**ASSIGNMENT 2 - DBMS**

**Question 1: Top 3 Departments with Highest Average Salary**

**SQL QUERY :**

SELECT

D.DepartmentID,

D.DepartmentName,

AVG(E.Salary) AS AvgSalary

FROM

Departments D

LEFT JOIN

Employees E ON D.DepartmentID = E.DepartmentID

GROUP BY

D.DepartmentID, D.DepartmentName

ORDER BY

AvgSalary DESC

LIMIT 3;

**EXPLANATION:**

LEFT JOIN: The LEFT JOIN ensures that all departments are included in the result set, even those without any employees. When there are no matching records in the Employees table for a department, the E.Salary will be NULL.

AVG(E.Salary): The AVG function calculates the average salary of employees in each department. If a department has no employees, the AVG function will return NULL for that department.

GROUP BY: The GROUP BY clause groups the result set by DepartmentID and DepartmentName, ensuring that the average salary is calculated for each department.

ORDER BY AvgSalary DESC: This sorts the departments by average salary in descending order, so the departments with the highest average salaries come first.

LIMIT 3: This limits the result to the top 3 departments with the highest average salary.

**Handling Departments with No Employees:**

LEFT JOIN ensures that departments with no employees are included in the result.

When a department has no employees, the AVG(E.Salary) function will return NULL because there are no salaries to average.

**SQL STATEMENTS TO CREATE THE TABLE WITH NECESSARIES:**

CREATE TABLE Departments (

DepartmentID NUMBER PRIMARY KEY,

DepartmentName VARCHAR2(100) NOT NULL

);

CREATE TABLE Employees (

EmployeeID NUMBER PRIMARY KEY,

EmployeeName VARCHAR2(100) NOT NULL,

Salary NUMBER,

DepartmentID NUMBER,

CONSTRAINT fk\_department

FOREIGN KEY (DepartmentID)

REFERENCES Departments(DepartmentID)

);

INSERT INTO Departments (DepartmentID, DepartmentName) VALUES (1, 'HR');

INSERT INTO Departments (DepartmentID, DepartmentName) VALUES (2, 'Finance');

INSERT INTO Departments (DepartmentID, DepartmentName) VALUES (3, 'Engineering');

INSERT INTO Departments (DepartmentID, DepartmentName) VALUES (4, 'Marketing');

INSERT INTO Employees (EmployeeID, EmployeeName, Salary, DepartmentID) VALUES (1, 'John Doe', 50000, 1);

INSERT INTO Employees (EmployeeID, EmployeeName, Salary, DepartmentID) VALUES (2, 'Jane Smith', 60000, 1);

INSERT INTO Employees (EmployeeID, EmployeeName, Salary, DepartmentID) VALUES (3, 'Alice Johnson', 75000, 2);

INSERT INTO Employees (EmployeeID, EmployeeName, Salary, DepartmentID) VALUES (4, 'Bob Brown', 55000, 3);

INSERT INTO Employees (EmployeeID, EmployeeName, Salary, DepartmentID) VALUES (5, 'Charlie Davis', 70000, 3);

INSERT INTO Employees (EmployeeID, EmployeeName, Salary, DepartmentID) VALUES (6, 'Diana Evans', 80000, NULL);

**Question 2: Retrieving Hierarchical Category Paths**

**SQL QUERY :**

WITH RECURSIVE CategoryHierarchy AS (

SELECT

CategoryID,

CategoryName,

ParentCategoryID,

CategoryName AS HierarchicalPath

FROM

Categories

WHERE

ParentCategoryID IS NULL

UNION ALL

SELECT

c.CategoryID,

c.CategoryName,

c.ParentCategoryID,

ch.HierarchicalPath || ' > ' || c.CategoryName AS HierarchicalPath

FROM

Categories c

INNER JOIN

CategoryHierarchy ch ON c.ParentCategoryID = ch.CategoryID

)

SELECT

CategoryID,

CategoryName,

HierarchicalPath

FROM

CategoryHierarchy

ORDER BY

HierarchicalPath;

**Explanation:**

To retrieve hierarchical category paths using recursive Common Table Expressions (CTEs) in SQL, we'll use a typical parent-child relationship in the categories table. Let's assume the Categories table has the following structure:

* CategoryID: Unique identifier for each category.
* CategoryName: Name of the category.
* ParentCategoryID: Identifier of the parent category (null if it's a root category).

**Common Table Expression (CTE) Definition:**

The CTE CategoryHierarchy is defined with WITH RECURSIVE to enable recursion.

Anchor Member:

The anchor member selects the root categories, which have no parent (ParentCategoryID IS NULL).

