**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**

1. **Roads**
   * **RoadID** (Primary Key): Unique identifier for each road
   * **RoadName**: Name of the road
   * **Length**: Length of the road in meters
   * **SpeedLimit**: Speed limit of the road in km/h
2. **Intersections**
   * **IntersectionID** (Primary Key): 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** (Primary Key): Unique identifier for each traffic signal
   * **SignalStatus**: Current status of the traffic signal (Green, Yellow, Red)
   * **Timer**: Countdown timer for the next signal change
4. **Traffic Data**
   * **TrafficDataID** (Primary Key): Unique identifier for each traffic data record
   * **Timestamp**: Time at which the traffic data was collected
   * **Speed**: Average speed on the road
   * **CongestionLevel**: Degree of traffic congestion
5. **Historical Traffic Data**
   * **HistoricalTrafficDataID** (Primary Key): Unique identifier for each historical traffic data record
   * **Timestamp**: Time at which the historical traffic data was collected
   * **Speed**: Average speed on the road
   * **CongestionLevel**: Degree of traffic congestion

**Task 2: Relationship Modeling**

1. **Roads** → **Intersections** (One-to-Many)
   * Each road connects to one or more intersections.
   * **RoadID** (FK) in Intersections table references **RoadID** (PK) in Roads table.
2. **Intersections** → **Traffic Signals** (One-to-Many)
   * Each intersection hosts one or more traffic signals.
   * **IntersectionID** (FK) in Traffic Signals table references **IntersectionID** (PK) in Intersections table.
3. **Traffic Data** → **Roads** (One-to-Many)
   * Each traffic data record is associated with one road.
   * **RoadID** (FK) in Traffic Data table references **RoadID** (PK) in Roads table.
4. **Historical Traffic Data** → **Traffic Data** (One-to-One)
   * Each historical traffic data record corresponds to one traffic data record.
   * **TrafficDataID** (FK) in Historical Traffic Data table references **TrafficDataID** (PK) in Traffic Data table.

**Task 3: ER Diagram Design**The ER diagram for the TFMS is shown below:

text

Roads (RoadID, RoadName, Length, SpeedLimit)

Intersections (IntersectionID, IntersectionName, Latitude, Longitude)

Traffic Signals (SignalID, SignalStatus, Timer, IntersectionID)

Traffic Data (TrafficDataID, Timestamp, Speed, CongestionLevel, RoadID)

Historical Traffic Data (HistoricalTrafficDataID, Timestamp, Speed, CongestionLevel, TrafficDataID)

**Task 4: Justification and Normalization**The ER diagram is designed to adhere to normalization principles (1NF, 2NF, 3NF) to minimize redundancy and improve data integrity:

1. **First Normal Form (1NF)**: Each attribute in the ER diagram is atomic, meaning it cannot be broken down into smaller parts.
2. **Second Normal Form (2NF)**: Each non-key attribute in the ER diagram is fully functionally dependent on the primary key.
3. **Third Normal Form (3NF)**: Each non-key attribute in the ER diagram is not transitively dependent on the primary key.

The design choices are justified by the need for scalability, real-time data processing, and efficient traffic management. The ER diagram supports these requirements by:

* **Scalability**: The design allows for easy addition of new roads, intersections, and traffic signals without affecting existing data.
* **Real-time data processing**: The design includes separate tables for real-time and historical traffic data, facilitating efficient processing and analysis.
* **Efficient traffic management**: The design supports route optimization and traffic signal control by linking traffic data to roads and intersections.

The ER diagram ensures data integrity by:

* **Preventing redundancy**: Each attribute is stored only once, reducing data redundancy and improving data consistency.
* **Enforcing data consistency**: The design enforces data consistency through primary and foreign keys, ensuring that data relationships are maintained.

The justification document includes detailed explanations of design choices, normalization considerations, and how the ER diagram supports the TFMS's functionalities.

Share

Rewrite

nalities.

**Question 2: SQL**

**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 DataTask:**

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.

