# DATA BASE MANAGEMENT SYSTEM

# ASSIGNMENT

**Question 1:**

**1. ER Diagram:**

**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 Question for Traffic Flow Management System (TFMS)**

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

**Relationships**

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

**Cardinality**: 1: N

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

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

**Cardinality:** M: N

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

1. 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)

1. 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)

1. 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)

**Justification and Normalization Considerations**

1. **Scalability**: The design allows for easy expansion and real-time data integration.
2. **Normalization**: 1NF, 2NF, 3NF principles ensure minimal redundancy and improved data integrity.
   * 1NF (First Normal Form): All entities have atomic values, with no repeating groups or arrays, ensuring that each attribute holds a single value.
   * 2NF (Second Normal Form): All attributes are fully functionally dependent on the primary key, ensuring no partial dependency.
   * 3NF (Third Normal Form): All non-key attributes are non-transitively dependent on the primary key, eliminating redundancy and ensuring data integrity.

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

| 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:**

1. A road can have multiple intersections (one-to-many): Roads -> Intersections
2. An intersection can have multiple roads (many-to-many): Intersections -> Roads
3. An intersection can have one traffic signal (one-to-one): Intersections -> Traffic Signals
4. A traffic signal is associated with one intersection (one-to-one): Traffic Signals -> Intersections
5. A road can have multiple traffic data points (one-to-many): Roads -> Traffic Data
6. Roads (1) ——————— (∞) Intersections
7. Intersections (1) ——————— (∞) Traffic Signals
8. Roads (1) ——————— (∞) Traffic Data

**Cardinality and Optionality:**

1. **One-to-One (1:1)**: A line with a "1" near both entities.
2. **One-to-Many (1: ∞)**: A line with a "1" near one entity and a "∞" near the other.
3. **Many-to-Many (∞: ∞)**: A line with "∞" near both entities.
4. **Optionality**:
   * A circle (O) represents that a relationship is optional.
   * A straight line ( | ) indicates that a relationship is mandatory.

**Notation Key Summary:**

**Rectangles** = Entities

**Ellipses** = Attributes

**Diamonds** = Relationships

**Lines** = Connect Entities

**1:1**, **1:∞**, **∞:∞** = Cardinality

**O** = Optional relationship

**|** = Mandatory relationship

**Question 2:**

1. **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:**

**LEFT JOIN** ensures all departments are included, even those without employees (resulting in **NULL** for **AvgSalary**).

The query calculates the average salary per department and sorts them in descending order, retrieving the top 3 departments.

1. **Question 2: Retrieving Hierarchical Category Paths**

**SQL Query:**

sql

Copy code

WITH RECURSIVE CategoryPath AS ( SELECT CategoryID, CategoryName, CategoryName AS Path

FROM Categories

WHERE ParentCategoryID IS NULL UNION ALL

SELECT C.CategoryID, C.CategoryName, CONCAT(CP.Path, ' > ', C.CategoryName) AS Path FROM Categories C INNER JOIN CategoryPath CP

ON C.ParentCategoryID = CP.CategoryID ) SELECT CategoryID, CategoryName, Path

FROM CategoryPath;

**Explanation:**

The recursive CTE starts with top-level categories and builds the hierarchical path by recursively joining child categories to their parents.

The **CONCAT** function is used to build the hierarchical path.

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

**SQL Query:**

sql

Copy code

WITH Months AS ( SELECT TO\_CHAR(ADD\_MONTHS(TRUNC(SYSDATE, 'YEAR'), LEVEL - 1), 'Month') AS MonthName, LEVEL AS MonthNumber

FROM DUAL CONNECT BY LEVEL <= 12 )

SELECT M.MonthName, COALESCE(COUNT(DISTINCT O.CustomerID), 0) AS CustomerCount

FROM Months M LEFT JOIN Orders O

ON TO\_CHAR(O.OrderDate, 'Month') = M.MonthName

WHERE TO\_CHAR(O.OrderDate, 'YYYY') = TO\_CHAR(SYSDATE, 'YYYY')

GROUP BY M.MonthName, M.MonthNumber

ORDER BY M.MonthNumber;

**Explanation:**

**Months CTE** generates all months of the current year.

**LEFT JOIN** ensures all months are included, and **COALESCE** handles months with no customer activity by setting the count to 0.

