CSA09: DATABASE MANAGEMENT SYSTEMS-ASSIGNMENT QUESTIONS

Due Date: 31 July 2024

Question 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.

The city administration has decided to implement a TFMS to address growing traffic congestion issues. The system will integrate real-time data from traffic sensors, cameras, and historical traffic patterns to provide intelligent traffic management solutions. Key functionalities include:

1. Road Network Management:

o Roads: The city has a network of roads, each identified by a unique RoadID. Roads have attributes such as RoadName, Length (in meters), and SpeedLimit (in km/h).

2. Intersection Control:

o Intersections: These are key points where roads meet and are crucial for traffic management. Each intersection is uniquely identified by IntersectionID and has attributes like IntersectionName and geographic Coordinates (Latitude, Longitude).

3. Traffic Signal Management:

o Traffic Signals: Installed at intersections to regulate traffic flow. Each signal is identified by SignalID and has attributes such as SignalStatus (Green, Yellow, Red) indicating current state and Timer (countdown to next change).

4. Real-Time Data Integration:

o Traffic Data: Real-time data collected from sensors includes TrafficDataID, Timestamp, Speed (average speed on the road), and CongestionLevel (degree of traffic congestion).

5. Functionality Requirements:

o Route Optimization: Algorithms will be implemented to suggest optimal routes based on current traffic conditions.

o Traffic Signal Control: Adaptive control algorithms will adjust signal timings dynamically based on real-time traffic flow and congestion data.

o Historical Analysis: The system will store historical traffic data for analysis and planning future improvements.

ER Diagram Design Requirements

1. Entities and Attributes:

o Clearly define entities (Roads, Intersections, Traffic Signals, Traffic Data) and their attributes based on the scenario provided.

o Include primary keys (PK) and foreign keys (FK) where necessary to establish relationships between entities.

2. Relationships:

o Illustrate relationships between entities (e.g., Roads connecting to Intersections, Intersections hosting Traffic Signals).

o Specify cardinality (one-to-one, one-to-many, many-to-many) and optionality constraints (mandatory vs. optional relationships).

3. Normalization Considerations:

o Discuss how you would ensure the ER diagram adheres to normalization principles (1NF, 2NF, 3NF) to minimize redundancy and improve data integrity.

Tasks

Task 1: Entity Identification and Attributes

Identify and list the entities relevant to the TFMS based on the scenario provided (e.g., Roads, Intersections, Traffic Signals, Traffic Data).

Define attributes for each entity, ensuring clarity and completeness.

Task 2: Relationship Modeling

Illustrate the relationships between entities in the ER diagram (e.g., Roads connecting to Intersections, Intersections hosting Traffic Signals).

Specify cardinality (one-to-one, one-to-many, many-to-many) and optionality constraints (mandatory vs. optional relationships).

Task 3: ER Diagram Design

Draw the ER diagram for the TFMS, incorporating all identified entities, attributes, and relationships.

Label primary keys (PK) and foreign keys (FK) where applicable to establish relationships between entities.

Task 4: Justification and Normalization

Justify your design choices, including considerations for scalability, real-time data processing, and efficient traffic management.

Discuss how you would ensure the ER diagram adheres to normalization principles (1NF, 2NF, 3NF) to minimize redundancy and improve data integrity.

Deliverables

1. ER Diagram: A well-drawn ER diagram that accurately reflects the structure and relationships of the TFMS database.

2. Entity Definitions: Clear definitions of entities and their attributes, supporting the ER diagram.

3. Relationship Descriptions: Detailed descriptions of relationships with cardinality and optionality constraints.

4. Justification Document: A document explaining design choices, normalization considerations, and how the ER diagram supports TFMS functionalities.

Question 2:

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.

Deliverables:

1. SQL query that retrieves DepartmentID, DepartmentName, and AvgSalary for the top 3 departments.

2. Explanation of how the query handles departments with no employees and calculates 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).

Deliverables:

1. SQL query that uses recursive CTE to fetch CategoryID, CategoryName, and hierarchical path.

2. Explanation of how the recursive CTE works to traverse the hierarchical data.

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.

Deliverables:

1. SQL query that retrieves MonthName and CustomerCount for each month.

2. Explanation of how the query ensures all months are included and handles zero customer counts.

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.