It initializes the HierarchicalPath with the CategoryName.

Recursive Member:

The recursive member joins the Categories table with the CTE CategoryHierarchy on the ParentCategoryID.

For each child category, it concatenates the current path (HierarchicalPath) with the child category's name using the || operator.

Final Select Statement:

The final SELECT retrieves all columns from the CTE.

The result is ordered by the HierarchicalPath for easier readability.

This query will generate a hierarchical path for each category, starting from the root and including all its subcategories. The hierarchical path is built by concatenating category names with ' > ' as a separator.

**Example Data and Output**

**Example Categories Table:**

CategoryID CategoryName ParentCategoryID

1 Electronics NULL

2 Computers 1

3 Laptops 2

4 Desktops 2

5 Mobile Phones 1

**Example Output:**

CategoryID CategoryName HierarchicalPath

1 Electronics Electronics

2 Computers Electronics > Computers

3 Laptops Electronics > Computers > Laptops

4 Desktops Electronics > Computers > Desktops

5 Mobile Phones Electronics > Mobile Phones

This query and explanation should help you understand how recursive CTEs work to traverse and display hierarchical data in a readable format.

**SQL COMMAND TO CREATE :**

CREATE TABLE Categories (

CategoryID NUMBER PRIMARY KEY,

CategoryName VARCHAR2(100) NOT NULL,

ParentCategoryID NUMBER,

CONSTRAINT fk\_parent\_category FOREIGN KEY (ParentCategoryID) REFERENCES Categories(CategoryID)

);

INSERT INTO Categories (CategoryID, CategoryName, ParentCategoryID) VALUES (1, 'Electronics', NULL);

INSERT INTO Categories (CategoryID, CategoryName, ParentCategoryID) VALUES (2, 'Furniture', NULL);

INSERT INTO Categories (CategoryID, CategoryName, ParentCategoryID) VALUES (3, 'Computers', 1);

INSERT INTO Categories (CategoryID, CategoryName, ParentCategoryID) VALUES (4, 'Mobile Phones', 1);

INSERT INTO Categories (CategoryID, CategoryName, ParentCategoryID) VALUES (5, 'Chairs', 2);

INSERT INTO Categories (CategoryID, CategoryName, ParentCategoryID) VALUES (6, 'Tables', 2);

INSERT INTO Categories (CategoryID, CategoryName, ParentCategoryID) VALUES (7, 'Laptops', 3);

INSERT INTO Categories (CategoryID, CategoryName, ParentCategoryID) VALUES (8, 'Desktops', 3);

INSERT INTO Categories (CategoryID, CategoryName, ParentCategoryID) VALUES (9, 'Smartphones', 4);

INSERT INTO Categories (CategoryID, CategoryName, ParentCategoryID) VALUES (10, 'Feature Phones', 4);

WITH CategoryHierarchy AS (

SELECT

CategoryID,

CategoryName,

ParentCategoryID,

CategoryName AS HierarchicalPath

FROM

Categories

WHERE

ParentCategoryID IS NULL

UNION ALL

SELECT

c.CategoryID,

c.CategoryName,

c.ParentCategoryID,

ch.HierarchicalPath || ' > ' || c.CategoryName AS HierarchicalPath

FROM

Categories c

INNER JOIN

CategoryHierarchy ch ON c.ParentCategoryID = ch.CategoryID

)

SELECT

CategoryID,

CategoryName,

HierarchicalPath

FROM

CategoryHierarchy

ORDER BY

HierarchicalPath;

**Question 3: Total Distinct Customers by Month**

**SQL QUERY :**

WITH Months AS (

SELECT LEVEL AS MonthNumber,

TO\_CHAR(TO\_DATE(LEVEL, 'MM'), 'Month') AS MonthName

FROM DUAL

CONNECT BY LEVEL <= 12

),

CustomerActivity AS (

SELECT

EXTRACT(MONTH FROM PurchaseDate) AS MonthNumber,

COUNT(DISTINCT CustomerID) AS CustomerCount

FROM

Purchases

WHERE

EXTRACT(YEAR FROM PurchaseDate) = EXTRACT(YEAR FROM SYSDATE)

GROUP BY

EXTRACT(MONTH FROM PurchaseDate)

)

SELECT

m.MonthName,

NVL(c.CustomerCount, 0) AS CustomerCount

FROM

Months m

LEFT JOIN

CustomerActivity c ON m.MonthNumber = c.MonthNumber

ORDER BY

m.MonthNumber;

**Explanation**

Months CTE:

The Months CTE generates a list of all months (1 through 12) with their respective names. LEVEL and CONNECT BY LEVEL <= 12 are used to generate numbers 1 to 12, which represent the months.

