Traffic Flow Management System (TFMS)

Name: Shaik Jabbar Basha

Reg no: 192325059

Question: 1

Task 1: Entity Identification and Attributes

Entities and Attributes:

1. Roads:
   - RoadID (PK)
   - RoadName
   - Length (in meters)
   - SpeedLimit (in km/h)

2. Intersections:
   - IntersectionID (PK)
   - IntersectionName
   - Latitude
   - Longitude

3. Traffic Signals:
   - SignalID (PK)
   - IntersectionID (FK)
   - SignalStatus (Green, Yellow, Red)
   - Timer (countdown to next change)

4. Traffic Data:
   - TrafficDataID (PK)
   - RoadID (FK)
   - Timestamp
   - Speed (average speed on the road)
   - CongestionLevel (degree of traffic congestion)

Task 2: Relationship Modeling

Relationships and Cardinality:

1. Roads to Intersections:

   - Relationship: Each road can connect to multiple intersections, and each intersection can be connected to multiple roads.

   - Cardinality: Many-to-Many

   - Implementation: Use a junction table, e.g., `Road_Intersection`.

2. Intersections to Traffic Signals:

   - Relationship: Each intersection can have multiple traffic signals, but each traffic signal is associated with one intersection.

   - Cardinality: One-to-Many

   - Implementation: Foreign key `IntersectionID` in `Traffic Signals`.

3. Roads to Traffic Data:

   - Relationship: Each road can have multiple traffic data entries, but each traffic data entry is associated with one road.

   - Cardinality: One-to-Many

   - Implementation: Foreign key `RoadID` in `Traffic Data`.

Task 3: ER Diagram Design

ER Diagram:

```plaintext
+----------------+        +------------------+        +--------------------+
|    Roads      |        |  Intersections  |        |  Traffic Signals |
+----------------+        +------------------+        +--------------------+
| RoadID (PK)   |1--------M| IntersectionID(PK)|1-------M| SignalID (PK)      |
| RoadName      |         | IntersectionName  |         | IntersectionID (FK)|
| Length        |         | Latitude          |         | SignalStatus       |
```

```
| SpeedLimit     |          | Longitude        |          | Timer             |
+----------------+          +------------------+          +--------------------+
       1
        \M
+-------------------+
|   Traffic Data    |
+-------------------+
| TrafficDataID (PK)|
| RoadID (FK)       |
| Timestamp         |
| Speed             |
| CongestionLevel   |
+-------------------+

+---------------------+
| Road_Intersection   |
+---------------------+
| RoadID (FK)         |
| IntersectionID (FK) |
+---------------------+
```

Task 4: Justification and Normalization

Justification for Design Choices:

1. Scalability:

- The design allows for easy addition of new roads, intersections, and traffic signals without altering the structure.

- Real-time data integration is facilitated by the `Traffic Data` entity, which can store high-frequency data points.

2. Real-Time Data Processing:

- Storing traffic data in a dedicated entity allows for efficient querying and analysis of real-time traffic conditions.

3. Efficient Traffic Management:

- The relationships and attributes chosen enable the system to dynamically manage and control traffic signals based on real-time traffic data and historical patterns.

Normalization Considerations:

1. First Normal Form (1NF):

- All attributes have atomic values.
- Each entity has a primary key, ensuring unique identification of records.

2. Second Normal Form (2NF):

- All non-key attributes are fully functionally dependent on the primary key.
- Composite primary keys, if any, are handled in the junction table `Road_Intersection`.

3. Third Normal Form (3NF):

- All attributes are functionally dependent only on the primary key, ensuring no transitive dependencies.

Deliverables

1. ER Diagram: Provided above in text form.
2. Entity Definitions:
   - Roads: RoadID (PK), RoadName, Length, SpeedLimit
   - Intersections: IntersectionID (PK), IntersectionName, Latitude, Longitude
   - Traffic Signals: SignalID (PK), IntersectionID (FK), SignalStatus, Timer
   - Traffic Data: TrafficDataID (PK), RoadID (FK), Timestamp, Speed, CongestionLevel
3. Relationship Descriptions:
   - Roads to Intersections: Many-to-Many (via `Road_Intersection` junction table)
   - Intersections to Traffic Signals: One-to-Many
   - Roads to Traffic Data: One-to-Many
4. Justification Document: Included in the section above, detailing scalability, real-time data processing, efficient traffic management, and normalization principles.

Question-2

Question 1: Top 3 Departments with Highest Average Salary

SQL Query:

sql

Copy code

```
SELECT DepartmentID, DepartmentName, AVG(Salary) AS AvgSalary
FROM Departments d
LEFT JOIN Employees e ON d.DepartmentID = e.DepartmentID
GROUP BY DepartmentID, DepartmentName
ORDER BY AvgSalary DESC
LIMIT 3;
```

Explanation:

- The LEFT JOIN ensures that all departments are included in the result, even those with no employees.
- The AVG(Salary) function calculates the average salary for each department. For departments with no employees, AVG(Salary) returns NULL.
- The GROUP BY clause groups the results by DepartmentID and DepartmentName.
- The ORDER BY AvgSalary DESC sorts the results by average salary in descending order.
- The LIMIT 3 restricts the result to the top 3 departments.

