UNIT - III:SQL & PL/SQL Lecture Hrs:12

Interactive SQL Part I - Types of Data Constraints - Computations done on Table Data - Oracle functions -Grouping Data from Tables in SQL - Sub queries – Joins - Concatenation data from table columns using the Union - Intersect and Minus Clause – Views – Sequences - Granting and Revoking Permissions - Advantages of PL/SQL - The Generic PL/SQL block - Control Structure - What is Cursor – Database Triggers - Types of Triggers.

Interactive SQL Part I

Interactive SQL refers to a mode where you enter SQL statements through a terminal or graphical interface and receive results immediately. Unlike embedded SQL, which is written inside an application's source code, Interactive SQL is used for browsing data, testing queries, and performing administrative tasks.

Interactive SQL Part I: Fundamentals

In a basic interactive session, you typically follow a sequence of defining, populating, and querying data.

- **Data Definition (DDL):** You start by creating a structure for your data. For example, a `STATION` table might store weather data.
- **Data Manipulation (DML):** Once the table exists, you insert records into it.
- **Data Querying:** You retrieve specific information using filters and sorting.

Example Session

The following sequence demonstrates a standard interactive workflow.

Create a Table:

```
sql
```

```
CREATE TABLE Students (ID INTEGER PRIMARY KEY,Name VARCHAR(50),Major VARCHAR(30));
```

Insert Data:

```
sql

INSERT INTO Students (ID, Name, Major) VALUES (101, 'Varun', 'Computer Science');
```

Query the Result:

```
sql

SELECT * FROM Students WHERE Major = 'Computer Science';
```

Interactive Features

- **Parameter Prompting:** Some implementations, such as those in Oracle, allow you to use variables that prompt for input before execution (e.g., `&user_id`).
- **Immediate Feedback:** Interactive tools like SAP SQL Anywhere (dbisql) provide a direct window to view table data and error messages instantly.
- **Command History:** Most interactive shells allow you to retrieve and edit previous commands using function keys or arrow keys.

Types of Data Constraints

Data constraints are rules applied to columns in a database table to enforce data integrity and ensure the accuracy and consistency of data.

Common types of data constraints with examples:

- **NOT NULL Constraint**: Ensures that a column cannot contain NULL values, meaning a value must always be provided for that column.
  - **Example:** In a Students table, the StudentID column is declared as NOT NULL to ensure every student has an ID.

SQL>CREATE TABLE Students (StudentID INT NOT NULL, Name VARCHAR(20);

UNIQUE Constraint: Ensures that all values in a column (or a set of columns) are distinct and no two rows have the same value for that column(s). It allows NULL values, but only one NULL value is permitted.

Example: In an Employees table, the Email column is declared as UNIQUE to ensure no two employees share the same email address.

Code: CREATE TABLE Employees(EmployeeID INT, Email VARCHAR(25) UNIQUE);

- **PRIMARY KEY Constraint:** Uniquely identifies each row in a table. It is a combination of NOT NULL and UNIQUE constraints, meaning the primary key column(s) must contain unique and non-null values. Each table can have only one primary key.
- Example: In a Products table, ProductID is designated as the PRIMARY KEY to uniquely identify each product.
- Code: CREATE TABLE Products (ProductID INT PRIMARY KEY, ProductName VARCHAR(25));

- **FOREIGN KEY Constraint:** Establishes a link between two tables, ensuring referential integrity. It refers to the primary key of another table, preventing actions that would create "orphan" records.

  - **Example:** In an Orders table, CustomerID is a FOREIGN KEY referencing the CustomerID (primary key) in the Customers table, ensuring that an order can only be placed by an existing customer.

    Code: CREATE TABLE Orders (OrderID INT PRIMARY KEY,
    OrderDate DATE,CustomerID INT,FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID));

- **CHECK Constraint:** Enforces a condition that all values in a column must satisfy.

- **Example:** In a Users table, a CHECK constraint ensures that the Age column stores values greater than or equal to 18.

  Code: CREATE TABLE Users (UserID INT PRIMARY KEY,
  UserName VARCHAR(20), Age INT CHECK (Age >= 18));

- **DEFAULT Constraint:** Provides a default value for a column when no value is explicitly specified during an INSERT operation.

- **Example:** In a Tasks table, the Status column is given a DEFAULT value of 'Pending'.

  Code: CREATE TABLE Tasks (TaskID INT PRIMARY KEY, TaskName VARCHAR(20), Status VARCHAR(50) DEFAULT 'Pending');

Oracle Functions

Oracle provides two main categories of functions: Built-in (SQL) Functions and User-Defined (PL/SQL) Functions.

Built-in (SQL) Functions

These are pre-defined functions provided by Oracle that can be used directly in SQL queries. They are categorized based on the data type they operate on or the type of value they return.

1. Numeric Functions: Perform operations on numbers.

- ABS(n): Returns the absolute value of n.

SQL>SELECT ABS(-10) FROM DUAL; -- Returns 10

- ROUND(n, decimal_places): Rounds n to the specified number of decimal places.

SQL>SELECT ROUND(123.456, 2) FROM DUAL; -- Returns 123.46

- MOD(m, n): Returns the remainder of m divided by n.

SQL> SELECT MOD(10, 3) FROM DUAL; -- Returns 1

2. Character Functions: Operate on character strings.

- UPPER(string): Converts a string to uppercase.

SQL> SELECT UPPER('hello') FROM DUAL; -- Returns 'HELLO'

- LENGTH(string): Returns the length of a string.

SQL> SELECT LENGTH('Oracle') FROM DUAL; -- Returns 6

- CONCAT(string1, string2): Concatenates two strings.

SQL> SELECT CONCAT('Hello', 'World') FROM DUAL; -- Returns 'HelloWorld'

3. Date Functions: Operate on date and time values.

- SYSDATE: Returns the current date and time.

SQL>SELECT SYSDATE FROM DUAL; -- Returns current date and time

- ADD_MONTHS(date, n): Adds n months to a date.

SQL>SELECT ADD_MONTHS(SYSDATE, 3) FROM DUAL; -- Returns date 3 months from now

4. Conversion Functions: Convert data from one type to another.

- TO_CHAR(value, format): Converts a number or date to a string.

SQL> SELECT TO_CHAR(SYSDATE, 'YYYY-MM-DD') FROM DUAL; -- Returns current date in 'YYYY-MM-DD' format

- TO_NUMBER(string): Converts a string to a number.

