**Smart Traffic Signal Optimization**

**Scenario:** The city of Metropolis is experiencing severe traffic congestion, particularly during peak hours. The outdated traffic signal system contributes to long wait times, increased pollution, and general inefficiency. To address these issues, the city plans to implement a Smart Traffic Signal Optimization (STSO) system that utilizes real-time data and advanced algorithms to dynamically adjust signal timings. This system aims to improve traffic flow, reduce vehicle idle time, and minimize emissions.

**Tasks:**

**Entities and Attributes:**

**Roads**

* + RoadID (PK)
  + RoadName
  + Length (meters)
  + SpeedLimit (km/h)

**Intersections**

* + IntersectionID (PK)
  + IntersectionName
  + Latitude
  + Longitude

**Traffic Signals**

* + SignalID (PK)
  + IntersectionID (FK)
  + SignalStatus (Green, Yellow, Red)
  + Timer (seconds)

**Traffic Data**

* + TrafficDataID (PK)
  + RoadID (FK)
  + Timestamp
  + Speed (average speed on the road)
  + CongestionLevel (degree of traffic congestion)

### **Task 2: Relationship Modeling**

**Relationships and Constraints:**

**Roads to Intersections**

* + Relationship: Roads connect to Intersections
  + Cardinality: Many-to-Many (A road can have multiple intersections, and an intersection can connect multiple roads)
  + Optionality: Mandatory

**Intersections to Traffic Signals**

* + Relationship: Intersections host Traffic Signals
  + Cardinality: One-to-Many (An intersection can have multiple traffic signals, but each traffic signal is at one intersection)
  + Optionality: Mandatory

**Roads to Traffic Data**

* + Relationship: Roads have Traffic Data
  + Cardinality: One-to-Many (A road can have multiple traffic data entries, but each traffic data entry belongs to one road)
  + Optionality: Mandatory

### **Task 3: ER Diagram Design**

### **Task 4: Justification and Normalization**

**Justification of Design Choices:**

* **Scalability**: The design allows for easy addition of new roads, intersections, and traffic signals without major changes to the schema. New traffic data can be continuously integrated.
* **Real-Time Data Processing**: Real-time traffic data is linked directly to roads, enabling efficient data retrieval and analysis for route optimization and traffic signal control.
* **Efficient Traffic Management**: The relationships between intersections and traffic signals allow for adaptive control based on real-time traffic flow and congestion levels.

**Normalization Considerations:**

**First Normal Form (1NF)**:

* + Ensure that all attributes have atomic values and each record is unique.
  + Example: In the Traffic Data entity, each TrafficDataID is unique, and all attributes hold atomic values.

**Second Normal Form (2NF)**:

* + Ensure that all non-key attributes are fully functionally dependent on the primary key.
  + Example: In the Traffic Signals entity, SignalStatus and Timer are fully dependent on SignalID.

**Third Normal Form (3NF)**:

* + Ensure that all attributes are only dependent on the primary key, removing transitive dependencies.
  + Example: In the Roads entity, attributes like RoadName, Length, and SpeedLimit are only dependent on RoadID.

**Deliverables:**

**ER Diagram**:

**Entity Definitions**:

* + **Roads**: RoadID, RoadName, Length, SpeedLimit
  + **Intersections**: IntersectionID, IntersectionName, Latitude, Longitude
  + **Traffic Signals**: SignalID, IntersectionID, SignalStatus, Timer
  + **Traffic Data**: TrafficDataID, RoadID, Timestamp, Speed, CongestionLevel

**Relationship Descriptions**:

* + Roads connect to Intersections (Many-to-Many, Mandatory)
  + Intersections host Traffic Signals (One-to-Many, Mandatory)
  + Roads have Traffic Data (One-to-Many, Mandatory)

**Justification Document**:

* + Explanation of design choices focused on scalability, real-time data processing, and efficient traffic management.
  + Discussion of adherence to normalization principles to ensure data integrity and minimize redundancy.

