**DBMS-ASSIGNMENT**

Name: K. Sravanthi

Reg.no: 192372127

1. **ER Diagram Question:**

**Traffic Flow Management System (TFMS)**

**Scenario**

You are tasked with designing an Entity-Relationship (ER) diagram for a Traffic Flow Management System (TFMS) used in a city to optimize traffic routes, manage intersections, and control traffic signals. The TFMS aims to enhance transportation efficiency by utilizing real-time data from sensors and historical traffic patterns.

**ER diagram for the Traffic Flow Management System (TFMS):**

**Entities and Attributes**

**Roads**

- RoadID (PK)

- RoadName

- Length (in meters)

- SpeedLimit (in km/h)

**Intersections**

- IntersectionID (PK)

- IntersectionName

- Latitude

- Longitude

**Traffic Signals**

- SignalID (PK)

- IntersectionID (FK referencing Intersections)

- SignalStatus (Green, Yellow, Red)

- Timer (countdown to next change)

**Traffic Data**

- TrafficDataID (PK)

- RoadID (FK referencing Roads)

- Timestamp

- Speed (average speed on the road)

- Congestion Level (degree of traffic congestion)

**Relationships**

A road can have multiple intersections (one-to-many): Roads -> Intersections

**Cardinality**: 1: N

**Optionality:** Mandatory (a road must have at least one intersection)

An intersection can have multiple roads (many-to-many): Intersections -> Roads

**Cardinality:** M: N

**Optionality:** Mandatory (an intersection must have at least one road)

An intersection can have one traffic signal (one-to-one): Intersections -> Traffic Signals

**Cardinality:** 1:1

**Optionality:** Optional (an intersection may not have a traffic signal)

A traffic signal is associated with one intersection (one-to-one): Traffic Signals -> Intersections

**Cardinality:** 1:1

**Optionality:** Mandatory (a traffic signal must be associated with an intersection)

A road can have multiple traffic data points (one-to-many): Roads -> Traffic Data

**Cardinality:** 1: N

**Optionality:** Mandatory (a road must have at least one traffic data point)

**Normalization Considerations**

**To ensure the ER diagram adheres to normalization principles, we can apply the following rules:**

**First Normal Form (1NF):** Each table cell must contain a single value. This is already satisfied in the ER diagram**.**

**Second Normal Form (2NF):** Each non-key attribute in a table must depend on the entire primary key. This is satisfied since each attribute in each entity depends on the primary key of that entity.

**Third Normal Form (3NF):** If a table is in 2NF, and a non-key attribute depends on another non-key attribute, then it should be moved to a separate table. This is not applicable in this ER diagram since there are no transitive dependencies.

+---------------+

| Roads |

+---------------+

| RoadID (PK) |

| RoadName |

| Length |

| SpeedLimit |

+---------------+

|

| 1: N

v

+---------------+

| Intersections |

+---------------+

| IntersectionID |

| IntersectionName|

| Latitude |

| Longitude |

+---------------+

|

| 1:1

v

+---------------+

| Traffic Signals|

+---------------+

| SignalID (PK) |

| IntersectionID |

| Signal Status |

| Timer |

+---------------+

|

| 1: N

v

+---------------+

| Traffic Data |

+---------------+

| TrafficDataID |

| RoadID |

| Timestamp |

| Speed |

| CongestionLevel|

+---------------+

**The relationships between the entities are represented by the lines connecting them:**

* A road can have multiple intersections (one-to-many): Roads -> Intersections
* An intersection can have multiple roads (many-to-many): Intersections -> Roads
* An intersection can have one traffic signal (one-to-one): Intersections -> Traffic Signals
* A traffic signal is associated with one intersection (one-to-one): Traffic Signals -> Intersections
* A road can have multiple traffic data points (one-to-many): Roads -> Traffic Data

**Here is a key to the notation used:**

* (PK) indicates a primary key
* 1: N indicates a one-to-many relationship
* 1:1 indicates a one-to-one relationship
* M: N indicates a many-to-many relationship
* indicates a mandatory relationship (i.e., a relationship that must exist)
* indicates an optional relationship (i.e., a relationship that may or may not exist)



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

**Task:**

* + - 1. **Write a SQL query to find the top 3 departments with the highest average salary of employees. Ensure departments with no employees show an average salary of NULL.**

**SQL Query**

WITH department\_salaries AS (

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

)

