SQL (STRUCUTED QUERY LANGUAGE)

Java Full Stack Trainee - Technoglobe IT Training & Placement, Bangalore, Marathahalli

1. What is Data?

A: Data is raw facts and figures collected for reference or analysis. It can be anything meaningful or meaningless that is processed to create useful information.

Example:

- Numbers: 25, 60, 150
- Text: "John", "Product A"

What is a Database (DB)?

A: A Database is an organized collection of data that is stored and managed to ensure easy access, retrieval

2. What is DBMS?

A: DBMS stands for **Database Management System**. It is software used to create, manage, and maintain databases, allowing users to store and retrieve data efficiently.

Example DBMS Software:

- Oracle DBMS
- Microsoft Access
- MySQL (Standalone)

Q: What are some key features of a DBMS?

A:

- Data storage, retrieval, and manipulation.
- Centralized data management.
- Data security and backup.

Advantages of DBMS:

1. **Data Redundancy Control:** Reduces duplicate data by centralizing storage.
2. **Data Integrity:** Ensures accuracy and consistency of data through constraints.
3. **Data Security:** Provides access control and protects sensitive data.
4. **Efficient Data Access:** Uses indexing and optimized query processing for faster data retrieval.
5. **Backup and Recovery:** Allows data to be backed up and restored in case of failure.
6. **Concurrent Access:** Supports multiple users accessing the data simultaneously without conflicts.
7. **Data Independence:** Changes in the data structure don't affect application programs.
8. **Scalability and Flexibility:** Handles large amounts of data and adapts to new requirements easily.

Disadvantages of a DBMS:

1. **Complexity:** Setting up and maintaining a DBMS can be complex, requiring skilled administrators and developers.
2. **Cost:** DBMS software can be expensive, and there are additional costs for hardware, training, and maintenance.

---

3. What is RDBMS?

A: RDBMS stands for **Relational Database Management System**. It is a type of DBMS that organizes data into related tables using rows and

columns. It uses **Structured Query Language (SQL)** for data management.

Example RDBMS Software:

- MySQL (as RDBMS)
- PostgreSQL
- Microsoft SQL Server
- Oracle Database

Q: What is the main difference between DBMS and RDBMS?
A:DBMS: Data is stored as files without relationships between data.

RDBMS: Data is stored in tables with relationships, ensuring data integrity through primary and foreign keys.

History of Data Base:-

Databases began in the 1960s with hierarchical and network models, followed by **E.F. Codd's relational model in the 1970s**, which introduced tables and SQL. Over time, they evolved to include NoSQL, cloud databases, and distributed systems.

Q: What is a Statement?
A: A statement is a collection of programs.

In General we have,

English

- Alphabets
- Words
- Grammar
- Sentence
- Paragraph
- Story

- Characters (Alphabets)
- Keywords (SQL Commands)
- Syntax
- SQL Query (Sentence)

SQL (STRUCUTED QUERY LANGUAGE)

Java Full Stack Trainee - Technoglobe IT Training & Placement, Bangalore, Marathahalli

What is Oracle?

Oracle is a powerful, industry-leading **Relational Database Management System (RDBMS)** that is used to store, retrieve, and manage large amounts of data efficiently. It was developed by Oracle Corporation and is widely used across industries for database management and application development.

Oracle databases are known for their robustness, scalability, and features like transaction processing, data warehousing, and distributed database systems.

Why is Oracle Required?

1. **Data Management:** Oracle helps in storing and organizing data systematically, making it easier to access and manage.
2. **Scalability:** Oracle databases can handle large amounts of data, making them ideal for growing businesses.
3. **High Performance:** With advanced optimization techniques, Oracle ensures fast data processing and query execution.
4. **Security:** Oracle provides advanced security features like data encryption, user access control, and auditing to safeguard data.

Importance of Oracle

1. **Industry Standard:** Oracle is used by organizations worldwide, making it a critical skill for IT professionals.

2. **Supports Big Data and Analytics:** Oracle integrates well with modern technologies like AI, machine learning, and big data for analytics.
3. **Cloud Integration:** Oracle databases can be deployed on the cloud, allowing for flexible and scalable solutions.
4. **Comprehensive Features:** It provides features like data integrity, transaction management, and high availability, which are essential for mission-critical systems.
5. **Job Opportunities:** Knowledge of Oracle opens up opportunities in database administration, development, and analytics.

What is an Oracle Client Application?

The Oracle client application is a software component that allows a computer (client) to connect to an Oracle database hosted on a server. It is essential for executing SQL queries and managing the database.

Key Features of Oracle Client Application:

1. **Database Connectivity:** Provides a secure connection between client machines and the Oracle database.
2. **SQL Query Execution:** Allows users to run SQL commands to retrieve, update, or delete data from the database.
3. **Tools for Developers:** Comes with tools like **SQL*Plus,** Oracle SQL Developer, and PL/SQL Developer for managing databases and developing applications.

How the Oracle Client Works

1. **Install on Local Machine:** The Oracle client application is installed on the user's computer.
2. **Connection to Server:** It connects to the Oracle database server using credentials (username, password, and connection string).
3. **Query Execution:** The user sends SQL queries through the client application, and the results are retrieved from the database server.

Common Tools in the Oracle Ecosystem

SQL*Plus: A command-line tool for executing SQL commands and PL/SQL scripts.

Oracle11g installation:-
Steps:-
1.Open oracle floder
2.open disk2 floder.
3.write -> right click on setup file and select run as administration and select 'yes'.

4.In the first screen 'select next'.
5.In the license page 'select accept the terms' and then select next.

6.In the destination floder 'select next'.

7.In the password page enter password,rememberd that password and select next.

8.In summary window select install.

9.In the setup status window select finish.

SQL (STRUCUTED QUERY LANGUAGE)

HOW TO OPEN :-

Open sql*plus:-

How to open sql*plus

1.go to start programs ,then select oracle folder.

2.select Run SQL commandline.

3.connect as DBA ,DB Administration.

PROCESSOR:-

SQL>CONNECT SYSTEM

ENTER PASSWORD:-XXXXX.

CONNECTED.

What is SQL?

SQL (Structured Query Language) is a standard programming language designed for managing and manipulating relational databases. It is used to perform operations like querying, updating, inserting, and deleting data stored in a database. SQL is essential for interacting with relational database systems such as MySQL, Oracle, Microsoft SQL Server, PostgreSQL, and others.

Why is SQL Required?

SQL is required for several reasons:

1. **Data Management:** It enables users to store, retrieve, and manipulate large volumes of structured data.
2. **User-Friendly:** SQL is easy to learn and use with its simple syntax for complex database tasks.

Advantages of SQL

1. **Ease of Use:** Simple syntax and commands make it easy to learn and use.
2. **Speed:** SQL allows quick retrieval and manipulation of data.
3. **Scalability:** Can handle small to large-scale databases efficiently.

Disadvantages of SQL

1. **Complexity in Advanced Queries:** Writing complex queries with subqueries and joins can be challenging for beginners.
2. **Resource Intensive:** Handling very large datasets may require substantial computing resources.

Basic Rules and Properties of SQL

1. **What is a Query?**
   A query is a request made to the database to perform operations such as retrieving, inserting, updating, or deleting data.
2. **End Every Query with a Semicolon (;):**
   In SQL, each query must end with a semicolon to indicate the end of the statement.
3. **Case Insensitivity:**
   SQL is not case-sensitive for commands, meaning SELECT is the same as select. However, data and identifiers (like table or column names) may be case-sensitive depending on the database.
4. **One Query at a Time:**
   By default, SQL executes one query at a time unless using batch processing.

5. **SQL Uses a Declarative Approach:**
   You specify *what* you want to do with the data, not *how* the database should process it.
6. **Database Engine Executes Queries:**
   Queries are executed by the **database engine**, which interprets and performs the requested operations.
7. **Use of Comments:**
   SQL allows comments for better readability:
   - Single-line comments: -- Comment text
   - Multi-line comments: /* Comment text */

Data Types

What is a Data Type?

A **data type** in SQL defines the type of data that can be stored in variable. It specifies the kind of value, such as numbers, text, dates, or binary data.

Main Types of SQL Data Types

1. **Character Data Types**
   Used to store text data.
   - **Examples**:
     - **CHAR (size): Fixed-length string.**
       Example: CHAR (5) stores "Hello" or "Tech " (always 5 characters, spaces added if shorter).
     - **VARCHAR2(size): Variable-length string.**
       Example: VARCHAR2(10) stores "Hello" or "SQL" (up to 10 characters).

---

2. **Numeric Data Types**
   Used to store numeric values.
   - **Examples**:
     - INT: Integer (whole number).
       Example: 100, -5.
     - DECIMAL(p, s) or NUMERIC(p, s): Fixed-point numbers with precision (p) and scale (s).
       Example: DECIMAL(5, 2) stores 123.45.

---

3. **Date and Time Data Types**
   Used to store dates, times, and timestamps.
   - **Examples**:
     - DATE: Stores only date.
       Example: '2024-12-31'.
     - DATETIME: Stores date and time.
       Example: '2024-12-31 23:59:59'.
     - TIME: Stores only time.
       Example: '14:30:00'.