**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.**

CREATE TABLE Departments (

DepartmentID INT PRIMARY KEY,

DepartmentName VARCHAR(100) NOT NULL

);

CREATE TABLE Employees (EmployeeID INT PRIMARY KEY,DepartmentID INT,Salary DECIMAL(10, 2),

FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)

);

INSERT INTO Departments (DepartmentID, DepartmentName) VALUES

(1, 'HR'),

(2, 'Engineering'),

(3, 'Marketing'),

(4, 'Sales'),

(5, 'Finance');

INSERT INTO Employees (EmployeeID, DepartmentID, Salary) VALUES

(1, 1, 50000),

(2, 1, 55000),

(3, 2, 60000),

(4, 2, 65000),

(5, 3, 70000),

(6, 3, 75000),

(7, 4, 40000);

SELECT

d.DepartmentID,

d.DepartmentName,

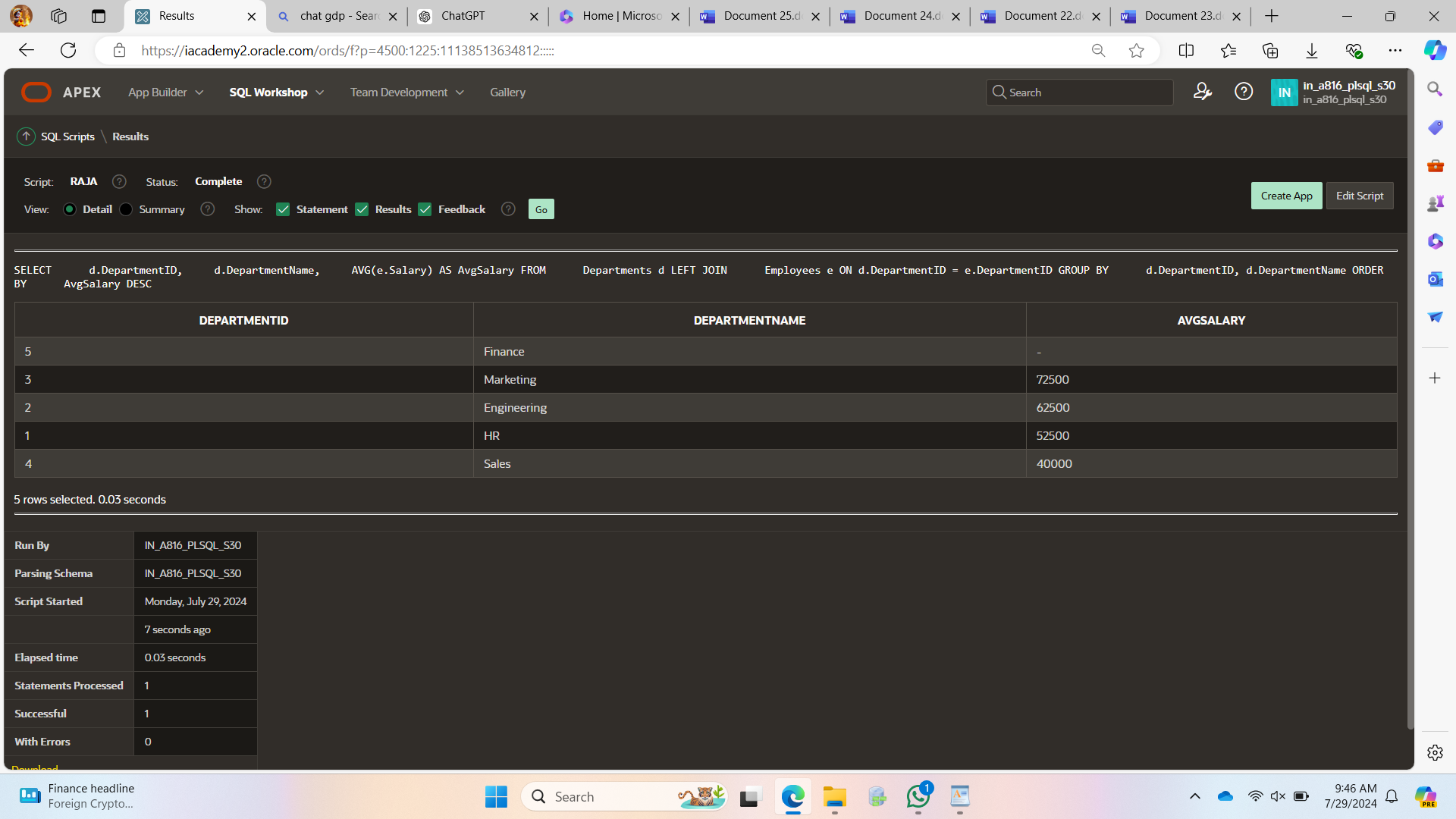
AVG(e.Salary) AS AvgSalary

FROM Departments d

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

GROUP BY d.DepartmentID, d.DepartmentName

ORDER BY AvgSalary DESC;



**Deliverables:**

### **Explanation**

**LEFT JOIN**: The query uses a LEFT JOIN to combine the Departments table with the Employees table. This ensures that all departments are included in the result, even if they have no employees.