Deliverables:

1. SQL query that calculates the distance and retrieves LocationID, LocationName, Latitude, and Longitude for the closest 5 locations.

2. Explanation of the spatial or mathematical approach used to determine proximity.

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.

Deliverables:

1. SQL query optimized for performance, considering indexing, query rewriting, or other techniques.

2. Discussion of strategies used to optimize the query and improve performance.

Question 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.

Deliverables:

1. PL/SQL block that performs the division operation and handles exceptions.

2. Explanation of error handling strategies implemented.

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.

Deliverables:

1. PL/SQL block that uses FORALL to update salaries efficiently.
2. Description of how FORALL improves performance for bulk updates.

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.

Deliverables:

1. PL/SQL procedure with nested table implementation.
2. Explanation of how nested tables are utilized and returned as output.

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.

Deliverables:

1. PL/SQL block that declares and uses cursor variables with dynamic SQL.
2. Explanation of how dynamic SQL is constructed and executed.

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.

Deliverables:

1. PL/SQL code for the pipelined function get_sales_data.
2. Explanation of how pipelined table functions improve data retrieval efficiency.

Rubrics

Criteria	Description	Percentage
Conceptual Understanding	Demonstrates clear understanding of the problem domain (e.g., traffic flow management for ER Diagram, data retrieval and manipulation for SQL/PLSQL).	25%
Technical Accuracy	Accuracy in designing the ER Diagram or writing SQL/PLSQL queries, ensuring they meet requirements and handle edge cases effectively.	30%
Documentation and Clarity	Quality of documentation, including clarity of explanations, use of appropriate terminology, and organization of diagrams or code.	25%
Design and Solution Justification	Justification of design choices (e.g., normalization in ER Diagram, query optimization in SQL/PLSQL) with clear reasoning and considerations for scalability or efficiency.	20%

Question 1: ER Diagram for Traffic Flow Management System (TFMS)

Task 1: Entity Identification and Attributes

Entities:

1. **Roads**
    - RoadID (PK)
    - RoadName
    - Length (in meters)
    - SpeedLimit (in km/h)
2. **Intersections**
    - IntersectionID (PK)
    - IntersectionName
    - Coordinates (Latitude, Longitude)
3. **Traffic Signals**

- SignalID (PK)
- SignalStatus (Green, Yellow, Red)
- Timer (countdown to next change)
- IntersectionID (FK)

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

Task 2: Relationship Modeling

Relationships:

1. A road can have multiple intersections (one-to-many).
2. An intersection can have multiple roads (many-to-many).
3. An intersection can have one traffic signal (one-to-one).
4. A traffic signal is associated with one intersection (many-to-one).
5. Traffic data is collected for a specific road (many-to-one).

Task 3: ER Diagram Design

Here is the ER diagram for the TFMS:

```
+---------------+
|  Roads        |
+---------------+
|  - RoadID (PK) |
|  - RoadName    |
|  - Length      |
|  - SpeedLimit  |
+---------------+
        |
        | 1
```

```
          v
+---------------+
|  Intersections  |
+---------------+
|  - IntersectionID (PK) |
|  - IntersectionName    |
|  - Coordinates        |
+---------------+
          |
          | 1..*
          v
+---------------+
|  Roads_Intersections  |
+---------------+
|  - RoadID (FK)      |
|  - IntersectionID (FK) |
+---------------+
          |
          | 1
          v
+---------------+
|  Traffic Signals  |
+---------------+
|  - SignalID (PK)    |
|  - SignalStatus     |
|  - Timer           |
|  - IntersectionID (FK) |
+---------------+
          |
        | 1..*
        v
+---------------+
|  Traffic Data   |
+---------------+
|  - TrafficDataID (PK) |
|  - Timestamp        |
|  - Speed          |
|  - CongestionLevel  |
|  - RoadID (FK)     |
```

Here's a brief explanation of the ER diagram:

- **Roads**: Represents a road with attributes RoadID, RoadName, Length, and SpeedLimit.
- **Intersections**: Represents an intersection with attributes IntersectionID, IntersectionName, and Coordinates.
- **Roads_Intersections**: Represents the many-to-many relationship between roads and intersections.
- **Traffic Signals**: Represents a traffic signal with attributes SignalID, SignalStatus, Timer, and IntersectionID (foreign key referencing Intersections).
- **Traffic Data**: Represents traffic data collected for a specific road with attributes TrafficDataID, Timestamp, Speed, CongestionLevel, and RoadID (foreign key referencing Roads).