---

4. **Binary Data Types**
   Used to store binary data like images, files, or binary strings.
   - **Examples**:
     - BINARY(size): Fixed-length binary data.
       Example: A file stored as BINARY(10).

- VARBINARY(size): Variable-length binary data.
  Example: A variable image file with a size limit of VARBINARY(100).

DBMS COMMANDS :-

1. Data Definition Language (DDL)

Used to define and modify the structure of database objects (e.g., tables, schemas).

- **Commands:**

  CREATE,ALTER,DROP,RENAME,TRUNCATE :-

1.CREATE: Creates a new database object (table, schema).

Syntax:- CREATE TABLE tablename (

Column1 datatype(size),
Column2 datatype(size),

);

2.ALTER: Modifies the structure of an existing table.

1.Add a Column:

Syntax:- ALTER TABLE table_name ADD (column_name datatype);

2.Modify a Column:

Syntax:- ALTER TABLE table_name MODIFY (column_name new_datatype);

3. Drop a Column:

Syntax:- ALTER TABLE table_name DROP COLUMN column_name;

3.DROP: Deletes a database object.

Syntax:- DROP TABLE TABLENAME;

4.RENAME:-The RENAME is used to change the name of an existing database object, such as a table
Syntax:- RENAME old_table_name TO new_table_name;

5.TRUNCATE: Removes all data from a table but retains its structure.

Syntax:- TRUNCATE TABLE TABLENAME;

EXAMPLE:-

```
CREATE TABLE Employees ( EmpID NUMBER(5),
Name VARCHAR2(50),
Age NUMBER(3),
Department VARCHAR2(30)
 );
```

1.To check the **current date format of your Oracle session**, you can use the following query:

SELECT * FROM NLS_SESSION_PARAMETERS WHERE PARAMETER = 'NLS_DATE_FORMAT';

2.IF YOU WANT CHECK HOW MAY TABLES ARE PRESENTED IN YOUR DATABASE:-

SQL >SELECT * FROM TAB:

In Oracle, the TAB table is a data dictionary view that **lists all the tables** owned by the current user.

2. Data Manipulation Language (DML)

Used to manipulate data within tables.

- **Commands**:

  INSERT,UPDATE,DELETE

1.INSERT:It is used to insert new record into the table.

Syntax:- INSERT INTO TABLENAME VALUES(VAL1,VAL2,….);

NOTE:-

1.No.of columns in the table no.of values in the query should be same.

2.Charater and date values should be in [‘ ‘] single quotations.

2.UPDATE: Modifies existing records in a table.

Syntax:- UPDATE table_name

SET column1 = value1,

column2 = value2,

...

WHERE condition;

3.DELETE: Removes records from a table.

Syntax:- DELETE FROM table_name WHERE condition;

3.Data Control Language (DCL) Commands

DCL commands in SQL are used to control access to data in a database.

HOW TO CREATE USER ACCOUNT ?

SQL> CREATE USER pragna IDENTIFIED BY pragna;

-- Output: User created.

SQL> GRANT RESOURCE, CONNECT, CREATE SESSION TO pragna;

-- Output: Grant succeeded.

IF YOU WANT TO CREATE 2 USERS ?

SYS AS SYSDBA means logging into the Oracle database as **the most powerful administrator**. It gives full control to manage the database, including starting, stopping, and performing recovery.

This is the query :-

SQL> connect sys as sysdba;

Enter password:pragna

Connected.

SQL> create user hema identified by madam;

User created.

SQL> grant resource,connect,create session to hema;

Grant succeeded.

IF YOU WANT CHECK HOW MANY USERS ARE PRESENTED IN DB?

SQL> SELECT username FROM all_users;

GRANT,REVOKE:-

1. GRANT Command

The GRANT command is used to give users permission to access specific database objects.

Syntax:

GRANT permission_type

ON table_name

TO user_name;

Explanation:

- **permission_type**: The type of access, like SELECT (to read data), INSERT (to add data), UPDATE (to modify data), or ALL (all permissions).

2.REVOKE Command

- The REVOKE command is used to remove previously granted permissions from a user or role to restrict access to specific database objects.
  Syntax:

```
REVOKE permission_type
ON table_name
FROM user_name;
```

4. TCL (Transaction Control Language) Commands

TCL commands are used to manage the **changes made by DML** (Data Manipulation Language) commands like INSERT, UPDATE, DELETE, and ensure data consistency and integrity in the database. These commands control transactions in the database.

---

1. COMMIT

The COMMIT command is used to save all the changes made during the current transaction permanently in the database. Once committed, changes cannot be rolled back.

Syntax:

COMMIT;

---

2. ROLLBACK

The ROLLBACK command is used to undo changes made during the current transaction and revert the database to its previous stable state.

Syntax:

ROLLBACK;

---

3. SAVEPOINT

The SAVEPOINT command is used to set a point in a transaction that you can roll back to later, without rolling back the entire transaction.

Syntax:

SAVEPOINT savepoint_name;

5.**DRL (Data Retrieval Language) :-**
This commands are used to retrieve data from the database. The main and widely recognized DRL command is **SELECT**:

Syntax:

SELECT [DISTINCT] column1, column2, ...

FROM table_name

[WHERE condition]

[GROUP BY column]

[HAVING condition]

[ORDER BY column [ASC|DESC]];

Features and Examples

1. **Retrieve All Columns:**

SELECT *

FROM employees;

- Fetches all columns and rows from the employees table.

2. **Retrieve Specific Columns:**

SELECT first_name, last_name, salary

FROM employees;

- Fetches only the first_name, last_name, and salary columns from the employees table.

DISTINCT Clause :-

The **DISTINCT** keyword is used in SQL to retrieve unique values from a column, eliminating **duplicate records** in the result set.

Syntax

SELECT DISTINCT column1, column2, ...

FROM table_name;

Table: employees

employee_id	first_name	last_name	department_id
1	John	Doe	10
2	Jane	Smith	20
3	Alice	Brown	10
4	Bob	White	30
5	Charlie	Green	20

Query: Retrieve unique department IDs

```
SELECT DISTINCT department_id
FROM employees;
```

Result:

department_id
10
20
30

Arithmetic Operators in Oracle

Arithmetic operators in Oracle are used to perform mathematical operations in SQL queries. The following are the common arithmetic operators:

Operator	Description	Example
+	Adds two numbers	a + b
-	Subtracts the second number from the first	a - b

Operator	Description	Example
*	Multiplies two numbers	a * b
/	Divides the first number by the second	a / b
% (MOD)	Returns the remainder of division	a % b

Example

Here's a detailed explanation of all arithmetic operators in Oracle with **examples** and their usage:

Table for Examples

We'll use the following table employee_salary for all examples:

emp_id	basic_salary	bonus	deduction
101	50000	10000	5000
102	40000	8000	3000
103	60000	12000	7000

---

1. Addition (+)

Calculate total earnings (basic salary + bonus).

```
SELECT emp_id,
    basic_salary,
    bonus,
    (basic_salary + bonus) AS total_earnings
FROM employee_salary;
```

Output:

emp_id	basic_salary	bonus	total_earnings
101	50000	10000	60000
102	40000	8000	48000
103	60000	12000	72000

---

2. Subtraction (-)

Calculate net salary after deductions.

```
SELECT emp_id,
    (basic_salary + bonus) AS total_earnings,
    deduction,
    ((basic_salary + bonus) - deduction) AS net_salary
FROM employee_salary;
```

Output:

emp_id	total_earnings	deduction	net_salary
101	60000	5000	55000
102	48000	3000	45000
103	72000	7000	65000

---

3. Multiplication (*)

Calculate annual basic salary (basic salary × 12).

```
SELECT emp_id,
    basic_salary,
```

```
    (basic_salary * 12) AS annual_salary
FROM employee_salary;
```

Output:

emp_id	basic_salary	annual_salary
101	50000	600000
102	40000	480000
103	60000	720000

---

4. Division (/)

Calculate the monthly deduction (deduction / 12).

```
SELECT emp_id,
    deduction,
    (deduction / 12) AS monthly_deduction
FROM employee_salary;
```

Output:

emp_id	deduction	monthly_deduction
101	5000	416.67
102	3000	250.00
103	7000	583.33

---

5. MOD (Modulo or Remainder)

Find the remainder when total earnings are divided by 1000.

```
SELECT emp_id,
    (basic_salary + bonus) AS total_earnings,
    MOD((basic_salary + bonus), 1000) AS remainder
FROM employee_salary;
```

Output:

emp_id	total_earnings	remainder
101	60000	0
102	48000	0
103	72000	0

Relational Operators:-

Relational operators in Oracle SQL are used to compare values between columns, expressions, or constants. They return a Boolean result (TRUE or FALSE) and are typically used in the WHERE clause of a query.

