigning IDs during insert operations and want to check the current value of the sequence or reset it.

🎯 In Real-Time Projects

- Here's how developers use metadata tables daily:
  - Data Engineers check USER_TAB_COLUMNS before data load
  - PL/SQL developers monitor USER_SOURCE before deployments
  - Report developers fix USER_VIEWS if dashboard fails
  - ETL teams use ALL_OBJECTS to recompile invalid packages

🔁 Summary : Full Comparison

Metadata View	Description	Best Used For
USER_TABLES	Lists all user-created tables	Check if a table exists, row stats
USER_TAB_COLUMNS	Shows all columns of your tables	Profile table design, null checks
ALL_OBJECTS	All objects (tables, packages, etc.)	Check status, created date
USER_VIEWS	Views and their SQL definition	Debug or edit complex views
USER_SOURCE	Full PL/SQL source code (line by line)	Extract code, backup, versioning
USER_PROCEDURES	List of PL/SQL procedures/functions	See if code is compiled or valid
USER_TRIGGERS	Lists all triggers created by user	Debug inserts, updates, deletes
USER_SEQUENCES	Shows all sequences and values	ID generation and tracking

🔷 PL/SQL Code Reusability Best Practices

◇ 1. Use Packages for Grouping Logic

🔹 What is it?

- A package is a collection of related procedures, functions, variables, cursors grouped together under a single name.

🔹 Why?

- Promotes modular programming
- Makes code easier to reuse and maintain
- Supports encapsulation (public vs private procedures)
- Improves performance (stored in memory)

💻 Syntax Example:

```
-- Package Specification
CREATE OR REPLACE PACKAGE emp_pkg IS
  PROCEDURE add_employee(p_id NUMBER, p_name VARCHAR2);
  FUNCTION get_employee(p_id NUMBER) RETURN VARCHAR2;
END emp_pkg;
/

-- Package Body
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
  PROCEDURE add_employee(p_id NUMBER, p_name VARCHAR2) IS
  BEGIN
    INSERT INTO employees (emp_id, emp_name) VALUES (p_id, p_name);
  END;

  FUNCTION get_employee(p_id NUMBER) RETURN VARCHAR2 IS
    v_name VARCHAR2(100);
  BEGIN
    SELECT emp_name INTO v_name FROM employees WHERE emp_id = p_id;
    RETURN v_name;
  END;
END emp_pkg;
/
```

🔹 How to reuse:

```
BEGIN
  emp_pkg.add_employee(101, 'Vaibhav');
  DBMS_OUTPUT.PUT_LINE(emp_pkg.get_employee(101));
END;
```

◇ 2. Use Procedures & Functions Instead of Repeating Code

🔹 When to use:

- Procedure → perform an action (insert, update, delete)
- Function → return a value (like a calculated result)

💡 Example

```
CREATE OR REPLACE FUNCTION calc_bonus (p_salary NUMBER)
RETURN NUMBER IS
BEGIN
```

```
  RETURN p_salary * 0.10;
END;
/
```

🔹 Usage:

SELECT emp_name, calc_bonus(salary) FROM employees;

📌 Instead of writing the bonus logic again and again, use this function in reports or triggers.

◇ 3. Use Constants and Global Variables

🔹 Why?

- Avoids hardcoding values
- Easier to update from one place
- Promotes maintainability

💡 Example

```
CREATE OR REPLACE PACKAGE constants_pkg IS
  bonus_rate CONSTANT NUMBER := 0.10;
END;
```

Use in other procedures:

sal * constants_pkg.bonus_rate;

◇ 4. Use Cursors & Cursor Variables

- Use explicit cursors or cursor FOR loops to reuse select logic, especially when looping through data in reports or batch jobs.

💡 Example

```
CURSOR emp_cur IS
  SELECT emp_id, salary FROM employees;

FOR rec IN emp_cur LOOP
  -- Reuse logic for all employees
END LOOP;
```

◇ 5. Write Generic & Parameterized Code

🔹 Why?

- Helps reuse the same code for different values
- Easy to plug and play in different modules

💡 Example

```
CREATE OR REPLACE PROCEDURE update_salary (p_id NUMBER, p_new_sal
NUMBER) IS
BEGIN
  UPDATE employees SET salary = p_new_sal WHERE emp_id = p_id;
END;
```

Use this in multiple reports, job schedulers, forms, etc.

◇ 6. Use Exception Blocks for Reusability

- Handle errors once and reuse the same standard error format across your app.

💡 Example

```
BEGIN
  -- logic
EXCEPTION
  WHEN OTHERS THEN
    log_error(SQLERRM); -- Reusable error logger
END;
```

Create a reusable error logger procedure:

```
CREATE OR REPLACE PROCEDURE log_error(p_msg VARCHAR2) IS
BEGIN
  INSERT INTO error_log(error_message, log_time)
  VALUES (p_msg, SYSDATE);
END;
```

◇ 7. Document and Comment the Code

- A best practice is to:
  - Add comments explaining inputs/outputs
  - Describe business logic
  - Mention version info or authors
- This makes code easier to reuse and maintain.

💡 Example

```
-- Procedure: update_salary
-- Author: Vaibhav D
-- Purpose: To update salary of employee
```

◇ 8. Avoid Copy-Pasting Logic Everywhere

- Instead of copying insert/update logic in multiple places:
  - Put it inside a procedure/package
  - Call it wherever needed
- This reduces bugs and inconsistencies

◇ 9. Use Modular Approach

- Break large code blocks into:
  - Helper procedures
  - Utility functions
  - Packages with private/public methods
- Modular code = clean, organized, testable

◇ 10. Plan for Future Use

- Always write PL/SQL with future use in mind:
  - Parameterize values
  - Avoid hard-coded table/column names
  - Keep logic separate from presentation

🔁 Summary Table

Best Practice	Benefit
Use Packages	Group logic, improve performance
Write Procedures/Functions	Avoid code repetition
Use Constants	Maintain values from one place
Use Cursors	Iterate reusable queries
Generic Parameters	Increase flexibility
Handle Exceptions Reusably	Central error management
Comment and Document Code	Easier to reuse and debug later
Avoid Hardcoding	Cleaner and scalable code
Modular Design	Easy to test and maintain