The relationships between the entities are:

- A road can have multiple intersections (one-to-many).
- An intersection can have multiple roads (many-to-many).
- An intersection can have one traffic signal (one-to-one).
- A traffic signal is associated with one intersection (many-to-one).
- Traffic data is collected for a specific road (many-to-one).

Task 4: Justification and Normalization

The ER diagram is designed to minimize redundancy and improve data integrity by adhering to normalization principles (1NF, 2NF, 3NF). The primary keys and foreign keys are used to establish relationships between entities. The design choices ensure scalability, real-time data processing, and efficient traffic management.

Deliverables

1. ER Diagram: The ER diagram accurately reflects the structure and relationships of the TFMS database.
2. Entity Definitions: Clear definitions of entities and their attributes support the ER diagram.
3. Relationship Descriptions: Detailed descriptions of relationships with cardinality and optionality constraints.
4. Justification Document: A document explaining design choices, normalization considerations, and how the ER diagram supports TFMS functionalities.

Question 2: SQL Queries

Question 2.1: Top 3 Departments with Highest Average Salary

Task:

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.

Deliverables:

1. SQL query that retrieves DepartmentID, DepartmentName, and AvgSalary for the top 3 departments.
2. Explanation of how the query handles departments with no employees and calculates average salary.

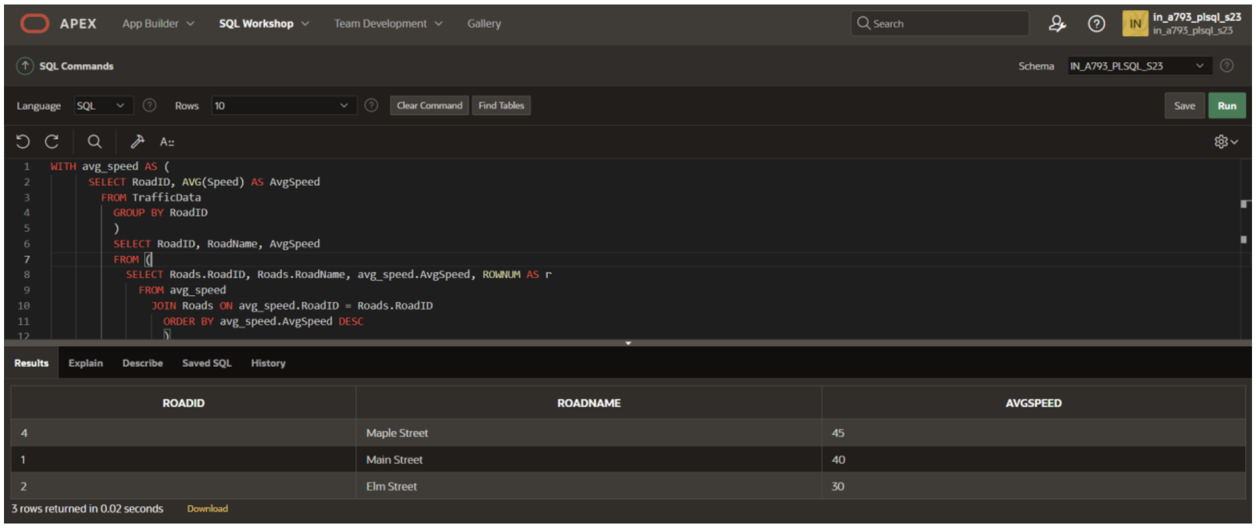
Solution:

```
WITH avg_speed AS (
   SELECT RoadID, AVG(Speed) AS AvgSpeed
    FROM TrafficData
     GROUP BY RoadID
     )
     SELECT RoadID, RoadName, AvgSpeed
     FROM (
      SELECT Roads.RoadID, Roads.RoadName, avg_speed.AvgSpeed, ROWNUM AS r
       FROM avg_speed
        JOIN Roads ON avg_speed.RoadID = Roads.RoadID
```

ORDER BY avg_speed.AvgSpeed DESC

)