TO\_CHAR(TO\_DATE(LEVEL, 'MM'), 'Month') converts the month numbers to their respective month names.

CustomerActivity CTE:

The CustomerActivity CTE calculates the count of distinct customers who made purchases each month in the current year.

EXTRACT(MONTH FROM PurchaseDate) extracts the month from the PurchaseDate.

EXTRACT(YEAR FROM PurchaseDate) = EXTRACT(YEAR FROM SYSDATE) filters the purchases to include only those made in the current year.

COUNT(DISTINCT CustomerID) counts the distinct customers for each month.

Final Select Statement:

The final SELECT joins the Months CTE with the CustomerActivity CTE using a LEFT JOIN.

NVL(c.CustomerCount, 0) ensures that months with no customer activity show a count of 0.

The result is ordered by MonthNumber to ensure the months appear in chronological order.

**SQL COMMAND TO CREATE TABLE :**

CREATE TABLE Customers (

CustomerID NUMBER PRIMARY KEY,

CustomerName VARCHAR2(100) NOT NULL

);

CREATE TABLE Purchases (

PurchaseID NUMBER PRIMARY KEY,

CustomerID NUMBER NOT NULL,

PurchaseDate DATE NOT NULL,

CONSTRAINT fk\_customer FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

**TO INSERT :**

INSERT INTO Customers (CustomerID, CustomerName) VALUES (1, 'John Doe');

INSERT INTO Customers (CustomerID, CustomerName) VALUES (2, 'Jane Smith');

INSERT INTO Customers (CustomerID, CustomerName) VALUES (3, 'Alice Johnson');

INSERT INTO Customers (CustomerID, CustomerName) VALUES (4, 'Bob Brown');

INSERT INTO Purchases (PurchaseID, CustomerID, PurchaseDate) VALUES (1, 1, DATE '2024-01-15');

INSERT INTO Purchases (PurchaseID, CustomerID, PurchaseDate) VALUES (2, 2, DATE '2024-02-20');

INSERT INTO Purchases (PurchaseID, CustomerID, PurchaseDate) VALUES (3, 1, DATE '2024-03-05');

INSERT INTO Purchases (PurchaseID, CustomerID, PurchaseDate) VALUES (4, 3, DATE '2024-03-25');

INSERT INTO Purchases (PurchaseID, CustomerID, PurchaseDate) VALUES (5, 2, DATE '2024-04-10');

INSERT INTO Purchases (PurchaseID, CustomerID, PurchaseDate) VALUES (6, 4, DATE '2024-05-14');

**Question 4: Finding Closest Locations**

**SQL QUERY :**

DECLARE

given\_lat NUMBER := 37.7749; -- example latitude

given\_lon NUMBER := -122.4194; -- example longitude

BEGIN

WITH DistanceCalculation AS (

SELECT

LocationID,

LocationName,

Latitude,

Longitude,

6371 \* ACOS(

COS(RADIANS(given\_lat)) \* COS(RADIANS(Latitude)) \* COS(RADIANS(Longitude) - RADIANS(given\_lon)) +

SIN(RADIANS(given\_lat)) \* SIN(RADIANS(Latitude))

) AS Distance

FROM

Locations

)

SELECT

LocationID,

LocationName,

Latitude,

Longitude,

Distance

FROM

DistanceCalculation

ORDER BY

Distance

FETCH FIRST 5 ROWS ONLY;

END;

**Explanation:**

The DECLARE block initializes the given latitude and longitude.

The DistanceCalculation CTE calculates the distance from the given point to each location using the Haversine formula.

The final SELECT statement retrieves the top 5 closest locations sorted by distance using the ORDER BY clause and limits the result to 5 rows with FETCH FIRST 5 ROWS ONLY.

**SQL COMMANDS TO CREATE TABLE :**

CREATE TABLE Locations (

LocationID NUMBER PRIMARY KEY,

LocationName VARCHAR2(100) NOT NULL,

Latitude NUMBER(9,6) NOT NULL,

Longitude NUMBER(9,6) NOT NULL

);

**TO INSERT :**

INSERT INTO Locations (LocationID, LocationName, Latitude, Longitude) VALUES (1, 'Location A', 37.7749, -122.4194);

INSERT INTO Locations (LocationID, LocationName, Latitude, Longitude) VALUES (2, 'Location B', 34.0522, -118.2437);

INSERT INTO Locations (LocationID, LocationName, Latitude, Longitude) VALUES (3, 'Location C', 40.7128, -74.0060);

INSERT INTO Locations (LocationID, LocationName, Latitude, Longitude) VALUES (4, 'Location D', 51.5074, -0.1278);

