**CSA0562: DATABASE MANAGEMENT SYSTEMS-ASSIGNMENT QUESTIONS**

* B.RISHITHA

192324130

**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**:
   * **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**:
   * **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**:
   * **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**:
   * **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**:
   * **Route Optimization**: Algorithms will be implemented to suggest optimal routes based on current traffic conditions.
   * **Traffic Signal Control**: Adaptive control algorithms will adjust signal timings dynamically based on real-time traffic flow and congestion data.
   * **Historical Analysis**: The system will store historical traffic data for analysis and planning future improvements.

ER Diagram Design Requirements

1. **Entities and Attributes**:
   * Clearly define entities (Roads, Intersections, Traffic Signals, Traffic Data) and their attributes based on the scenario provided.
   * Include primary keys (PK) and foreign keys (FK) where necessary to establish relationships between entities.
2. **Relationships**:
   * Illustrate relationships between entities (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).
3. **Normalization Considerations**:
   * 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.

Solution:

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**: Geographic latitude of the intersection.
   * **Longitude**: Geographic longitude of the intersection.
3. **Traffic Signals**
   * **SignalID** (PK): Unique identifier for each traffic signal.
   * **IntersectionID** (FK): Identifier of the intersection where the signal is located.
   * **SignalStatus**: Current status of the signal (Green, Yellow, Red).
   * **Timer**: Countdown timer for the next signal change.
4. **Traffic Data**
   * **TrafficDataID** (PK): Unique identifier for each traffic data record.
   * **Timestamp**: Date and time of the data collection.
   * **RoadID** (FK): Identifier of the road where data is collected.
   * **Speed**: Average speed on the road.
   * **CongestionLevel**: Degree of traffic congestion.

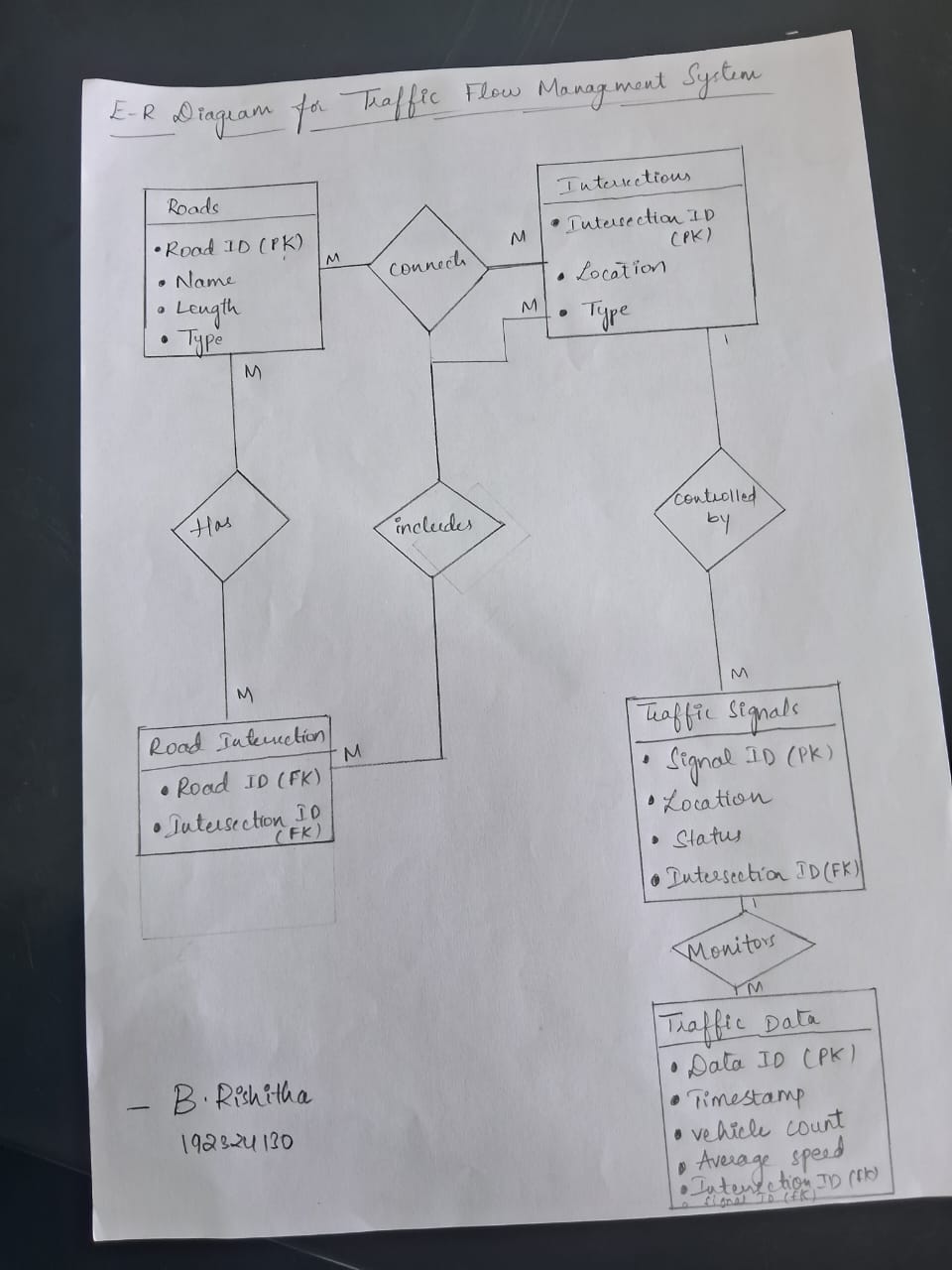
Task 2: Relationship Modeling

**Relationships:**

1. **Roads and Intersections**
   * One road can connect to many intersections (one-to-many).
   * Each intersection can be connected by many roads (many-to-many).
2. **Intersections and Traffic Signals**
   * One intersection can host many traffic signals (one-to-many).
   * Each traffic signal is located at one intersection (many-to-one).
3. **Roads and Traffic Data**
   * One road can have many traffic data records (one-to-many).
   * Each traffic data record is associated with one road (many-to-one).