SQL>SELECT TO_NUMBER('123') + 10 FROM DUAL; -- Returns 133

5. Aggregate Functions: Perform calculations on a set of rows and return a single result.

- COUNT(*): Counts the number of rows.

SQL>SELECT COUNT(*) FROM employees;

- AVG(column): Calculates the average of a column.

SQL> SELECT AVG(salary) FROM employees;

Grouping Data from tables in SQL

GROUP BY statement groups rows that have the same values in one or more columns. It is commonly used to create summaries, such as total sales by region or number of users by age group.

Its main features include:

- Used with the SELECT statement.
- Groups rows after filtering with WHERE.
- Can be combined with aggregate functions like SUM(), COUNT(), AVG(), etc.
- Filter grouped results using the HAVING clause.
- Comes after WHERE but before HAVING and ORDER BY.

Query execution order: FROM -> WHERE -> GROUP BY -> HAVING -> SELECT -> ORDER BY.

Syntax:

SELECT column1, aggregate_function(column2)
FROM table_name WHERE condition GROUP BY column1, column2;

Parameters:

- aggregate_function: function used for aggregation, e.g., SUM(), AVG(), COUNT().
- table_name: name of the table from which data is selected.

- condition: Optional condition to filter rows before grouping (used with WHERE).
- column1, column2: Columns on which the grouping is applied.

Examples of GROUP BY

Let's assume that we have a Student table. We will insert some sample data into this table and then perform operations using GROUP BY to understand how it groups rows based on a column and aggregates data.

CREATE TABLE student (name VARCHAR(50),year INT, subject VARCHAR(50));

INSERT INTO student (name, year, subject) VALUES('Avery', 1,

'Mathematics'),('Elijah', 2, 'English'),('Harper', 3, 'Science'),

('James', 1, 'Mathematics'),('Charlotte', 2, 'English'),('Benjamin', 3, 'Science');

Output

name	year	subject
Avery	1	Mathematics
Elijah	2	English
Harper	3	Science
James	1	Mathematics
Charlotte	2	English
Benjamin	3	Science

Example 1: Group By Single Column

When we group by a single column, rows with the same value in that column are combined. For example, grouping by subject shows how many students are enrolled in each subject.

Query:

SELECT subject, COUNT(*) AS Student_Count FROM Student GROUP BY subject;

Output

subject	Student_Count
English	2
Mathematics	2
Science	2

Explanation: Each subject appears twice in the table, so the count for English, Mathematics and Science is 2.

Example 2: Group By Multiple Columns

Using GROUP BY with multiple columns groups rows that share the same values in those columns. For example, grouping by subject and year will combine rows with the same subject–year pair and we can count how many students fall into each group.

Query:

SELECT subject, year, COUNT(*) FROM Student GROUP BY subject, year;

Output

subject	year	count(*)

subject	year	count(*)
English	2	2
Mathematics	1	2
Science	3	2

Explantion: Students with the same subject and year are grouped together. Since each subject–year pair occurs twice, the count is 2 for every group.

HAVING Clause in GROUP BY Clause

HAVING clause is used to filter results after grouping, especially when working with aggregate functions like SUM(), COUNT() or AVG(). Unlike WHERE, it applies conditions on grouped data.

In this section, we will use Employee table(emp) first insert some sample data, then perform GROUP BY queries combined with HAVING.

CREATE TABLE emp (emp_no INT PRIMARY KEY,name VARCHAR(50), sal DECIMAL(10,2),age INT);

INSERT INTO emp (emp_no, name, sal, age) VALUES
(1, 'Liam', 50000.00, 25),(2, 'Emma', 60000.50, 30),
(3, 'Noah', 75000.75, 35),(4, 'Olivia', 45000.25, 28),
(5, 'Ethan', 80000.00, 32),(6, 'Sophia', 65000.00, 27),
(7, 'Mason', 55000.50, 29),(8, 'Isabella', 72000.75, 31),
(9, 'Logan', 48000.25, 26),(10, 'Mia', 83000.00, 33);

SELECT * FROM emp;

Output

emp_no	name	sal	age
1	Liam	50000.00	25
2	Emma	60000.50	30
3	Noah	75000.75	35
4	Olivia	45000.25	28
5	Ethan	80000.00	32
6	Sophia	65000.00	27
7	Mason	55000.50	29
8	Isabella	72000.75	31
9	Logan	48000.25	26
10	Mia	83000.00	33

Example 1: Filter by Total Salary

In this query, we group employees by name and display only those whose total salary is greater than 50,000.

```
SELECT NAME, SUM(sal) FROM Emp GROUP BY name
HAVING SUM(sal)>50000;
```

Output

name	SUM(sal)
Emma	60000.50
Noah	75000.75
Ethan	80000.00
Sophia	65000.00
Mason	55000.50
Isabella	72000.75
Mia	83000.00

Explanation: Only employees whose total salary exceeds 50,000 appear in the result.

Set Operations

Set operations in SQL are used to combine or exclude the results of two or more SELECT statements into a single result set. The primary set operators are **UNION**, **UNION ALL**, **INTERSECT**, and **EXCEPT** (or **MINUS** in Oracle).

Key Rules for Using Set Operations

To use set operations successfully, the following rules must be followed:

- Both queries must return the **same number of columns**.
- The corresponding columns in both queries must have **compatible data types**.
- The order of columns in the SELECT statements must be the same.
- The ORDER BY clause can only be used once, at the very end of the combined query, to sort the final result set.

Example

1. UNION

The UNION operator combines the result sets of two or more SELECT statements and returns only the **distinct (unique) rows**, removing any duplicates.

Example: Get a list of all unique employee names from both tables.

```
SELECT Name FROM Employees_A UNION SELECT Name FROM Employees_B;
```

UNION ALL

The UNION ALL operator combines the result sets of two or more SELECT statements, but it **includes all duplicate rows**. This operator is generally faster than UNION because it does not have to perform the extra step of filtering out duplicates.

Example: Get a list of all employee names from both tables, including duplicates.

```
SELECT Name FROM Employees_A UNION ALL SELECT Name FROM Employees_B;
```

INTERSECT

The INTERSECT operator returns only the rows that are **common** to all the SELECT statements.

Example: Get the names of employees who are present in *both* Employees_A and Employees_B.

```
SELECT Name FROM Employees_A INTERSECT SELECT Name FROM Employees_B;
```

EXCEPT (or MINUS)

The EXCEPT operator returns rows that are in the result set of the **first** SELECT statement but **not** in the result set of the **second** SELECT statement. Note that EXCEPT is supported by most databases (like SQL Server, PostgreSQL, and MySQL 8.0+), while Oracle uses the synonym MINUS. The order of the queries matters for this operation.