Relational Operators in Oracle

Operator	Description
=	Equal to
!= or <>	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

Operator	Description
BETWEEN	Checks if a value is within a range
IN	Checks if a value matches any value in a list
NOT IN	Checks if a value does not match any value in a list
LIKE	Pattern matching
IS NULL	Checks if a value is NULL
IS NOT NULL	Checks if a value is not NULL

Examples

emp_id	ename	salary	dept_id	joining_date
101	John	60000	10	01-JAN-2023
102	Alice	75000	20	15-MAR-2023
103	Bob	50000	10	20-APR-2023
104	Eve	80000	30	05-MAY-2023

---

1. Equal To (=)

Find all employees in department 10:

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

Output:

emp_id	ename	salary	dept_id	joining_date
101	John	60000	10	01-JAN-2023
103	Bob	50000	10	20-APR-2023

---

2. Not Equal To (!= or <>)

Find all employees not in department 10:

```
SELECT *
FROM employees
WHERE dept_id != 10;
```

Output:

emp_id	ename	salary	dept_id	joining_date
102	Alice	75000	20	15-MAR-2023
104	Eve	80000	30	05-MAY-2023

---

3. Greater Than (>)

Find all employees with a salary greater than 60000:

```
SELECT *
FROM employees
WHERE salary > 60000;
```

Output:

emp_id	ename	salary	dept_id	joining_date
102	Alice	75000	20	15-MAR-2023

emp_id	ename	salary	dept_id	joining_date
104	Eve	80000	30	05-MAY-2023

---

4. Less Than (<)

Find all employees with a salary less than 70000:

```
SELECT *
FROM employees
WHERE salary < 70000;
```

Output:

emp_id	ename	salary	dept_id	joining_date
101	John	60000	10	01-JAN-2023
103	Bob	50000	10	20-APR-2023

---

5. Between (BETWEEN)

Find all employees with a salary between 50000 and 75000:

```
SELECT *
FROM employees
WHERE salary BETWEEN 50000 AND 75000;
```

Output:

emp_id	ename	salary	dept_id	joining_date
101	John	60000	10	01-JAN-2023
103	Bob	50000	10	20-APR-2023

emp_id	ename	salary	dept_id	joining_date
102	Alice	75000	20	15-MAR-2023

---

6. IN (IN)

Find all employees who work in departments 10 or 30:

```
SELECT *
FROM employees
WHERE dept_id IN (10, 30);
```

Output:

emp_id	ename	salary	dept_id	joining_date
101	John	60000	10	01-JAN-2023
103	Bob	50000	10	20-APR-2023
104	Eve	80000	30	05-MAY-2023

---

7. NOT IN (IN)

Find all employees who are not work in departments 10 or 20:

```
SELECT emp_id, ename, salary, dept_id, joining_date
FROM employees
WHERE dept_id NOT IN (10, 20);
```

8. LIKE

Find all employees whose names start with A:

```
SELECT *
FROM employees
```

```
WHERE ename LIKE 'A%';
```

Output:

emp_id	ename	salary	dept_id	joining_date
102	Alice	75000	20	15-MAR-2023

---

9. IS NULL

Find all employees who have not received a salary (assuming some salary values are NULL):

```
SELECT *
FROM employees
WHERE salary IS NULL;
```

---

10. IS NOT NULL

Find all employees who have received a salary:

```
SELECT *
FROM employees
WHERE salary IS NOT NULL;
```

LIKE OPERATOR:-

Wildcard Character: %

- **%**: Represents zero, one, or multiple characters. It can be placed at the beginning, middle, or end of the string, depending on what you're looking for.

Pattern	Description	Example Results
'%A'	Values that end with "A"	Anna, Tina, Jessica
'A%'	Values that start with "A"	Anna, Aaron
'%A%'	Values that contain "A" anywhere in the string	Anna, Tina, Jessica
'_A%'	Values where the second character is "A"	Sarah, Jana
'A_'	Values where the first character is "A" and the second character is any one character	Anna, Aaro, Aaron

---

Product table:-

How to calculate the discount price:-

Product Table Structure

Product_ID	Product_Name	Price	Quantity	Discount_Percentage
101	Laptop	50000	2	10
102	Mobile	20000	5	5
103	Headphones	1500	10	20

General:-

1. Calculate the **Total Price** before discount:
   Total_Price = Price * Quantity
2. Calculate the **Discount Amount**:
   Discount_Amount = (Total_Price * Discount_Percentage) / 100
3. Calculate the **Final Price** after applying the discount:
   Final_Price = Total_Price - Discount_Amount

SQL Query

```
SELECT
    Product_ID,
    Product_Name,
    Price,
    Quantity,
    (Price * Quantity) AS Total_Price,
```

```
    ((Price * Quantity) * Discount_Percentage / 100) AS
Discount_Amount,

    ((Price * Quantity) - ((Price * Quantity) * Discount_Percentage /
100)) AS Final_Price

FROM

    Product;
```

Output

Product_ID	Product_Name	Price	Quantity	Total_Price	Discount_Amount	Final_Price
101	Laptop	50000	2	100000	10000	90000
102	Mobile	20000	5	100000	5000	95000
103	Headphones	1500	10	15000	3000	12000

Aggregate Functions

Aggregate functions in SQL are used to perform calculations on multiple rows of a table and **return a single result.** Common aggregate functions include **SUM(), AVG(), COUNT(), MIN(), and MAX().**

Example: Product Table (PRODUCTS)

Assume we have a table named **PRODUCTS** with the following data:

PRODUCT_ID	PRODUCT_NAME	CATEGORY	PRICE	QUANTITY
1	Laptop	Electronics	50000	10
2	Smartphone	Electronics	30000	20
3	Refrigerator	Home	45000	5
4	Washing Machine	Home	40000	8
5	Air Conditioner	Home	35000	7

---

1. SUM() Function: Total of a Column

- **Use Case**: Calculate the total value of all products in stock.

```
SELECT
  SUM(PRICE * QUANTITY) AS TOTAL_VALUE
FROM
  PRODUCTS;
```

Result:

TOTAL_VALUE
3,300,000

Explanation:

- The total value is calculated as:
  (50000 * 10) + (30000 * 20) + (45000 * 5) + (40000 * 8) + (35000 * 7) = 3,300,000.

---

2. AVG() Function: Average Value

- **Use Case**: Calculate the average price of all products.

```
SELECT
  AVG(PRICE) AS AVERAGE_PRICE
FROM
  PRODUCTS;
```

Result:

AVERAGE_PRICE
40000

Explanation:

- The average is calculated as:
  (50000 + 30000 + 45000 + 40000 + 35000) / 5 = 40000.

---

3. COUNT() Function: Count Rows

- **Use Case**: Count the number of products in the table.

```
SELECT
  COUNT(*) AS TOTAL_PRODUCTS
FROM
  PRODUCTS;
```

Result:

TOTAL_PRODUCTS
5

Explanation:

- The COUNT(*) function counts all rows in the table, which is 5 in this case.

---

4. MIN() Function: Minimum Value

- **Use Case**: Find the cheapest product price.

```
SELECT
  MIN(PRICE) AS MINIMUM_PRICE
FROM
  PRODUCTS;
```

Result:

MINIMUM_PRICE
30000

Explanation:

- The minimum price in the table is 30000 (Smartphone).

---

5. MAX() Function: Maximum Value

- **Use Case**: Find the most expensive product price.

```
SELECT
  MAX(PRICE) AS MAXIMUM_PRICE
FROM
  PRODUCTS;
```

Result:

MAXIMUM_PRICE

50000

Explanation:

- The maximum price in the table is 50000 (Laptop).

String Functions in SQL:

String functions in SQL are **built-in functions** used to manipulate, process, and retrieve information from string (text) data. They allow operations like changing case, extracting substrings, removing unwanted characters, concatenating strings, and measuring string length.

Example Table: CUSTOMERS

CUSTOMER_ID	CUSTOMER_NAME	EMAIL
1	John Doe	john.doe@example.com
2	Alice Smith	alice.smith@example.com
3	Bob Johnson	bob.johnson@example.com

---

1. UPPER() Function

Converts all characters in a string to uppercase.

Query:

```
SELECT CUSTOMER_NAME, UPPER(CUSTOMER_NAME) AS UPPERCASE_NAME
FROM CUSTOMERS;
```

Result:

CUSTOMER_NAME	UPPERCASE_NAME
John Doe	JOHN DOE
Alice Smith	ALICE SMITH
Bob Johnson	BOB JOHNSON

---

2. LOWER() Function

Converts all characters in a string to lowercase.

Query:

```
SELECT CUSTOMER_NAME, LOWER(CUSTOMER_NAME) AS LOWERCASE_NAME
FROM CUSTOMERS;
```

Result:

CUSTOMER_NAME	LOWERCASE_NAME
John Doe	john doe
Alice Smith	alice smith
Bob Johnson	bob johnson

---

3. SUBSTRING() Function

Extracts a portion of a string starting from a specified position.