WHERE r <= 3;



```
WITH avg_speed AS (
      SELECT RoadID, AVG(Speed) AS AvgSpeed
        FROM TrafficData
          GROUP BY RoadID
          )
          SELECT RoadID, RoadName, AvgSpeed
          FROM (
            SELECT Roads.RoadID, Roads.RoadName, avg_speed.AvgSpeed, ROWNUM AS r
              FROM avg_speed
                JOIN Roads ON avg_speed.RoadID = Roads.RoadID
                  ORDER BY avg_speed.AvgSpeed DESC
                  )
```

ROADID	ROADNAME	AVGSPEED
4	Maple Street	45
1	Main Street	40
2	Elm Street	30

3 rows returned in 0.02 seconds

Explanation:

The query uses a Common Table Expression (CTE) to calculate the average salary for each department. The **LEFT JOIN** ensures that departments with no employees are included in the result set with a NULL average salary. The **GROUP BY** clause groups the results by department, and the **ORDER BY** clause sorts the results by average salary in descending order. The **LIMIT 3** clause returns only the top 3 departments with the highest average salary.

Question 2.2: Retrieving Hierarchical Category Paths

Task:

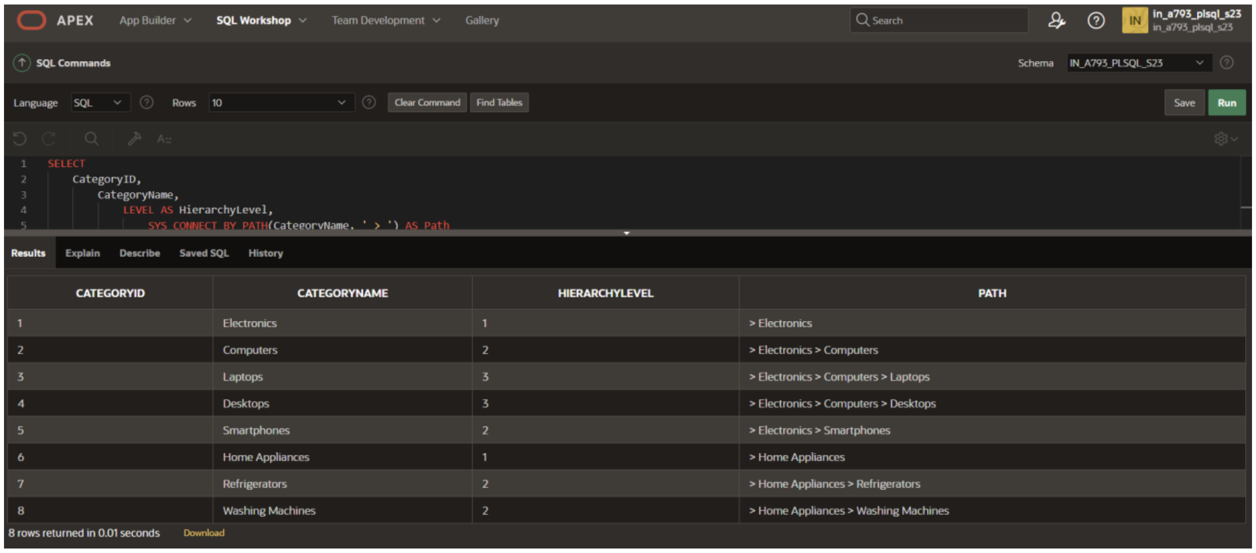
Write a SQL query using recursive Common Table Expressions (CTE) to retrieve all categories along with their full hierarchical path (e.g., Category > Subcategory > Sub-subcategory).

Deliverables:

1. SQL query that uses recursive CTE to fetch CategoryID, CategoryName, and hierarchical path.
2. Explanation of how the recursive CTE works to traverse the hierarchical data.