Example: Get the names of employees who are in Employees_A but not in Employees_B.

```
SELECT Name FROM Employees_A EXCEPT -- Use MINUS in Oracle
SELECT Name FROM Employees_B;
```

Subqueries

A **subquery** (also known as an inner query or inner select) is a SQL query nested inside a larger query (the outer query). Subqueries can be used in various parts of a SQL statement, such as the SELECT, INSERT, UPDATE, or DELETE clauses, and are most commonly found in the WHERE clause.

Key Characteristics of Subqueries:

- A subquery must be enclosed in parentheses ().
- It can retrieve data or evaluate a condition for the main query.
- The ORDER BY clause cannot be used within a subquery (unless within a top-level SELECT of a view or ORDER BY of a subquery in the FROM clause).
- Subqueries can return a single value (scalar subquery), a single column of multiple values, or multiple columns of multiple values.

Types of Subqueries:

1. **Scalar Subquery:** Returns exactly one row and one column.
2. **Column Subquery:** Returns multiple rows, but only one column. Often used with IN, ANY, ALL.
3. **Row Subquery:** Returns a single row with one or more columns.

4. **Correlated Subquery:** The inner query depends on the outer query's data and executes once for every row processed by the outer query.
5. **Non-Correlated Subquery:** The inner query executes independently of the outer query, just once.

Non-Correlated Subquery Example (Scalar Subquery)

This is a basic non-correlated subquery that runs once to find a value used by the outer query.

Scenario: Find the names of employees who have a salary greater than the average salary of all employees.

Tables: Employees

mpID	Name	Department	Salary
1	Alice	HR	60000
2	Bob	IT	75000
3	Charlie	IT	80000
4	David	HR	62000

SQL Query:

```
SELECT Name, Salary FROM Employees WHERE Salary > (SELECT AVG(Salary) FROM Employees);
```

Explanation:

1. The inner subquery (SELECT AVG(Salary) FROM Employees) runs first and returns a single value (e.g., 69250).
2. The outer query then executes as SELECT Name, Salary FROM Employees WHERE Salary > 69250.

Result:

Name	Salary

Bob	75000
Charlie	80000

Non-Correlated Subquery Example (Column Subquery with IN)

Scenario: Find all employees who work in the 'IT' or 'HR' departments (assuming department IDs are used in the main table).

Tables: Employees (with DeptID), Departments (with DeptID, DeptName)

```
SELECT Name FROM Employees WHERE DeptID IN (SELECT DeptID FROM Departments WHERE DeptName IN ('IT', 'HR'));
```

Explanation:

1. The inner subquery returns a column of DeptID values for 'IT' and 'HR'.
2. The outer query uses these IDs to filter employees.

Correlated Subquery Example

Scenario: Find all employees whose salary is greater than the *average salary of their own department*.

```
SELECT Name, Department, Salary FROM Employees E1 WHERE Salary > (SELECT AVG(Salary)FROM Employees E2 WHERE E2.Department = E1.Department);
```

Explanation:

1. The outer query selects a row (aliased as E1).
2. The inner subquery executes for that specific row, calculating the average salary *only* for the department of E1.
3. The outer query compares E1.Salary to that departmental average.
4. This process repeats for every row in the Employees table.

A **subquery** (also known as an inner query or nested query) is an SQL query nested inside another SQL statement, such as SELECT, INSERT, UPDATE, or DELETE. It is used to return data that will be used by the main query (outer query) to complete its operation.

Key Concepts

- **Enclosed in Parentheses:** Subqueries must always be enclosed within parentheses ().
- **Execution Order:** Typically, the inner query executes first, and its result is passed to the outer query. (Correlated subqueries are an exception, where the inner query executes once for each row of the outer query).
- **Placement:** Subqueries can be used in the SELECT, FROM, WHERE, and HAVING clauses.
- **Operators:** They can be used with comparison operators like =, >, <, >=, <=, or with multiple-row operators like IN, NOT IN, ANY, ALL, and EXISTS.

Example: Using a Subquery in a WHERE Clause

A common use case is to filter records in one table based on an aggregate result from the same or another table.

Scenario: Find the employees who earn more than the average salary of all employees.

Using a Subquery in a WHERE Clause

A common use case is to filter records in one table based on an aggregate result from the same or another table.

Scenario: Find the employees who earn more than the average salary of all employees.

Tables: Employees

EmpID	Name	Department	Salary
1	Alice	HR	50000
2	Bob	IT	70000
3	Charlie	Sales	60000

4	David	IT	80000

```
SELECT Name,Salary FROM Employees WHERE Salary > (SELECT AVG(Salary) FROM Employees);
```

Explanation:

1. The **inner query** (SELECT AVG(Salary) FROM Employees) calculates the average salary of all employees (which is 65000).
2. The **outer query** then uses this single value to filter the results: SELECT Name, Salary FROM Employees WHERE Salary > 65000.

Result:

Name	Salary
Bob	70000
David	80000

Sequences

SQL sequences are database objects used **to generate unique numeric values automatically**. They are commonly used for **creating primary keys or other unique identifiers** and can be **shared across multiple tables.**

Creating a Sequence

The basic syntax for creating a sequence is as follows. The parameters allow customization of the sequence's behavior:

```
CREATE SEQUENCE sequence_name
[ AS integer_type ]
[ START WITH initial_value ]
[ INCREMENT BY increment_value ]
[ MINVALUE min_value | NO MINVALUE ]
[ MAXVALUE max_value | NO MAXVALUE ]
[ CYCLE | NO CYCLE ]
```

[CACHE size | NO CACHE];

- **sequence_name**: A unique name for the sequence.
- **AS integer_type**: The data type for the sequence (e.g., INT, BIGINT, DECIMAL). BIGINT is the default.
- **START WITH**: The first value the sequence will return.
- **INCREMENT BY**: The value by which the sequence increases (or decreases, if negative) with each call.
- **MINVALUE / MAXVALUE**: The lower and upper limits of the sequence.
- **CYCLE / NO CYCLE**: Determines if the sequence restarts from the minimum (or maximum) value after reaching its limit. NO CYCLE is the default.
- **CACHE / NO CACHE**: Improves performance by pre-allocating values in memory, but can cause gaps if the server stops abnormally.