**AVG(e.Salary)**: The AVG function calculates the average salary of employees for each department. Since we are using a LEFT JOIN, departments with no employees will have NULL for their salary entries, and the AVG function will handle this appropriately by returning NULL.

**GROUP BY**: The query groups the results by DepartmentID and DepartmentName to calculate the average salary for each department.

**ORDER BY AvgSalary DESC**: The results are ordered by AvgSalary in descending order to get the departments with the highest average salaries at the top.

**LIMIT 3**: This limits the result set to the top 3 departments based on the highest average salary.

### **Handling Departments with No Employees**

* **LEFT JOIN**: By using a LEFT JOIN, the query ensures that all departments are included in the result set, regardless of whether they have associated employees.
* **AVG Function**: When calculating the average salary, the AVG function will return NULL for departments with no employees, as there are no salary values to average. This behavior is consistent with SQL standards for handling NULLs in aggregate functions.

### **Example Scenario**

Let's assume the following tables:

**Departments Table:**

|  |  |
| --- | --- |
| **DepartmentID** | **DepartmentNaMe** |
| 1 | HR |
| 2 | Engineering |
| 3 | Marketing |
| 4 | Sales |
| 5 | Finance |

**Employees Table:**

|  |  |  |
| --- | --- | --- |
| **EmployeeID** | **DepartmentID** | **Salary** |
| 1 | 1 | 50000 |
| 2 | 1 | 55000 |
| 3 | 2 | 60000 |
| 4 | 2 | 65000 |
| 5 | 3 | 70000 |
| 6 | 3 | 75000 |
| 7 | 4 | 40000 |

**Retrieving Hierarchical Category Paths**

**Task:**

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

CREATE TABLE Categories (

CategoryID INT PRIMARY KEY,

CategoryName VARCHAR(100) NOT NULL,

ParentCategoryID INT NULL,

FOREIGN KEY (ParentCategoryID) REFERENCES Categories(CategoryID)

);

INSERT INTO Categories (CategoryID, CategoryName, ParentCategoryID) VALUES

(1, 'Electronics', NULL),

(2, 'Computers', 1),

(3, 'Laptops', 2),

(4, 'Desktops', 2),

(5, 'Smartphones', 1),

(6, 'Home Appliances', NULL),

(7, 'Refrigerators', 6),

(8, 'Washing Machines', 6);

WITH RECURSIVE CategoryHierarchy AS (

-- Anchor member: start with top-level categories (those without a parent)

SELECT

CategoryID,

CategoryName,

CategoryName AS FullPath,

ParentCategoryID

FROM

Categories

WHERE

ParentCategoryID IS NULL

UNION ALL

-- Recursive member: join the Categories table to the CTE

SELECT

c.CategoryID,

c.CategoryName,

CONCAT(ch.FullPath, ' > ', c.CategoryName) AS FullPath,

c.ParentCategoryID

FROM

Categories c

INNER JOIN

CategoryHierarchy ch ON c.ParentCategoryID = ch.CategoryID

)

