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.**

ANSWER:

Task 1: Entity Identification and

Attributes Entities and Attributes

1. **Roads**
   - **RoadID (PK): Unique identifier for each road**
   - **RoadName: Name of the road**
   - **Length: Length of the road in meters**
   - **SpeedLimit: Speed limit on the road in km/h**
2. **Intersections**
   - **IntersectionID (PK): Unique identifier for each intersection**
   - **IntersectionName: Name of the intersection**
   - **Latitude: Latitude coordinate of the intersection**
   - **Longitude: Longitude coordinate of the intersection**
3. **Traffic Signals**
   - **SignalID (PK): Unique identifier for each traffic signal**
   - **IntersectionID (FK): Foreign key referencing IntersectionID in Intersections**
   - **SignalStatus: Current status of the signal (Green, Yellow, Red)**
   - **Timer: Countdown to the next change in signal status**
4. **Traffic Data**

- **TrafficDataID (PK): Unique identifier for each traffic data entry**
- **RoadID (FK): Foreign key referencing RoadID in Roads**
- **Timestamp: Time when the data was collected**
- **Speed: Average speed of the traffic on the road**
- **CongestionLevel: Degree of traffic congestion**

Task 2: Relationship Modeling

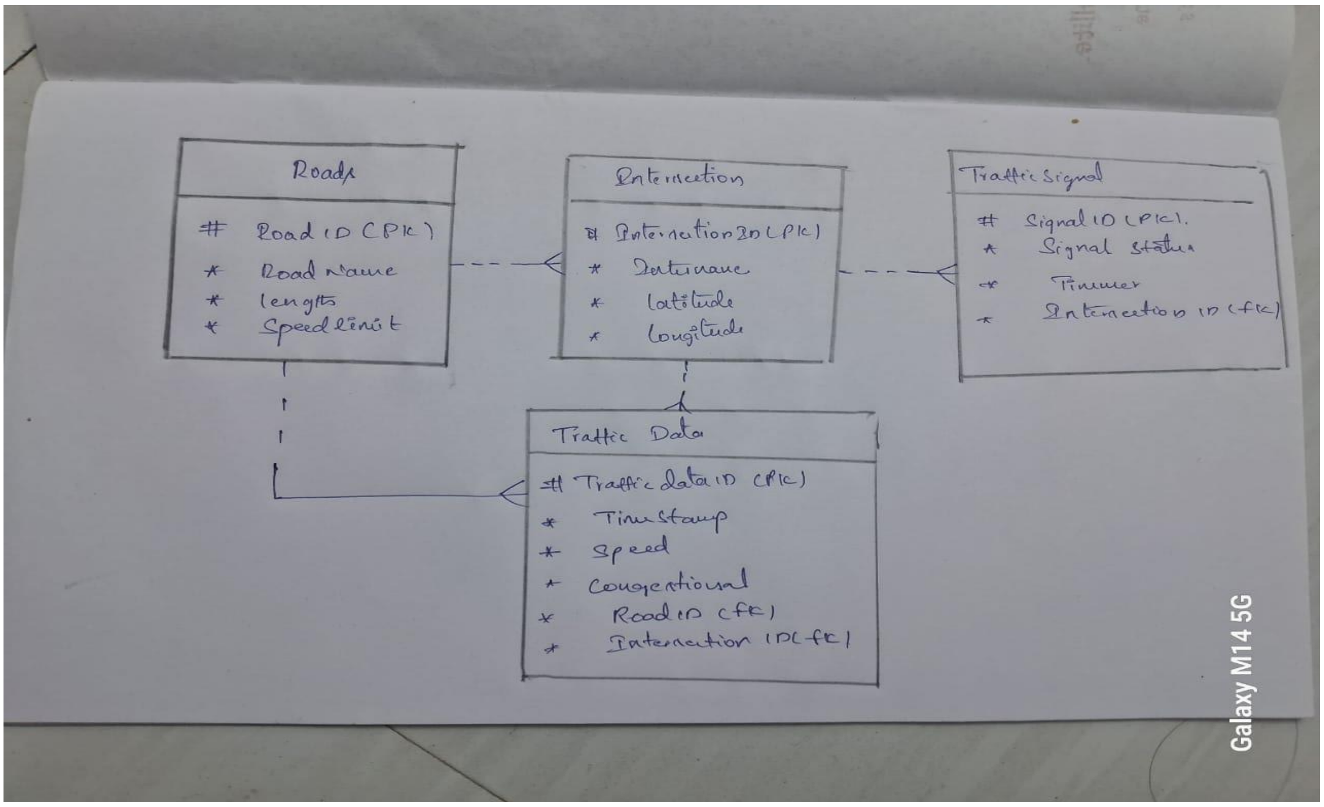
Relationships

1. **Roads to Intersections**
   - **A road can connect to many intersections (one-to-many relationship)**
   - **An intersection must connect to at least one road (mandatory relationship)**
2. **Intersections to Traffic Signals**
   - **An intersection can host multiple traffic signals (one-to-many relationship)**
   - **A traffic signal must be hosted at one intersection (mandatory relationship)**
3. **Roads to Traffic Data**
   - **A road can have multiple traffic data entries (one-to-many relationship)**
   - **Traffic data must be associated with one road (mandatory relationship)**

Cardinality and Optionality

1. **Roads to Intersections**
   - **One road can have multiple intersections**
   - **One intersection can be connected to multiple roads (many-to-many relationship). This can be represented through a linking table (Road_Intersections).**
2. **Intersections to Traffic Signals**
   - **One intersection can have multiple traffic signals**
   - **One traffic signal must be at one intersection**
3. **Roads to Traffic Data**
   - **One road can have multiple traffic data entries**
   - **One traffic data entry must be associated with one road**

Task 3: ERD Diagram



Task 4: Justification and

Normalization Justification

- **Scalability: The design can accommodate additional roads, intersections, signals, and traffic data entries without major changes.**