Task 3: ER Diagram Design

**ER Diagram:**



Task 4: Justification and Normalization

**Justification:**

• To fulfill all TFMS functions, ensuring scalability and real-time data processing speeds, the ER diagram was made to capture all necessary entities as well as their attributes.

• Traffic signals can be controlled, traffic routes managed and traffic data analyzed efficiently by system through clear definition of relationships.

**Normalization:**

1NF (First Normal Form):

• It is important to ensure that all attributes have atomic values and each record is unique.

• All entities have unique primary keys while no attribute has more than one value.

2NF (Second Normal Form):

• It is important to ensure that all non-key attributes are fully functionally dependent on the

primary key.

• Every attribute relates directly with the primary key of the entity.

3NF (Third Normal Form):

• Since no attribute should depend upon another other than being fully functionally

dependent on the primary key, this should also be true for all attributes in a table relating

to third normal form.

• In order to remove transitive dependencies all non-key attributes are removed from the

Entities.

**Question 2:**

* **Question 1:**

**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.

Solution:

TASK

DROP TABLE IF EXISTS Employees;

DROP TABLE IF EXISTS Departments;

CREATE TABLE IF NOT EXISTS Departments (

department\_id INTEGER PRIMARY KEY,

department\_name TEXT NOT NULL

);

CREATE TABLE IF NOT EXISTS Employees (

employee\_id INTEGER PRIMARY KEY,

first\_name TEXT NOT NULL,

last\_name TEXT NOT NULL,

email TEXT NOT NULL UNIQUE,

phone\_number TEXT,

hire\_date DATE NOT NULL,

job\_id INTEGER NOT NULL,

salary DECIMAL(10, 2),

manager\_id INTEGER,

department\_id INTEGER,

FOREIGN KEY (department\_id) REFERENCES Departments(department\_id),

FOREIGN KEY (manager\_id) REFERENCES Employees(employee\_id)

);

INSERT INTO Departments (department\_id, department\_name) VALUES

(1, 'HR'),

(2, 'Engineering'),

(3, 'Sales'),

(4, 'Marketing'),

(5, 'Support');

INSERT INTO Employees (employee\_id, first\_name, last\_name, email, phone\_number, hire\_date, job\_id, salary, manager\_id, department\_id) VALUES

(1, 'Sumiya', 'Shaik', 'sumiya12@example.com', '1234567890', '2020-01-15', 101, 60000, NULL, 1),

(2, 'Aarna', 'reddy', 'aarnalucky@example.com', '1234567891', '2021-02-20', 102, 70000, 1, 2),

(3, 'Preethi', 'Rishi', 'Preethi@example.com', '1234567892', '2019-03-25', 103, 80000, 1, 2),

(4, 'Siri', 'cristie', 'siricristie@example.com', '1234567893', '2018-04-30', 104, 50000, 2, 3),

(5, 'Vijay', 'Kumar', 'vijaykumar@example.com', '1234567894', '2017-05-05', 105, 90000, 2, 4);