Syntax:

```
SUBSTR(string, start_position, length)
```

- string: The input string

- start_position: The starting position (1-based index)
- length: The number of characters to extract (optional)

Query:

```
SELECT CUSTOMER_NAME, SUBSTR(CUSTOMER_NAME, 1, 4) AS
FIRST_NAME_PART

FROM CUSTOMERS;
```

Result:

CUSTOMER_NAME	FIRST_NAME_PART
John Doe	John
Alice Smith	Alic
Bob Johnson	Bob

Explanation: Extracts the first 4 characters from the CUSTOMER_NAME column.

---

4. TRIM() Function

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

TRIM() Function

The TRIM() function is used to remove unwanted characters (usually spaces) from both the **beginning** and **end** of a string. You can also specify a specific character to remove.

Syntax:

TRIM([LEADING | TRAILING | BOTH] [character] FROM string)

- **LEADING**: Removes the specified character from the beginning of the string.
- **TRAILING**: Removes the specified character from the end of the string.
- **BOTH**: Removes the specified character from both the beginning and end (default).

Example of TRIM() Function with Syntax:

We will demonstrate the usage of **LEADING**, **TRAILING**, and **BOTH** options of the TRIM() function with the string '---apple---'.

Table Example: FRUITS

ID	FRUIT_NAME
1	'---apple---'
2	'---banana---'

---

1. Remove Leading Characters

Query:

```
SELECT
   FRUIT_NAME AS ORIGINAL_NAME,
   TRIM(LEADING '-' FROM FRUIT_NAME) AS NO_LEADING
FROM
   FRUITS;
```

Output:

ORIGINAL_NAME	NO_LEADING
'---apple---'	'apple---'
'---banana---'	'banana---'

Explanation:

- Removes all dashes (-) only from the **start** of the string.

---

2. Remove Trailing Characters

Query:

```
SELECT
   FRUIT_NAME AS ORIGINAL_NAME,
   TRIM(TRAILING '-' FROM FRUIT_NAME) AS NO_TRAILING
FROM
   FRUITS;
```

Output:

ORIGINAL_NAME	NO_TRAILING
'---apple---'	'---apple'
'---banana---'	'---banana'

Explanation:

- Removes all dashes (-) only from the **end** of the string.

---

3. Remove Both Leading and Trailing Characters

Query:

```
SELECT
```

```
    FRUIT_NAME AS ORIGINAL_NAME,
    TRIM(BOTH '-' FROM FRUIT_NAME) AS NO_BOTH
FROM
    FRUITS;
```

Output:

ORIGINAL_NAME	NO_BOTH
'---apple---'	'apple'
'---banana---'	'banana'

Explanation:

- Removes all dashes (-) from **both the beginning and end** of the string.

---

5. CONCAT() Function

Combines two or more strings into one.

Query:

```
SELECT CUSTOMER_NAME, CONCAT(CUSTOMER_NAME, '
(Customer)') AS FULL_INFO
FROM CUSTOMERS;
```

Result:

CUSTOMER_NAME	FULL_INFO
John Doe	John Doe (Customer)
Alice Smith	Alice Smith (Customer)

CUSTOMER_NAME	FULL_INFO
Bob Johnson	Bob Johnson (Customer)

---

6. LENGTH() Function

Returns the length of a string in characters.

Query:

```
SELECT CUSTOMER_NAME, LENGTH(CUSTOMER_NAME) AS NAME_LENGTH
FROM CUSTOMERS;
```

Result:

CUSTOMER_NAME	NAME_LENGTH
John Doe	8
Alice Smith	11
Bob Johnson	11

Explanation: Counts the total number of characters in the CUSTOMER_NAME.

Combined Query with All Functions

Query:

```
SELECT
    CUSTOMER_NAME,
    UPPER(CUSTOMER_NAME) AS UPPERCASE_NAME,
    LOWER(CUSTOMER_NAME) AS LOWERCASE_NAME,
    SUBSTR(CUSTOMER_NAME, 1, 4) AS FIRST_NAME_PART,
```

```
    TRIM(' ' FROM CUSTOMER_NAME) AS TRIMMED_NAME,
    CONCAT(CUSTOMER_NAME, ' (Customer)') AS FULL_INFO,
    LENGTH(CUSTOMER_NAME) AS NAME_LENGTH
FROM CUSTOMERS;
```

Result:

CUSTOMER_NAME	UPPERCASE_NAME	LOWERCASE_NAME	FIRST_NAME_PART	TRIMMED_NAME	FULL_INFO	NAME_LENGTH
John Doe	JOHN DOE	john doe	John	John Doe	John Doe (Customer)	8
Alice Smith	ALICE SMITH	alice smith	Alic	Alice Smith	Alice Smith (Customer)	11
Bob Johnson	BOB JOHNSON	bob johnson	Bob	Bob Johnson	Bob Johnson (Customer)	11

Clauses:-

1. GROUP BY Clause

Definition:

The GROUP BY clause groups rows that have the same values in specified columns into aggregated data. It's often used with aggregate functions **like SUM(), AVG(), COUNT(), etc.**

Example Table: EMPLOYEE

EMP_ID	EMP_NAME	DEPARTMENT	SALARY	CITY
101	John	IT	60000	New York
102	Alice	HR	50000	Los Angeles
103	Bob	IT	70000	Chicago
104	Eve	Marketing	45000	New York
105	Charlie	IT	55000	Chicago
106	David	HR	48000	Los Angeles
107	Grace	IT	67000	Chicago

SQL Query Using GROUP BY:

We want to calculate the **total salary for each department**.

```
SELECT DEPARTMENT, SUM(SALARY) AS TOTAL_SALARY
FROM EMPLOYEE
GROUP BY DEPARTMENT;
```

Output:

DEPARTMENT	TOTAL_SALARY
IT	252000
HR	98000
Marketing	45000

Explanation:

- The GROUP BY clause groups all rows by the **DEPARTMENT** column.
- For each department, the SUM(SALARY) function calculates the total salary.

---

2. HAVING Clause

Definition:

The HAVING clause filters the grouped data created by the GROUP BY clause. It's similar to the WHERE clause **but operates on aggregated data.**

If you want to see only those departments where the total salary exceeds 90,000.

SQL Query Using HAVING:

```
SELECT DEPARTMENT, SUM(SALARY) AS TOTAL_SALARY
FROM EMPLOYEE
GROUP BY DEPARTMENT
HAVING SUM(SALARY) > 90000;
```

Output:

DEPARTMENT	TOTAL_SALARY
IT	252000

Explanation:

- The GROUP BY clause groups the data by **DEPARTMENT**.
- The HAVING clause filters out departments where the total salary (SUM(SALARY)) is less than 90,000.

IF you want to see highest salary along with that person name:-

```
SELECT EMP_NAME, SALARY
FROM EMPLOYEE
GROUP BY EMP_NAME, SALARY
HAVING SALARY = (SELECT MAX(SALARY) FROM EMPLOYEE);
```

Explanation

1. **GROUP BY Clause**:
    - Groups the data by EMP_NAME and SALARY. This is necessary because HAVING works on grouped data.
2. **HAVING Clause**:
    - Filters the grouped results to keep only the rows where the salary matches the maximum salary from the subquery (SELECT MAX(SALARY) FROM EMPLOYEE).

Output:

EMP_NAME	SALARY
Bob	70000
Grace	70000

Why Use GROUP BY and HAVING?

- **GROUP BY** is required when aggregating data or when using HAVING.
- **HAVING** is used to filter aggregated or grouped data, unlike WHERE, which filters rows before grouping.

---

3. ORDER BY Clause

Definition:

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

SQL Query Using ORDER BY:

```
SELECT EMP_NAME, SALARY
FROM EMPLOYEE
ORDER BY SALARY DESC;
```

Output:

EMP_NAME	SALARY
Bob	70000

EMP_NAME	SALARY
Grace	67000
John	60000
Charlie	55000
Alice	50000
David	48000
Eve	45000

Explanation:

- The ORDER BY SALARY DESC sorts the rows based on the SALARY column in descending order.
- If you wanted to sort in ascending order, you would use ORDER BY SALARY ASC.

Combining GROUP BY, HAVING, and ORDER BY

Scenario:

Let's combine everything! :-

1. Calculate the total salary for each department.
2. Filter out departments with a total salary of 90,000 or less.
3. Sort the result in descending order of total salary.

SQL Query:

```
SELECT DEPARTMENT, SUM(SALARY) AS TOTAL_SALARY
FROM EMPLOYEE
```

```
GROUP BY DEPARTMENT
HAVING SUM(SALARY) > 90000
ORDER BY TOTAL_SALARY DESC;
```

Output:

DEPARTMENT	TOTAL_SALARY
IT	252000

Explanation:

1. GROUP BY DEPARTMENT: Groups rows by the department.
2. SUM(SALARY): Calculates the total salary for each group (department).
3. HAVING SUM(SALARY) > 90000: Filters out groups where the total salary is less than 90,000 .
4. ORDER BY TOTAL_SALARY DESC: Sorts the result in descending order of total salary.