- **Real-Time Data Processing: The structure supports real-time updates and historical data storage, essential for adaptive signal control and route optimization.**
- **Efficient Traffic Management: Clear relationships between roads, intersections, and traffic signals enable precise control and monitoring.**

Normalization

1NF: Each attribute contains only atomic values.

2NF: No partial dependencies; attributes depend fully on the primary key.

3NF: No transitive dependencies; non-key attributes depend only on the primary key.

This ER diagram ensures data integrity and minimizes redundancy by properly organizing entities and their relationships.

ANSWER:

Question 2: SQL Question 1: Top 3 Departments with Highest Average Salary Task: Deliverables: 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. 1. SQL query that retrieves DepartmentID, DepartmentName, and Avg Salary 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: Deliverables: 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). 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 1: Top 3 Departments with Highest

Average Salary SQL Query

```
SELECT

  d.Department

  ID,

  d.Department

  Name,

  AVG(e.Salary) AS

AvgSalary FROM

  Departme

nts d LEFT

JOIN

  Employees e ON d.DepartmentID =
e.DepartmentID GROUP BY
  d.DepartmentID,
d.DepartmentName
ORDER BY
  AvgSalary
DESC
LIMIT 3;
```

Explanation

1. **LEFT JOIN: We use a LEFT JOIN to include all departments, even those without employees. If a department has no employees, the corresponding Salary value will be NULL.**
2. **GROUP BY: We group by DepartmentID and DepartmentName to calculate the average salary for each department.**
3. **AVG Function: The AVG function calculates the average salary. For departments with no employees, the AVG function will return NULL.**

4. **ORDER BY: We order the results by AvgSalary in descending order to get the departments with the highest average salary at the top.**
5. **LIMIT: We limit the results to the top 3 departments.**

Question 2: Retrieving Hierarchical

Category Paths SQL Query

```
WITH RECURSIVE
 CategoryPaths AS (
 SELECT
   c.CategoryI
   D,
   c.Category
   Name,
   c.ParentCategoryID,
   CAST(c.CategoryName AS
 VARCHAR(255)) AS Path FROM
   Categor
 ies c
 WHERE
   c.ParentCategoryI
 D IS NULL UNION
 ALL
 SELECT
   c.CategoryI
   D,
   c.Category
   Name,
   c.ParentCategoryID,
   CONCAT(cp.Path, ' > ',
 c.CategoryName) AS Path FROM
   Categor
 ies c
 INNER
 JOIN
   CategoryPaths cp ON c.ParentCategoryID = cp.CategoryID
)
SEL
ECT
 CategoryI
 D,
 CategoryN
 ame, Path
FROM
```