SELECT

d.department\_id AS DepartmentID,

d.department\_name AS DepartmentName,

AVG(e.salary) AS AvgSalary

FROM

Departments d

LEFT JOIN

Employees e ON d.department\_id = e.department\_id

GROUP BY

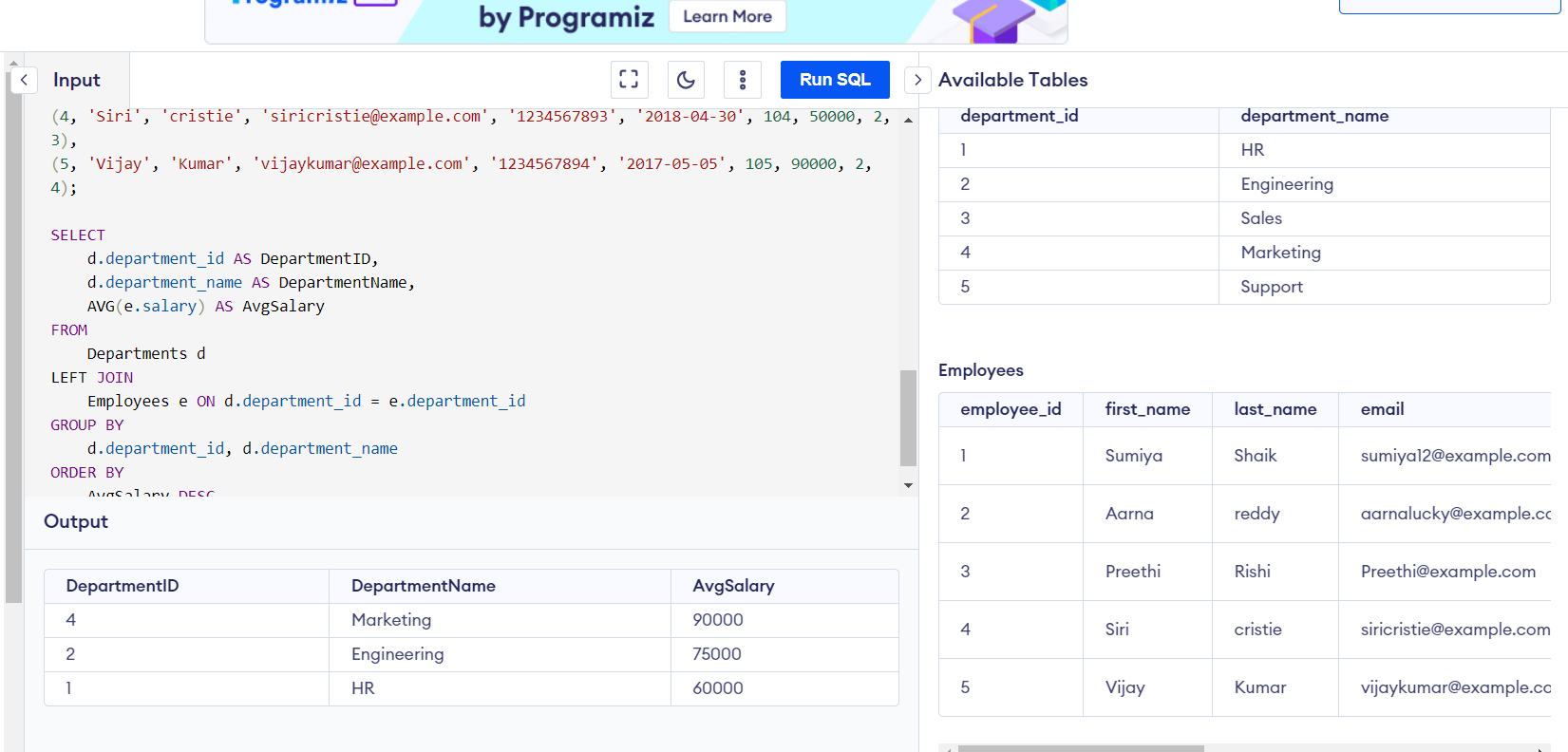
d.department\_id, d.department\_name

ORDER BY

AvgSalary DESC

LIMIT 3;

OUTPUT:



deliverables

Handling Empty Departments:

Left Join: Left join is used in this query between the Departments table and the Employees table. With left join, all records from the left table (Departments) are returned with matched records from the right table (Employees). If there is no match, then NULL appears on the side of Employees.

This means that if a department has no employees, it will still appear in result set with NULL for employee-related fields.

Calculating Average Salaries:

Average Function: The average function calculates a mean salary of employees within each department. The average function calculates the mean of salary values for each department.

Grouping: In order to calculate an average salary for each department, we use GROUP BY clause to group them by department\_id and department\_name.

Null Handling: Whenever AVG encounters NULL values it ignores them while calculating. All joined salaries have NULL values in case of departments without employees hence AVG gives NULL as result for average salary

Sorting and Limiting Results:

Order By Clause: ORDER BY AvgSalary DESC sorts results based on calculated average salary in descending order.

Limit Clause: To obtain top 3 departments

* **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.

Task

Solution:

**DROP TABLE IF EXISTS Categories;**

**CREATE TABLE Categories (**

**CategoryID INT PRIMARY KEY,**

**CategoryName VARCHAR(100),**

**ParentCategoryID INT**

**);**

**INSERT INTO Categories (CategoryID, CategoryName, ParentCategoryID) VALUES**

**(1, 'Electronics', NULL),**

**(2, 'Computers', 1),**

**(3, 'Laptops', 2),**

**(4, 'Desktops', 2),**

**(5, 'Smartphones', 1),**

**(6, 'Cameras', 1),**

**(7, 'DSLR', 6),**

**(8, 'Mirrorless', 6),**

**(9, 'Home Appliances', NULL),**

**(10, 'Refrigerators', 9),**

**(11, 'Microwaves', 9);**

**WITH RECURSIVE CategoryHierarchy AS (**

**SELECT**

**CategoryID,**

**CategoryName,**

**CategoryName AS HierarchicalPath,**

**ParentCategoryID**

**FROM**

**Categories**

**WHERE**

**ParentCategoryID IS NULL**

**UNION ALL**

**SELECT**

**c.CategoryID,**

**c.CategoryName,**

**ch.HierarchicalPath || ' > ' || c.CategoryName AS HierarchicalPath,**

**c.ParentCategoryID**

**FROM**

**Categories c**

**INNER JOIN**

**CategoryHierarchy ch ON ch.CategoryID = c.ParentCategoryID**

**)**

**SELECT**

**CategoryID,**

**CategoryName,**

**HierarchicalPath**

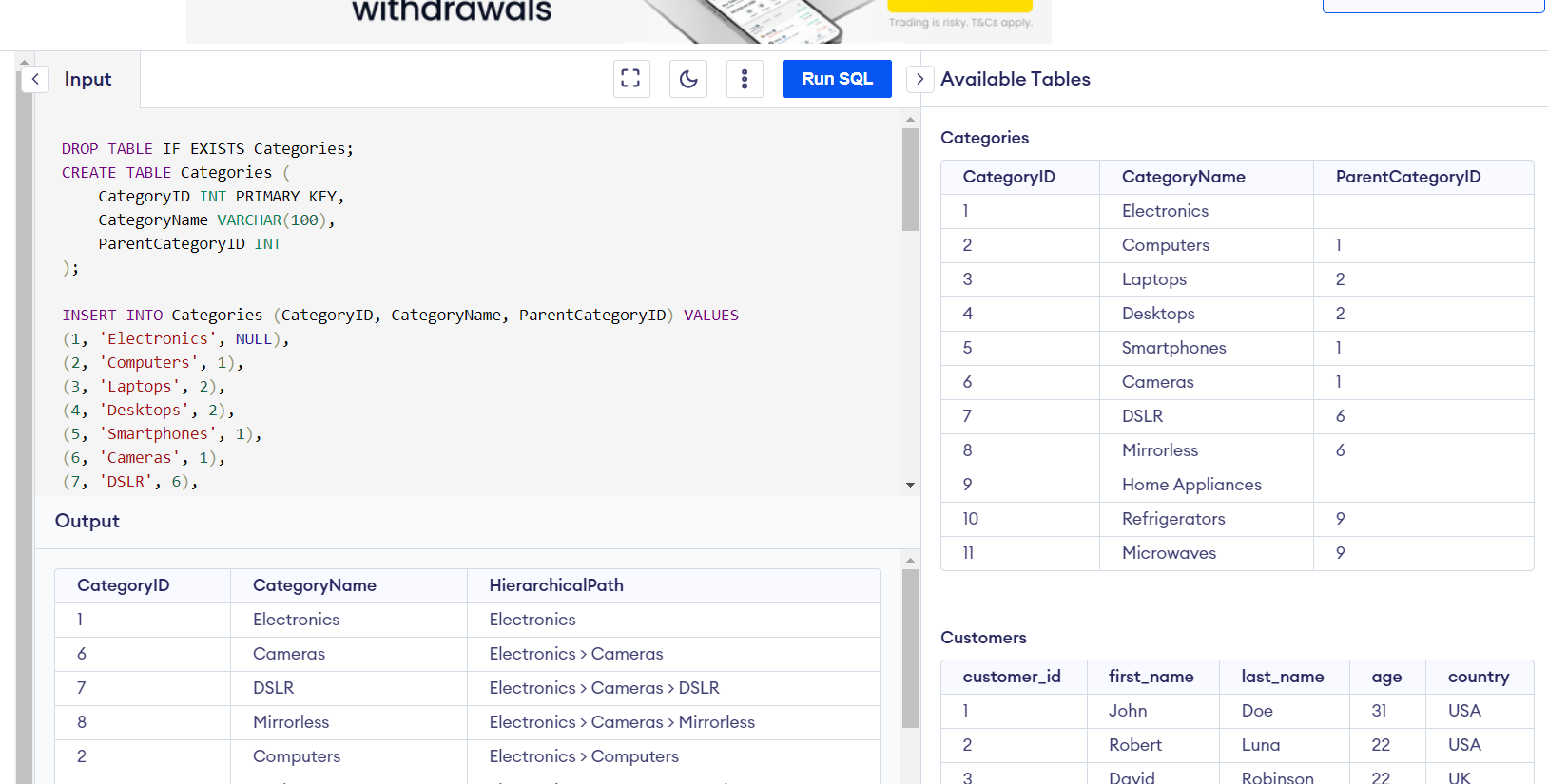
**FROM**

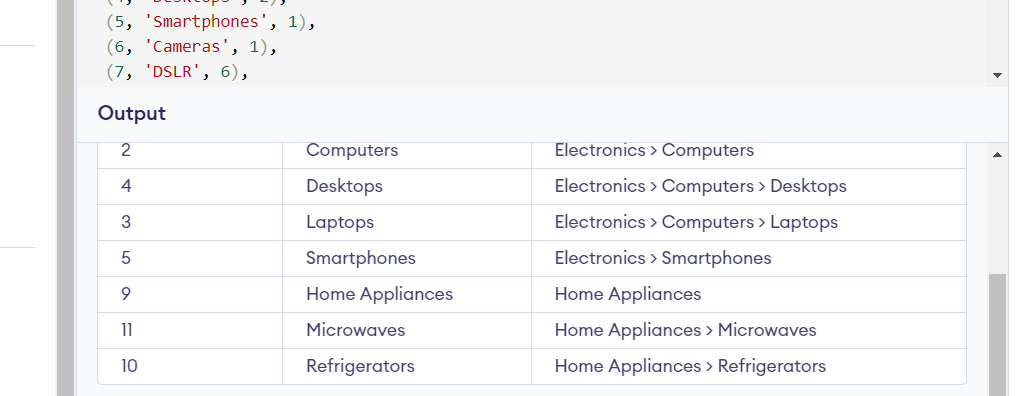
**CategoryHierarchy**

**ORDER BY**

**HierarchicalPath;**

**OUTPUT:**





**Deliverables**

**This SQL query given uses a recursive Common Table Expression (CTE) to traverse hierarchical data in the Categories table. Here’s how it works:**

**Anchor Member:**

**The recursion starting point is called the anchor member. It selects all root categories from the Categories table. A root category is identified by having null in the ParentCategoryID column.**

**The hierarchical path for these root categories is initialized to their own names.**

**Example:**

**SELECT**

**CategoryID,**

**CategoryName,**

**CategoryName AS HierarchicalPath,**

**ParentCategoryID**

**FROM**

**Categories**

**WHERE**

**ParentCategoryID IS NULL**

**With this query, some entries would be like Electronics, Home Appliances and so on.**

**Recursive Member:**

**The recursion step appends each category’s name to its parent category’s hierarchy path. In this regard, a recursive member connects the Categories table with the CategoryHierarchy CTE.**

**This final step recursively builds up for each category, full hierarchical path.**

**Example:**

**SELECT**

**c.CategoryID,**

**c.CategoryName,**

**ch.HierarchicalPath || ' > ' || c.CategoryName AS HierarchicalPath,**

**c.ParentCategoryID**

**FROM**

**Categories c**

**INNER JOIN**

**CategoryHierarchy ch ON ch.CategoryID =c.ParentCategoryID**

**Final Query**:

* The final **SELECT** statement retrieves all categories along with their full hierarchical paths.
* The results are ordered by the **HierarchicalPath** to present the hierarchy in a readable format.

**SELECT**

**CategoryID,**

**CategoryName,**

**HierarchicalPath**

**FROM**

**CategoryHierarchy**

**ORDER BY**

**HierarchicalPath**

**Example of Hierarchical Path Output:**

The output of the query would look something like this:

| **CategoryID** | **CategoryName** | **HierarchicalPath** |
| --- | --- | --- |
| 1 | Electronics | Electronics |
| 2 | Computers | Electronics > Computers |
| 3 | Laptops | Electronics > Computers > Laptops |
| 4 | Desktops | Electronics > Computers > Desktops |
| 5 | Smartphones | Electronics > Smartphones |
| 6 | Cameras | Electronics > Cameras |
| 7 | DSLR | Electronics > Cameras > DSLR |
| 8 | Mirrorless | Electronics > Cameras > Mirrorless |
| 9 | Home Appliances | Home Appliances |
| 10 | Refrigerators | Home Appliances > Refrigerators |
| 11 | Microwaves | Home Appliances > Microwaves |

* **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.