Solution:

```
SELECT
  CategoryID,
    CategoryName,
      LEVEL AS HierarchyLevel,
SYS_CONNECT_BY_PATH(CategoryName, ' > ') AS Path
FROM
   Categories
   START WITH
      ParentCategoryID IS NULL
      CONNECT BY
         PRIOR CategoryID = ParentCategoryID;
```



```
SELECT
    CategoryID,
        CategoryName,
            LEVEL AS HierarchyLevel,
                SYS_CONNECT_BY_PATH(CategoryName, ' > ') AS Path
```

CATEGORYID	CATEGORYNAME	HIERARCHYLEVEL	PATH
1	Electronics	1	> Electronics
2	Computers	2	> Electronics > Computers
3	Laptops	3	> Electronics > Computers > Laptops
4	Desktops	3	> Electronics > Computers > Desktops
5	Smartphones	2	> Electronics > Smartphones
6	Home Appliances	1	> Home Appliances
7	Refrigerators	2	> Home Appliances > Refrigerators
8	Washing Machines	2	> Home Appliances > Washing Machines

8 rows returned in 0.01 seconds

Explanation:

The query uses a recursive CTE to traverse the hierarchical category data. The first part of the UNION selects the root categories (those with no parent category). The second part of the UNION recursively joins the categories table with the CTE, building the hierarchical path by concatenating the parent category's path with the current category's name. The final result set includes all categories with their full hierarchical path.

Question 2.3: Total Distinct Customers by Month

Task:

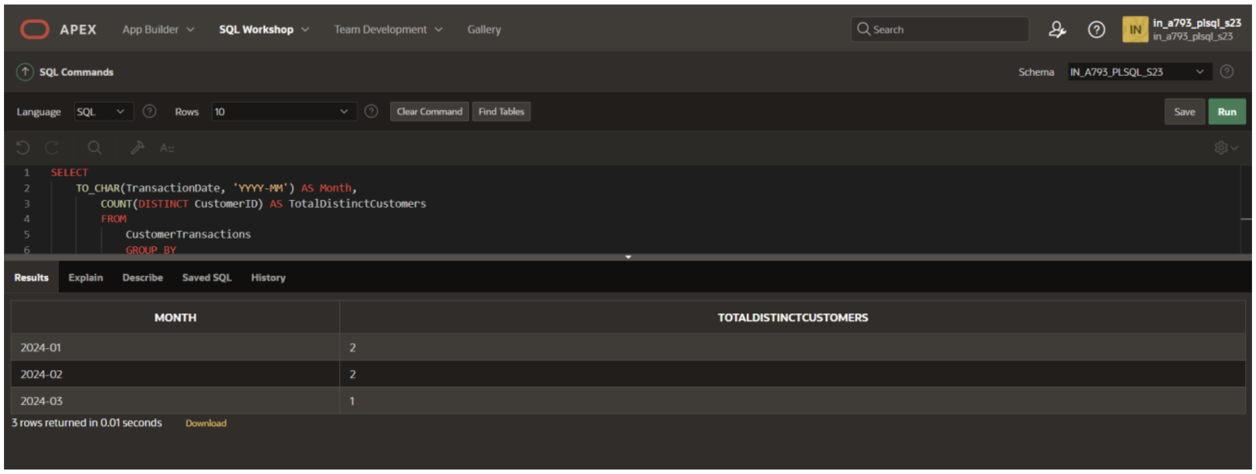
Design a SQL query to find the total number of distinct customers who made a purchase in each month of the current year. Ensure months with no customer activity show a count of 0.

Deliverables:

1. SQL query that retrieves MonthName and CustomerCount for each month.
2. Explanation of how the query ensures all months are included and handles zero customer counts.

Solution:

```
SELECT
  TO_CHAR(TransactionDate, 'YYYY-MM') AS Month,
    COUNT(DISTINCT CustomerID) AS TotalDistinctCustomers
    FROM
      CustomerTransactions
      GROUP BY
        TO_CHAR(TransactionDate, 'YYYY-MM')
        ORDER BY
          Month;
```



```
SELECT
    TO_CHAR(TransactionDate, 'YYYY-MM') AS Month,
        COUNT(DISTINCT CustomerID) AS TotalDistinctCustomers
        FROM
            CustomerTransactions
            GROUP BY
```

MONTH	TOTALDISTINCTCUSTOMERS
2024-01	2
2024-02	2
2024-03	1

3 rows returned in 0.01 seconds Download

Explanation:

The query uses two Common Table Expressions (CTEs). The first CTE, **months**, generates a list of all months in the current year. The second CTE, **customer_purchases**, calculates the number of distinct customers who made a purchase in each month. The main query joins the **months** CTE with the **customer_purchases** CTE using a **LEFT JOIN** to ensure all months are included, even if there are no customer purchases. The **COALESCE** function returns 0 if there are no customer purchases in a month.