CategoryPaths;

Explanation

1. **Common Table Expression (CTE):** We use a recursive CTE named CategoryPaths to build the hierarchical paths.
2. **Base Case:** The initial part of the CTE selects categories with no parent (ParentCategoryID IS NULL), setting the Path to the category's name.
3. **Recursive Case:** The recursive part joins the Categories table with the CategoryPaths CTE on the ParentCategoryID. It appends the current category's name to the path built so far using CONCAT.
4. **Result:** The final SELECT statement retrieves the CategoryID, CategoryName, and the complete hierarchical path (Path) for each category.

This approach ensures that we can traverse and build the hierarchical paths for all categories, starting from the root categories and moving downwards through their children.

Question 3: Total Distinct

Customers by Month SQL Query

sql

Copy code

```sql
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'
),
CustomerActiv
  ity AS (
  SELECT
```

```
    EXTRACT(MONTH FROM
    PurchaseDate) AS MonthNum,
    COUNT(DISTINCT CustomerID) AS
    CustomerCount
  FROM
    Purch
  ases
  WHER
  E
    EXTRACT(YEAR FROM PurchaseDate) =
  EXTRACT(YEAR FROM CURRENT_DATE) GROUP BY
    EXTRACT(MONTH FROM PurchaseDate)
)
SEL
ECT
  m.MonthName,
  COALESCE(ca.CustomerCount, 0)
AS CustomerCount FROM
  Month
s m
LEFT
JOIN
  CustomerActivity ca ON m.MonthNum =
ca.MonthNum ORDER BY
  m.MonthNum;
```

Explanation

1. **Months CTE: This Common Table Expression (CTE)**

generates a list of all months with their respective numerical representation and names.

2. **CustomerActivity CTE: This CTE calculates the count of distinct customers for each month of the current year.**
   - **It extracts the month and year from the PurchaseDate and filters for the current year.**
   - **The distinct customer count for each month is then grouped and calculated.**
3. **LEFT JOIN: We perform a LEFT JOIN between the Months CTE and the CustomerActivity CTE to ensure all months are included, even if there are no purchases.**
4. **COALESCE: The COALESCE function replaces NULL values (which occur when there are no purchases in a month) with 0.**
5. **ORDER BY: Finally, we order the results by MonthNum to ensure the months are listed chronologically.**

Question 4: Finding Closest Locations

SQL Query

sql

Copy

code

```
SELE
CT
  LocationI
  D,
  LocationN
  ame,
  Latitude,
  Longit
  ude,(
    3959 * acos(
      cos(radians(@given_latitude)) * cos(radians(Latitude)) *
cos(radians(Longitude) - radians(@given_longitude)) +
      sin(radians(@given_latitude)) * sin(radians(Latitude))
    )
  ) AS
Distance
FROM
  Locati
ons
ORDE
R BY
  Dista
nce
```

LIMIT

5;

Explanation

1. **Haversine Formula: This query uses the Haversine formula to calculate the great-circle distance between two points on the Earth's surface specified by latitude and longitude.**
   - **The formula: 3959 * acos(cos(radians(lat1)) * cos(radians(lat2)) * cos(radians(lon2) - radians(lon1)) + sin(radians(lat1)) * sin(radians(lat2))) gives the distance in miles (use 6371 for kilometers).**
2. **Variables: @given_latitude and @given_longitude are placeholders for the given point's latitude and longitude.**
3. **Distance Calculation: The query calculates the distance for each location from the given point.**
4. **ORDER BY: The results are ordered by the calculated Distance to find the closest locations.**
5. **LIMIT: The query limits the result to the top 5 closest locations.**

Question 5: Optimizing Query for Orders Table

SQL Query

sql

Copy

code

```
SELE
CT
  OrderID
  ,
  Custome
  rID,
  OrderDa
  te,
  TotalAm
  ount
FROM
  Ord
ers
WHE
RE
  OrderDate >= CURRENT_DATE -
INTERVAL '7 days' ORDER BY
  OrderDate DESC;
```

Discussion of Optimization Strategies

1. **Indexing:**
   - **Ensure an index exists on the OrderDate column to improve the speed of the WHERE clause filter.**
   - **Consider a composite index on (OrderDate, OrderID) if the**

query often sorts or filters by OrderDate.

