CSA09: DATABASE MANAGEMENT SYSTEMS-ASSIGNMENT QUESTIONS

Due Date: 31 July 2024

M.Mahathi

192324098

Question 1:

ER Diagram Question: Traffic Flow Management System (TFMS)

Entities and Attributes

1. Roads
   * RoadID (PK)
   * RoadName
   * Length
   * SpeedLimit
2. Intersections
   * IntersectionID (PK)
   * IntersectionName
   * Latitude
   * Longitude
3. Traffic Signals
   * SignalID (PK)
   * SignalStatus
   * Timer
   * IntersectionID (FK)
4. Traffic Data
   * TrafficDataID (PK)
   * Timestamp
   * Speed
   * CongestionLevel
   * RoadID (FK)

Relationships

1. Roads to Intersections
   * Type: One-to-Many (1

)

* + Description: A road can have multiple intersections, meaning a single road can connect to multiple intersections.
  + Cardinality: 1
  + Optionality: Mandatory (a road must have at least one intersection)

1. Intersections to Roads
   * Type: Many-to-Many (M

)

* + Description: An intersection can connect multiple roads, and each road can connect to multiple intersections.
  + Cardinality: M
  + Optionality: Mandatory (an intersection must connect to at least one road)

1. Intersections to Traffic Signals
   * Type: One-to-One (1:1)
   * Description: An intersection can have one traffic signal, and each traffic signal is associated with one intersection.
   * Cardinality: 1:1
   * Optionality: Optional (an intersection may not have a traffic signal)
2. Traffic Signals to Intersections
   * Type: One-to-One (1:1)
   * Description: A traffic signal is associated with one intersection, ensuring that each signal corresponds to one specific intersection.
   * Cardinality: 1:1
   * Optionality: Mandatory (a traffic signal must be associated with an intersection)
3. Roads to Traffic Data
   * Type: One-to-Many (1

)

* + Description: A road can have multiple traffic data points, reflecting different data collected over time.
  + Cardinality: 1
  + Optionality: Mandatory (a road must have at least one traffic data point)

Justification and Normalization Considerations

1. Scalability:
   * The ERD design supports scalability by ensuring that roads, intersections, and traffic signals can be added or modified without disrupting the existing structure. Real-time data integration allows for continuous monitoring and traffic flow optimization.
2. Normalization:
   * 1NF (First Normal Form): All entities have attributes with atomic values, and there are no repeating groups or arrays. Each attribute holds a single value, ensuring clarity and consistency.
   * 2NF (Second Normal Form): All non-key attributes are fully functionally dependent on the primary key. This eliminates partial dependency, ensuring that all attributes are related to the entity's primary key.
   * 3NF (Third Normal Form): All non-key attributes are non-transitively dependent on the primary key, ensuring that data is stored efficiently with no redundancy. This maintains data integrity and minimizes the risk of anomalies.

ERD Representation

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

| Roads |

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

| RoadID (PK) |

| RoadName |

| Length |

| SpeedLimit |

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

|

| 1: N

v

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

| Intersections |

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

| IntersectionID(PK)|

| IntersectionName |

| Latitude |

| Longitude |

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

|

| 1:1

v

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

| Traffic Signals|

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

| SignalID (PK) |

| IntersectionID (FK)|

| Signal Status |

| Timer |

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

|

| 1: N

v

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

| Traffic Data |

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

| TrafficDataID (PK)|

| RoadID (FK) |

| Timestamp |

| Speed |

| CongestionLevel |

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

Cardinality and Optionality Notations

* 1:1: Represents a one-to-one relationship where each entity is associated with exactly one other entity.
* 1:∞: Represents a one-to-many relationship where one entity can be associated with multiple entities on the other side.
* ∞:∞: Represents a many-to-many relationship where multiple entities can be associated with multiple entities on the other side.
* O: Optional relationship, indicating that the relationship is not mandatory.
* |: Mandatory relationship, indicating that the relationship must exist.

Notation Key Summary

* Rectangles: Entities
* Ellipses: Attributes
* Diamonds: Relationships
* Lines: Connect Entities
* 1:1, 1:∞, ∞:∞: Cardinality
* O: Optional relationship
* |: Mandatory relationship

Conclusion

This ERD design for the TFMS is structured to support efficient traffic management, allowing for real-time data processing and analysis. The design adheres to normalization principles, ensuring data integrity and minimizing redundancy. The relationships between entities are clearly defined with appropriate cardinality and optionality, supporting the system's functionality.