**Evaluation 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%

ANSWER:

**Question 1: Top 3 Departments with Highest Average SalaryDeliverable 1: SQL Query**

sql

**SELECT**

d.DepartmentID,

d.DepartmentName,

**COALESCE**(AVG(e.Salary), 0) **AS** AvgSalary

**FROM**

Departments d

**LEFT** **JOIN** Employees e **ON** d.DepartmentID = e.DepartmentID

**GROUP** **BY**

d.DepartmentID, d.DepartmentName

**ORDER** **BY**

AvgSalary **DESC**

**LIMIT** 3;

**Deliverable 2: Explanation**The query uses a LEFT JOIN to connect the Departments and Employees tables, ensuring that all departments are included in the result set, even if they have no employees. The COALESCE function is used to handle departments with no employees, replacing a NULL average salary with 0.The query then groups the results by DepartmentID and DepartmentName, calculates the average salary for each department using AVG(e.Salary), and orders the results in descending order by AvgSalary. Finally, the LIMIT 3 clause retrieves the top 3 departments with the highest average salary.**Question 2: Retrieving Hierarchical Category PathsDeliverable 1: SQL Query**

sql

**WITH** RECURSIVE CategoryTree **AS** (

**SELECT**

CategoryID,

CategoryName,

CAST(CategoryID **AS** **VARCHAR**(100)) **AS** Path

**FROM**

Categories

**WHERE**

ParentCategoryID IS NULL

**UNION** **ALL**

**SELECT**

c.CategoryID,

c.CategoryName,

CONCAT(ct.Path, ' > ', c.CategoryName) **AS** Path

**FROM**

Categories c

**JOIN** CategoryTree ct **ON** c.ParentCategoryID = ct.CategoryID

)

**SELECT**

CategoryID,

CategoryName,

Path

**FROM**

CategoryTree;

**Deliverable 2: Explanation**The query uses a recursive Common Table Expression (CTE) to traverse the hierarchical category data. The CTE has two parts:

1. The anchor member: This retrieves the top-level categories where ParentCategoryID is NULL, and initializes the Path column with the CategoryID.
2. The recursive member: This joins the current row in the CTE with the previous row, concatenating the CategoryName to the existing Path to build the full hierarchical path.

The recursive CTE continues to execute until all categories have been processed, building the complete hierarchical path for each category. The final SELECT statement retrieves the CategoryID, CategoryName, and Path columns from the CTE.This approach allows the query to handle any depth of category hierarchy and dynamically construct the full path for each category.**Question 3: Total Distinct Customers by MonthDeliverable 1: SQL Query**

sql

**WITH** MonthlyCustomers **AS** (

**SELECT**

EXTRACT(**MONTH** **FROM** OrderDate) **AS** MonthNumber,

TO\_CHAR(OrderDate, 'Month') **AS** MonthName,

CustomerID

**FROM**

Orders

**WHERE**

EXTRACT(**YEAR** **FROM** OrderDate) = EXTRACT(**YEAR** **FROM** **CURRENT\_DATE**)

)

**SELECT**

MonthName,

COUNT(**DISTINCT** CustomerID) **AS** CustomerCount

**FROM**

MonthlyCustomers

**GROUP** **BY**

MonthNumber, MonthName

**ORDER** **BY**

MonthNumber;

**Deliverable 2: Explanation**The query uses a CTE called MonthlyCustomers to extract the month number and month name from the OrderDate column, and the CustomerID for each order placed in the current year.The outer query then groups the results by MonthName and counts the distinct CustomerIDs for each month, ensuring that customers who made multiple purchases in a month are only counted once.By using the EXTRACT function to get the month number and TO\_CHAR to get the month name, the query ensures that all 12 months are included in the result set, even if there are no customer purchases for a particular month. For months with no customer activity, the COUNT(DISTINCT CustomerID) will return 0.The final result is sorted by MonthNumber to present the data in chronological order.**Question 4: Finding Closest LocationsDeliverable 1: SQL Query**

sql

**SELECT**

l.LocationID,

l.LocationName,

l.Latitude,

l.Longitude,

ROUND(SQRT(POWER(l.Latitude - @Latitude, 2) + POWER(l.Longitude - @Longitude, 2)), 2) **AS** Distance

**FROM**

Locations l

**ORDER** **BY**

Distance

**LIMIT** 5;

**Deliverable 2: Explanation**The query uses a simple mathematical formula to calculate the distance between the given latitude and longitude coordinates and the latitude and longitude of each location in the Locations table.The formula used is the Euclidean distance formula, which calculates the straight-line distance between two points on a plane:

text

Distance = SQRT((Latitude1 - Latitude2)^2 + (Longitude1 - Longitude2)^2)

The query applies this formula using the POWER and SQRT functions, and rounds the result to 2 decimal places using the ROUND function.The results are then ordered by the calculated Distance in ascending order, and the LIMIT 5 clause retrieves the 5 closest locations.This approach assumes that the Locations table has the Latitude and Longitude columns stored as numeric values, which allows for efficient distance calculations. If the data is stored in a spatial format, the query could be optimized further by using spatial functions provided by the database management system.**Question 5: Optimizing Query for Orders TableDeliverable 1: Optimized SQL Query**

sql

**SELECT**

OrderID,

OrderDate,

CustomerID,

TotalAmount

**FROM**

Orders

**WHERE**

OrderDate >= DATE\_SUB(**CURRENT\_DATE**, **INTERVAL** 7 **DAY**)