Question 2: Retrieving Hierarchical Category Paths

SQL Query:

sql

Copy code

```
WITH RECURSIVE CategoryHierarchy AS (
  SELECT CategoryID, CategoryName, NULL AS ParentCategoryID, CategoryName AS HierarchicalPath
  FROM Categories
  WHERE ParentCategoryID IS NULL
  UNION ALL
  SELECT c.CategoryID, c.CategoryName, c.ParentCategoryID, CONCAT(ch.HierarchicalPath, ' > ', c.CategoryName) AS HierarchicalPath
  FROM Categories c
  INNER JOIN CategoryHierarchy ch ON c.ParentCategoryID = ch.CategoryID
)
```

SELECT CategoryID, CategoryName, HierarchicalPath

FROM CategoryHierarchy;

Explanation:

- The WITH RECURSIVE clause defines a Common Table Expression (CTE) named CategoryHierarchy.
- The first part of the CTE (before UNION ALL) selects the root categories (those with ParentCategoryID IS NULL).
- The second part recursively joins the Categories table to build the hierarchical path for each category.
- The CONCAT function constructs the full hierarchical path by concatenating parent paths with the current category name.
- The final SELECT statement retrieves the CategoryID, CategoryName, and HierarchicalPath.

Question 3: Total Distinct Customers by Month

SQL Query:

sql

Copy code

WITH Months AS (

SELECT 1 AS MonthNum, 'January' AS MonthName UNION ALL

SELECT 2, 'February' UNION ALL

SELECT 3, 'March' UNION ALL

SELECT 4, 'April' UNION ALL

SELECT 5, 'May' UNION ALL

SELECT 6, 'June' UNION ALL

SELECT 7, 'July' UNION ALL

SELECT 8, 'August' UNION ALL

SELECT 9, 'September' UNION ALL

SELECT 10, 'October' UNION ALL

SELECT 11, 'November' UNION ALL

SELECT 12, 'December'

),

CustomerActivity AS (

SELECT MONTH(OrderDate) AS MonthNum, COUNT(DISTINCT CustomerID) AS CustomerCount

FROM Orders

WHERE YEAR(OrderDate) = YEAR(CURRENT_DATE)

GROUP BY MONTH(OrderDate)

)

SELECT m.MonthName, COALESCE(ca.CustomerCount, 0) AS CustomerCount

FROM Months m

LEFT JOIN CustomerActivity ca ON m.MonthNum = ca.MonthNum;

Explanation:

- The Months CTE generates a list of all months in the year.
- The CustomerActivity CTE counts the distinct customers who made purchases in each month of the current year.
- The final SELECT statement performs a LEFT JOIN between Months and CustomerActivity to include all months, substituting NULL with 0 using COALESCE.

Question 4: Finding Closest Locations

SQL Query:

sql

Copy code

SELECT LocationID, LocationName, Latitude, Longitude,

(3959 * 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

LIMIT 5;

Explanation:

- The query uses the Haversine formula to calculate the distance between two points on the Earth's surface given their latitude and longitude.
- Replace @latitude and @longitude with the specified point coordinates.
- The ORDER BY Distance clause sorts the results by calculated distance.
- The LIMIT 5 clause restricts the result to the closest 5 locations.

Question 5: Optimizing Query for Orders Table

SQL Query:

sql

Copy code

SELECT OrderID, CustomerID, OrderDate, TotalAmount

FROM Orders

```
WHERE OrderDate >= DATE_SUB(CURRENT_DATE, INTERVAL 7 DAY)
ORDER BY OrderDate DESC;
```

Optimization Discussion:

- Ensure an index on the OrderDate column to speed up the retrieval of recent orders.
- The query uses the DATE_SUB function to calculate the date 7 days ago from the current date.
- Filtering records based on the indexed OrderDate column reduces the number of rows to sort and return.
- The ORDER BY OrderDate DESC clause ensures the results are sorted by the most recent orders first.