Solution:

**-- Create a table for testing**

**CREATE TABLE purchases (**

**customer\_id INT,**

**purchase\_date DATE**

**);**

**-- Insert some test data**

**INSERT INTO purchases (customer\_id, purchase\_date)**

**VALUES**

**(1, '2023-01-01'),**

**(2, '2023-01-15'),**

**(1, '2023-02-01'),**

**(3, '2023-03-01'),**

**(2, '2023-04-01'),**

**(1, '2023-05-01'),**

**(4, '2023-06-01'),**

**(2, '2023-07-01'),**

**(1, '2023-08-01'),**

**(5, '2023-09-01'),**

**(2, '2023-10-01'),**

**(1, '2023-11-01'),**

**(6, '2023-12-01');**

**-- Query to get the customer count for each month**

**SELECT**

**m.month,**

**IFNULL(cc.customer\_count, 0) AS customer\_count**

**FROM**

**(SELECT 1 AS month UNION SELECT 2 UNION SELECT 3 UNION SELECT 4 UNION SELECT 5 UNION SELECT 6 UNION SELECT 7 UNION SELECT 8 UNION SELECT 9 UNION SELECT 10 UNION SELECT 11 UNION SELECT 12) m**

**LEFT JOIN (**

**SELECT**

**MONTH(purchase\_date) AS month,**

**COUNT(DISTINCT customer\_id) AS customer\_count**

**FROM**

**purchases**

**WHERE**

**YEAR(purchase\_date) = YEAR(CURRENT\_DATE)**

**GROUP BY**

**MONTH(purchase\_date)**

**) cc ON m.month = cc.month**

**ORDER BY**

**m.month;**

**Deliverables**

* Here is the SQL query that retrieves the **MonthName** and **CustomerCount** for each month:

SELECT

m.month\_name,

IFNULL(cc.customer\_count, 0) AS customer\_count

FROM

(SELECT 'January' AS month\_name, 1 AS month UNION

SELECT 'February', 2 UNION

SELECT 'March', 3 UNION

SELECT 'April', 4 UNION

SELECT 'May', 5 UNION

SELECT 'June', 6 UNION

SELECT 'July', 7 UNION

SELECT 'August', 8 UNION

SELECT 'September', 9 UNION

SELECT 'October', 10 UNION

SELECT 'November', 11 UNION

SELECT 'December', 12) m

LEFT JOIN (

SELECT

MONTHNAME(purchase\_date) AS month\_name,

COUNT(DISTINCT customer\_id) AS customer\_count

FROM

purchases

WHERE

YEAR(purchase\_date) = YEAR(CURRENT\_DATE)

GROUP BY

MONTHNAME(purchase\_date)

) cc ON m.month\_name = cc.month\_name

ORDER BY

m.month;

The following is an explanation of how the query ensures all months are included and handles zero customer counts:

It makes sure to include all months by creating a list of all 12 months with corresponding month numbers and month names, for example, 'January', 'February', etc. The LEFT JOIN guarantees all months will show up in the result set, even if there are no customer purchases in a particular month.

The IFNULL function replaces NULL values in the column customer\_count with 0. This will ensure that those months, for which no customer made any purchase, will be part of the result set with customer\_count as 0.

Left join also helps in handling counts of zero customers, as all rows from the m subquery are returned, even if there are no matching rows in the cc subquery.

The LEFT JOIN and IFNULL used guarantee that all months will be represented in the result set, and that months with no purchases by customers are handled correctly to return a count of zero for customer\_count.

* **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.

Solution:

**CREATE TABLE locations (**

**LocationID INT,**

**LocationName VARCHAR(255),**

**Latitude DECIMAL(10, 8),**

**Longitude DECIMAL(11, 8)**

**);**

**INSERT INTO locations (LocationID, LocationName, Latitude, Longitude)**

**VALUES**

**(1, 'Location 1', 37.7749, -122.4194),**

**(2, 'Location 2', 37.7859, -122.4364),**

**(3, 'Location 3', 37.7963, -122.4574),**

**(4, 'Location 4', 37.8067, -122.4784),**

**(5, 'Location 5', 37.8171, -122.4994),**

**(6, 'Location 6', 37.8275, -122.5204),**

**(7, 'Location 7', 37.8379, -122.5414),**

**(8, 'Location 8', 37.8483, -122.5624),**

**(9, 'Location 9', 37.8587, -122.5834),**

**(10, 'Location 10', 37.8691, -122.6044);**

**SELECT**

**l.LocationID,**

**l.LocationName,**

**l.Latitude,**

**l.Longitude,**

**( 3959 \* acos( cos( radians(37.7749) )**

**\* cos( radians( l.Latitude ) )**

**\* cos( radians( l.Longitude ) - radians(-122.4194) )**

**+ sin( radians(37.7749) )**

**\* sin( radians( l.Latitude ) ) ) ) AS distance**

**FROM**

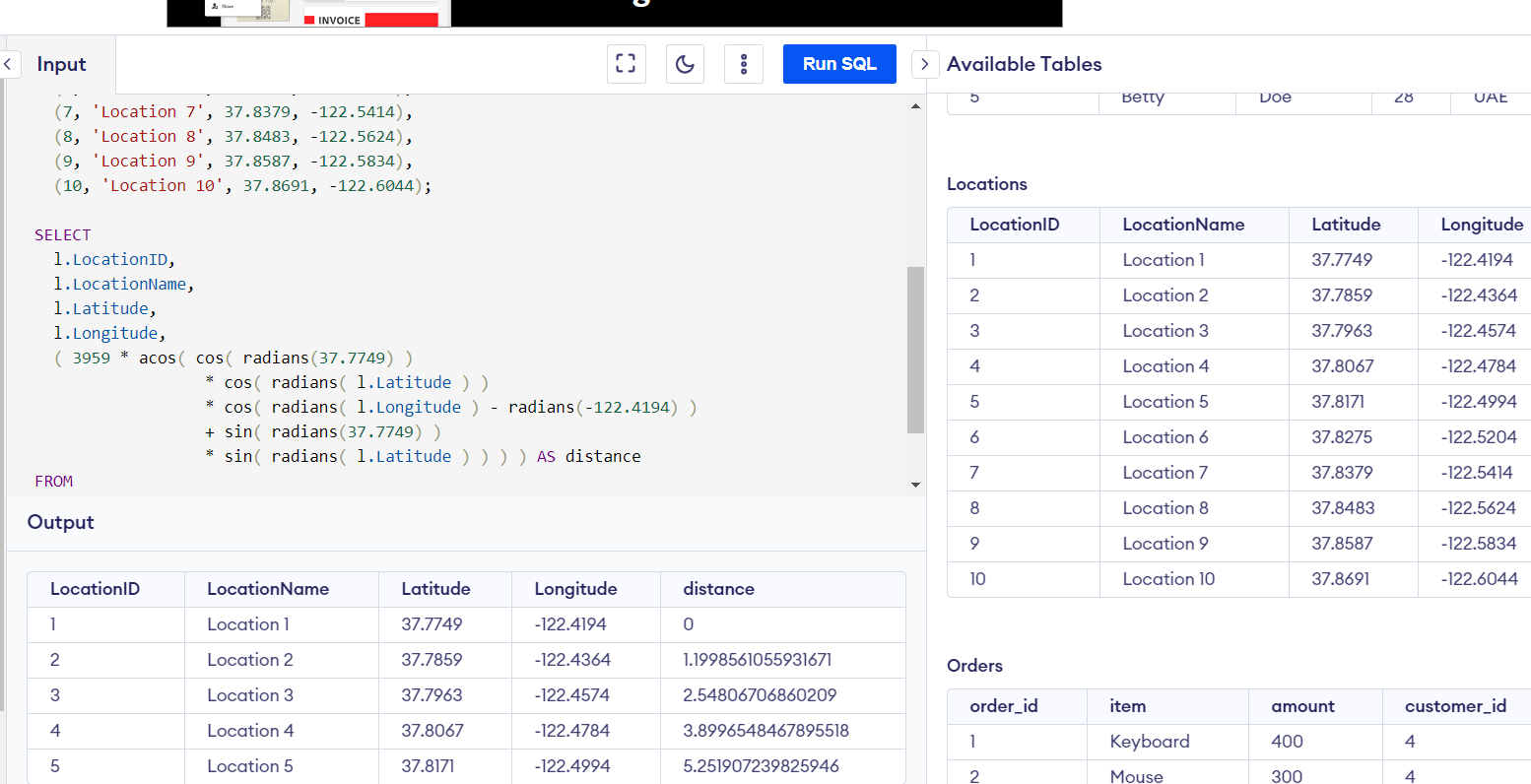
**locations l**

**ORDER BY**

**distance**

**LIMIT 5;**

OUTPUT:



Deliverables

SELECT

l.LocationID,

l.LocationName,

l.Latitude,

l.Longitude,

( 3959 \* acos( cos( radians(37.7749) )

\* cos( radians( l.Latitude ) )

\* cos( radians( l.Longitude ) - radians(-122.4194) )

+ sin( radians(37.7749) )

\* sin( radians( l.Latitude ) ) ) ) AS distance

FROM

locations l

ORDER BY

distance

LIMIT 5;

The query uses the **Haversine formula**, a well-known algorithm for calculating the distance between two points on a sphere (such as the Earth) given their longitudes and latitudes. The formula is:

**a = sin²(Δlat/2) + cos(lat1) \* cos(lat2) \* sin²(Δlong/2)** **c = 2 \* atan2(sqrt(a), sqrt(1-a))** **distance = R \* c**

where:

* **lat1** and **lat2** are the latitudes of the two points
* **long1** and **long2** are the longitudes of the two points
* **Δlat** is the difference in latitude between the two points
* **Δlong** is the difference in longitude between the two points
* **R** is the radius of the Earth (approximately 3959 miles or 6371 kilometers)

In the query, we use the **acos** (arccosine) and **sin** functions to calculate the distance between the given point **(37.7749, -122.4194)** and each location in the **locations** table. The **radians** function is used to convert the latitude and longitude values from degrees to radians.

* **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.

Solution:

CREATE INDEX idx\_order\_date ON Orders(order\_date);

SELECT order\_id, customer\_id, order\_date, amount

FROM Orders

WHERE order\_date >= NOW() - INTERVAL 7 DAY

ORDER BY order\_date DESC;

Deliverables

1. **Indexing:**
   * **Create an Index on order\_date:** Indexes are crucial for enhancing query performance, especially on large tables. Creating an index on the **order\_date** column allows the database to quickly locate rows that match the condition **order\_date >= NOW() - INTERVAL 7 DAY**. This avoids scanning the entire table and reduces query execution time.

CREATE INDEX idx\_order\_date ON Orders(order\_date);

**Why Indexing Helps:**

* + **Faster Data Retrieval:** With an index, the database can use a more efficient data structure to retrieve rows, significantly speeding up queries that involve filtering and sorting.
  + **Reduces I/O Operations:** By narrowing down the search to indexed data, the database minimizes the number of I/O operations needed.

1. **Query Rewriting:**
   * **Efficient Date Calculation:** Using **NOW() - INTERVAL 7 DAY** is an efficient way to filter records based on a relative date range. This method is well-supported and optimized in MySQL for date comparisons.

WHERE order\_date >= NOW() - INTERVAL 7 DAY

**Why Rewriting Helps:**

* + **Optimized Execution Plans:** Date functions are processed efficiently, especially when combined with indexing. This results in better execution plans and faster query performance.
  + **Select Specific Columns:** Rather than using **SELECT \***, specifying only the columns you need (**order\_id**, **customer\_id**, **order\_date**, **amount**) reduces the amount of data transferred and processed, which can improve query speed.

SELECT order\_id, customer\_id, order\_date, amount

1. **Database Configuration:**
   * **Ensure Index Utilization:** Verify that your database configuration is optimized for using indexes. This includes checking settings related to indexing and query performance.
2. **Partitioning (Optional for Extremely Large Tables):**
   * **Consider Table Partitioning:** For very large datasets, partitioning the table by date can further optimize performance. This divides the table into smaller, more manageable pieces, which can reduce the amount of data scanned for each query.

