ASSIGNMENT-02

Question 1: Top 3 Departments with Highest Average Salary

TABLE:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| EMPLOYEE ID | EMPLOYEE NAME | DEPARTMENT NAME | DEPARTMENT  ID | SALARY |
| 192311 | SARATHA | IT | 1 | 50,000 |
| 192313 | SHANKAR | MARKETING | 2 | 85,000 |
| 192314 | RAVI | SALES | 3 | 70,000 |
| 192312 | JAYANTHI | HR | 4 | 1,00,000 |

QUERY FOR CREATING TABLE:

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

EmployeeName VARCHAR(50),

DepartmentName VARCHAR(50),

DepartmentID INT

);

DELIVERABLES:

1. SQL QUERY:

SELECT

d.DepartmentID,

d.DepartmentName,

AVG(s.Salary) AS AvgSalary

FROM

Employees e

JOIN

Departments d ON e.DepartmentID = d.DepartmentID

JOIN

Salaries s ON e.EmployeeID = s.EmployeeID

GROUP BY

d.DepartmentID, d.DepartmentName

ORDER BY

AvgSalary DESC

LIMIT 3;

1. EXPLANATION:

* **LEFT JOIN:** This ensures that all departments are included in the result set, even those with no matching employees in the Employee table. If there is no matching employee, the salary will be NULL.
* **AVG(e.SALARY)**: This calculates the average salary for each department. Departments with no employees will have a NULL value for the average salary.
* **GROUP BY d.DEPARTMENT\_ID, d.DEPARTMENT\_NAME**: This groups the results by department, allowing us to calculate the average salary for each department.
* **ORDER BY d.DEPARTMENT\_ID**: This orders the results by department ID for better readability.

**RESULT TABLE**:

Top 3 department

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Employee id | Employee name | Department name | Department id | Salary |
| 192312 | JAYANTHI | HR | 4 | 1,00,00 |
| 192313 | SHANKAR | MARKETING | 2 | 85,000 |
| 192314 | RAVI | SALES | 3 | 70,000 |
| 192311 | SARATHA | IT | 1 | 50,000 |

Question 2: Retrieving Hierarchical Category Paths

TABLE:

|  |  |  |
| --- | --- | --- |
| CATEGORY\_ID | CATEGORY\_NAME | PARENT\_ID |
| 1 | ELECTRONICS | NULL |
| 2 | COMPUTERS | 1 |
| 3 | LAPTOPS | 2 |
| 4 | DESKTOPS | 2 |
| 5 | MOBILES | 1 |
| 6 | SMART PHONES | 5 |
| 7 | ACCESSORIES | 1 |

QUERY FOR TABLE:

Create the Category table

CREATE TABLE Category (

CATEGORY\_ID INT PRIMARY KEY,

CATEGORY\_NAME VARCHAR(255) NOT NULL,

PARENT\_ID INT,

FOREIGN KEY (PARENT\_ID) REFERENCES Category(CATEGORY\_ID)

);

Insert data into the Category table

INSERT INTO Category (CATEGORY\_ID, CATEGORY\_NAME, PARENT\_ID) VALUES

(1, 'Electronics', NULL),

(2, 'Computers', 1),

(3, 'Laptops', 2),

(4, 'Desktops', 2),

(5, 'Mobiles', 1),

(6, 'Smartphones', 5),

(7, 'Accessories', 1);

DELIVERABLES:

1. SQL QUERY:

WITH RECURSIVE CategoryHierarchy AS (

-- Anchor member: select top-level categories

SELECT

CATEGORY\_ID,

CATEGORY\_NAME,

PARENT\_ID,

CATEGORY\_NAME AS FullPath

FROM

Category

WHERE

PARENT\_ID IS NULL

UNION ALL

-- Recursive member: join the CTE with the table to find child categories

SELECT

c.CATEGORY\_ID,

c.CATEGORY\_NAME,

c.PARENT\_ID,

CONCAT(ch.FullPath, ' > ', c.CATEGORY\_NAME) AS FullPath

FROM

Category c

INNER JOIN

CategoryHierarchy ch

ON

c.PARENT\_ID = ch.CATEGORY\_ID

)