SELECT

DepartmentID,

DepartmentName,

AvgSalary

FROM

department\_salaries

ORDER BY

AvgSalary DESC

LIMIT 3;

**Explanation:**

This SQL query finds the top 3 departments with the highest average salary of employees. It does this by:

* Joining the **Departments** table with the **Employees** table on the **DepartmentID** column
* Calculating the average salary for each department using the **AVG** function
* Including departments with no employees in the result set, with a **NULL** value for average salary
* Sorting the result set in descending order by average salary
* Returning only the top 3 departments with the highest average salary

**Question 2: Retrieving Hierarchical Category Paths**

**Task:**

* + - 1. **Write a SQL query using recursive Common Table Expressions (CTE) to retrieve all categories along with their full hierarchical path (e.g., Category > Subcategory > Sub-subcategory).**

**SQL Query**

WITH RECURSIVE category\_hierarchy AS (

-- Anchor query: Select top-level categories

SELECT

CategoryID,

CategoryName,

CAST (CategoryName AS VARCHAR(MAX)) AS HierarchicalPath,

0 AS Level

FROM

Categories

WHERE

ParentCategoryID IS NULL

UNION ALL

-- Recursive query: Select child categories

SELECT

c.CategoryID,

c.CategoryName,

ch.HierarchicalPath + ' > ' + c.CategoryName,

ch. Level + 1

FROM

Categories c

INNER JOIN

category\_hierarchy ch ON c. ParentCategoryID = ch. CategoryID

)

SELECT

CategoryID,

CategoryName,

HierarchicalPath

FROM

category\_hierarchy

ORDER BY

HierarchicalPath;

**Explanation of Recursive CTE**

This SQL query uses a recursive Common Table Expression (CTE) to retrieve all categories along with their full hierarchical path. Here's what it does:

* It starts with the top-level categories (categories with no parent)
* Then, it recursively adds child categories to the result set, building the hierarchical path by concatenating the parent category's path with the child category's name
* The query continues to add child categories until it reaches the lowest level of the hierarchy
* Finally, it returns the category ID, category name, and the full hierarchical path for each category

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

**Task:**

1. **Design a SQL query to find the total number of distinct customers who made a purchase in each month of the current year. Ensure months with no customer activity show a count of 0.**

**SQL Query**

WITH Months AS (

SELECT

DATENAME (MONTH, DATEFROMPARTS (YEAR (GETDATE ()), m, 1)) AS MonthName

FROM

(VALUES (1), (2), (3), (4), (5), (6), (7), (8), (9), (10), (11), (12)) AS Months(m)

),

CustomerPurchases AS (

SELECT

DATENAME (MONTH, o. OrderDate) AS MonthName,

COUNT (DISTINCT c. CustomerID) AS CustomerCount

FROM

Customers c

INNER JOIN

Orders o ON c. CustomerID = o. CustomerID

WHERE

YEAR (o. OrderDate) = YEAR (GETDATE ())

GROUP BY

DATENAME (MONTH, o. OrderDate)

)

SELECT

m.MonthName,

COALESCE (cp. CustomerCount, 0) AS CustomerCount

FROM

Months m

LEFT JOIN CustomerPurchases cp ON m. MonthName = cp. MonthName

ORDER BY

m.MonthName;

**Explanation of the Query**

* **Months CTE**: The first CTE, **Months**, generates a list of all 12 months of the year using a derived table with **VALUES** clause. The **DATENAME** function is used to get the month name from the month number.
* **Customer Purchases CTE**: The second CTE, **Customer Purchases**, retrieves the number of distinct customers who made a purchase in each month of the current year. It joins the **Customers** and **Orders** tables, filters the orders by the current year, and groups the results by month.
* **LEFT JOIN**: The main query uses a **LEFT JOIN** to combine the **Months** CTE with the **Customer Purchases** CTE. This ensures that all months are included in the result set, even if there are no customer purchases in a particular month.
* **COALESCE**: The **COALESCE** function is used to replace **NULL** values in the **Customer Count** column with 0. This handles the case where there are no customer purchases in a month, ensuring that the result set shows a count of 0 for that month.
* **ORDER BY**: Finally, the result set is sorted by month name to ensure that the months are in the correct order.

**Question 4: Finding Closest Locations**

**Task:**

* 1. **Write a SQL query to find the closest 5 locations to a given point specified by latitude and longitude. Use spatial functions or advanced mathematical calculations for proximity**.