Examples

Sequence with Specific Range and Cycle

This sequence starts at 10, increments by 5, has a maximum of 50, and restarts (cycles) from 10 when the maximum is reached.

sql

```
CREATE SEQUENCE seq_OrderTracking

AS DECIMAL(3, 0)

START WITH 10

INCREMENT BY 5

MAXVALUE 50

MINVALUE 10

CYCLE;
```

Using Sequences

Once a sequence is created, you can use the NEXT VALUE FOR function to retrieve the next unique number.

1. Retrieving the Next Value Independently

You can test the sequence or get a value without inserting it into a table immediately.

```
sql
SELECT NEXT VALUE FOR seq_CustomerID AS NextID;
```

2. Using a Sequence in an INSERT Statement

Sequences are commonly used to automatically populate a primary key column when inserting new records into a table.

```
sql
-- First, create a sample table
CREATE TABLE Customers (CustomerID INT PRIMARY KEY,CustomerName VARCHAR(50));

-- Insert data using the sequence
INSERT INTO Customers (CustomerID, CustomerName)VALUES (NEXT VALUE FOR seq_CustomerID, 'Akash');

INSERT INTO Customers (CustomerID, CustomerName) VALUES (NEXT VALUE FOR seq_CustomerID, 'Ram');

-- View the results
SELECT * FROM Customers;
```

Result:

CustomerID	CustomerName
1	Akash
2	Ram

3. Using a Sequence with a Default Constraint

You can set a sequence as the default value for a column, similar to an IDENTITY column, to automatically manage ID generation upon insert.

```
sql
-- Create a table with the sequence as a default
CREATE TABLE Products (ProductID INT PRIMARY KEY DEFAULT (NEXT
VALUE FOR seq_OrderTracking),ProductName VARCHAR(50));

-- Insert data without specifying the ProductID
INSERT INTO Products (ProductName) VALUES ('Laptop');

INSERT INTO Products (ProductName) VALUES ('Mouse');

-- View the results
SELECT * FROM Products;
```

Granting and Revoking Permissions

In SQL, the GRANT and REVOKE statements (part of Data Control Language or DCL) are used to manage **database permissions** (also known as privileges) for users, roles, or the public.

GRANT: Assigning Permissions

The GRANT statement is used to provide access rights to a user. These rights can include the ability to SELECT, INSERT, UPDATE, or DELETE data from a specific object (like a table or view).

ProductID	ProductName
10	Laptop
15	Mouse

Syntax:

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

You can grant multiple privileges in a single statement by separating them with commas. The WITH GRANT OPTION allows the recipient to, in turn, grant those same permissions to other users.

1. Granting SELECT Privilege to a User

To grant the SELECT privilege on a table named "Employees" to a user named "Arun", use the following command:

GRANT SELECT ON Employees TO 'Arun'@'localhost;'

2. Granting Multiple Privileges

To grant multiple privileges, such as SELECT, INSERT, DELETE, and UPDATE to the user "Arun",

GRANT SELECT, INSERT, DELETE, UPDATE ON Employees TO 'Arun@'localhost';

3. Granting All Privileges

To grant all privileges on the "Employees" table to "Arun", use:

GRANT ALL ON Employees TO 'Arun'@'localhost';

4. Granting Privileges to All Users

To grant a specific privilege (e.g., SELECT) to all users on the "Employees" table, execute:

GRANT SELECT ON Employees TO '*'@'localhost';

In the above example the "*****" symbol is used to grant select permission to all the users of the table "Employees".

5. Granting Privileges on Functions/Procedures

While using functions and procedures, the Grant statement can be used to grant users the ability to execute the functions and procedures in MySQL.

Granting Execute Privilege: Execute privilege gives the ability to execute a function or procedure.

Syntax:

GRANT EXECUTE ON [PROCEDURE | FUNCTION] object TO user;

Example for granting **EXECUTE** privilege on a function named "**CalculateSalary**":

GRANT EXECUTE ON FUNCTION CalculateSalary TO 'Arun'@'localhost';

- To verify, the database administrator can use SHOW GRANTS FOR user1; to see the assigned privileges.

REVOKE: Removing Permissions

The REVOKE statement does the opposite of GRANT; it withdraws privileges that were previously granted to a user.

Syntax:

REVOKE privilege_name ON object_name FROM user_name;

Various Ways of Revoking Privileges From a User

Below are the different ways of revoking privileges from a user in MySQL.

1. Revoking SELECT Privilege

To revoke Select Privilege to a table named "Employees" where User Name is Arun, the following revoke statement should be executed.

```
REVOKE SELECT ON  Employees FROM 'Arun'@'localhost';
```

2. Revoking Multiple Privileges

To revoke multiple Privileges to a user named "**Arun**" in a table "users", the following revoke statement should be executed.

```
REVOKE SELECT, INSERT, DELETE, UPDATE ON Users FROM
'Arun'@'localhost';
```

3. Revoking All Privileges

To revoke all the privileges to a user named "Arun" in a table "users", the following revoke statement should be executed.

```
REVOKE ALL ON Users FROM 'Arun'@'localhost';
```

Revoking Privileges on Functions/Procedures

While using functions and procedures, the revoke statement can be used to revoke the privileges from users which have been EXECUTE privileges in the past.

Syntax:

```
REVOKE EXECUTE ON [ PROCEDURE | FUNCTION ] object FROM User;
```

1. Revoking EXECUTE Privileges on a Function in MySQL

If there is a function called "CalculateSalary" and you want to revoke EXECUTE access to the user named Arun, then the following revoke statement should be executed.

```
REVOKE EXECUTE ON FUNCTION Calculatesalary FROM 'Arun'@'localhost';
```

2. Revoking EXECUTE Privileges to All Users on a Function in MySQL

If there is a function called "CalculateSalary" and you want to revoke EXECUTE access to all the users, then the following revoke statement should be executed.

```
REVOKE EXECUTE ON FUNCTION Calculatesalary FROM '*'@'localhost';
```

3. Revoking EXECUTE Privilege to a Users on a Procedure in MySQL

If there is a procedure called "DBMSProcedure" and you want to revoke EXECUTE access to the user named Arun, then the following revoke statement should be executed.

```
REVOKE EXECUTE ON PROCEDURE DBMSProcedure FROM 'Arun'@'localhost';
```

4. Revoking EXECUTE Privileges to all Users on a Procedure in MySQL

If there is a procedure called "DBMSProcedure" and we want to revoke EXECUTE access to all the users, then the following revoke statement should be executed.

```
REVOKE EXECUTE ON PROCEDURE DBMSProcedure FROM '*'@'localhost';
```

Advantages of PL/SQL

A PL/SQL block is a fundamental, structured unit of code in Oracle Database's procedural language extension to SQL. It is used to **group related SQL statements and procedural logic into a single unit** that the database engine can execute efficiently. This structure is foundational to building sophisticated database applications, improving **performance, modularity, and error handling** capabilities.