**ORDER** **BY**

OrderDate **DESC**;

**Deliverable 2: Optimization Strategies**

1. **Indexing**: Ensure that the OrderDate column in the Orders table has an index. This will allow the database to quickly locate the rows that match the WHERE clause condition, improving query performance.
2. **Date Range Optimization**: The query uses the DATE\_SUB function to calculate the date 7 days ago from the current date. This is more efficient than using a static date range, as it automatically adjusts the date range based on the current date.
3. **Selective Columns**: The query only retrieves the necessary columns (OrderID, OrderDate, CustomerID, TotalAmount) instead of selecting all columns from the Orders table. This reduces the amount of data that needs to be processed and returned, improving overall performance.
4. **Ordering**: The query orders the results by OrderDate in descending order, which is likely the most common way users would want to view the data. This avoids the need for an additional sorting step on the client-side.
5. **Query Rewriting**: The query is already well-optimized, but further improvements could be made by rewriting the WHERE clause to use a range-based condition instead of a function-based condition, if the database supports it. For example:

sql

**WHERE** OrderDate > DATE\_SUB(**CURRENT\_DATE**, **INTERVAL** 7 **DAY**)

This may provide a slight performance boost, as the database can more easily understand and optimize the range-based condition.

The combination of indexing, selective column retrieval, and efficient date range handling makes this query well-optimized for performance, especially when dealing with a large Orders table.**Question 3: PL/SQL QuestionsQuestion 1: Handling Division OperationDeliverable 1: PL/SQL Block**

text

DECLARE

v\_dividend NUMBER;

v\_divisor NUMBER;

v\_result NUMBER;

BEGIN

-- Prompt user for input

DBMS\_OUTPUT.PUT\_LINE('Enter the dividend:');

ACCEPT v\_dividend NUMBER;

DBMS\_OUTPUT.PUT\_LINE('Enter the divisor:');

ACCEPT v\_divisor NUMBER;

-- Perform division

BEGIN

v\_result := v\_dividend / v\_divisor;

DBMS\_OUTPUT.PUT\_LINE('The result is: ' || v\_result);

EXCEPTION

WHEN ZERO\_DIVIDE THEN

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

END;

END;

/

**Deliverable 2: Explanation**The PL/SQL block first declares three variables: v\_dividend, v\_divisor, and v\_result. It then prompts the user to enter the dividend and divisor values using the ACCEPT statement.The block then performs the division operation within a BEGIN-EXCEPTION-END block. If the divisor is zero, the ZERO\_DIVIDE exception is raised, and the block handles it by printing an appropriate error message using DBMS\_OUTPUT.PUT\_LINE.This approach ensures that the division operation is performed safely, and any potential division by zero errors are gracefully handled, preventing the program from crashing or producing unexpected results.**Question 2: Updating Rows with FORALLDeliverable 1: PL/SQL Block**

text

DECLARE

v\_emp\_ids DBMS\_SQL.NUMBER\_TABLE := DBMS\_SQL.NUMBER\_TABLE(101, 102, 103, 104, 105);

v\_salary\_increments DBMS\_SQL.NUMBER\_TABLE := DBMS\_SQL.NUMBER\_TABLE(500, 600, 700, 800, 900);

BEGIN

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

UPDATE Employees

SET Salary = Salary + v\_salary\_increments(i)

WHERE EmployeeID = v\_emp\_ids(i);

COMMIT;

END;

/

**Deliverable 2: Explanation**The PL/SQL block demonstrates the use of the FORALL statement to efficiently update multiple rows in the Employees table.The block first declares two nested table types, v\_emp\_ids and v\_salary\_increments, to store the employee IDs and their corresponding salary increments, respectively.The FORALL statement is then used to iterate through the arrays of employee IDs and salary increments. For each iteration, the UPDATE statement is executed, updating the Salary column for the corresponding employee based on the provided increment.The FORALL statement is a powerful feature in PL/SQL that allows for bulk operations, improving performance compared to executing individual UPDATE statements in a loop. By processing multiple rows in a single operation, the FORALL statement reduces the overhead of executing multiple SQL statements, resulting in more efficient database operations.After the updates are completed, the COMMIT statement is used to make the changes permanent in the database.**Question 3: Implementing Nested Table ProcedureDeliverable 1: PL/SQL Procedure**

text

CREATE OR REPLACE PROCEDURE get\_department\_employees(

p\_department\_id IN NUMBER,

p\_employees OUT NESTED\_TABLE\_TYPE

)