**SQL Query**

WITH ClosestLocations AS (

SELECT

l.LocationID,

l.LocationName,

l.Latitude,

l.Longitude,

(

3959 \* ACOS (

COS(RADIANS(@lat)) \* COS (RADIANS (l. Latitude)) \*

COS (RADIANS (l. Longitude) - RADIANS(@lon)) +

SIN(RADIANS(@lat)) \* SIN (RADIANS (l. Latitude))

)

) AS Distance

FROM

Locations l

)

SELECT

LocationID,

LocationName,

Latitude,

Longitude,

Distance

FROM

ClosestLocations

ORDER BY

Distance

LIMIT 5;

**Explanation of the Mathematical Approach**

The query uses the Haversine formula, a well-known algorithm for calculating the distance between two points on a sphere (such as the Earth) given their longitudes and latitudes. The formula is implemented using a combination of trigonometric functions and mathematical operations.

**Here's a breakdown of the formula:**

* **3959** is the average radius of the Earth in miles.
* **ACOS** calculates the inverse cosine of the angle between the two points.
* **COS(RADIANS(@lat))** and **COS (RADIANS (l. Latitude))** calculate the cosine of the latitude angles.
* **COS (RADIANS (l. Longitude) - RADIANS(@lon))** calculates the cosine of the longitude difference.
* **SIN(RADIANS(@lat))** and **SIN (RADIANS (l. Latitude))** calculate the sine of the latitude angles.
* The formula combines these values to calculate the distance between the two points.

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

**Task:**

1. **Write a SQL query to retrieve orders placed in the last 7 days from a large Orders table, sorted by order date in descending order.**

**SQL Query**

SELECT

o. OrderID,

o. OrderDate,

o. CustomerID,

o. TotalAmount

FROM

Orders o

WHERE

o. OrderDate >= DATEADD (DAY, -7, CONVERT (DATE, GETDATE ()))

ORDER BY

o. OrderDate DESC;

**Discussion of Strategies Used to Optimize the Query**

* **Indexing**: Creating an index on the **Order Date** column can significantly improve query performance. This is because the query filters on this column, and an index can help the database quickly locate the relevant rows.
* **Date arithmetic**: Instead of using a function like **DATEDIFF** to calculate the date range, I used the **DATEADD** function to subtract 7 days from the current date. This allows the database to use an index on the **Order Date** column.
* **Convert to date**: By converting the **GETDATE ()** function to a **DATE** data type, I ensured that the database can use an index on the **Order Date** column. This is because the **GETDATE ()** function returns a **DATETIME** value, which may not be compatible with the **DATE** data type of the **Order Date** column.
* **Avoid using functions in the WHERE clause**: By rewriting the query to use a date range instead of a function like **DATEDIFF**, I avoided the need for the database to evaluate the function for each row. This can improve performance, especially for large tables.
* **Optimized sorting**: By sorting the results in descending order by **Order Date**, I ensured that the database can use an index on this column to optimize the sorting operation.

**3.**

**PL/SQL Questions**

**Question 1: Handling Division Operation**

**Task:**

* + - 1. **Write a PL/SQL block to perform a division operation where the divisor is obtained from user input. Handle the ZERO\_DIVIDE exception gracefully with an appropriate error message.**

**PL/SQL Query**

DECLARE

dividend NUMBER: = 10;

divisor NUMBER;

result NUMBER;

BEGIN

divisor: = &enter\_divisor; -- obtain divisor from user input

BEGIN

result: = dividend / divisor;

DBMS\_OUTPUT.PUT\_LINE ('Result: ' || result);

EXCEPTION

WHEN ZERO\_DIVIDE THEN

DBMS\_OUTPUT.PUT\_LINE ('Error: Division by zero');

END;

END;

**Explanation of error handling strategies implemented:**

The code uses a nested BEGIN-END block to catch the specific exception obtained from a division operation. The divisor is obtained from user input, and the division operation is performed inside the inner BEGIN-END block. If a division by zero exception is raised, the EXCEPTION block immediately follows the division operation and catches the ZERO\_DIVIDE exception. Inside the EXCEPTION block, an error message is displayed indicating that a division by zero has occurred. This approach ensures that the program does not terminate abruptly due to the exception, but instead handles it gracefully and provides a meaningful error message to the user.