Key advantages include:

- **High Performance**: PL/SQL significantly reduces network traffic between the application and the database by allowing an entire block of SQL statements to be sent to the database at once, rather than executing single queries individually. Stored procedures, functions, and packages are compiled once and stored in executable form, making subsequent calls efficient and fast.

- **Tight Integration with SQL**: PL/SQL is seamlessly integrated with SQL. It supports all SQL data types and allows the use of all SQL data manipulation, transaction control, and cursor control statements directly within PL/SQL blocks, eliminating the need for data type conversions between the two languages.

- **Procedural Capabilities**: Unlike pure SQL, PL/SQL provides procedural language features such as variables, conditional statements (IF-THEN-ELSE), and loops (FOR, WHILE). This allows developers to implement complex business logic and control flow directly within the database.

- **Code Reusability and Modularity**: PL/SQL allows for the creation of subprograms (procedures and functions) and packages, which are stored in the database and can be invoked repeatedly by multiple applications. This promotes modular design, code reuse, and easier maintenance since changes only need to be applied in one central location.

- **Robust Error Handling**: PL/SQL provides an effective exception-handling mechanism. When an error occurs during program execution, control is transferred to the exception section, allowing the developer to manage the error gracefully and prevent the entire application from crashing.

- **Enhanced Security**: Stored procedures in PL/SQL can abstract data access, meaning users can be granted access to execute a procedure but not the underlying tables. This provides a strong layer of security and data integrity.

- **Portability and Scalability**: PL/SQL applications are highly portable and can run on any operating system or platform where the Oracle database is supported without code changes. The use of stored programs also enhances scalability by centralizing processing on the server, which is shared among many users.

- **High Productivity**: By offering a full range of software engineering features and allowing for compact, readable code, PL/SQL saves time during development and debugging.

The Generic PL/SQL Block

The fundamental unit of code in Oracle PL/SQL is an **anonymous block**. This generic block structure is used to combine SQL and procedural logic. It is a block-structured language element, designed to improve performance and manage errors effectively.

A PL/SQL block is logically divided into three distinct sections, defined by specific keywords:

1. Declarative Section (DECLARE)

This section is **optional**. It is the area where all local identifiers are defined. Items declared here, such as variables, constants, cursors, user-defined exceptions, and local subprograms (nested procedures or functions), exist only for the duration of the block's execution.

- **Keyword:** DECLARE

2. Executable Section (BEGIN...END)

This section is **mandatory**. It contains the core logic of the block. It holds the SQL statements and procedural statements (control flow structures like loops and IF statements, assignment statements, etc.) that perform the desired tasks. Every PL/SQL block must have a BEGIN and an END statement.

- **Keywords:** BEGIN and END

3. Exception Handling Section (EXCEPTION)

This section is **optional**. It provides a structured way to handle runtime errors (exceptions) that might occur within the executable section. Code here defines specific handlers for anticipated errors (e.g., NO_DATA_FOUND) or general handlers for unexpected errors (WHEN OTHERS). If an error occurs in the BEGIN...END block, control is transferred to this section.

- **Keyword:** EXCEPTION

Key Characteristics:

- **Block Structure:** Blocks can be nested within one another, allowing for modular and organized code.
- **Performance:** Executing a single block reduces network overhead compared to sending multiple individual SQL statements to the database.
- **Delimiter:** The forward slash (/) character is commonly used in client tools like SQL*Plus to execute the anonymous block

Example:PL/SQL Code:

```
SET SERVEROUTPUT ON; -- Command to enable output display in the client tool

DECLARE
  -- Declarative Section: Define a local variable
  greeting_message VARCHAR2(50);

BEGIN
```

```
  -- Executable Section: Assign a value and print it
  greeting_message := 'Welcome to the world of PL/SQL blocks!';
  DBMS_OUTPUT.PUT_LINE(greeting_message);

EXCEPTION
  -- Exception Handling Section: A simple generic handler
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('An unexpected error occurred.');

END;
/
```

Result (Output in the SQL Client Console):

```
Welcome to the world of PL/SQL blocks!

PL/SQL procedure successfully completed.
```

PL/SQL Conditional Structure

Control structures in PL/SQL manage the flow of program execution, enabling decision-making, repetition, and sequence modification. Decision-making statements in **programming languages** decide the direction of the flow of program execution. Conditional Statements available in PL/SQL are defined below:

1. **IF THEN**
2. **IF THEN ELSE**
3. **NESTED-IF-THEN**
4. **IF THEN ELSIF-THEN-ELSE Ladder**

1. IF THEN

if then the statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not

i.e if a certain condition is true then a block of statement is executed otherwise not.

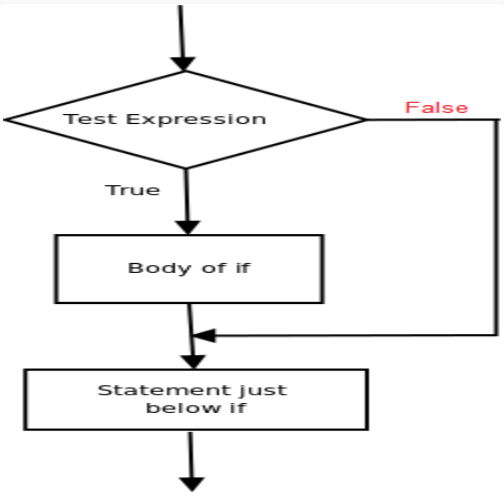
Syntax:

```
if condition then
-- do something
end if;
```

Here, condition after evaluation will be either true or false. if statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not. if and endif consider as a block here.