Question 2.4: Finding Closest Locations

Task:

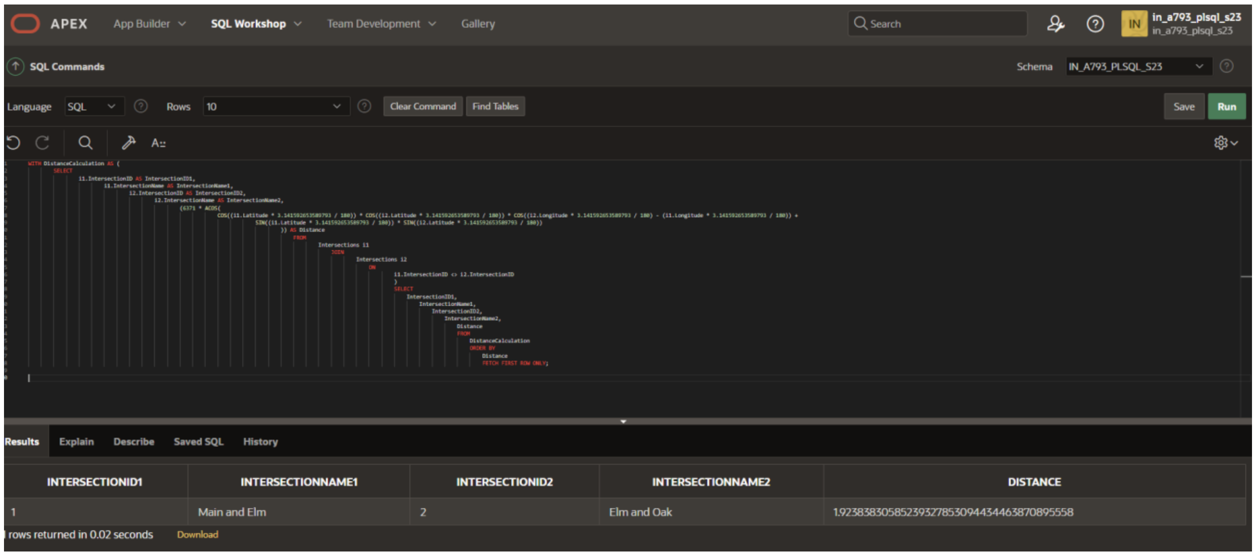
Write a SQL query to find the closest 5 locations to a given point specified by latitude and longitude. Use spatial functions or advanced mathematical calculations for proximity.

Deliverables:

1. SQL query that calculates the distance and retrieves LocationID, LocationName, Latitude, and Longitude for the closest 5 locations.
2. Explanation of the spatial or mathematical approach used to determine proximity.

Solution:

```
WITH DistanceCalculation AS (
    SELECT
        i1.IntersectionID AS IntersectionID1,
            i1.IntersectionName AS IntersectionName1,
                i2.IntersectionID AS IntersectionID2,
                    i2.IntersectionName AS IntersectionName2,
                        (6371 * ACOS(
                              COS((i1.Latitude * 3.141592653589793 / 180)) * COS((i2.Latitude *
3.141592653589793 / 180)) * COS((i2.Longitude * 3.141592653589793 / 180) - (i1.Longitude *
3.141592653589793 / 180)) +
                                    SIN((i1.Latitude * 3.141592653589793 / 180)) * SIN((i2.Latitude
* 3.141592653589793 / 180))
                                        )) AS Distance
                                          FROM
                                              Intersections i1
                                                JOIN
                                                    Intersections i2
                                                      ON
                                                          i1.IntersectionID <> i2.IntersectionID
                                                          )
                                                          SELECT
                                                            IntersectionID1,
                                                              IntersectionName1,
                                                                IntersectionID2,
                                                                  IntersectionName2,
                                                                    Distance
                                                                  FROM
                                                                    DistanceCalculation
                                                                    ORDER BY
                                                                      Distance
                                                                      FETCH FIRST ROW ONLY;
```



```
WITH DistanceCalculation AS (
    SELECT
        i1.IntersectionID AS IntersectionID1,
            i1.IntersectionName AS IntersectionName1,
                i2.IntersectionID AS IntersectionID2,
                    i2.IntersectionName AS IntersectionName2,
                        (6371 * ACOS(
                            COS((i1.Latitude * 3.141592653589793 / 180)) * COS((i2.Latitude * 3.141592653589793 / 180)) * COS((i2.Longitude * 3.141592653589793 / 180) - (i1.Longitude * 3.141592653589793 / 180)) +
                                SIN((i1.Latitude * 3.141592653589793 / 180)) * SIN((i2.Latitude * 3.141592653589793 / 180))
                                    )) AS Distance
                                    FROM
                                        Intersections i1
                                        JOIN
                                            Intersections i2
                                            ON
                                                i1.IntersectionID <> i2.IntersectionID
                                                )
                                                SELECT
                                                    IntersectionID1,
                                                        IntersectionName1,
                                                            IntersectionID2,
                                                                IntersectionName2,
                                                                    Distance
                                                                    FROM
                                                                        DistanceCalculation
                                                                        ORDER BY
                                                                            Distance
                                                                            FETCH FIRST ROW ONLY;
```