Question-3:

Question 1: Handling Division Operation

PL/SQL Block:

```plsql
DECLARE
    numerator NUMBER := 100;  -- Example numerator
    divisor NUMBER;
    result NUMBER;
BEGIN
    -- Obtain divisor from user input
    divisor := &divisor;

    -- Perform division
    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 `DECLARE` section initializes the numerator and declares variables for the divisor and result.

- The divisor is obtained from user input using substitution variables (`&divisor`).

- The division operation is performed inside the `BEGIN...END` block.

- If a division by zero occurs, the `ZERO_DIVIDE` exception is caught and an appropriate error message is displayed using `DBMS_OUTPUT.PUT_LINE`.

## Question 2: Updating Rows with FORALL

**PL/SQL Block:**

```plsql
DECLARE
   TYPE EmployeeIDArray IS TABLE OF Employees.EmployeeID%TYPE;
   TYPE SalaryIncrementArray IS TABLE OF NUMBER;

   employee_ids EmployeeIDArray := EmployeeIDArray(101, 102, 103);  -- Example IDs
   salary_increments SalaryIncrementArray := SalaryIncrementArray(500, 600, 700);
BEGIN
   FORALL i IN INDICES OF employee_ids
      UPDATE Employees
      SET Salary = Salary + salary_increments(i)
      WHERE EmployeeID = employee_ids(i);

   DBMS_OUTPUT.PUT_LINE('Salaries updated successfully.');
END;
/
```

Explanation:
```

- The `DECLARE` section defines two nested table types for employee IDs and salary increments.

- Example data is initialized for both arrays.

- The `FORALL` statement iterates over the indices of the `employee_ids` array, performing bulk updates on the `Employees` table.

- The `FORALL` statement improves performance by reducing context switches between SQL and PL/SQL engines.

Question 3: Implementing Nested Table Procedure

PL/SQL Procedure:

```plsql
CREATE OR REPLACE TYPE EmployeeTableType IS TABLE OF Employees%ROWTYPE;


CREATE OR REPLACE PROCEDURE GetEmployeesByDepartment (
   p_DepartmentID IN Employees.DepartmentID%TYPE,
   p_EmployeeTable OUT EmployeeTableType
) IS
BEGIN
   SELECT * BULK COLLECT INTO p_EmployeeTable
   FROM Employees
   WHERE DepartmentID = p_DepartmentID;


   DBMS_OUTPUT.PUT_LINE('Employees retrieved successfully.');
END;
/
```

Explanation

- A nested table type `EmployeeTableType` is created to hold employee records.

- The `GetEmployeesByDepartment` procedure accepts a department ID and returns a collection of employees in the specified department.

- The `BULK COLLECT` statement fetches the employees into the nested table.

- Nested tables are used to store and return multiple rows as a single collection.

Question 4: Using Cursor Variables and Dynamic SQL

PL/SQL Block:

```plsql
DECLARE
    TYPE EmployeeCurType IS REF CURSOR;
    emp_cursor EmployeeCurType;
    emp_rec Employees%ROWTYPE;
    salary_threshold NUMBER := &salary_threshold;  -- Example threshold
BEGIN
    OPEN emp_cursor FOR 'SELECT EmployeeID, FirstName, LastName FROM Employees WHERE Salary > :1' USING salary_threshold;

    LOOP
        FETCH emp_cursor INTO emp_rec;
        EXIT WHEN emp_cursor%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('ID: ' || emp_rec.EmployeeID || ', Name: ' || emp_rec.FirstName || ' ' || emp_rec.LastName);
    END LOOP;

    CLOSE emp_cursor;
END;
/
```

Explanation:

- A cursor variable type `EmployeeCurType` is defined.
- The `emp_cursor` cursor variable is declared.
- The cursor is opened with a dynamic SQL statement that selects employees with a salary above the threshold, using a bind variable (`:1`).
- The loop fetches and processes each row, displaying employee details.
- The cursor is closed after processing.

Question 5: Designing Pipelined Function for Sales Data

PL/SQL Code:

```plsql
CREATE OR REPLACE TYPE SalesRecordType IS OBJECT (
  OrderID NUMBER,
  CustomerID NUMBER,
  OrderAmount NUMBER
);

CREATE OR REPLACE TYPE SalesRecordTableType IS TABLE OF SalesRecordType;

CREATE OR REPLACE FUNCTION get_sales_data (
  p_month IN NUMBER,
  p_year IN NUMBER
) RETURN SalesRecordTableType PIPELINED IS
  rec SalesRecordType;
BEGIN
  FOR r IN (SELECT OrderID, CustomerID, OrderAmount
        FROM Orders
        WHERE EXTRACT(MONTH FROM OrderDate) = p_month
         AND EXTRACT(YEAR FROM OrderDate) = p_year) LOOP
    rec := SalesRecordType(r.OrderID, r.CustomerID, r.OrderAmount);
    PIPE ROW(rec);
  END LOOP;
  RETURN;
END;
/
```

Explanation:

- A record type `SalesRecordType` and a table type `SalesRecordTableType` are defined to represent the sales data.

- The `get_sales_data` function is a pipelined table function that retrieves sales data for a given month and year.

- The `FOR` loop fetches each record matching the criteria and pipes it using `PIPE ROW`.

- Pipelined functions return results as they are produced, improving data retrieval efficiency by reducing memory consumption and allowing parallel processing.