ALTER TABLE Orders PARTITION BY RANGE (YEAR(order\_date)) ( PARTITION p2023 VALUES

LESS THAN (2024), PARTITION p2024 VALUES LESS THAN (2025), PARTITION p2025

VALUES LESS THAN (2026) );

**Why Partitioning Helps:**

* + **Smaller Data Scans:** Partitioning limits the amount of data scanned by the query, as each partition contains a subset of the data. This can significantly speed up queries on large tables.

**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.

Solution:

DECLARE

dividend NUMBER := 100;

divisor NUMBER;

result NUMBER;

BEGIN

divisor := &divisor\_input; -- Replace with actual method for getting user input

BEGIN

result := dividend / divisor;

DBMS\_OUTPUT.PUT\_LINE('The result of the division is: ' || result);

EXCEPTION

WHEN ZERO\_DIVIDE THEN

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

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred: ' || SQLERRM);

END;

END;

**Exception Handling with ZERO\_DIVIDE:**

Purpose: The ZERO\_DIVIDE exception is specifically designed to handle cases where a division by zero occurs.

Implementation: The EXCEPTION block catches this specific exception and provides a user-friendly error message indicating that division by zero is not allowed.

Code:

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

**Handling Other Exceptions with WHEN OTHERS:**

Purpose: The WHEN OTHERS exception handler is a catch-all for any exceptions not explicitly handled by other WHEN clauses. It ensures that any unexpected errors are captured and reported.

Implementation: This block captures and displays a generic error message along with the error details using SQLERRM, which provides the error message associated with the exception.

Code:

WHEN OTHERS THEN DBMS\_OUTPUT.PUT\_LINE('An unexpected error occurred: ' || SQLERRM);

In real-world applications, user input might be handled through forms or interfaces rather than direct PL/SQL prompts, and error messages would be managed according to the application's requirements and user interface.

* **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.

Solution:

DECLARE

-- Arrays to hold employee IDs and salary increments

TYPE id\_array IS TABLE OF Employees.employee\_id%TYPE;

TYPE salary\_array IS TABLE OF NUMBER;

v\_employee\_ids id\_array := id\_array(101, 102, 103); -- Example employee IDs

v\_salary\_increments salary\_array := salary\_array(500, 600, 700); -- Example salary increments

BEGIN

-- Use FORALL to perform the update in bulk

FORALL i IN INDICES OF v\_employee\_ids

UPDATE Employees

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

WHERE employee\_id = v\_employee\_ids(i);

-- Optionally, commit the changes if not in an implicit transaction

COMMIT;

DBMS\_OUTPUT.PUT\_LINE('Salaries updated successfully.');

EXCEPTION

WHEN OTHERS THEN

-- Handle exceptions if any

DBMS\_OUTPUT.PUT\_LINE('An error occurred: ' || SQLERRM);

ROLLBACK; -- Optionally roll back the changes in case of error

END;

/

Description of How FORALL Improves Performance for Bulk Updates

Bulk Operations:

Purpose: FORALL is used to perform bulk operations efficiently. It allows you to execute the same SQL operation (e.g., UPDATE, INSERT, DELETE) for multiple rows with a single statement, rather than executing individual operations in a loop.

Efficiency: By processing multiple rows in a single operation, FORALL reduces the number of context switches between PL/SQL and SQL engines, which improves performance significantly for bulk updates.

Performance Improvement:

Reduces Context Switching: Traditional row-by-row processing involves many context switches between PL/SQL and SQL engines, which can be slow. FORALL minimizes these switches by sending all the data to the SQL engine in one batch.

Optimizes SQL Execution: FORALL executes the SQL statements in a single batch operation, which allows the SQL engine to optimize the execution plan and handle the bulk operation more efficiently.

Minimizes Network Round-Trips: For remote databases, reducing the number of network round-trips between the application and the database server further improves performance.

Error Handling:

Exception Handling: Errors encountered during bulk operations are handled collectively. If an error occurs, you can catch it using the EXCEPTION block and handle it accordingly, which provides robustness to the bulk update operation.

This approach is highly effective for scenarios where you need to perform the same operation on many rows, making it a powerful tool for bulk data manipulation in PL/SQL.

* **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.

Solution:

CREATE OR REPLACE TYPE employee\_obj AS OBJECT (

employee\_id NUMBER,

first\_name VARCHAR2(50),

last\_name VARCHAR2(50),

email VARCHAR2(100)

);

/

CREATE OR REPLACE TYPE employee\_tab AS TABLE OF employee\_obj;

/

CREATE OR REPLACE PROCEDURE get\_employees\_by\_department (

p\_department\_id IN NUMBER,

p\_employee\_list OUT employee\_tab

) AS

BEGIN

-- Initialize the nested table

p\_employee\_list := employee\_tab();

-- Retrieve employees for the given department

SELECT employee\_obj(employee\_id, first\_name, last\_name, email)

BULK COLLECT INTO p\_employee\_list

FROM employees

WHERE department\_id = p\_department\_id;

-- Handle the case where no employees are found

IF p\_employee\_list.COUNT = 0 THEN

p\_employee\_list.EXTEND;

END IF;

END get\_employees\_by\_department;

/

1. **Nested Table Type Definition**:
   * employee\_obj: An object type representing an employee with attributes like employee\_id, first\_name, last\_name, and email.
   * employee\_tab: A nested table type that can store multiple employee\_obj objects.
2. **Procedure (get\_employees\_by\_department)**:
   * **Input Parameter (p\_department\_id)**: The ID of the department for which employees are to be retrieved.
   * **Output Parameter (p\_employee\_list)**: The nested table that will contain the employee data.
   * **Initialization**: The nested table is initialized to hold employee data.
   * **Data Retrieval**: A SELECT statement fetches the employees belonging to the specified department and populates the nested table using BULK COLLECT.
   * **Handling Empty Results**: If no employees are found, the procedure ensures that the output nested table is initialized properly.