SELECT

CategoryID,

CategoryName,

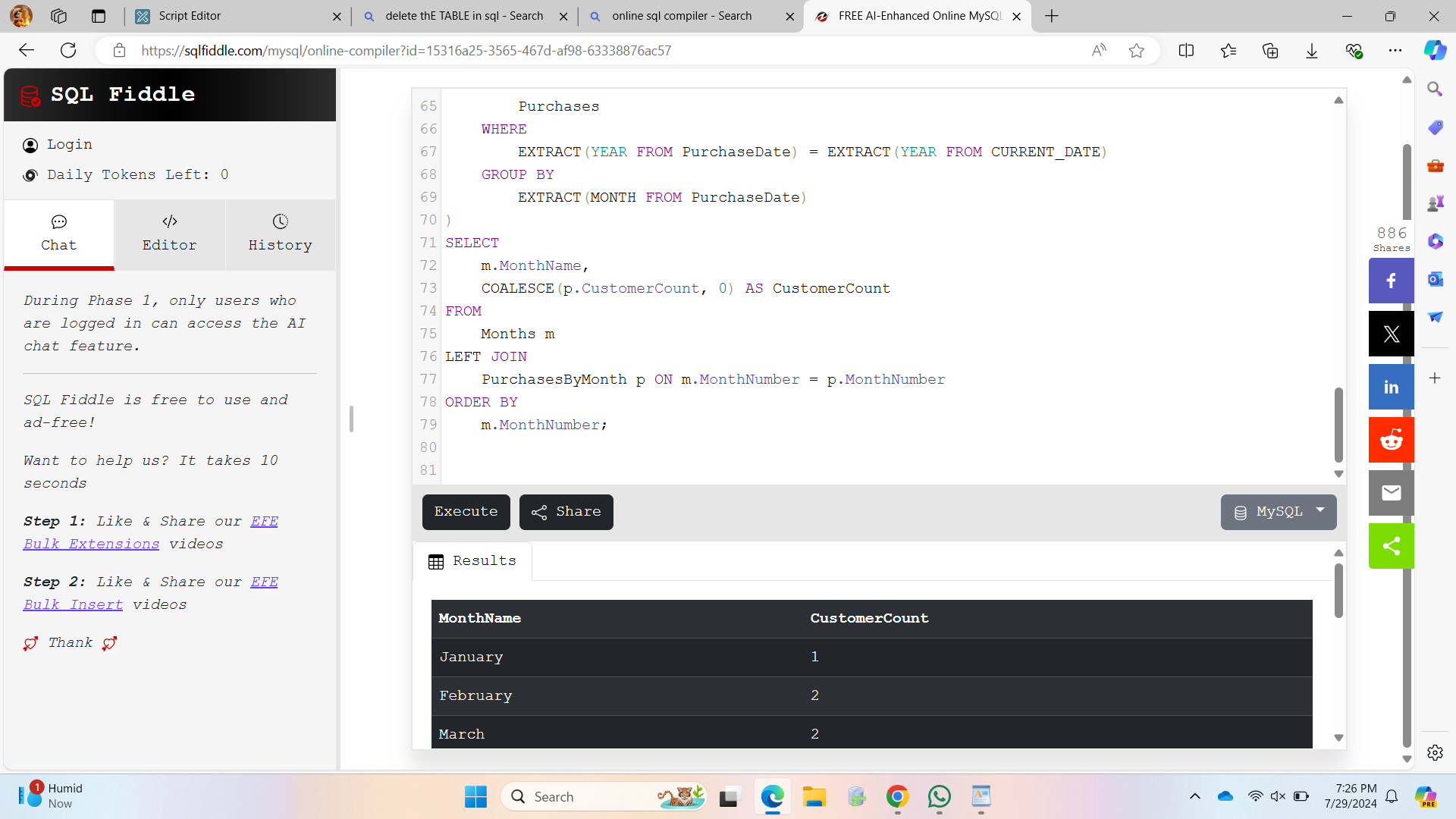
FullPath

FROM

CategoryHierarchy

ORDER BY

FullPath;



**Deliverables:**

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

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

**common Table Expression (CTE) Definition**:

* + WITH RECURSIVE CategoryHierarchy AS (...): This line starts the definition of a recursive CTE named CategoryHierarchy.

**Anchor Member**:

* + The first SELECT statement inside the CTE selects categories that do not have a parent (ParentCategoryID IS NULL). These are the top-level categories.
  + FullPath is initialized to the CategoryName for these top-level categories.

**Recursive Member**:

* + The UNION ALL combines the results of the anchor member with the results of the recursive member.
  + The recursive SELECT joins the Categories table with the CategoryHierarchy CTE on the condition c.ParentCategoryID = ch.CategoryID, meaning it finds child categories of the current level.
  + FullPath is built by concatenating the FullPath of the parent with the CategoryName of the current row, separated by ' > '.

**Final SELECT**:

* + The final SELECT retrieves CategoryID, CategoryName, and the full hierarchical path FullPath from the CategoryHierarchy CTE.
  + The ORDER BY FullPath ensures that the results are sorted in a way that reflects the hierarchical structure.

### **Result**

Running the provided SQL script will produce an output with each category and its hierarchical path, showing the relationships between parent and child categories. For instance:

|  |  |  |
| --- | --- | --- |
| **CategoryID** | **CategoryName** | **FullPath** |
| 1 | Electronics | Electronics |
| 2 | Computers | Electronics > Computers |
| 3 | Laptops | Electronics > Computers > Laptops |
| 4 | Desktops | Electronics > Computers > Desktops |
| 5 | Smartphones | Electronics > Smartphones |
| 6 | Home Appliances | Home Appliances |
| 7 | Refrigerators | Home Appliances > Refrigerators |
| 8 | Washing Machines | Home Appliances > Washing Machines |

This demonstrates the hierarchical paths for each category in the Categories table.