SELECT

CATEGORY\_ID,

CATEGORY\_NAME,

FullPath

FROM

CategoryHierarchy

ORDER BY

FullPath;

1. EXPLANATION:

**How the Recursive CTE Works**

1. **Anchor Member**:
   * The first part of the CTE (Anchor member) selects the top-level categories where PARENT\_ID IS NULL.
   * It initializes the FullPath column with the CATEGORY\_NAME.
2. **Recursive Member**:

* The second part of the CTE (Recursive member) joins the original table with the CTE itself to find child categories.
* For each row in the CategoryHierarchy CTE, it finds rows in the Category table where PARENT\_ID matches CATEGORY\_ID.
* It concatenates the FullPath of the parent category with the current CATEGORY\_NAME
* The process repeats, finding subcategories and building the full path for each one. The recursive member runs until no more child categories are found.

**Step-by-Step Traversal**:

**First Iteration**:

* + Parent: Electronics (CATEGORY\_ID = 1)
  + Children: Computers, Mobiles, Accessories

**Second Iteration**:

* + Parent: Computers (CATEGORY\_ID = 2)
  + Children: Laptops, Desktops
  + Parent: Mobiles (CATEGORY\_ID = 5)
  + Children: Smartphones

**Third Iteration**:

* + Parent: Laptops (CATEGORY\_ID = 3) - No further children
  + Parent: Desktops (CATEGORY\_ID = 4) - No further children
  + Parent: Smartphones (CATEGORY\_ID = 6) - No further children

**3.Final SELECT**:

* Retrieves all the results from the CategoryHierarchy CTE.
* Orders the results by FullPath for better readability.

**RESULT TABLE:**

|  |  |  |
| --- | --- | --- |
| CATEGORY\_ID | CATEGORY\_NAME | FULL PATH |
| 1 | Electronics | Electronics |
| 2 | Computers | Electronics>Computers |
| 3 | Desktop | Electronics>Computers>Desktops |
| 4 | Laptop | Electronics>Computers>Laptops |
| 5 | Mobiles | Electronics>Mobiles |
| 6 | Smart phones | Electronics>Mobiles>Smartphones |
| 7 | accessories | Electronics>Accessories |

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

TABLE:

|  |  |  |
| --- | --- | --- |
| ID | CUSTOMER\_ID | PURCHASE DATE |
| 192311 | 1 | 01-04-2024 |
| 192312 | 2 | 04-04-2024 |
| 192313 | 3 | 15-04-2024 |
| 192314 | 4 | 20-04-2024 |

QUERY FOR CREATING TABLE:

CREATE TABLE purchases (

id INT PRIMARY KEY,

customer\_id INT,

purchase\_date DATE

);

INSERT INTO purchases (id, customer\_id, purchase\_date) VALUES

(192311, 1, '2024-04-01'),

(192312, 2, '2024-04-04'),

(192313, 3, '2024-04-15'),a

(192314, 4, '2024-04-20');

DELIVERABLES:

1. QUERY:

WITH RECURSIVE Months AS (

SELECT

DATE\_FORMAT(CURDATE(), '%Y-01-01') AS month\_start

UNION ALL

SELECT

DATE\_ADD(month\_start, INTERVAL 1 MONTH)

FROM

Months

WHERE

month\_start < DATE\_FORMAT(CURDATE(), '%Y-12-01')

)

SELECT

DATE\_FORMAT(Months.month\_start, '%Y-%m') AS month,

COALESCE(COUNT(DISTINCT purchases.customer\_id), 0) AS distinct\_customers

FROM

Months

LEFT JOIN

purchases ON DATE\_FORMAT(purchases.purchase\_date, '%Y-%m') = DATE\_FORMAT(Months.month\_start, '%Y-%m')

GROUP BY

Months.month\_start

ORDER BY

Months.month\_start;

1. EXPLANATION:

**CTE Months:**

This Common Table Expression generates a list of the first day of each month in the current year.