---

why we use HAVING instead of WHERE :-

1. The Key Difference Between HAVING and WHERE :-

- **WHERE Clause:**
  - Filters rows **before grouping** the data.
  - Works on individual rows.
  - Cannot work with aggregate functions (e.g., SUM(), COUNT(), AVG()).
- **HAVING Clause:**

- Filters rows **after grouping** the data.
- Works on grouped data (aggregates like SUM, AVG, etc.).
- Specifically designed to filter data resulting from aggregate functions.

To see only duplicated data:-

Sql query:-

```
SELECT emp_name, salary, COUNT(*)
FROM emtble
GROUP BY emp_name, salary
HAVING COUNT(*) > 1;
```

Constraints in SQL

A **constraint** is a rule or condition that is applied to data in a table to ensure that it adheres to certain standards, ensuring data integrity, consistency, and accuracy. Constraints help enforce rules at the database level.

Here are the **types of constraints** in SQL:

---

1. NOT NULL Constraint

- **Definition**: Ensures that a column cannot have a NULL value. It guarantees that a field will always contain a value when data is inserted or updated.

2. UNIQUE Constraint

- **Definition**: Ensures that all values in a column or a set of columns are unique, meaning no two rows can have the same value in that column or combination of columns. It allows NULL values (but each NULL must be unique).

3. PRIMARY KEY Constraint

- **Definition**: A combination of the NOT NULL and UNIQUE constraints. It uniquely identifies each record in a table. A table can have only one primary key, and the values in the primary key columns must be unique and not NULL.

4. FOREIGN KEY Constraint

- **Definition**: Establishes and enforces a link between the columns in two tables. The foreign key in one table points to the primary key or a unique key in another table. It maintains referential integrity by ensuring that a value in the foreign key column matches a value in the referenced table.

5. CHECK Constraint

- **Definition**: Ensures that all values in a column satisfy a specific condition or set of conditions. This can be used to limit the range of values or enforce business rules.

6. DEFAULT Constraint

- **Definition**: Provides a default value for a column when no value is specified during an insert operation. This helps to maintain consistency by automatically filling in a predefined value when data is missing.

  Example:-

  1. Employee Table (emp45)

- **Columns**:

- id (Primary Key): The unique identifier for each employee.
- name (Unique): Employee's name, which must be unique.
- age: Employee's age, with a constraint to ensure it is valid (greater than or equal to 18).
- department_id: The foreign key that links to the Department table.
- join_date: The date when the employee joins, default generated by the system.
- salary: Employee's salary, with a default value of 250000.

2. Department Table (dep45)

- **Columns**:
  - department_id (Primary Key): The unique identifier for each department.
  - name: The name of the department.
  - designation: The designation of the department (e.g., "HR", "Finance", etc.).

Tables creation:-

```
-- Create the Department table (dep45)

CREATE TABLE dep45 (

    department_id number(20) PRIMARY KEY, -- Primary key for department_id

    name VARCHAR2(100) NOT NULL, -- Department name cannot be NULL

    designation VARCHAR2(100) NOT NULL -- Department designation cannot be NULL
```

```
);


-- Create the Employee table (emp45)
CREATE TABLE emp45 (
   id number(20) PRIMARY KEY, -- Primary key for employee id
   name VARCHAR2(100) NOT NULL UNIQUE, -- Name cannot be NULL and must be unique
   age number(30) CHECK (age >= 18), -- Age must be 18 or greater
   department_id number(30), -- Foreign key referencing department_id in dep45
   join_date DATE DEFAULT SYSDATE, -- Automatically set the current system date as the join date
   salary number(10,4) DEFAULT 250000, -- Default salary is 250000 if not specified
FOREIGN KEY (department_id) REFERENCES dep45(department_id) -- Foreign key linking employee to department
);
```

Logical Operators in Oracle

Logical operators in SQL are used to combine multiple conditions in a WHERE clause to filter data based on specific conditions. These operators help in making complex queries by evaluating multiple logical expressions.

Types of Logical Operators in Oracle

1. **AND Operator**

- The AND operator is used when all conditions must be TRUE for a record to be included in the result.
- It returns TRUE only if both conditions are met; otherwise, it returns FALSE.

2. **OR Operator**
   - The OR operator is used when at least one condition must be TRUE for a record to be included.
   - It returns TRUE if any of the conditions evaluate to TRUE; otherwise, it returns FALSE.
3. **NOT Operator**
   - The NOT operator is used to reverse the result of a condition.
   - If a condition is TRUE, NOT makes it FALSE, and if it is FALSE, NOT makes it TRUE.

- Find employees in the IT department who earn more than 50,000 OR belong to HR.
- Find employees who are NOT in Finance AND earn more than 40,000.
- Find employees who work in IT OR HR but NOT in Finance.
- Find employees who do NOT have a manager AND belong to HR or Finance.
- Find employees who either earn more than 55,000 OR do NOT belong to HR.
- Find employees who belong to IT OR HR but NOT earning less than 50,000.
- Find employees who are NOT in IT AND either earn below 40,000 OR have no manager.

Employees Table:

employee_id	name	department	salary	manager_id
1	Alice	IT	60000	101
2	Bob	HR	45000	102
3	Charlie	Finance	55000	103
4	David	IT	40000	104
5	Emma	Sales	70000	NULL
6	Frank	HR	38000	NULL
7	Grace	IT	52000	101
8	Harry	Finance	42000	103
9	Ivy	IT	48000	104
10	Jack	HR	62000	105

---

1. Employees in IT earning > 50,000 OR in HR

Query:

```
SELECT * FROM employees
WHERE (department = 'IT' AND salary > 50000) OR department =
'HR';
```

Output:

employee_id	name	department	salary	manager_id
1	Alice	IT	60000	101
2	Bob	HR	45000	102
6	Frank	HR	38000	NULL
7	Grace	IT	52000	101
10	Jack	HR	62000	105

Explanation:

- The query selects employees in the **IT department** who earn more than 50,000 (Alice, Grace).
- It also selects employees in the **HR department** regardless of salary (Bob, Frank, Jack).

---

2. Employees NOT in Finance AND earning > 40,000

Query:

```
SELECT * FROM employees
WHERE NOT department = 'Finance' AND salary > 40000;
```

Output:

employee_id	name	department	salary	manager_id
1	Alice	IT	60000	101

employee_id	name	department	salary	manager_id
2	Bob	HR	45000	102
5	Emma	Sales	70000	NULL
7	Grace	IT	52000	101
9	Ivy	IT	48000	104
10	Jack	HR	62000	105

Explanation:

- The query excludes employees in the **Finance department**.
- From the remaining employees, it only selects those earning more than 40,000 (Alice, Bob, Emma, Grace, Ivy, Jack).

---

3. Employees in IT OR HR but NOT in Finance

Query:

```
SELECT * FROM employees
WHERE (department = 'IT' OR department = 'HR') AND NOT department = 'Finance';
```

Output:

employee_id	name	department	salary	manager_id
1	Alice	IT	60000	101
2	Bob	HR	45000	102
4	David	IT	40000	104
6	Frank	HR	38000	NULL
7	Grace	IT	52000	101

employee_id	name	department	salary	manager_id
9	Ivy	IT	48000	104
10	Jack	HR	62000	105

Explanation:

- The query selects employees who are either in **IT** or **HR**.
- It **excludes** those in the **Finance department**.

---

4. Employees who do NOT have a manager AND belong to HR or Finance

Query:

```
SELECT * FROM employees
WHERE manager_id IS NULL AND (department = 'HR' OR department = 'Finance');
```

Output:

employee_id	name	department	salary	manager_id
6	Frank	HR	38000	NULL

Explanation:

- The query selects employees who **do not have a manager** (manager_id IS NULL).
- It further filters to include only those from the **HR** or **Finance** department (Frank).

- **manager_id IS NULL**: This condition selects employees who **do not** have a manager. In other words, **their manager_id field is NULL. This is because, in databases, NULL represents the**

absence of a value. So, for an employee to "not have a manager," the manager_id should be NULL.

- **(department = 'HR' OR department = 'Finance')**: This condition selects employees who belong to either the **HR** department **or** the **Finance** department.

What happens when these conditions are combined?

The query will return employees who meet **both** of the following conditions:

1. **Do not have a manager** (manager_id IS NULL).
2. Belong to the **HR** or **Finance** department.

Key takeaway:

The **NOT** operator works by reversing the condition. If an employee does not meet the condition inside the parentheses, the **NOT** will make it true, and the employee will be included. In this case, **Bob** is in HR, so he is **automatically included** because the condition (department = 'IT' AND salary > 60000) is false for him.

---

5. Employees who either earn more than 55,000 OR do NOT belong to HR

Query:

```
SELECT * FROM employees
WHERE salary > 55000 OR NOT department = 'HR';
```

Output:

employee_id	name	department	salary	manager_id
1	Alice	IT	60000	101
3	Charlie	Finance	55000	103
4	David	IT	40000	104
5	Emma	Sales	70000	NULL
7	Grace	IT	52000	101
8	Harry	Finance	42000	103
9	Ivy	IT	48000	104
10	Jack	HR	62000	105

Explanation:

- The query selects employees who either earn more than 55,000 (Alice, Jack) or those who are not in **HR** (everyone except HR department).