**Question 2: Updating Rows with FORALL**

**Task:**

1. **Use the FORALL statement to update multiple rows in the Employees table based on arrays of employee IDs and salary increments.**

**PL/SQL Query**

DECLARE

TYPE emp\_id\_array IS VARRAY (10) OF NUMBER;

TYPE salary\_inc\_array IS VARRAY (10) OF NUMBER;

emp\_ids emp\_id\_array: = emp\_id\_array (101, 102, 103, 104, 105);

salary\_incs salary\_inc\_array: = salary\_inc\_array (1000, 2000, 3000, 4000, 5000);

BEGIN

FORALL i IN 1.emp\_ids. COUNT

UPDATE Employees

SET salary = salary + salary\_incs(i)

WHERE employee\_id = emp\_ids(i);

COMMIT;

END;

**Description of how FORALL improves performance for bulk updates:**

The FORALL statement is a bulk DML (Data Manipulation Language) statement that allows you to execute a single DML statement multiple times, with each iteration using a different set of values from an array. In this example, we use two arrays, emp\_ids and salary\_incs, to store the employee IDs and corresponding salary increments.

1. **Reduced number of SQL statements**: Instead of executing multiple individual UPDATE statements, one for each employee, the FORALL statement executes a single UPDATE statement that updates multiple rows in a single operation.
2. **Improved performance**: By reducing the number of SQL statements, the FORALL statement minimizes the overhead of parsing, binding, and executing individual statements.
3. **Enhanced concurrency**: FORALL statements can take advantage of Oracle's parallel processing capabilities, allowing multiple updates to be executed concurrently, further improving performance.
4. **Error handling**: If an error occurs during the execution of the FORALL statement, Oracle rolls back the entire operation, ensuring data consistency and integrity.

**Question 3: Implementing Nested Table Procedure**

**Task:**

* + - 1. **Implement a PL/SQL procedure that accepts a department ID as input, retrieves employees belonging to the department, stores them in a nested table type, and returns this collection as an output parameter.**

**PL/SQL Query**

CREATE OR REPLACE TYPE employee\_type AS OBJECT (

employee\_id NUMBER,

first\_name VARCHAR2(20),

last\_name VARCHAR2(20)

);

CREATE OR REPLACE TYPE employee\_collection AS TABLE OF employee\_type;

CREATE OR REPLACE PROCEDURE get\_department\_employees(

p\_department\_id IN NUMBER,

p\_employees OUT employee\_collection

) AS

BEGIN

SELECT employee\_type (e. employee\_id, e. first\_name, e. last\_name)

BULK COLLECT INTO p\_employees

FROM Employees e

WHERE e. department\_id = p\_department\_id;

END get\_department\_employees;

**Explanation of how nested tables are utilized and returned as output:**

1. **Nested Table Type**: We define a nested table type **employee\_collection** as a table of **employee\_type** objects. This allows us to store a collection of employee records.
2. **Procedure Definition**: The **get\_department\_employees** procedure takes two parameters: **p\_department\_id** (input) and **p\_employees** (output). The output parameter **p\_employees** is of type **employee\_collection**.
3. **BULK COLLECT INTO**: Inside the procedure, we use the **BULK COLLECT INTO** statement to retrieve the employees belonging to the specified department and store them in the **p\_employees** collection. This statement allows us to fetch multiple rows into a single collection.
4. **Returning the Collection**: The **p\_employees** collection is returned as an output parameter, which can be accessed by the calling program.

**Question 4: Using Cursor Variables and Dynamic SQL**

**Task:**

* + - 1. **Write a PL/SQL block demonstrating the use of cursor variables (REF CURSOR) and dynamic SQL. Declare a cursor variable for querying EmployeeID, FirstName, and LastName based on a specified salary threshold.**

**PL/SQL Query**

DECLARE

-- Declare a REF CURSOR type

TYPE ref\_cursor IS REF CURSOR;

-- Declare a cursor variable

v\_cursor ref\_cursor;

-- Declare variables to hold the dynamic SQL statement and its parameters

v\_sql\_stmt VARCHAR2(200);

v\_salary\_threshold NUMBER: = 50000;

-- Declare variables to hold the query results

v\_employee\_id NUMBER;

v\_first\_name VARCHAR2(20);

v\_last\_name VARCHAR2(20);

BEGIN

-- Construct the dynamic SQL statement

v\_sql\_stmt := 'SELECT employee\_id, first\_name, last\_name FROM Employees WHERE salary > :salary\_threshold';

-- Open the cursor variable with the dynamic SQL statement

OPEN v\_cursor FOR v\_sql\_stmt USING v\_salary\_threshold;