* DATE\_FORMAT(CURDATE(), '%Y-01-01'): Generates the first day of January of the current year.
* DATE\_ADD(month\_start, INTERVAL 1 MONTH): Adds one month to the previous month.
* The WHERE clause ensures we stop at December of the current year.

**Main Query**:

* LEFT JOIN: Joins the generated list of months with the purchases table on the month part of the date.
* COUNT(DISTINCT purchases.customer\_id): Counts the distinct customers for each month.
* COALESCE(..., 0): Ensures months with no purchases show a count of 0.
* GROUP BY and ORDER BY: Groups the results by month and orders them chronologically.

RESULT TABLE:

|  |  |
| --- | --- |
| MONTH | DISTINCT CUSTOMER |
| 01-04-2024 | 0 |
| 02-04-2024 | 0 |
| 03-04-2024 | 0 |
| 04-04-2024 | 4 |
| 05-04-2024 | 0 |
| 06-04-2024 | 0 |
| 07-04-2024 | 0 |
| 08-04-2024 | 0 |
| 09-04-2024 | 0 |
| 10-04-2024 | 0 |
| 11-04-2024 | 0 |
| 12-04-2024 | 0 |

**Question 4: Finding Closest Locations**

**TABLE:**

|  |  |  |  |
| --- | --- | --- | --- |
| ID | NAME | LATITUDE | LONGITUDE |
| 1 | LOCATION A | 40.1728 | -0.8796 |
| 2 | LOCATION B | 05.1379 | -6.7895 |
| 3 | LOCATION C | 03.3478 | -3.4567 |
| 4 | LOCATION D | 54.1234 | -2.4567 |
| 5 | LOCATION E | 76.9768 | 2.3522 |
| 6 | LOCATION F | 18.5678 | 139.651 |

QUERY FOR CREATING TABLE:

Create the table

CREATE TABLE Locations (

ID INT PRIMARY KEY,

NAME VARCHAR(50),

LATITUDE DECIMAL(8, 4),

LONGITUDE DECIMAL(8, 4)

);

Insert the data

INSERT INTO Locations (ID, NAME, LATITUDE, LONGITUDE) VALUES

(1, 'LOCATION A', 40.1728, -0.8796),

(2, 'LOCATION B', 5.1379, -6.7895),

(3, 'LOCATION C', 3.3478, -3.4567),

(4, 'LOCATION D', 54.1234, -2.4567),

(5, 'LOCATION E', 76.9768, 2.3522),

(6, 'LOCATION F', 18.5678, 139.651);

DELIVERABLES:

* 1. QUERY:

WITH LocationDistances AS (

SELECT

l1.ID AS LocationID1,

l1.NAME AS LocationName1,

l1.LATITUDE AS Latitude1,

l1.LONGITUDE AS Longitude1,

l2.ID AS LocationID2,

l2.NAME AS LocationName2,

l2.LATITUDE AS Latitude2,

l2.LONGITUDE AS Longitude2,

111.045 \* DEGREES(ACOS(COS(RADIANS(l1.LATITUDE))

\* COS(RADIANS(l2.LATITUDE))

\* COS(RADIANS(l1.LONGITUDE) –

RADIANS(l2.LONGITUDE))

+ SIN(RADIANS(l1.LATITUDE))

\* SIN(RADIANS(l2.LATITUDE)))) AS distance

FROM

Locations l1

CROSS JOIN

Locations l2

WHERE

l1.ID != l2.ID

)

SELECT

LocationID2 AS LocationID,

LocationName2 AS Name,

Latitude2 AS Latitude,

Longitude2 AS Longitude,

distance

FROM

LocationDistances

ORDER BY

distance

LIMIT 5;

* 1. EXPLANATION:

### Haversine Formula:

The Haversine formula is an equation that can be used to find the shortest distance between two points on a sphere, given their longitudes and latitudes. This distance is along the surface of the sphere.