Key takeaway:

- The **OR** operator means **either** one of the conditions has to be true for a record to be included.
- **Jack**, who is in the HR department, is included because his **salary is greater than 55,000**.
- Employees who do **not belong to HR** are also included, regardless of their salary.

Conclusion:

The query includes:

- Employees with a **salary greater than 55,000**.
- Employees **not in HR** (regardless of salary).

---

6. Employees who belong to IT OR HR but NOT earning less than 50,000

Query:

SELECT * FROM employees

WHERE (department = 'IT' OR department = 'HR') AND NOT salary < 50000;

Output:

employee_id	name	department	salary	manager_id
1	Alice	IT	60000	101
7	Grace	IT	52000	101
10	Jack	HR	62000	105

Explanation:

- The query selects employees who are either in **IT** or **HR**.
- It further filters to only include those earning **50,000 or more** (Alice, Grace, Jack).

What happens here?

- The query **first filters for employees in the IT or HR departments**.
- It then **excludes employees earning less than 50,000** (because **NOT salary < 50000** means "salary >= 50,000").

Key takeaway:

- The **NOT salary < 50000** condition does **not** mean "salary can't be above 50,000." It actually means **"salary must be 50,000 or more"**.
- The **NOT** operator negates the condition to reverse its meaning.

---

7. Employees who are NOT in IT AND either earn below 40,000 OR have no manager

Query:

```
SELECT * FROM employees
WHERE NOT department = 'IT' AND (salary < 40000 OR manager_id IS NULL);
```

Output:

employee_id	name	department	salary	manager_id
5	Emma	Sales	70000	NULL
6	Frank	HR	38000	NULL

Explanation:

- The query excludes employees from **IT**.
- It includes employees who either earn below 40,000 (Frank) or have no manager (Emma, Frank).

So, the results for employees are:

- **Emma**: She is **not** in the IT department and has **no manager** (manager_id is NULL). Hence, she meets the second condition.
- **Frank**: He is **not** in the IT department and has a salary of 38,000, which meets the first condition (salary < 40000).

In a table might be empty column along with null values both having ?

```
SELECT *
FROM your_table
WHERE manager IS NULL
  OR manager = '';
```

Explanation:

- IS NULL: This condition checks for NULL values.
- manager = '': This checks for empty strings (''), assuming the column is of VARCHAR or TEXT type.

Joins in SQL

Joins in SQL are used to **combine rows** from two or more tables based on a related column.

Types of Joins in SQL

There are **five main types** of joins:

1. **INNER JOIN**
2. **LEFT JOIN (LEFT OUTER JOIN)**
3. **RIGHT JOIN (RIGHT OUTER JOIN)**
4. **FULL JOIN (FULL OUTER JOIN)**
5. **CROSS JOIN**

Types of Joins in SQL – Definitions

1. **INNER JOIN** – Returns only the matching records from both tables based on a common column.
2. **LEFT JOIN (LEFT OUTER JOIN)** – Returns all records from the left table and the matching records from the right table. If no match is found, NULL is returned for the right table's columns.
3. **RIGHT JOIN (RIGHT OUTER JOIN)** – Returns all records from the right table and the matching records from the left table. If no match is found, NULL is returned for the left table's columns.
4. **FULL JOIN (FULL OUTER JOIN)** – Returns all records from both tables, with NULL values where there is no match in either table.
5. **CROSS JOIN** – Returns the Cartesian product of both tables, meaning every row from the first table is paired with every row from the second table.

Comparison Table

Join Type	Matches Required?	Unmatched Rows Shown?	Example Use Case
INNER JOIN	Yes, in both tables	❌ No	Find employees with assigned departments
LEFT JOIN	Yes, in left table	✅ Yes, from left	List all employees, even those without departments
RIGHT JOIN	Yes, in right table	✅ Yes, from right	List all departments, even if no employees are assigned

Join Type	Matches Required?	Unmatched Rows Shown?	Example Use Case
FULL JOIN	No, includes all	✅ Yes, from both	Show all employees and all departments, even without matches
CROSS JOIN	No matching column needed	✅ Yes, all combinations	Generate test cases, all possible combinations

---

Conclusion

- If you need **only matching records** → Use **INNER JOIN**
- If you need **all records from the left table** → Use **LEFT JOIN**
- If you need **all records from the right table** → Use **RIGHT JOIN**
- If you need **all records from both tables** → Use **FULL JOIN**
- If you need **all combinations of records** → Use **CROSS JOIN**

Table creation:-

```
CREATE TABLE emp55 (
   EmpID NUMBER(20) PRIMARY KEY,
   Name VARCHAR2(50),
   DeptID NUMBER(20)
);
CREATE TABLE dep55 (
   DeptID NUMBER(20) PRIMARY KEY,
   DeptName VARCHAR2(50)
);
```

Employees Table (emp55)

EmpID	Name	DeptID
1	Alice	101
2	Bob	102
3	Charlie	103
4	David	NULL

Departments Table (dep55)

DeptID	DeptName
101	HR
102	IT
103	Finance
104	Marketing

Now, let's see each **SQL JOIN example** with the expected output and explanation.

1. INNER JOIN Example

- **Returns only matching records from both tables.**

Query:

```
SELECT e.EmpID, e.Name, d.DeptName
FROM emp55 e
```

INNER JOIN dep55 d ON e.DeptID = d.DeptID;

Output:

EmpID	Name	DeptName
1	Alice	HR
2	Bob	IT
3	Charlie	Finance

Explanation:

- **Only employees who have a matching DeptID in dep55 are included.**
- **David** is missing because his DeptID is NULL.

---

2. LEFT JOIN Example

- **Returns all records from emp55 (left table) and matching records from dep55 (right table).**

Query:

```
SELECT e.EmpID, e.Name, d.DeptName
FROM emp55 e
LEFT JOIN dep55 d ON e.DeptID = d.DeptID;
```

Output:

EmpID	Name	DeptName
1	Alice	HR
2	Bob	IT
3	Charlie	Finance

EmpID	Name	DeptName
4	David	NULL

Explanation:

- **All employees are included.**
- **David appears, but since he has no DeptID, DeptName is NULL.**

---

3. RIGHT JOIN Example

- **Returns all records from dep55 (right table) and matching records from emp55 (left table).**

Query:

```
SELECT e.EmpID, e.Name, d.DeptName
FROM emp55 e
RIGHT JOIN dep55 d ON e.DeptID = d.DeptID;
```

Output:

EmpID	Name	DeptName
1	Alice	HR
2	Bob	IT
3	Charlie	Finance
NULL	NULL	Marketing

Explanation:

- **All departments are included.**
- **The Marketing department is listed, but since no employee is assigned to it, EmpID and Name are NULL.**

---

4. FULL OUTER JOIN Example

- **Returns all records from both tables. If there's no match, it returns NULL.**

Query:

```
SELECT e.EmpID, e.Name, d.DeptName
FROM emp55 e
FULL OUTER JOIN dep55 d ON e.DeptID = d.DeptID;
```

Output:

EmpID	Name	DeptName
1	Alice	HR
2	Bob	IT
3	Charlie	Finance
4	David	NULL
NULL	NULL	Marketing

Explanation:

- **All employees and all departments** are included.
- **David is listed, but his department is NULL.**
- **Marketing is listed, but no employee is assigned to it.

5.CROSS JOIN Query:

A **CROSS JOIN** in SQL returns the **Cartesian Product**, pairing **each row from the first table with every row from the second table**.

Query:-

```
SELECT e.EmpID, e.Name, d.DeptID, d.DeptName
FROM emp55 e
```

```
CROSS JOIN dep55 d;
```

EmpID	Name	DeptID	DeptName
1	Alice	101	HR
1	Alice	102	IT
1	Alice	103	Finance
1	Alice	104	Marketing
2	Bob	101	HR
2	Bob	102	IT
2	Bob	103	Finance
2	Bob	104	Marketing
3	Charlie	101	HR
3	Charlie	102	IT
3	Charlie	103	Finance
3	Charlie	104	Marketing
4	David	101	HR
4	David	102	IT
4	David	103	Finance
4	David	104	Marketing

Explanation:

- Since emp55 has **4 rows** and dep55 has **4 rows**, the **CROSS JOIN** generates **4 × 4 = 16 rows**.
- **Each employee is paired with every department.**
- This type of join is useful for generating all possible combinations of two datasets.

Alternative Syntax (Without CROSS JOIN):-

SELECT e.EmpID, e.Name, d.DeptID, d.DeptName

FROM emp55 e, dep55 d;

This also performs a CROSS JOIN (Cartesian Product) in Oracle.

When to Use CROSS JOIN?

✅ When you **need all possible combinations** of two datasets.
❌ Avoid using it on large datasets **without a filter** (it can generate millions of rows).

Date Operators:-

Date operators in SQL are used to perform operations on date values. They help manipulate and compare date data types, either by adding or subtracting intervals, or by comparing two dates. These operators are essential when working with date-related calculations or when filtering records based on date values.