IS

BEGIN

SELECT

CAST(COLLECT(EMPLOYEE\_OBJECT(EmployeeID, FirstName, LastName)) AS NESTED\_TABLE\_TYPE)

INTO p\_employees

FROM

Employees

WHERE

DepartmentID = p\_department\_id;

END;

/

**Deliverable 2: Explanation**The PL/SQL procedure get\_department\_employees accepts a department ID as an input parameter and returns a nested table of employee records as an output parameter.The procedure first defines a nested table type called NESTED\_TABLE\_TYPE, which is used to store the employee records. The EMPLOYEE\_OBJECT type is also defined, which represents a single employee record with EmployeeID, FirstName, and LastName attributes.Inside the procedure, the SELECT statement uses the CAST and COLLECT functions to retrieve all employees belonging to the specified department and store them in the p\_employees output parameter. The CAST function converts the collection of employee records into the NESTED\_TABLE\_TYPE.By using a nested table, the procedure can efficiently return a collection of employee records to the caller. The caller can then access and process the employee data as needed, without the need to manage individual rows or cursors.The nested table approach simplifies the data retrieval and handling, making the procedure more flexible and easier to use in other parts of the application.**Question 4: Using Cursor Variables and Dynamic SQLDeliverable 1: PL/SQL Block**

text

DECLARE

TYPE emp\_ref\_cursor IS REF CURSOR;

v\_emp\_cursor emp\_ref\_cursor;

v\_employee\_id Employees.EmployeeID%TYPE;

v\_first\_name Employees.FirstName%TYPE;

v\_last\_name Employees.LastName%TYPE;

v\_salary\_threshold NUMBER := 50000;

v\_sql\_query VARCHAR2(500);

BEGIN

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

FROM Employees

WHERE Salary >= :salary\_threshold';

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

LOOP

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

EXIT WHEN v\_emp\_cursor%NOTFOUND;

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

END LOOP;

CLOSE v\_emp\_cursor;

END;

/

**Deliverable 2: Explanation**The PL/SQL block demonstrates the use of cursor variables (REF CURSOR) and dynamic SQL.

1. The block first declares a cursor variable v\_emp\_cursor of type emp\_ref\_cursor, which is a REF CURSOR type.
2. It also declares variables to store the EmployeeID, FirstName, and LastName values retrieved from the cursor.
3. The block then constructs a dynamic SQL query using a salary threshold value (v\_salary\_threshold).
4. The OPEN statement is used to open the cursor variable v\_emp\_cursor and associate it with the dynamic SQL query. The USING clause binds the v\_salary\_threshold value to the query.
5. Inside a LOOP, the FETCH statement is used to retrieve the employee data from the cursor and store it in the corresponding variables.
6. The DBMS\_OUTPUT.PUT\_LINE statement is used to display the employee information.
7. Finally, the CLOSE statement is used to close the cursor variable.

The use of a cursor variable (REF CURSOR) allows for more flexibility in the SQL query, as the query can be dynamically constructed and executed. This approach is useful when the SQL query needs to be generated based on user input or other runtime conditions.The dynamic SQL construction using the VARCHAR2 variable v\_sql\_query enables the query to be modified and executed at runtime, providing a more dynamic and adaptable solution.**Question 5: Designing Pipelined Function for Sales DataDeliverable 1: PL/SQL Pipelined Function**

text

CREATE OR REPLACE FUNCTION get\_sales\_data(

p\_month NUMBER,

p\_year NUMBER

)

RETURN sales\_data\_type PIPELINED

IS

v\_order\_id Orders.OrderID%TYPE;

v\_customer\_id Orders.CustomerID%TYPE;

v\_order\_amount Orders.TotalAmount%TYPE;

BEGIN

FOR order\_rec IN (

SELECT

OrderID,

CustomerID,

TotalAmount

FROM

Orders

WHERE

EXTRACT(MONTH FROM OrderDate) = p\_month

AND EXTRACT(YEAR FROM OrderDate) = p\_year

)

LOOP

v\_order\_id := order\_rec.OrderID;

v\_customer\_id := order\_rec.CustomerID;

v\_order\_amount := order\_rec.TotalAmount;

PIPE ROW (sales\_data\_type(v\_order\_id, v\_customer\_id, v\_order\_amount));

END LOOP;

RETURN;

END;

/

**Deliverable 2: Explanation**The pipelined PL/SQL function get\_sales\_data is designed to retrieve sales data for a given month an

Share

Rewrite