INSERT INTO Locations (LocationID, LocationName, Latitude, Longitude) VALUES (5, 'Location E', 48.8566, 2.3522);

INSERT INTO Locations (LocationID, LocationName, Latitude, Longitude) VALUES (6, 'Location F', 35.6895, 139.6917);

INSERT INTO Locations (LocationID, LocationName, Latitude, Longitude) VALUES (7, 'Location G', 55.7558, 37.6173);

INSERT INTO Locations (LocationID, LocationName, Latitude, Longitude) VALUES (8, 'Location H', 28.7041, 77.1025);

INSERT INTO Locations (LocationID, LocationName, Latitude, Longitude) VALUES (9, 'Location I', -33.8688, 151.2093);

INSERT INTO Locations (LocationID, LocationName, Latitude, Longitude) VALUES (10, 'Location J', -23.5505, -46.6333);

**Question 5: Optimizing Query for Orders Table**

**SQL QUERY :**

SELECT

OrderID,

CustomerID,

OrderDate,

TotalAmount

FROM

Orders

WHERE

OrderDate >= SYSDATE - 7

ORDER BY

OrderDate DESC;

**EXPLANATION :**

**Discussion of Strategies to Optimize the Query**

Indexing:

Create an index on the OrderDate column to speed up the retrieval of records based on the date filter. Indexes improve the performance of queries that filter on specific columns.

Composite indexes could also be considered if multiple columns are frequently used in conjunction (e.g., OrderDate and CustomerID).

Partitioning:

For very large tables, consider partitioning the table based on the OrderDate column. This technique helps by breaking the table into smaller, more manageable pieces (partitions), each of which can be queried more efficiently.

Use Efficient Date Functions:

Ensure the date function used in the WHERE clause is efficient. In the example, SYSDATE - 7 is used to get the date 7 days ago. This calculation is efficient and avoids unnecessary function calls.

Limit the Result Set:

If you only need a subset of the data (e.g., top 1000 records), you can limit the result set to reduce the load on the database.

Query Rewriting:

Ensure the query is written in a way that the database optimizer can generate the most efficient execution plan. The query provided is straightforward and leverages an indexed column in the WHERE clause.

Analyze and Gather Statistics:Regularly gather statistics on the table and indexes to help the optimizer choose the best execution plan.

**Summary of Optimizations**

Indexing: Creating an index on the OrderDate column ensures that the query can quickly locate the relevant rows.

Partitioning: For extremely large tables, partitioning by OrderDate can further enhance performance.

Efficient Date Functions: Using efficient date functions like SYSDATE - 7 helps avoid unnecessary computations.

Limiting Result Set: If applicable, limiting the number of rows returned reduces the amount of data processed.

Query Rewriting: Ensuring the query is straightforward helps the optimizer generate an efficient execution plan.

Statistics: Regularly gathering statistics ensures the optimizer has the information needed to choose the best plan.

**SQL COMMANDS TO CREATE TABLE:**

CREATE TABLE Orders (

OrderID NUMBER PRIMARY KEY,

CustomerID NUMBER NOT NULL,

OrderDate DATE NOT NULL,

TotalAmount NUMBER NOT NULL

);

**TO INSERT :**

INSERT INTO Orders (OrderID, CustomerID, OrderDate, TotalAmount) VALUES (1, 100, TO\_DATE('2024-07-21', 'YYYY-MM-DD'), 150.00);

INSERT INTO Orders (OrderID, CustomerID, OrderDate, TotalAmount) VALUES (2, 101, TO\_DATE('2024-07-22', 'YYYY-MM-DD'), 200.00);

INSERT INTO Orders (OrderID, CustomerID, OrderDate, TotalAmount) VALUES (3, 102, TO\_DATE('2024-07-23', 'YYYY-MM-DD'), 250.00);

INSERT INTO Orders (OrderID, CustomerID, OrderDate, TotalAmount) VALUES (4, 103, TO\_DATE('2024-07-24', 'YYYY-MM-DD'), 300.00);

INSERT INTO Orders (OrderID, CustomerID, OrderDate, TotalAmount) VALUES (5, 104, TO\_DATE('2024-07-25', 'YYYY-MM-DD'), 350.00);

INSERT INTO Orders (OrderID, CustomerID, OrderDate, TotalAmount) VALUES (6, 105, TO\_DATE('2024-07-26', 'YYYY-MM-DD'), 400.00);

INSERT INTO Orders (OrderID, CustomerID, OrderDate, TotalAmount) VALUES (7, 106, TO\_DATE('2024-07-27', 'YYYY-MM-DD'), 450.00);