This procedure allows the caller to retrieve a list of employees in a department as a collection, making it easier to process or manipulate the data further in PL/SQL or in the application layer.

* **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.

Solution:

DECLARE

TYPE ref\_cursor\_type IS REF CURSOR;

c\_employee ref\_cursor\_type;

v\_employee\_id employees.employee\_id%TYPE;

v\_first\_name employees.first\_name%TYPE;

v\_last\_name employees.last\_name%TYPE;

v\_salary\_threshold NUMBER := 50000; -- Example salary threshold

v\_sql\_query VARCHAR2(4000);

BEGIN

-- Construct the dynamic SQL query

v\_sql\_query := 'SELECT employee\_id, first\_name, last\_name FROM employees WHERE salary > :salary\_threshold';

-- Open the cursor with dynamic SQL

OPEN c\_employee FOR v\_sql\_query USING v\_salary\_threshold;

-- Fetch data from the cursor

LOOP

FETCH c\_employee INTO v\_employee\_id, v\_first\_name, v\_last\_name;

EXIT WHEN c\_employee%NOTFOUND;

-- Process the fetched data (e.g., output to screen, log, etc.)

DBMS\_OUTPUT.PUT\_LINE('Employee ID: ' || v\_employee\_id ||

', First Name: ' || v\_first\_name ||

', Last Name: ' || v\_last\_name);

END LOOP;

-- Close the cursor

CLOSE c\_employee;

END;

/

1. **Ref Cursor Type Declaration**:
   * ref\_cursor\_type IS REF CURSOR;: Defines a cursor variable type.
2. **Cursor Variable Declaration**:
   * c\_employee ref\_cursor\_type;: Declares a cursor variable that can point to a result set.
3. **Dynamic SQL Construction**:
   * v\_sql\_query VARCHAR2(4000);: A variable to hold the dynamic SQL query string.
   * The SQL query is constructed dynamically, allowing the query to change based on conditions or parameters at runtime. In this case, it selects employee\_id, first\_name, and last\_name from the employees table where the salary exceeds a specified threshold.
4. **Opening the Cursor with Dynamic SQL**:
   * OPEN c\_employee FOR v\_sql\_query USING v\_salary\_threshold;: Opens the cursor with the dynamically constructed SQL statement. The USING clause binds the v\_salary\_threshold variable to the :salary\_threshold placeholder in the SQL statement.
5. **Fetching Data**:
   * The loop fetches each row from the cursor into local variables (v\_employee\_id, v\_first\_name, v\_last\_name) until no more rows are available (EXIT WHEN c\_employee%NOTFOUND;).
   * In this example, the fetched data is displayed using DBMS\_OUTPUT.PUT\_LINE.
6. **Closing the Cursor**:
   * CLOSE c\_employee;: Closes the cursor to release resources.

**Dynamic SQL and Cursor Variables**

* **Dynamic SQL**: This is SQL code that is constructed and executed at runtime, allowing flexibility in building queries based on runtime conditions or inputs. This is useful when the SQL structure is not known until the code executes.
* **Cursor Variables (REF CURSOR)**: These allow the result set of a query to be referenced and manipulated dynamically. They are particularly useful when the result set of a query is determined at runtime and may change based on different conditions.
* **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.

Solution:

CREATE OR REPLACE TYPE sales\_record AS OBJECT (

order\_id NUMBER,

customer\_id NUMBER,

order\_amount NUMBER

);

/

CREATE OR REPLACE TYPE sales\_table AS TABLE OF sales\_record;

/

CREATE OR REPLACE FUNCTION get\_sales\_data (

p\_month IN NUMBER,

p\_year IN NUMBER

) RETURN sales\_table PIPELINED IS

v\_order\_id NUMBER;

v\_customer\_id NUMBER;

v\_order\_amount NUMBER;

BEGIN

FOR rec IN (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) LOOP

PIPE ROW (sales\_record(rec.order\_id, rec.customer\_id, rec.order\_amount));

END LOOP;

RETURN;

END get\_sales\_data;

/

1. **Record and Table Types**:
   * sales\_record: An object type representing a sales record with order\_id, customer\_id, and order\_amount attributes.
   * sales\_table: A nested table type that can hold multiple sales\_record objects.
2. **Pipelined Function (get\_sales\_data)**:
   * **Input Parameters**:
     + p\_month: The month for which sales data is to be retrieved.
     + p\_year: The year for which sales data is to be retrieved.
   * **Local Variables**:
     + v\_order\_id, v\_customer\_id, v\_order\_amount: Variables to hold the sales data.
   * **Cursor FOR Loop**:
     + The loop retrieves the order\_id, customer\_id, and order\_amount from the sales table for orders placed in the specified month and year.
   * **PIPE ROW**:
     + PIPE ROW (sales\_record(...)): Sends each fetched row to the caller as a sales\_record object. This is the key feature of pipelined functions, allowing rows to be returned one at a time as they are fetched.
   * **RETURN**:
     + The RETURN statement indicates the end of the function. It is necessary even though it does not return a specific value because the function is pipelined.

**Pipelined Table Functions and Efficiency**

Pipelined table functions improve data retrieval efficiency in several ways:

1. **Streaming Data**: Pipelined functions return rows incrementally as they are produced, which means the calling query can start processing rows immediately without waiting for the entire result set to be generated. This can significantly reduce response time for large datasets.
2. **Reduced Memory Usage**: By not requiring the entire result set to be materialized in memory before returning data, pipelined functions can handle larger datasets more efficiently, reducing memory overhead.
3. **Parallel Processing**: Pipelined functions can take advantage of Oracle's parallel execution capabilities, where multiple rows can be processed concurrently, improving performance for large-scale data retrieval and processing.
4. **Flexibility**: Pipelined functions provide a flexible way to encapsulate complex logic in a function that can be queried like a regular table. This makes it easier to integrate procedural logic with SQL queries.