Question 2:

Question 1: Top 3 Departments with Highest Average Salary

SQL:

SELECT TOP 3

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;

Explanation:

* Handling Departments with No Employees:
  + The LEFT JOIN ensures all departments are included, even those without employees, resulting in an AvgSalary of NULL for such departments.
* Calculating Average Salary:
  + The AVG function calculates the average salary for employees in each department. Departments are sorted by average salary in descending order, and the TOP 3 limits the results to the top 3.

Question 2: Retrieving Hierarchical Category Paths

SQL:

WITH CategoryCTE AS (

SELECT

CategoryID,

CategoryName,

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

ParentCategoryID

FROM

Categories

WHERE

ParentCategoryID IS NULL

UNION ALL

SELECT

c.CategoryID,

c.CategoryName,

CAST(CONCAT(cte.HierarchicalPath, ' > ', c.CategoryName) AS VARCHAR(MAX)) AS HierarchicalPath,

c.ParentCategoryID

FROM

Categories c

INNER JOIN

CategoryCTE cte ON cte.CategoryID = c.ParentCategoryID

)

SELECT

CategoryID,

CategoryName,

HierarchicalPath

FROM

CategoryCTE

ORDER BY

HierarchicalPath;

Explanation:

* Recursive CTE Mechanism:
  + The recursive CTE begins with root categories and builds the full hierarchical path by iterating through each child category and concatenating the parent’s path.

Question 3: Total Distinct Customers by Month

SQL:

WITH Months AS (

SELECT

DATEFROMPARTS(YEAR(GETDATE()), number, 1) AS FirstDayOfMonth

FROM

master.dbo.spt\_values

WHERE

type = 'P' AND number BETWEEN 1 AND 12

)

SELECT

DATENAME(MONTH, m.FirstDayOfMonth) AS MonthName,

COUNT(DISTINCT o.CustomerID) AS CustomerCount

FROM

Months m

LEFT JOIN

Orders o ON MONTH(o.OrderDate) = MONTH(m.FirstDayOfMonth)

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

GROUP BY

DATENAME(MONTH, m.FirstDayOfMonth)

ORDER BY

MIN(m.FirstDayOfMonth);

Explanation:

* Including All Months:
  + The CTE Months generates all months in the current year. A LEFT JOIN ensures all months are included, showing 0 for months with no customer activity.

Question 4: Finding Closest Locations

SQL:

SELECT TOP 5

LocationID,

LocationName,

Latitude,

Longitude,

(

6371 \* acos(

cos(radians(@Latitude)) \* cos(radians(Latitude)) \*

cos(radians(Longitude) - radians(@Longitude)) +

sin(radians(@Latitude)) \* sin(radians(Latitude))

)

) AS Distance

FROM

Locations

ORDER BY

Distance ASC;

Explanation:

* Proximity Calculation:
  + The query uses the Haversine formula to calculate the distance between two points on Earth, based on latitude and longitude. The TOP 5 returns the closest 5 locations.

Question 5: Optimizing Query for Orders Table

SQL:

SELECT

OrderID,

OrderDate,

CustomerID,

TotalAmount

FROM

Orders WITH (INDEX(OrderDate\_Index))

WHERE

OrderDate >= DATEADD(DAY, -7, GETDATE())

ORDER BY

OrderDate DESC;

Explanation:

Query Optimization:

An index on OrderDate (OrderDate\_Index) speeds up the retrieval of recent orders. Filtering by OrderDate and sorting by it in descending order are optimized by this index, reducing the scan time for large datasets.

Question 3:

PL/SQL Questions

Question 1: Handling Division Operation

Task: 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 Block:

DECLARE

numerator NUMBER := 100; -- Example numerator

divisor NUMBER;

result NUMBER;

BEGIN

-- Accepting divisor from user input

divisor := &input\_divisor;

-- Performing division operation

result := numerator / divisor;

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

EXCEPTION

WHEN ZERO\_DIVIDE THEN

DBMS\_OUTPUT.PUT\_LINE('Error: Division by zero is not allowed.');

END;

/

Explanation:

* The block attempts to divide the numerator by the divisor.
* If the divisor is zero, the ZERO\_DIVIDE exception is raised, and the block handles it by printing an appropriate error message.