Types of Date Operators

1. **Arithmetic Date Operators:** These are used to perform calculations on date values, such as adding or subtracting days, months, or years.
    - Addition (+): Adds days (or intervals) to a date.

- Subtraction (-): Subtracts days (or intervals) from a date.

2. **Relational Date Operators:** These are used to compare dates and determine whether one date is before, after, or equal to another date.
    - Equality (=): Checks if two dates are the same.
    - Not Equal (!= or <>): Checks if two dates are not the same.
    - Greater Than (>): Checks if one date is after another.
    - Less Than (<): Checks if one date is before another.
    - Greater Than or Equal To (>=): Checks if one date is the same or after another date.
    - Less Than or Equal To (<=): Checks if one date is the same or before another date.
3. **Functions:** These are used for specific date manipulations and operations.
    - MONTHS_BETWEEN: Finds the number of months between two dates.
    - ADD_MONTHS: Adds a specified number of months to a date.
    - NEXT_DAY: Finds the next occurrence of a specific weekday after a given date.
    - LAST_DAY: Finds the last day of the month for a given date.

Note:-

The **TO_DATE function** in Oracle is used to convert a string to a DATE data type. It allows you to specify the format of the string you're converting so that Oracle can correctly interpret the date.

Oracle Date Format check:-

To check our **oracle DB date format:-**

```
SELECT *
FROM NLS_SESSION_PARAMETERS
WHERE PARAMETER = 'NLS_DATE_FORMAT';
```

Explanation:

- This query retrieves the value of the NLS_DATE_FORMAT parameter specifically for your session from the NLS_SESSION_PARAMETERS view.
- The result will show the current date format being used for your session.

Example Output:

If the format is DD-MON-YY, the output would look like:

```
PARAMETER            | VALUE
---------------------|-----------
NLS_DATE_FORMAT      | DD-MON-YY
```

Changing the Date Format for the Session:

If you want to change the date format for your session, you can run the following:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'YYYY-MM-DD';
```

This will change the date format to YYYY-MM-DD for the current session.

Table: Employeee

emp_id	emp_name	joining_date
1	John Doe	2024-02-01
2	Jane Smith	2023-12-15
3	Alice Brown	2022-06-10

---

1. Arithmetic Date Operators

(A) + (Addition)

Adds days to a date.

Example Query:

```
SELECT emp_name, joining_date, joining_date + 10 AS new_date
FROM Employeee;
```

Output:

emp_name	joining_date	new_date
John Doe	2024-02-01	2024-02-11
Jane Smith	2023-12-15	2023-12-25
Alice Brown	2022-06-10	2022-06-20

Explanation:

- **Added 10 days** to the joining date of each employee.

(B) - (Subtraction)

Subtracts days from a date.

Example Query:

```
SELECT emp_name, joining_date, joining_date - 5 AS new_date
FROM Employeee;
```

Output:

emp_name	joining_date	new_date
John Doe	2024-02-01	2024-01-27
Jane Smith	2023-12-15	2023-12-10
Alice Brown	2022-06-10	2022-06-05

Explanation:

- **Subtracted 5 days** from each employee's joining date.

(C) MONTHS_BETWEEN

Finds the number of months between two dates.

Example Query:

```
SELECT emp_name, joining_date, MONTHS_BETWEEN(SYSDATE,
joining_date) AS months_difference
FROM Employeee;
```

Output:

emp_name	joining_date	months_difference
John Doe	2024-02-01	0.30
Jane Smith	2023-12-15	1.60
Alice Brown	2022-06-10	19.50

Explanation:

- **MONTHS_BETWEEN** calculates how many months have passed since the joining date up to the current date (SYSDATE).

(D) ADD_MONTHS

Adds months to a date.

Example Query:

SELECT emp_name, joining_date, ADD_MONTHS(joining_date, 3) AS new_date

FROM Employeee;

Output:

emp_name	joining_date	new_date
John Doe	2024-02-01	2024-05-01
Jane Smith	2023-12-15	2024-03-15
Alice Brown	2022-06-10	2022-09-10

Explanation:

- **Added 3 months** to the joining date.

(E) NEXT_DAY

Finds the next occurrence of a specific weekday.

Example Query:

SELECT emp_name, joining_date, NEXT_DAY(joining_date, 'FRIDAY') AS next_friday

FROM Employeee;

Output:

emp_name	joining_date	next_friday
John Doe	2024-02-01	2024-02-02
Jane Smith	2023-12-15	2023-12-22
Alice Brown	2022-06-10	2022-06-17

Explanation:

- **NEXT_DAY** gives the next **Friday** after the employee's joining date.

(F) LAST_DAY

Finds the last day of the month.

Example Query:

```
SELECT emp_name, joining_date, LAST_DAY(joining_date) AS last_day_of_month
FROM Employeee;
```

Output:

emp_name	joining_date	last_day_of_month
John Doe	2024-02-01	2024-02-29
Jane Smith	2023-12-15	2023-12-31
Alice Brown	2022-06-10	2022-06-30

Explanation:

- **LAST_DAY** returns the last day of the month for the given joining date.

---

2. Relational Date Operators

(A) = (Equality)

Checks if two dates are equal.

Example Query:

```
SELECT emp_name, joining_date
FROM Employeee
WHERE joining_date = TO_DATE('2024-02-01', 'YYYY-MM-DD');
```

Output:

emp_name	joining_date
John Doe	2024-02-01

Explanation:

- This checks if any employee's joining date matches **2024-02-01**.

(B) != or <> (Not Equal)

Checks if two dates are not equal.

Example Query:

```
SELECT emp_name, joining_date
FROM Employeee
WHERE joining_date != TO_DATE('2024-02-01', 'YYYY-MM-DD');
```

Output:

emp_name	joining_date
Jane Smith	2023-12-15
Alice Brown	2022-06-10

Explanation:

- This returns employees whose **joining_date** is not equal to **2024-02-01**.

(C) > (Greater Than)

Checks if one date is after another.

Example Query:

SELECT emp_name, joining_date

FROM Employeee

WHERE joining_date > TO_DATE('2023-12-01', 'YYYY-MM-DD');

Output:

emp_name	joining_date
John Doe	2024-02-01

Explanation:

- This query checks if the **joining_date** is **after December 1, 2023**.

(D) < (Less Than)

Checks if one date is before another.

Example Query:

SELECT emp_name, joining_date

FROM Employeee

WHERE joining_date < TO_DATE('2023-12-01', 'YYYY-MM-DD');

Output:

emp_name joining_date

Alice Brown 2022-06-10

Explanation:

- This query checks if the **joining_date** is **before December 1, 2023**.

(E) >= (Greater Than or Equal To)

Checks if a date is on or after another date.

Example Query:

```
SELECT emp_name, joining_date
FROM Employeee
WHERE joining_date >= TO_DATE('2023-12-01', 'YYYY-MM-DD');
```

Output:

emp_name joining_date

John Doe 2024-02-01

Jane Smith 2023-12-15

Explanation:

- This checks if **joining_date** is on or after **December 1, 2023**.

(F) <= (Less Than or Equal To)

Checks if a date is on or before another date.

Example Query:

```
SELECT emp_name, joining_date
```

```
FROM Employeee

WHERE joining_date <= TO_DATE('2023-12-01', 'YYYY-MM-DD');
```

Output:

emp_name	joining_date
Alice Brown	2022-06-10

Explanation:

- This checks if **joining_date** is on or before **December 1, 2023**.

Summary

- **Arithmetic Operators** allow for adding and subtracting days from dates, or calculating the months between dates.
- **Relational Operators** compare dates for equality, greater/less than, or on/before conditions.

SETS in Oracle

In Oracle, **sets** refer to **set operations**, which are used to combine the results of multiple SELECT queries. These operations help retrieve data from two or more tables based on certain conditions.

Oracle supports the following **set operations**:

1. **UNION** – Combines results from multiple queries and removes duplicates.
2. **UNION ALL** – Combines results but keeps duplicates.
3. **INTERSECT** – Returns only common records between queries.
4. **MINUS** – Returns records from the first query that do not exist in the second query.

Creating two tables:-

1. Sales23 Table Creation (Sales Data for 2023)

```
CREATE TABLE Sales23 (
    OrderID NUMBER(6) PRIMARY KEY,      -- OrderID with 6 digits
    Product VARCHAR2(50),               -- Product Name with a max length of 50 characters
    Amount NUMBER(10,4)                 -- Amount with up to 10 digits in total, 4 after the decimal point
);
```

2. Sales22 Table Creation (Sales Data for 2022)

```
CREATE TABLE Sales22 (
    OrderID NUMBER(6) PRIMARY KEY,      -- OrderID with 6 digits
    Product VARCHAR2(50),               -- Product Name with a max length of 50 characters
    Amount NUMBER(10,4)                 -- Amount with up to 10 digits in total, 4 after the decimal point
);
```

Tables and Sample Data

Sales23 Table (Sales Data for 2023)

OrderID	Product	Amount
201	Laptop	55000
202	Mobile	35000
203	Tablet	25000

Sales22 Table (Sales Data for 2022)

OrderID	Product	Amount
202	Mobile	35000
203	Tablet	25000
204	Monitor	28000

---

1. UNION (Removes Duplicates)