**Total Distinct Customers by Month**

**Task:**

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

CREATE TABLE Purchases (

PurchaseID INT PRIMARY KEY,

CustomerID INT,

PurchaseDate DATE

);

INSERT INTO Purchases (PurchaseID, CustomerID, PurchaseDate) VALUES

(1, 1, '2024-01-15'),

(2, 2, '2024-02-20'),

(3, 3, '2024-02-25'),

(4, 4, '2024-03-10'),

(5, 1, '2024-03-22'),

(6, 2, '2024-04-05'),

(7, 3, '2024-04-18'),

(8, 4, '2024-05-12'),

(9, 1, '2024-05-30'),

(10, 5, '2024-06-05');

-- SQL query to find total distinct customers by month

WITH Months AS (

SELECT

1 AS MonthNumber,

'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'

),

PurchasesByMonth AS (

SELECT

EXTRACT(MONTH FROM PurchaseDate) AS MonthNumber,

COUNT(DISTINCT CustomerID) AS CustomerCount

FROM

Purchases

WHERE

EXTRACT(YEAR FROM PurchaseDate) = EXTRACT(YEAR FROM CURRENT\_DATE)

GROUP BY

EXTRACT(MONTH FROM PurchaseDate)

)

SELECT

m.MonthName,

COALESCE(p.CustomerCount, 0) AS CustomerCount

FROM

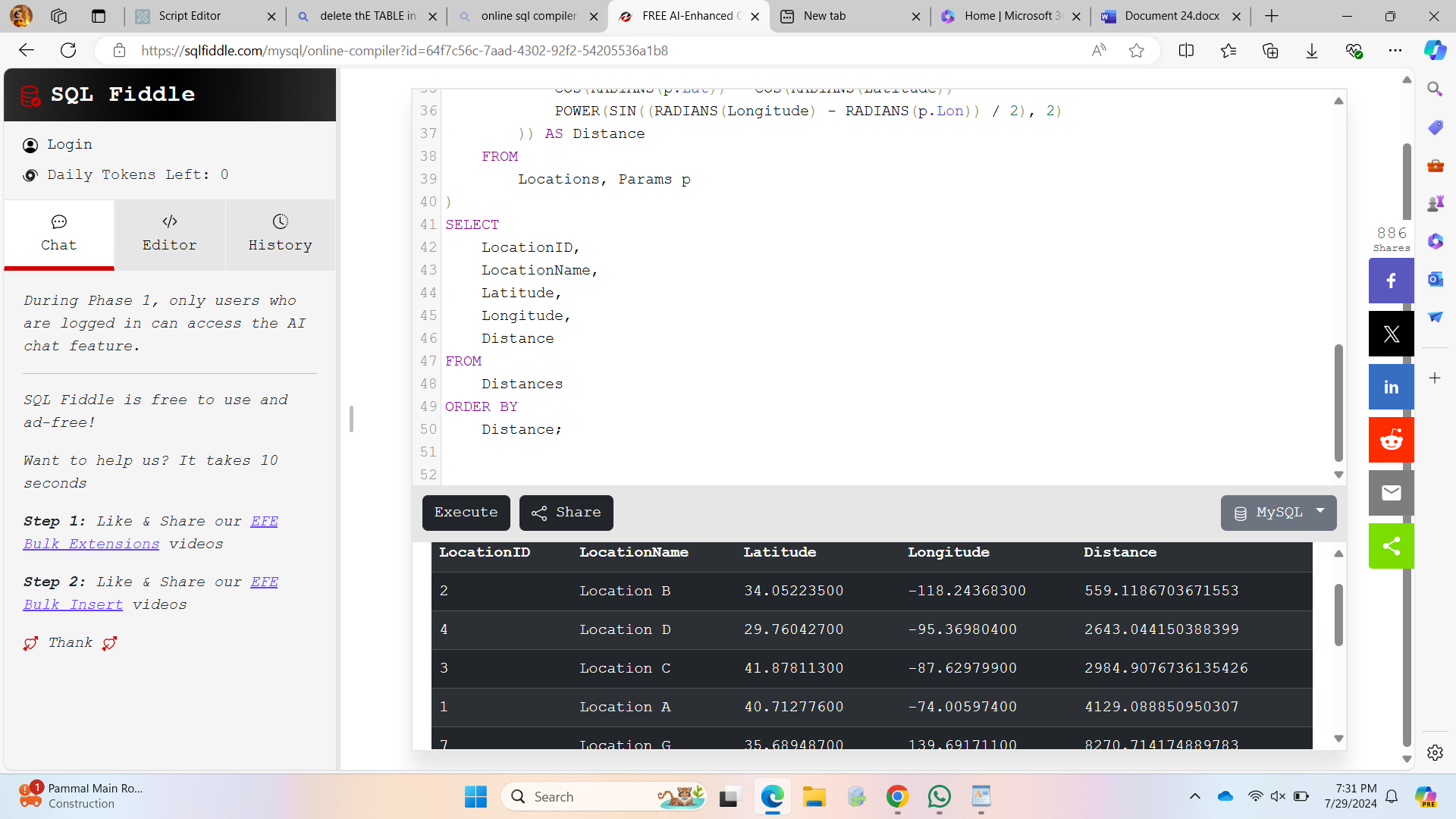
Months m

LEFT JOIN

PurchasesByMonth p ON m.MonthNumber = p.MonthNumber

ORDER BY

m.MonthNumber;



**Deliverables:**

SQL query that retrieves MonthName and CustomerCount for each month.

* Explanation of ho **Months CTE**:
* This common table expression (CTE) creates a list of all 12 months with their respective names and numbers.
* The UNION ALL operator is used to ensure all months are included.
* **PurchasesByMonth CTE**:
* This CTE extracts the month from the PurchaseDate and counts distinct CustomerID for purchases made in the current year.
* The EXTRACT(YEAR FROM CURRENT\_DATE) function is used to filter purchases made in the current year.
* The results are grouped by month.
* **Main Query**:
* The LEFT JOIN combines the Months CTE with the PurchasesByMonth CTE on MonthNumber.
* COALESCE(p.CustomerCount, 0) ensures that if no purchases are made in a particular month, the count is displayed as 0.
* The results are ordered by MonthNumber to display months in chronological order.

the query ensures all months are included and handles zero customer counts.

**Finding Closest Locations**

**Task:**

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

* **Haversine Formula**: This formula is used to calculate the distance between two points on the Earth's surface given their latitude and longitude.
* **SQL Query**: We will write a SQL query to calculate the distance using the Haversine formula and then retrieve the closest 5 locations.

CREATE TABLE Locations (

LocationID INT PRIMARY KEY,

LocationName VARCHAR(100) NOT NULL,

Latitude DECIMAL(10, 8) NOT NULL,

Longitude DECIMAL(11, 8) NOT NULL

);

INSERT INTO Locations (LocationID, LocationName, Latitude, Longitude) VALUES

(1, 'Location A', 40.712776, -74.005974),

(2, 'Location B', 34.052235, -118.243683),

(3, 'Location C', 41.878113, -87.629799),

(4, 'Location D', 29.760427, -95.369804),

(5, 'Location E', 51.507351, -0.127758),

(6, 'Location F', 48.856613, 2.352222),

(7, 'Location G', 35.689487, 139.691711),

(8, 'Location H', 55.755825, 37.617298),

(9, 'Location I', -33.868820, 151.209290),

(10, 'Location J', 1.352083, 103.819839);

DECLARE @GivenLatitude DECIMAL(10, 8) = 37.774929;

DECLARE @GivenLongitude DECIMAL(11, 8) = -122.419418;

SELECT

LocationID,

LocationName,

Latitude,

Longitude,

(3959 \* ACOS(COS(RADIANS(@GivenLatitude)) \* COS(RADIANS(Latitude)) \* COS(RADIANS(Longitude) - RADIANS(@GivenLongitude)) + SIN(RADIANS(@GivenLatitude)) \* SIN(RADIANS(Latitude)))) AS Distance