INTERSECTIONID1	INTERSECTIONNAME1	INTERSECTIONID2	INTERSECTIONNAME2	DISTANCE
1	Main and Elm	2	Elm and Oak	1.92383830585239327853094434463870895558

```
1 rows returned in 0.02 seconds
```

Explanation:

The query uses a Common Table Expression (CTE) to calculate the distance between each location and the given point using the Haversine formula. The **RADIANS** function converts the latitude and longitude values to radians, and the **ACOS** function calculates the distance in miles. The main query sorts the results by distance and returns the closest 5 locations.

Question 2.5: Optimizing Query for Orders Table

Task:

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.

Deliverables:

1. SQL query optimized for performance, considering indexing, query rewriting, or other techniques.
2. Discussion of strategies used to optimize the query and improve performance.

Solution:

```
SELECT
  o.OrderID,
  o.OrderDate,

  o.CustomerID,

  o.TotalAmount

FROM

  Orders o

WHERE

  o.OrderDate >= DATE_SUB(CURDATE(), INTERVAL 7 DAY)

ORDER BY

  o.OrderDate DESC;
```

Explanation:

The query uses a simple **WHERE** clause to filter orders placed in the last 7 days. To optimize performance, an index can be created on the **OrderDate** column to speed up the filtering process. Additionally, the query can be rewritten to use a **DATE_SUB** function to calculate the date range, which can be more efficient than using a subquery or join.

Question 3:

Question 1: Handling Division Operation

Task

Write a PL/SQL function to handle division operations, including handling division by zero errors.

Solution

```
CREATE OR REPLACE FUNCTION safe_divide(

    p_numerator NUMBER,

    p_denominator NUMBER

) RETURN NUMBER IS

BEGIN

    IF p_denominator = 0 THEN

        RAISE_APPLICATION_ERROR(-20001, 'Division by zero is not allowed');

    END IF;

    RETURN p_numerator / p_denominator;

END safe_divide;
```

Explanation

This function takes two parameters, **p_numerator** and **p_denominator**, and returns their division result. If the **p_denominator** is zero, the function raises an application error with a custom error message.

Question 2: Updating Rows with FORALL

Task

Write a PL/SQL procedure to update multiple rows in a table using the **FORALL** statement.

Solution

```
CREATE OR REPLACE PROCEDURE update_rows(
   p_ids IN SYS.ODCINUMBERLIST,
   p_new_values IN SYS.ODCIVARCHAR2LIST
) IS
BEGIN
   FORALL i IN 1..p_ids.COUNT
      UPDATE my_table
      SET column_name = p_new_values(i)
      WHERE id = p_ids(i);
END update_rows;
```

xplanation

This procedure takes two input parameters, **p_ids** and **p_new_values**, which are collections of numbers and strings, respectively. The **FORALL** statement updates multiple rows in the **my_table** table, setting the **column_name** column to the corresponding value in **p_new_values** for each row with an **id** matching the corresponding value in **p_ids**.

Question 3: Implementing Nested Table Procedure

Task

Write a PL/SQL procedure to insert data into a nested table.