Question 2: Updating Rows with FORALL

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

PL/SQL Block:

DECLARE

TYPE t\_emp\_ids IS TABLE OF employees.employee\_id%TYPE INDEX BY PLS\_INTEGER;

TYPE t\_salary\_increments IS TABLE OF employees.salary%TYPE INDEX BY PLS\_INTEGER;

l\_emp\_ids t\_emp\_ids;

l\_salary\_increments t\_salary\_increments;

BEGIN

-- Example data

l\_emp\_ids(1) := 101;

l\_emp\_ids(2) := 102;

l\_emp\_ids(3) := 103;

l\_salary\_increments(1) := 1000;

l\_salary\_increments(2) := 1200;

l\_salary\_increments(3) := 1100;

FORALL i IN l\_emp\_ids.FIRST .. l\_emp\_ids.LAST

UPDATE employees

SET salary = salary + l\_salary\_increments(i)

WHERE employee\_id = l\_emp\_ids(i);

COMMIT;

END;

/

Explanation:

* The FORALL statement is used to efficiently update multiple rows in bulk.
* This improves performance compared to updating rows one at a time in a loop.

Question 3: Implementing Nested Table Procedure

Task: 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 Procedure:

CREATE OR REPLACE PROCEDURE get\_department\_employees (

p\_department\_id IN employees.department\_id%TYPE,

p\_employees OUT SYS.ODCIVARCHAR2LIST

) IS

TYPE t\_employee\_list IS TABLE OF employees.last\_name%TYPE;

l\_employees t\_employee\_list;

BEGIN

SELECT last\_name

BULK COLLECT INTO l\_employees

FROM employees

WHERE department\_id = p\_department\_id;

p\_employees := SYS.ODCIVARCHAR2LIST();

p\_employees.EXTEND(l\_employees.COUNT);

FOR i IN 1..l\_employees.COUNT LOOP

p\_employees(i) := l\_employees(i);

END LOOP;

END;

/

Explanation:

* A nested table is used to store the list of employees retrieved by department.
* The procedure returns this list as an output parameter.

Question 4: Using Cursor Variables and Dynamic SQL

Task: 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 Block:

DECLARE

TYPE ref\_cursor IS REF CURSOR;

c\_emp ref\_cursor;

v\_employee\_id employees.employee\_id%TYPE;

v\_first\_name employees.first\_name%TYPE;

v\_last\_name employees.last\_name%TYPE;

v\_sql VARCHAR2(4000);

v\_salary\_threshold NUMBER := &salary\_threshold;

BEGIN

v\_sql := 'SELECT employee\_id, first\_name, last\_name FROM employees WHERE salary > :1';

OPEN c\_emp FOR v\_sql USING v\_salary\_threshold;

LOOP

FETCH c\_emp INTO v\_employee\_id, v\_first\_name, v\_last\_name;

EXIT WHEN c\_emp%NOTFOUND;

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

END LOOP;

CLOSE c\_emp;

END;

/

Explanation:

* The block demonstrates dynamic SQL using a cursor variable (REF CURSOR).
* The SQL query is dynamically constructed and executed with a salary threshold provided by the user.

Question 5: Designing Pipelined Function for Sales Data

Task: 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 Function:

CREATE OR REPLACE TYPE t\_sales\_record IS OBJECT (

order\_id NUMBER,

customer\_id NUMBER,

order\_amount NUMBER

);

CREATE OR REPLACE TYPE t\_sales\_table IS TABLE OF t\_sales\_record;

CREATE OR REPLACE FUNCTION get\_sales\_data(p\_month NUMBER, p\_year NUMBER)

RETURN t\_sales\_table PIPELINED

IS

CURSOR c\_sales IS

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;

r\_sales c\_sales%ROWTYPE;

BEGIN

OPEN c\_sales;

LOOP

FETCH c\_sales INTO r\_sales;

EXIT WHEN c\_sales%NOTFOUND;

PIPE ROW(t\_sales\_record(r\_sales.order\_id, r\_sales.customer\_id, r\_sales.order\_amount));

END LOOP;

CLOSE c\_sales;

END;

/

Explanation:

* The pipelined function returns a table of records as its output.
* Pipelined functions allow for efficient data retrieval by streaming results as they are produced, rather than returning the entire result set at once.

Each of these PL/SQL blocks and functions is designed to handle specific tasks efficiently, demonstrating different techniques like exception handling, bulk operations, dynamic SQL, and pipelined table functions.