Example:

```
declare
-- declare the values here
begin
if condition then
dbms_output.put_line('output');
end if;
dbms_output.put_line('output2');
end;
```

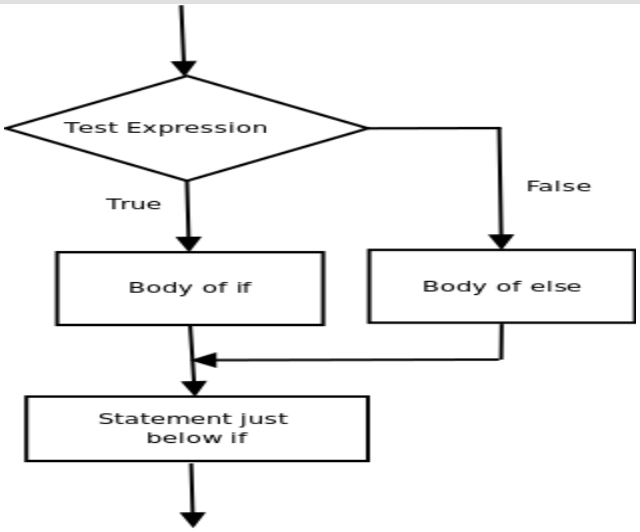


2. IF THEN ELSE

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the else statement. We can use the else statement with if statement to execute a block of code when the condition is false.

Syntax:-

```
if (condition) then
    -- Executes this block if condition is true
else
    -- Executes this block if condition is false
```



Example:-

```
-- pl/sql program to illustrate If else statement
declare
num1 number:= 10;
num2 number:= 20;
begin
if num1 < num2 then
dbms_output.put_line('i am in if block');
ELSE
dbms_output.put_line('i am in else Block');
```

```
end if;
dbms_output.put_line('i am not in if or else Block');
end;
```

Output:-

```
i'm in if Block
i'm not in if and not in else Block
```

The block of code following the else statement is executed as the condition present in the if statement is false after calling the statement which is not in block(without spaces).

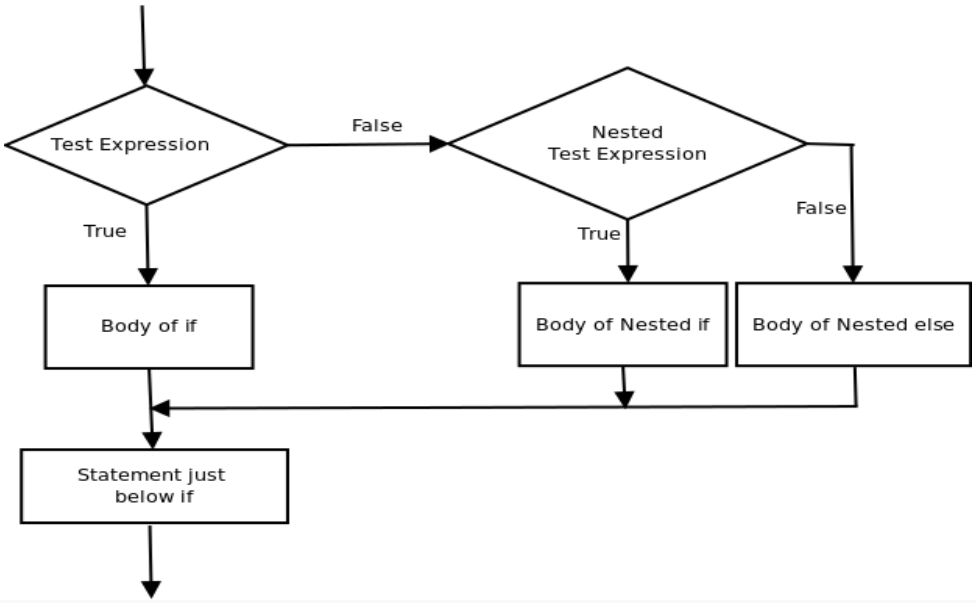
3. NESTED-IF-THEN

A nested if-then is an if statement that is the target of another if statement.

Nested if-then statements mean an if statement inside another if statement.

Yes, PL/SQL allows us to nest if statements within if-then statements. i.e, we can place an if then statement inside another if then statement. Syntax:-

```
if (condition1) then
   -- Executes when condition1 is true
   if (condition2) then
     -- Executes when condition2 is true
   end if;
end if;
```




```
-- pl/sql program to illustrate nested If statement
declare
num1 number:= 10;
num2 number:= 20;
num3 number:= 20;
begin
if num1 < num2 then
dbms_output.put_line('num1 small num2');
  if num1 < num3 then
  dbms_output.put_line('num1 small num3 also');
  end if;
end if;
dbms_output.put_line('after end if');
end;
```

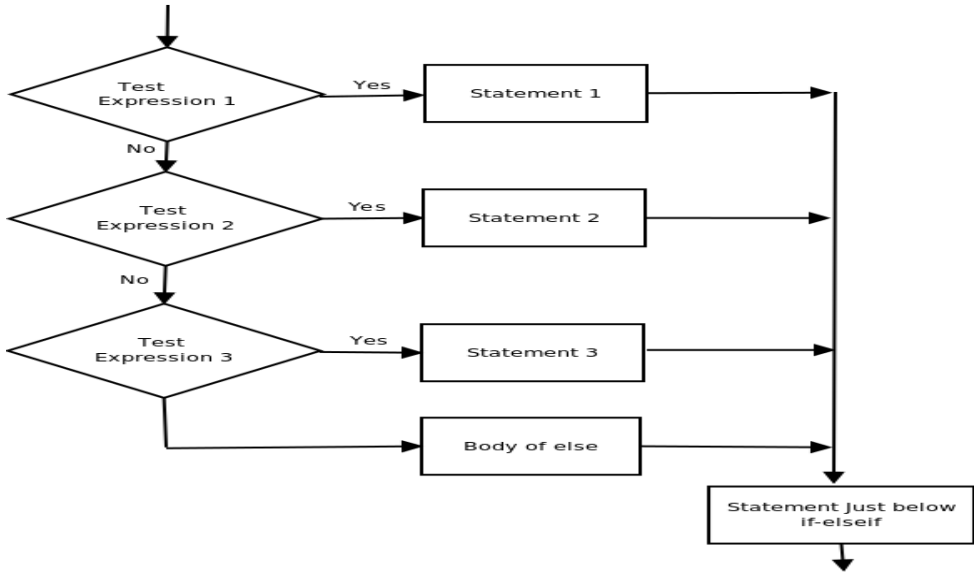
```
Output:
num1 small num2
num1 small num3 also
after end if
```

4. IF THEN ELSIF-THEN-ELSE Ladder

A user can decide among multiple options. The if then statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. Syntax:-

```
if (condition) then
    --statement
elsif (condition) then
    --statement
.
.
else
    --statement
endif
```

Flow Chart:-



Example:-

```
-- pl/sql program to illustrate if-then-elif-then-else ladder
declare
num1 number:= 10;
```

```
num2 number:= 20;
begin
if num1 < num2 then
dbms_output.put_line('num1 small');
ELSEIF num1 = num2 then
dbms_output.put_line('both equal');
ELSE
dbms_output.put_line('num2 greater');
end if;
dbms_output.put_line('after end if');
end;
```

Output:-

```
num1 small
after end if
```

2. Iterative Control (Loops)

Loops repeat a block of statements until a specific condition is met or a set number of iterations is complete.