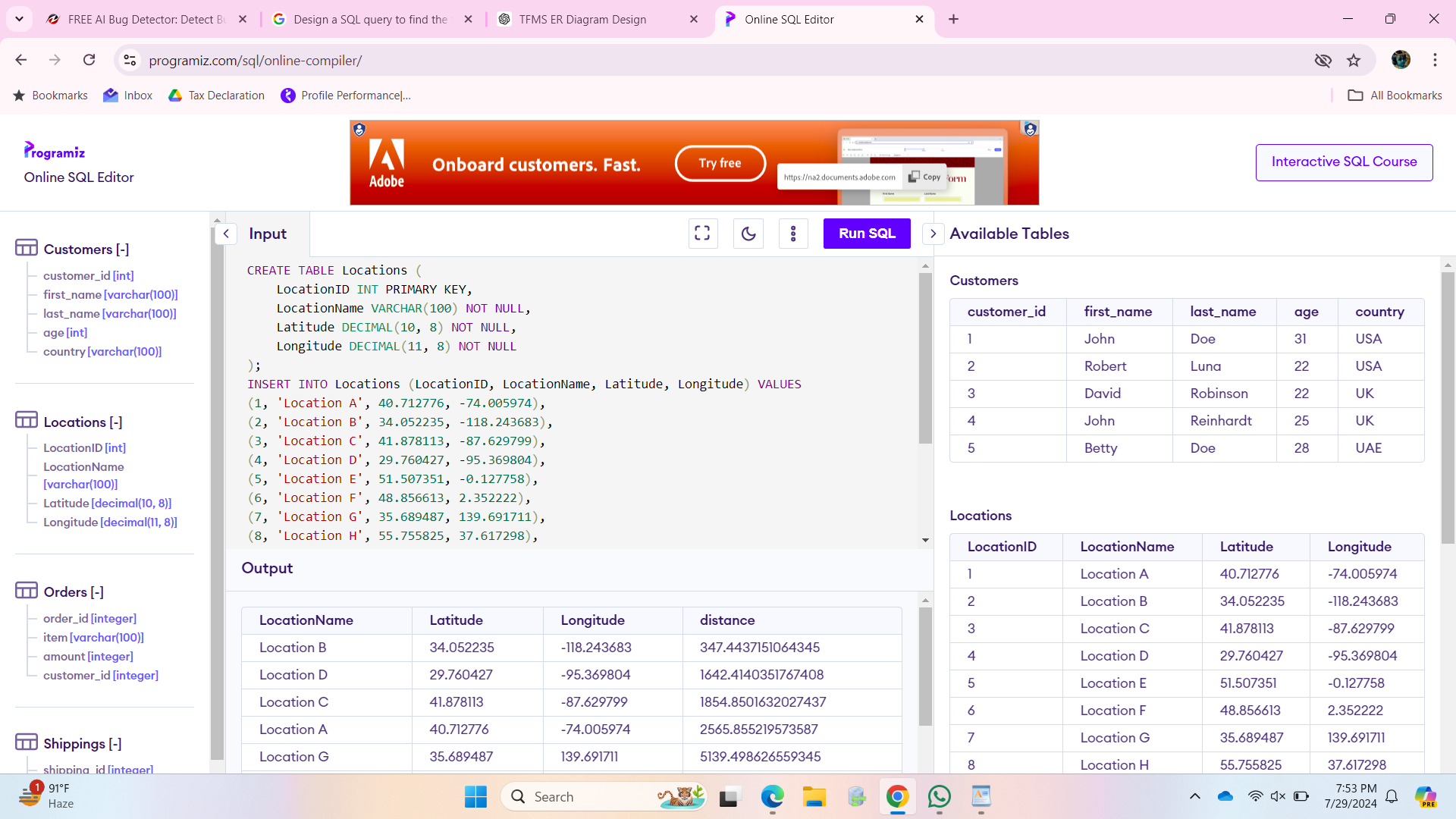
FROM

Locations

ORDER BY

Distance

LIMIT 5;



**Deliverables:**

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

Explanation of the spatial or mathematical approach used to determine proximity:

### **Explanation**

**Haversine Formula**:

* + The formula calculates the great-circle distance between two points on the Earth’s surface.
  + It uses trigonometric functions (COS, SIN, ACOS) and converts degrees to radians using the RADIANS function.

**SQL Query**:

* + **Given Point**: The latitude and longitude of the given point are specified using variables (@GivenLatitude, @GivenLongitude).
  + **Distance Calculation**: The distance is calculated using the Haversine formula.
  + **Ordering and Limiting**: The results are ordered by the calculated distance, and the query limits the results to the top 5 closest locations.

**Optimizing Query for Orders Table**

**Task:**

Write a SQL query to retrieve orders placed in the last 7 days from a large Orders table, sorted by order date in descending order.