### Steps to Calculate Proximity:

1. **Convert Degrees to Radians:** Convert the latitude and longitude values from degrees to radians, as trigonometric functions in the Haversine formula require radians.
2. **Calculate Differences:** Compute the differences between the latitudes and longitudes of the two points.
3. **Apply Haversine Formula:** Use the Haversine formula to calculate the distance between the two points.
4. **Compare Distances:** Repeat the above steps for all pairs of locations to determine the proximity of each location to the others.

RESULT:

|  |  |  |  |
| --- | --- | --- | --- |
| ID | NAME | LATITUDE | LONGITUDE |
| 1 | LOCATION A | 40.1728 | -0.8796 |
| 2 | LOCATION B | 05.1379 | -6.7895 |
| 3 | LOCATION C | 03.3478 | -3.4567 |
| 4 | LOCATION D | 54.1234 | -2.4567 |
| 5 | LOCATION E | 76.9768 | 2.3522 |
| 6 | LOCATION F | 18.5678 | 139.65 |

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

**TABLE:**

|  |  |  |  |
| --- | --- | --- | --- |
| ORDER ID | CUSTOMER ID | ORDER DATE | ORDER AMOUNT |
| 105 | 1001 | 27-07-2024 | 150.00 |
| 104 | 1002 | 26-07-2024 | 165.00 |
| 103 | 1003 | 25-07-2024 | 180.00 |
| 102 | 1004 | 24-07-2024 | 185.00 |
| 101 | 1005 | 23-07-2024 | 170.00 |
| 100 | 1006 | 22-07-2024 | 140.00 |
| 99 | 1007 | 21-07-2024 | 145.00 |

QUERY FOR TABLE:

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

OrderDate DATE,

OrderAmount DECIMAL(10, 2)

);

INSERT INTO Orders (OrderID, CustomerID, OrderDate, OrderAmount) VALUES

(105, 1001, '2024-07-27', 150.00),

(104, 1002, '2024-07-26', 200.00),

(103, 1003, '2024-07-25', 300.00),

(102, 1004, '2024-07-24', 250.00),

(101, 1005, '2024-07-23', 100.00),

(100, 1006, '2024-07-22', 175.00),

(99, 1007, '2024-07-21', 225.00);

DELIVERABLES:

1. QUERY:

SELECT OrderID, CustomerID, OrderDate, OrderAmount

FROM Orders

WHERE OrderDate >= DATE\_SUB(CURDATE(), INTERVAL 7 DAY)

ORDER BY OrderDate DESC;

1. EXPLANATION:

### Discussion on Optimization Strategies

* **Indexing**: The creation of an index on the OrderDate column will allow the database to quickly locate the rows corresponding to the desired date range, reducing the need for a full table scan.
* **Limiting Columns Selected**: Selecting only the necessary columns minimizes data transfer and processing overhead.

**Partitioning**:

By partitioning the table, the database can skip entire partitions that do not match the date range condition, thus speeding up the query.

* **Use of EXPLAIN**: Analyzing the query execution plan helps identify inefficient operations and provides insights for further optimization.
* **Query Caching**: Implementing caching mechanisms can store frequently accessed data, reducing the load on the database and improving response times for repeated queries.
* **Proper Data Types**: Using appropriate data types for columns ensures that the database can efficiently manage and query the data.

RESULT:

|  |  |  |  |
| --- | --- | --- | --- |
| ORDER ID | CUSTOMER ID | ORDER DATE | ORDER AMOUNT |
| 105 | 1001 | 27-07-2024 | 150.00 |
| 104 | 1002 | 26-07-2024 | 200.00 |
| 103 | 1003 | 25-07-2024 | 300.00 |
| 102 | 1004 | 24-07-2024 | 250.00 |
| 101 | 1005 | 23-07-2024 | 100.00 |
| 100 | 1006 | 22-07-2024 | 175.00 |
| 99 | 1007 | 21-07-2024 | 225.00 |