2. **Query Rewriting:**
   - **The query uses CURRENT_DATE - INTERVAL '7 days' to avoid functions on the column, which can hinder the use of indexes.**
3. **Partitioning:**
   - **If the table is extremely large, consider partitioning the Orders table by date. This can significantly improve query performance for date-based queries.**
4. **Limiting Columns:**
   - **Retrieve only necessary columns (OrderID, CustomerID, OrderDate, TotalAmount) to reduce I/O and memory usage.**
5. **Batch Processing:**
   - **For very large datasets, consider processing in batches to avoid long-running queries and potential timeouts.**

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.

ANSWER:

Question 1: Handling Division Operation

1. **PL/SQL Block for Division Operation and Exception Handling**

plsql

Copy

code

```
DECL
ARE
  numerator NUMBER := 100;  --
  Example numerator divisor
  NUMBER;
  result
NUMBER;
BEGIN
  -- Get the divisor from user input (assuming a value
  is already assigned) divisor := &divisor;

  -- Perform the division
  result := numerator / divisor;


  -- Display the result
  DBMS_OUTPUT.PUT_LINE('
Result: ' || result); EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Error: Division by
  zero is not allowed.'); WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Er
ror: ' || SQLERRM); END;
```

2. **Explanation of Error Handling Strategies**

- **ZERO_DIVIDE Exception: This specific exception is handled to catch division by zero errors. An appropriate error message is displayed using DBMS_OUTPUT.PUT_LINE.**
- **OTHERS Exception: This generic exception handler catches all other exceptions that may occur. It provides an error message that includes the specific error using SQLERRM.**

Question 2: Updating Rows with FORALL

1. **PL/SQL Block Using FORALL**

plsql

Copy

code

```
DECL
ARE
  TYPE emp_id_array IS TABLE OF
  employees.employee_id%TYPE; TYPE
  salary_inc_array IS TABLE OF
  employees.salary%TYPE;
  emp_ids emp_id_array := emp_id_array(101, 102, 103);
  salary_incs salary_inc_array :=
salary_inc_array(500, 1000, 1500); BEGIN
  FORALL i IN emp_ids.FIRST
    .. emp_ids.LAST UPDATE
    employees
    SET salary = salary +
    salary_incs(i) WHERE
    employee_id =
    emp_ids(i);

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

END;

2. **Description of FORALL Performance Improvement**

- **FORALL Statement: This statement allows for bulk binding, which reduces context switches between the PL/SQL engine and the SQL engine. Instead of executing an update statement for each row individually, FORALL sends all updates in a single context switch, improving performance significantly, especially for large data sets.**

Question 3: Implementing Nested Table Procedure

1. **PL/SQL Procedure with Nested Table Implementation**

plsql

Copy code

```
CREATE OR REPLACE TYPE emp_rec IS OBJECT (
  employee_id
  NUMBER,
  first_name
  VARCHAR2(50),
  last_name
  VARCHAR2(50),
  salary NUMBER
);

CREATE OR REPLACE TYPE emp_table IS TABLE OF emp_rec;

CREATE OR REPLACE PROCEDURE
  get_employees_by_dept( p_dept_id IN
  NUMBER,
  p_employees OUT emp_table
) IS
```

```
BEGIN
  SELECT emp_rec(employee_id,
  first_name, last_name, salary) BULK
  COLLECT INTO p_employees
  FROM employees
  WHERE department_id = p_dept_id;