A. `BASIC LOOP`

Definition: An infinite loop structure that requires an explicit `EXIT` or `EXIT WHEN` statement to terminate.

```
DECLARE
    v_counter NUMBER := 1;
BEGIN
    LOOP
        DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
        v_counter := v_counter + 1;
        EXIT WHEN v_counter > 3; -- Exit after 3 iterations
    END LOOP;
```

```
END;
```

Result (Output):

```
Counter: 1
Counter: 2
Counter: 3
```

B. WHILE LOOP

Definition: The condition is checked *before* each iteration. The loop continues as long as the condition remains `TRUE`

```
DECLARE
    v_counter NUMBER := 1;
BEGIN
    WHILE v_counter <= 3 LOOP
        DBMS_OUTPUT.PUT_LINE('Counter: ' || v_counter);
        v_counter := v_counter + 1;
    END LOOP;
END;
/
```

Result (Output):

```
Counter: 1
Counter: 2
Counter: 3
```

C. FOR LOOP

Definition: Used for a fixed number of iterations. It automatically manages a loop counter within a specified range.

sql

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('Looping from 1 to 3:');
    FOR i IN 1..3 LOOP
        DBMS_OUTPUT.PUT_LINE('Iteration number: ' || i);
    END LOOP;
END;
/
```

Result (Output):

```
Looping from 1 to 3:
```

```
Iteration number: 1
Iteration number: 2
Iteration number: 3
```

3. Sequential Control

These structures alter the standard top-down execution sequence.

A. GOTO

Definition: Transfers control unconditionally to a specific labeled statement within the same program block.

```
DECLARE
    v_count NUMBER := 0;
BEGIN
    <<start_loop>> -- Label definition
    v_count := v_count + 1;
    DBMS_OUTPUT.PUT_LINE('Current count: ' || v_count);

    IF v_count < 3 THEN
        GOTO start_loop; -- Jump back to the label
    END IF;
    DBMS_OUTPUT.PUT_LINE('Finished execution.');
END;
/
```

Result (Output):

```
Current count: 1
Current count: 2
Current count: 3
Finished execution.
```

B. NULL

Definition: A statement that does nothing. It is used as a placeholder in conditional or exception handling blocks where a statement is syntactically required but no action is logically needed.

```
DECLARE
    v_status VARCHAR2(10) := 'INACTIVE';
BEGIN
```

```
    DBMS_OUTPUT.PUT_LINE('Checking status...');
    IF v_status = 'ACTIVE' THEN -- Call some procedure here
        NULL; -- Placeholder
    ELSE
        NULL; -- Do nothing for inactive status
    END IF;
    DBMS_OUTPUT.PUT_LINE('Check complete.');
END;
/
```

Result (Output):

```
Checking status...
Check complete.
```

Cursor

A **cursor** in PL/SQL is a control structure used to traverse and process the rows in a result set one row at a time. It acts as a pointer to a private memory area (context area) where the results of a SQL query are stored.

Cursors are essential for situations where complex logic, conditional updates, or row-by-row data manipulation is required, which is difficult to achieve with standard set-based SQL operations alone

Types of Cursors

PL/SQL supports two types of cursors:

1. **Implicit Cursors**

   Automatic: Automatically created and managed by Oracle for all DML statements (`INSERT`, `UPDATE`, `DELETE`) and single-row `SELECT INTO` statements.

   Limited Control: Programmers have limited control and cannot explicitly open, fetch, or close them. The database engine handles the entire lifecycle.

   Attributes: Information can be accessed using the `SQL` cursor attributes, such as `SQL%ROWCOUNT`, `SQL%FOUND`, `SQL%NOTFOUND`, and `SQL%ISOPEN` (which is always `FALSE` for implicit cursors).

2. Explicit Cursors

- **User-defined:** Declared and managed explicitly by the programmer in the declaration section of a PL/SQL block.
- **Greater Control:** Used for `SELECT` statements that return multiple rows, offering full control over the cursor lifecycle: `DECLARE`, `OPEN`, `FETCH`, and `CLOSE`.
- **Lifecycle:** Involves four distinct steps:

1. **Declaration:** Defining the cursor and its associated `SELECT` query.
2. **Opening:** Executing the query and populating the result set in memory.
3. **Fetching:** Retrieving one row at a time into PL/SQL variables or a record.
4. **Closing:** Releasing the memory and resources associated with the cursor.

Example 1: Implicit Cursor (UPDATE Statement)
This program uses an implicit cursor to give all customers a salary increase and uses the `SQL%ROWCOUNT` attribute to report the number of affected rows.

```
SET SERVEROUTPUT ON;

DECLARE
    total_rows NUMBER;
BEGIN
 -- Implicit cursor is used automatically here
    UPDATE Customers SET Salary = Salary + 500;
-- Get the number of rows affected by the immediately
preceding SQL statement
    total_rows := SQL%ROWCOUNT;
    DBMS_OUTPUT.PUT_LINE(total_rows || ' customers had
their salary updated.');
END;
```

```
/
```

Expected Result

The output in the console will be:

```
5 customers had their salary updated.

PL/SQL procedure successfully completed.
```

If you query the `Customers` table afterward, the salaries will reflect the increase

Example 2: Explicit Cursor (`CURSOR FOR` Loop)

This program uses an explicit cursor with a `FOR` loop, which is the simplest way to process multiple rows. PL/SQL automatically handles the declaration, opening, fetching, and closing

```
SET SERVEROUTPUT ON;

DECLARE

    -- Declare the cursor with the query

    CURSOR c_customers IS SELECT id, name, address FROM
Customers;

BEGIN

    DBMS_OUTPUT.PUT_LINE('--- Customer List ---');

    -- Iterate through the cursor using a FOR loop
(automatic management)

    FOR customer_record IN c_customers LOOP

  DBMS_OUTPUT.PUT_LINE('ID: '||customer_record.ID ||',
Name: ' || customer_record.NAME ||', Address: ' ||
customer_record.ADDRESS);

    END LOOP;

    DBMS_OUTPUT.PUT_LINE('--- End of List ---');

END;