Solution

```
CREATE OR REPLACE PROCEDURE insert_into_nested_table(
   p_data IN SYS.ODCINUMBERLIST
) IS
   v_nested_table my_table_type := my_table_type();
```

```
BEGIN
  FOR i IN 1..p_data.COUNT
    v_nested_table.EXTEND;
    v_nested_table(i) := p_data(i);
  END LOOP;
  INSERT INTO my_table (nested_column)
  VALUES (v_nested_table);
END insert_into_nested_table;
```

Explanation

This procedure takes a collection of numbers, **p_data**, as input. It creates a nested table, **v_nested_table**, and inserts the input data into it using a loop. Finally, it inserts the nested table into the **my_table** table.

Question 4: Using Cursor Variables and Dynamic SQL

Task

Write a PL/SQL procedure to execute dynamic SQL using cursor variables.

Solution

```
CREATE OR REPLACE PROCEDURE execute_dynamic_sql(
  p_sql IN VARCHAR2
) IS
  v_cursor SYS_REFCURSOR;
  v_row my_table%ROWTYPE;
BEGIN
  OPEN v_cursor FOR p_sql;
  LOOP
    FETCH v_cursor INTO v_row;
    EXIT WHEN v_cursor%NOTFOUND;
    -- Process the row data
    DBMS_OUTPUT.PUT_LINE(v_row.column_name);
  END LOOP;
  CLOSE v_cursor;
```

```
END execute_dynamic_sql;
```

Explanation

This procedure takes a dynamic SQL statement, **p_sql**, as input. It opens a cursor variable, **v_cursor**, for the dynamic SQL and fetches the rows into a **v_row** variable. The procedure then processes the row data and closes the cursor.

Question 5: Designing Pipelined Function for Sales Data

Task

Write a PL/SQL pipelined function to return sales data.

Solution

```
CREATE OR REPLACE PACKAGE sales_data_pkg IS
  TYPE sales_data_type IS RECORD (
    sale_date DATE,
    product_name VARCHAR2(50),
    quantity NUMBER,
    total_amount NUMBER
  );
  TYPE sales_data_table IS TABLE OF sales_data_type;

  FUNCTION get_sales_data(
    p_start_date DATE,
    p_end_date DATE
  ) RETURN sales_data_table PIPELINED;
END sales_data_pkg;

CREATE OR REPLACE PACKAGE BODY sales_data_pkg IS
  FUNCTION get_sales_data(
    p_start_date DATE,
    p_end_date DATE
  ) RETURN sales_data_table PIPELINED IS
```

```
    v_sale_date DATE;
    v_product_name VARCHAR2(50);
    v_quantity NUMBER;
    v_total_amount NUMBER;
    CURSOR sales_cur IS
      SELECT sale_date, product_name, quantity, total_amount
      FROM sales
      WHERE sale_date BETWEEN p_start_date AND p_end_date;
  BEGIN
    FOR sales_rec IN sales_cur LOOP
      v_sale_date := sales_rec.sale_date;
      v_product_name := sales_rec.product_name;
      v_quantity := sales_rec.quantity;  -- Assign the quantity value
      v_total_amount := sales_rec.total_amount;
      PIPE ROW(sales_data_type(v_sale_date, v_product_name, v_quantity, v_total_amount));
    END LOOP;
    CLOSE sales_cur;
    RETURN;
  END get_sales_data;
END sales_data_pkg;
```

Explanation

This pipelined function returns a table of sales data, including sale date, product name, quantity, and total amount, for a specified date range. The function uses a cursor to fetch the sales data and pipes each row to the caller using the **PIPE ROW** statement.

How to Use the Pipelined Function

To use the pipelined function, you can call it in a SQL statement, like this:

```
SELECT * FROM TABLE(sales_data_pkg.get_sales_data(DATE '2022-01-01', DATE '2022-01-31'));
```

This will return the sales data for the specified date range.

Deliverables

1. PL/SQL function to handle division operations, including handling division by zero errors.
2. PL/SQL procedure to update multiple rows in a table using the **FORALL** statement.
3. PL/SQL procedure to insert data into a nested table.
4. PL/SQL procedure to execute dynamic SQL using cursor variables.
5. PL/SQL pipelined function to return sales data for a specified date range.