  -- If no employees found, return an
  empty nested table IF p_employees IS
  NULL THEN
    p_employees := emp_table();
```

END
IF;

END;

2. Explanation of Nested Tables Utilization and Return

- **Nested Tables: These are collections that can be used to store sets of data within a single variable. In this procedure, an object type emp_rec is defined to store employee details. A nested table type emp_table is created to hold multiple employee records.**
- **Returning Nested Tables: The procedure get_employees_by_dept accepts a department ID and returns a collection of employees (nested table) belonging to that department. The BULK COLLECT INTO clause is used to fetch multiple rows into the nested table efficiently.**

These implementations handle exceptions, improve performance, and utilize advanced PL/SQL features like collections and bulk operations.

Question 4: Using Cursor Variables and Dynamic SQL

1. **PL/SQL Block Demonstrating Cursor Variables with Dynamic SQL**

plsql

Copy

code

```
DECL

ARE

  TYPE ref_cursor IS

  REF CURSOR;

  c_ref_cursor

  ref_cursor;
  v_sql VARCHAR2(4000);
  v_salary_threshold NUMBER := &salary_threshold;  -- Assume user

  provides salary threshold v_employee_id

  employees.employee_id%TYPE;
  v_first_name

  employees.first_name%TYPE;

  v_last_name

  employees.last_name%TYPE;

BEGIN

  -- Construct the dynamic SQL statement

  v_sql := 'SELECT employee_id, first_name, last_name FROM
  employees WHERE salary >
:salary_threshold';


  -- Open the cursor for the dynamic SQL statement

  OPEN c_ref_cursor FOR v_sql USING v_salary_threshold;


  -- Fetch the

  resultsLOOP
```

```
    FETCH c_ref_cursor INTO v_employee_id,

    v_first_name, v_last_name; EXIT WHEN

    c_ref_cursor%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE('Employee ID: ' || v_employee_id || ',
    Name: ' || v_first_name || ' '
||

  v_last_na

  me); END

  LOOP;


  -- Close the

  cursor

  CLOSE

  c_ref_cursor;

END;
```

2. Explanation of Dynamic SQL Construction and Execution

- Dynamic SQL Construction: The SQL query is built as a string and can include placeholders for bind variables. This allows the query to be modified at runtime.
- Execution: The dynamic SQL is executed using cursor variables (REF CURSOR). The OPEN ... FOR statement opens the cursor for the dynamic SQL, and bind variables are passed using the USING clause.
- Fetching Results: The results are fetched in a loop, and the cursor is closed after all rows are processed. This method is flexible and allows queries to be constructed and executed dynamically based on runtime conditions.

Question 5: Designing Pipelined Function for Sales Data

1. PL/SQL Code for Pipelined Function

plsql

Copy code

```
CREATE OR REPLACE TYPE sales_rec IS OBJECT (
  order_id NUMBER,
  customer_id
  NUMBER,
  order_amount
  NUMBER
);

CREATE OR REPLACE TYPE sales_table IS TABLE OF sales_rec;

CREATE OR REPLACE FUNCTION get_sales_data(p_month IN
NUMBER, p_year IN NUMBER) RETURN sales_table PIPELINED
IS
```

```
  CURSOR c_sales IS
    SELECT order_id,
    customer_id, order_amount
    FROM sales
    WHERE EXTRACT(MONTH FROM
     order_date) = p_month AND
     EXTRACT(YEAR FROM order_date)
     = p_year;
  v_sales_row
c_sales%ROWTYPE;
BEGIN
  OPEN
  c_sales;
  LOOP
    FETCH c_sales INTO
    v_sales_row; EXIT
    WHEN
    c_sales%NOTFOUND
    ;
    PIPE ROW(sales_rec(v_sales_row.order_id,
v_sales_row.customer_id, v_sales_row.order_amount));
  END LOOP;
  CLOSE
  c_sales;
  RETURN
  ;
END;
```

2. Explanation of Pipelined Table Functions

- **Pipelined Table Functions: These functions allow the return of** rows one at a time, which canbe consumed by the calling query immediately. This method enhances performance by starting the data retrieval process before the entire result set is fully fetched.

- **Efficiency:** The pipelined function processes and returns rows in a streaming manner, reducing memory usage and improving responsiveness for large data sets. It allows for parallel processing and can be more efficient than returning a large collection all at once.

These implementations demonstrate advanced PL/SQL features like cursor variables, dynamic SQL, and pipelined functions, providing efficient and flexible data processing solutions.