1. **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 the given point and each location.

The closest 5 locations are selected based on the calculated distance.

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

**SQL Query:**

sql

Copy code

SELECT OrderID, OrderDate, CustomerID, TotalAmount

FROM Orders

WHERE OrderDate >= SYSDATE - 7

ORDER BY OrderDate DESC;

**Optimization Strategies:**

**Indexing**: Ensure an index exists on **OrderDate** to speed up retrieval.

**Query Rewrite**: The **SYSDATE - 7** is computed once, reducing computation during execution.

**Partitioning**: If the table is large, partition by **OrderDate** to improve performance.

**Question 3:**

* + 1. **Question 1: Handling Division Operation**

**PL/SQL Block:**

plsql

Copy code

DECLARE numerator NUMBER := 100; denominator NUMBER; result NUMBER;

BEGIN -- Get denominator input denominator := &denominator; -- Perform division result := numerator / denominator;

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

EXCEPTION WHEN ZERO\_DIVIDE THEN DBMS\_OUTPUT.PUT\_LINE

('Error: Division by zero is not allowed.');

END;

**Explanation:**

**ZERO\_DIVIDE** exception is handled gracefully by displaying an error message instead of causing the program to fail.

* + 1. **Question 2: Updating Rows with FORALL**

**PL/SQL Block:**

plsql

Copy code

DECLARE TYPE t\_emp\_ids IS TABLE OF Employees.EmployeeID%TYPE;

TYPE t\_increments IS TABLE OF Employees.Salary%TYPE;

emp\_ids t\_emp\_ids := t\_emp\_ids(101, 102, 103);

increments t\_increments := t\_increments(500, 700, 600);

BEGIN FORALL i IN emp\_ids.FIRST .. emp\_ids.LAST

UPDATE Employees SET Salary = Salary + increments(i)

WHERE EmployeeID = emp\_ids(i);

END;

**Explanation:**

**FORALL** improves performance by performing bulk updates in a single context switch between PL/SQL and SQL, reducing overhead.

* + 1. **Question 3: Implementing Nested Table Procedure**

**PL/SQL Procedure:**

plsql

Copy code

CREATE OR REPLACE PROCEDURE get\_employees\_by\_dept ( p\_dept\_id IN Departments.DepartmentID%TYPE, p\_employees OUT t\_emp\_table ) IS TYPE t\_emp\_table IS TABLE OF Employees%ROWTYPE;

v\_employees t\_emp\_table;

BEGIN SELECT \* BULK COLLECT INTO v\_employees

FROM Employees

WHERE DepartmentID = p\_dept\_id;

p\_employees := v\_employees;

END;

**Explanation:**

**Nested Tables** allow for storing and returning collections of employee data, enabling easy manipulation of the dataset within PL/SQL.

* + 1. **Question 4: Using Cursor Variables and Dynamic SQL**

**PL/SQL Block:**

plsql

Copy code

DECLARE TYPE t\_cursor IS REF CURSOR;

c\_emp\_cursor t\_cursor; v\_sql VARCHAR2(200);

v\_employee Employees%ROWTYPE;

v\_salary\_threshold NUMBER := 50000;

BEGIN v\_sql := 'SELECT EmployeeID, FirstName, LastName

FROM Employees

WHERE Salary > :salary\_threshold';

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

LOOP FETCH c\_emp\_cursor INTO v\_employee;

EXIT WHEN c\_emp\_cursor%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE(v\_employee.EmployeeID || ' - ' || v\_employee.FirstName || ' ' || v\_employee.LastName); END LOOP;

CLOSE c\_emp\_cursor; END;

**Explanation:**

**Dynamic SQL** with **REF CURSOR** allows for flexible query execution based on runtime conditions, here retrieving employees with salaries above a specified threshold.

* + 1. **Question 5: Designing Pipelined Function for Sales Data**

**PL/SQL Function:**

plsql

Copy code

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

RETURN t\_sales\_table PIPELINED IS v\_sales\_row t\_sales\_table%ROWTYPE; 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 PIPE ROW (r.OrderID, r.CustomerID, r.OrderAmount);

END LOOP;

RETURN;

END;

**Explanation:**

**Pipelined Functions** enable the function to return rows iteratively, improving performance by streaming results back to the caller as they are generated, rather than waiting for the entire dataset to be prepared.