/    Expected Result
```

```
The output in the console will list each customer's
details:
--- Customer List ---
ID: 1, Name: Ramesh, Address: Ahmedabad
ID: 2, Name: Khilan, Address: Delhi
ID: 3, Name: Kaushik, Address: Kota
ID: 4, Name: Chaitali, Address: Mumbai
ID: 5, Name: Hardik, Address: Bhopal
--- End of List ---
PL/SQL procedure successfully completed.
```

Database Triggers and Types

A **PL/SQL trigger** is a stored procedural code that the Oracle engine executes automatically in response to specific events in the database, such as DML statements (INSERT, UPDATE, DELETE), DDL statements (CREATE, ALTER, DROP), or database operations (LOGON, LOGOFF, SHUTDOWN).
Triggers are used to enforce complex business rules, maintain data integrity, provide logging/auditing, and automate related actions.

Trigger Types in PL/SQL

Triggers can be categorized based on their timing, level, and the type of event that fires them.

1.Based on Timing

`BEFORE` Trigger: Fires *before* the triggering event occurs. This is often used for data validation or modifying the data before it is written to the table.

`AFTER` Trigger: Fires *after* the triggering event completes. This is useful for auditing changes, logging events in another table, or performing actions based on the final data state.

`INSTEAD OF` Trigger: Used with views that are not directly updatable. The trigger fires *instead of* the DML statement, allowing you to perform the necessary DML operations on the underlying base tables manually within the trigger body.

2 .Based on Level

Row-Level Trigger: Fires once for *each row* affected by the triggering statement. This type is specified using the `FOR EACH ROW` clause and can access the old and new column values using the `:OLD` and `:NEW` pseudorecords.

Statement-Level Trigger: Fires only once per *triggering statement*, regardless of the number of rows affected. This is efficient for actions that don't depend on individual row data, such as a security check on the time of day a statement is issued.

3.Based on Event Type

DML Triggers: Fire in response to `INSERT`, `UPDATE`, or `DELETE` statements on a table or view.

DDL Triggers: Fire in response to Data Definition Language events like `CREATE`, `ALTER`, or `DROP`.

Database Event Triggers: Fire for database operations such as `LOGON`, `LOGOFF`, `STARTUP`, or `SHUTDOWN`.

Example: A `BEFORE EACH ROW` Trigger

This example demonstrates a `BEFORE EACH ROW` DML trigger that ensures an employee's salary increase does not exceed 10% of their old salary.

```
PL/SQL
CREATE OR REPLACE TRIGGER update_salary_check
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
    -- Check if the new salary is more than 10% greater
than the old salary
    IF :NEW.salary > :OLD.salary * 1.1 THEN
        -- If it is, raise an application error to prevent
the update
        RAISE_APPLICATION_ERROR(-20001, 'Salary increase
cannot exceed 10%');
```

```
    END IF;
END;
/
```

Explanation:

- `CREATE OR REPLACE TRIGGER update_salary_check`: Creates or replaces a trigger named `update_salary_check`.
- `BEFORE UPDATE OF salary ON employees`: Specifies that the trigger fires *before* an `UPDATE` operation on the `salary` column of the `employees` table.
- `FOR EACH ROW`: Indicates a row-level trigger, so it checks each affected row individually.
- `:NEW.salary` and `:OLD.salary`: Access the current (new) and original (old) values of the `salary` column for the row being processed.
- `RAISE_APPLICATION_ERROR`: If the condition is met, this function stops the DML operation and returns a user-defined error message.

Computation done on table Data

SQL provides two primary ways to perform computations on table data: using **arithmetic operators within queries** for row-level calculations, and employing **aggregate functions** for summary calculations across groups of rows.

Sample Data

We'll use a hypothetical Products table with the following structure and data:

Table Structure and Data

ProductID	ProductName	UnitPrice	QuantityInStock	Category
1	Laptop	35000	5	Electronics

2	Keyboard	800	10	Accessories
3	Mouse	500	10	Accessories
4	Monitor	3000	8	Electronics

Sql>CREATE TABLE Products (ProductID INT PRIMARY KEY, ProductName VARCHAR(50),UnitPrice number(5), QuantityInStock INT, Category VARCHAR(50));

Sql>INSERT INTO Products VALUES((1, 'Laptop', 35000, 5, 'Electronics');

Sql>INSERT INTO Products VALUES(2, 'Keyboard', 800, 10, 'Accessories');

Sql>INSERT INTO Products VALUES(3, 'Mouse', 500, 10, 'Accessories');

Sql>INSERT INTO Products VALUES(4, 'Monitor', 3000, 8, 'Electronics');

The GROUP BY Query

This query calculates the total monetary value of all stock within each distinct category.

Sql>SELECT Category, SUM(UnitPrice * QuantityInStock) AS CategoryStockValue FROM Products GROUP BY Category;

Result of the Query

Category	CategoryStockValue
Accessories	

Electronics

1. Using Arithmetic Operators (Row-Level Calculations)

Arithmetic operators (+, -, *, /) can be used in the SELECT clause to perform calculations on each individual row and display the result as a new (calculated) column.

- **Example: Calculate the total value of stock for each product.**
  We multiply the UnitPrice by the QuantityInStock for each row.
  The AS keyword is used to give the new computed column an alias, like StockValue.

SQL>SELECT ProductName,UnitPrice, QuantityInStock,(UnitPrice * QuantityInStock) AS StockValue FROM Products;

Result:

ProductName	UnitPrice	QuantityInStock	StockValue
Laptop	35000	50	60000.00
Keyboard		200	20000.00
Mouse	50.00	150	7500.00

2. Using Aggregate Functions (Summary Calculations)

Aggregate functions (like SUM(), AVG(), COUNT(), MIN(), MAX()) operate on a set of values (a column or a group of columns) and return a single summary value.

- **Example: Calculate the total value of all products in stock.**
  We use the SUM() function on the result of the UnitPrice * QuantityInStock expression.

Sql>SELECT SUM(UnitPrice * QuantityInStock) AS Total_Inventory_Value FROM Products;

Result:

Total_Inventory_Value
87500.00

Example: Find the average unit price and total count of products.

Sql>SELECT AVG(UnitPrice) AS AveragePrice, COUNT(ProductID) AS NumberOfProducts -- *COUNT(*)* FROM Products;

Result:

AveragePrice	NumberOfProducts
450.000000	3

- **Example: Using GROUP BY for grouped computations.**
  If the table had a Category column, we could group by category to get the total stock value per category.

Assuming a 'Category' column exists in the table

SELECT Category,SUM(UnitPrice * QuantityInStock) AS CategoryStockValue FROM Products GROUP BY Category;