```
SELECT OrderID, Product, Amount FROM Sales23
UNION
SELECT OrderID, Product, Amount FROM Sales22;
```

Output (Unique Orders from Both Years, No Duplicates)

OrderID	Product	Amount
201	Laptop	55000
202	Mobile	35000
203	Tablet	25000
204	Monitor	28000

Explanation:

- OrderID 202 and 203 are common in both tables, so they appear only once.
- OrderID 201 and 204 are unique, so they are included.

---

2. UNION ALL (Keeps Duplicates)

```
SELECT OrderID, Product, Amount FROM Sales23
UNION ALL
SELECT OrderID, Product, Amount FROM Sales22;
```

Output (All Orders, Including Duplicates)

OrderID	Product	Amount
201	Laptop	55000
202	Mobile	35000
203	Tablet	25000
202	Mobile	35000
203	Tablet	25000
204	Monitor	28000

Explanation:

- OrderID 202 and 203 appear twice because they exist in both tables.
- OrderID 201 and 204 are unique, so they appear only once.

---

3. INTERSECT (Common Records in Both Tables)

```
SELECT OrderID, Product, Amount FROM Sales23
INTERSECT
SELECT OrderID, Product, Amount FROM Sales22;
```

Output (Only Common Orders in Both Years)

OrderID	Product	Amount
202	Mobile	35000

203 Tablet 25000

Explanation:

- Only OrderID 202 and 203 are present in both tables, so only these records are shown.

---

4. MINUS (Records in Sales23 but NOT in Sales22)

```
SELECT OrderID, Product, Amount FROM Sales23
MINUS
SELECT OrderID, Product, Amount FROM Sales22;
```

Output (Records in Sales23 but NOT in Sales22)

OrderID Product Amount

201 Laptop 55000

Explanation:

- OrderID 201 exists **only in Sales23**, so it is included in the result.
- OrderID 202 and 203 exist in both tables, so they are removed.

Summary Table

Set Operation	Usage	Duplicates?	Example Output
UNION	Combines results from both tables	No Duplicates	Unique records only

UNION ALL	Combines results from both tables	Keeps Duplicates	All records shown
INTERSECT	Returns common records from both tables	No Duplicates	Only matching records
MINUS	Returns records in the first table but not in the second	No Duplicates	Unique records from Sales23

What is a Subquery in Oracle?

A **subquery** is a query nested inside another SQL query. It is used to retrieve data that will be used in the main query as a condition for selection, insertion, updating, or deletion.

Key Points:

- A subquery is enclosed within parentheses ().
- It can be used in **SELECT, INSERT, UPDATE, DELETE** statements.
- It can return a single value (scalar subquery) or multiple values (row/column subquery).
- Only **DML (Data Manipulation Language) commands** like INSERT, UPDATE, and DELETE work with subqueries.

Table Creation :_

```
CREATE TABLE em11 (
    EmpID NUMBER(5) PRIMARY KEY,
    Name VARCHAR2(50),
```

```
    Salary NUMBER(10,2),
    Department VARCHAR2(50)
);
```

Table em11 Structure:

EmpID	Name	Salary	Department
101	John	60000.00	HR
102	Alice	75000.00	IT
103	Bob	50000.00	Finance

1.INSERT Using Subquery

Insert an employee into em11 where the salary is equal to the maximum salary of an existing employee.

```
INSERT INTO em11 (EmpID, Name, Salary, Department)
VALUES (104, 'David', (SELECT MAX(Salary) FROM em11), 'IT');
```

Explanation:

- The **subquery** (SELECT MAX(Salary) FROM em11) fetches the highest salary from em11.
- The **outer query** inserts a new employee **David** with that salary.

Output (After Insert):

EmpID	Name	Salary	Department
101	John	60000.00	HR
102	Alice	75000.00	IT
103	Bob	50000.00	Finance

EmpID	Name	Salary	Department
104	David	75000.00	IT

2.UPDATE Using Subquery

Increase the salary of employees in em11 whose salary is below the average salary of all employees.

```
UPDATE em11
SET Salary = Salary + 5000
WHERE Salary < (SELECT AVG(Salary) FROM em11);
```

Explanation:

- The **subquery** (SELECT AVG(Salary) FROM em11) calculates the average salary.
- The **outer query** increases the salary by **5000** for employees who earn **less than the average salary**.

Output (After Update):

EmpID	Name	Salary	Department
101	John	65000.00	HR
102	Alice	75000.00	IT
103	Bob	55000.00	Finance
104	David	75000.00	IT

3.DELETE Using Subquery

Delete employees whose salary is equal to the minimum salary in em11.

```
DELETE FROM em11
WHERE Salary = (SELECT MIN(Salary) FROM em11);
```

Explanation:

- The **subquery** (SELECT MIN(Salary) FROM em11) finds the lowest salary.
- The **outer query** deletes employees with this lowest salary.

Output (After Delete):

EmpID	Name	Salary	Department
101	John	65000.00	HR
102	Alice	75000.00	IT
104	David	75000.00	IT

Summary of DML Subquery Operations

DML Command	Use Case	Query Example
INSERT	Add new records based on existing data	INSERT INTO em11 (EmpID, Name, Salary, Department) VALUES (104, 'David', (SELECT MAX(Salary) FROM em11), 'IT');
UPDATE	Modify records based on a condition from another table/query	UPDATE em11 SET Salary = Salary + 5000 WHERE Salary < (SELECT AVG(Salary) FROM em11);
DELETE	Remove records based on a condition	DELETE FROM em11 WHERE Salary = (SELECT MIN(Salary) FROM em11);

DML Command	Use Case	Query Example
	from another table/query	

Key Takeaways

Subqueries allow us to dynamically retrieve and use values within INSERT, UPDATE, and DELETE statements.
DML Subqueries work efficiently for operations based on existing data conditions.
Best Practice: Use subqueries when working with comparisons like MAX(), MIN(), and AVG().

What is a View in Oracle?

A **View** in Oracle is a virtual table that is based on the result of a **SELECT** query. It does not store data physically but provides a way to look at the data from one or more tables in a structured way.

Importance of Views

Security: Restricts direct access to sensitive data by displaying only specific columns.
Simplicity: Simplifies complex SQL queries for users.
Data Abstraction: Hides table structure complexity.
Reusability: Allows frequently used queries to be saved as a view.
Consistency: Ensures a uniform data representation across multiple users.

Basic Syntax for Creating a View

```
CREATE VIEW view_name AS

SELECT column1, column2, ...
```

```
FROM table_name
WHERE condition;
```

Example Using Table emp19

1. Table Creation (emp19)

```
CREATE TABLE emp19 (
    EmpID NUMBER(6) PRIMARY KEY,
    Name VARCHAR2(50),
    Salary NUMBER(10,2),
    Department VARCHAR2(50)
);
```

---

emp19 table contains:

EmpID	Name	Salary	Department
101	John	60000.00	HR
102	Alice	75000.00	IT
103	Bob	50000.00	Finance

If your Oracle DB not allowed to create the view ,you need to get access.
connect to your systemdb:-

```
SQL> CONNECT sys/password AS SYSDBA;
Connected.
SQL> GRANT CREATE VIEW TO pragna;
```

Grant succeeded.

SQL> connect pragna/pragna;

Connected.

SQL> create view emp_view as select * from emp19 where salary >50000;

View created.

---

2. Create a View (emp_view)

We will create a view that includes only employees earning more than **50,000**.

CREATE VIEW emp_view AS

SELECT EmpID, Name, Salary, Department

FROM emp19

WHERE Salary > 50000;

The emp_view contains:

EmpID	Name	Salary	Department
101	John	60000.00	HR
102	Alice	75000.00	IT

Performing DML Operations on the View

3.1 Insert Data into the View

INSERT INTO emp_view (EmpID, Name, Salary, Department)

VALUES (104, 'David', 80000.00, 'Marketing');

Output (Now emp19 also has this record, since views do not store data separately):

EmpID	Name	Salary	Department
101	John	60000.00	HR
102	Alice	75000.00	IT
103	Bob	50000.00	Finance
104	David	80000.00	Marketing

3.2 Update Data in the View

```
UPDATE emp_view
SET Salary = 85000.00
WHERE EmpID = 104;
```

Output (Now salary for EmpID 104 is updated in emp19 as well):

EmpID	Name	Salary	Department
101	John	60000.00	HR
102	Alice	75000.00	IT
103	Bob	50000.00	Finance
104	David	85000.00	Marketing

3.3 Delete Data from the View

```
DELETE FROM emp_view WHERE EmpID = 101;
```

Output (John is removed from emp19 and emp_view):

EmpID	Name	Salary	Department
102	Alice	75000.00	IT

EmpID	Name	Salary	Department
103	Bob	50000.00	Finance
104	David	85000.00	Marketing