-- Fetch the query results

LOOP

FETCH v\_cursor INTO v\_employee\_id, v\_first\_name, v\_last\_name;

EXIT WHEN v\_cursor%NOTFOUND;

-- Process the query results

DBMS\_OUTPUT.PUT\_LINE ('Employee ID: ' || v\_employee\_id || ', Name: ' || v\_first\_name || ' ' || v\_last\_name);

END LOOP;

-- Close the cursor variable

CLOSE v\_cursor;

END;

**Explanation of how dynamic SQL is constructed and executed:**

1. Declaring a REF CURSOR type: We declare a REF CURSOR type, which is a pointer to a cursor.
2. Declaring a cursor variable: We declare a cursor variable v\_cursor of type REF CURSOR.
3. Constructing the dynamic SQL statement: We construct a dynamic SQL statement as a string, using a placeholder: salary\_threshold for the salary threshold parameter.
4. Opening the cursor variable: We open the cursor variable with the dynamic SQL statement, passing the v\_salary\_threshold parameter as a bind variable.
5. Fetching the query results: We fetch the query results using a LOOP statement, which iterates over the rows returned by the cursor.
6. Processing the query results: We process the query results by printing the employee ID, first name, and last name to the console.
7. Closing the cursor variable: We close the cursor variable to release system resources.

Dynamic SQL is constructed by concatenating strings to form a SQL statement. In this example, we construct a SELECT statement with a WHERE clause that filters employees based on a salary threshold. The: salary**\_threshold** placeholder is replaced with the actual value of **v\_salary\_threshold** when the cursor is opened.

**Question 5: Designing Pipelined Function for Sales Data**

**Task:**

* + - 1. **Design a pipelined PL/SQL function get\_sales\_data that retrieves sales data for a given month and year. The function should return a table of records containing OrderID, CustomerID, and OrderAmount for orders placed in the specified month and year.**

**PL/SQL Query:**

CREATE OR REPLACE PACKAGE sales\_data\_pkg AS

TYPE sales\_data\_rec IS RECORD (

order\_id NUMBER,

customer\_id NUMBER,

order\_amount NUMBER

);

TYPE sales\_data\_tbl IS TABLE OF sales\_data\_rec;

FUNCTION get\_sales\_data(p\_month IN NUMBER, p\_year IN NUMBER)

RETURN sales\_data\_tbl PIPELINED;

END sales\_data\_pkg;

CREATE OR REPLACE PACKAGE BODY sales\_data\_pkg AS

FUNCTION get\_sales\_data(p\_month IN NUMBER, p\_year IN NUMBER)

RETURN sales\_data\_tbl PIPELINED AS

BEGIN

FOR cur\_rec IN (

SELECT order\_id, customer\_id, order\_amount

FROM orders

WHERE EXTRACT (MONTH FROM order\_date) = p\_month

AND EXTRACT (YEAR FROM order\_date) = p\_year

) LOOP

PIPE ROW (sales\_data\_rec(cur\_rec.order\_id, cur\_rec.customer\_id, cur\_rec.order\_amount));

END LOOP;

RETURN;

END get\_sales\_data;

END sales\_data\_pkg;

**Explanation of how pipelined table functions improve data retrieval efficiency:**

1. Pipelined Functions: A pipelined function is a type of table function that returns a table of records in a pipelined manner, allowing the caller to process the data in a streaming fashion.
2. Improved Efficiency: Pipelined table functions improve data retrieval efficiency in several ways:
   * Reduced Memory Usage: By processing the data in a pipelined manner, the function only needs to store a small amount of data in memory at a time, reducing the memory footprint and improving performance.
   * Faster Data Retrieval: Pipelined functions can return data to the caller as soon as it is available, rather than waiting for the entire result set to be generated. This can significantly improve data retrieval times, especially for large datasets.
   * Improved Scalability: Pipelined functions can handle large datasets more efficiently, making them more scalable and suitable for big data applications.
3. How it Works: In the get\_sales\_data function, we use a cursor to iterate over the orders table, filtering the data based on the specified month and year. For each row, we use the PIPE ROW statement to return a single record to the caller, allowing the data to be processed in a streaming fashion.
4. Caller Perspective: From the caller's perspective, the pipelined function returns a table of records that can be processed using a cursor or a loop. The caller can process the data as it is returned, without having to wait for the entire result set to be generated.