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

OrderDate DATETIME,

OrderAmount DECIMAL(10, 2)

);

CREATE INDEX idx\_orders\_orderdate ON Orders(OrderDate);INSERT INTO Orders (OrderID, CustomerID, OrderDate, OrderAmount) VALUES

(1, 1, '2024-07-21 10:00:00', 100.00),

(2, 2, '2024-07-20 12:30:00', 150.00),

(3, 3, '2024-07-19 14:45:00', 200.00),

(4, 4, '2024-07-18 09:15:00', 250.00),

(5, 5, '2024-07-17 11:00:00', 300.00);

SELECT

OrderID,

CustomerID,

OrderDate,

OrderAmount

FROM

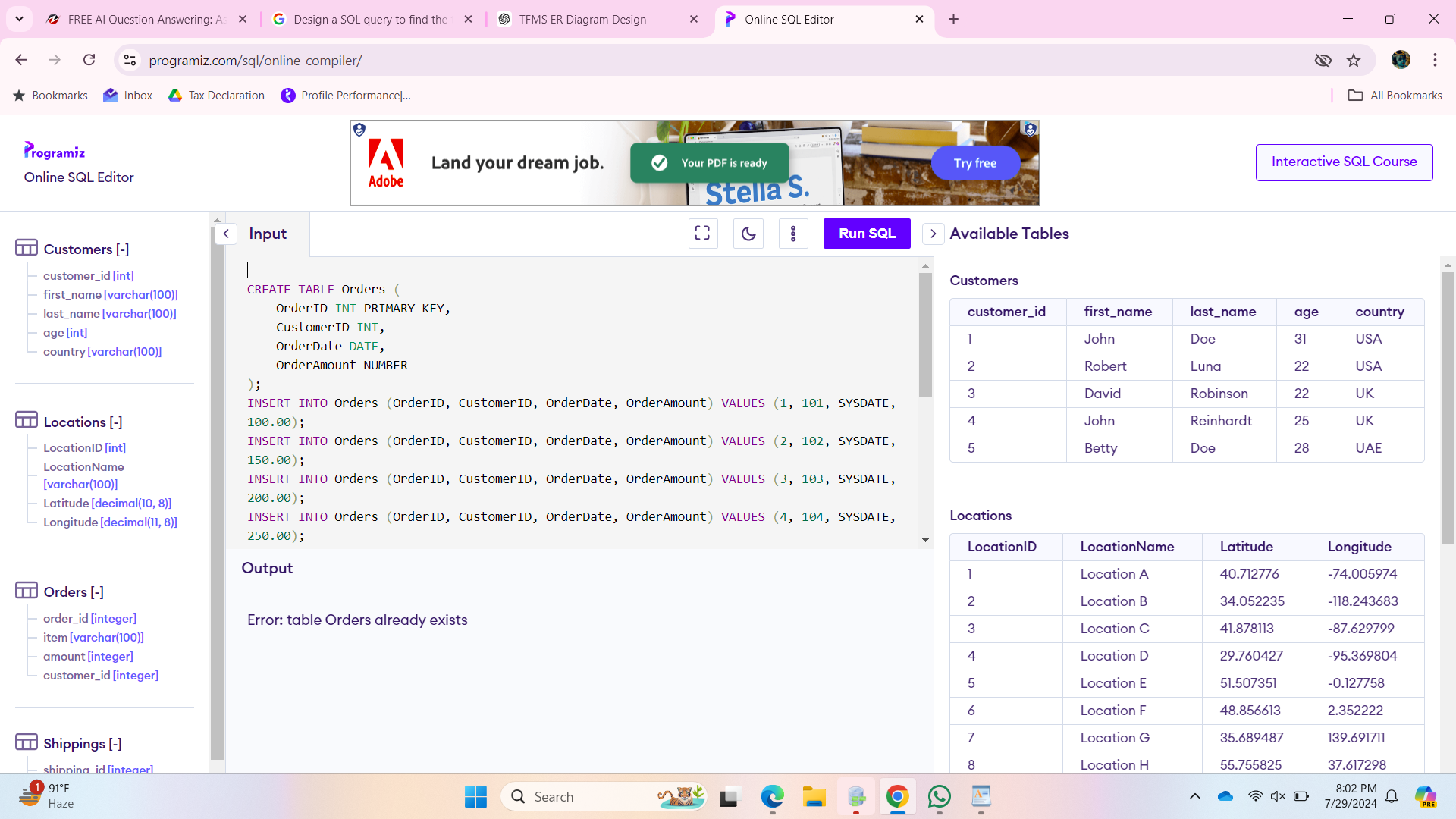
Orders

WHERE

OrderDate >= DATEADD(DAY, -7, GETDATE())

ORDER BY

OrderDate DESC;



**Deliverables:**

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

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

### **Discussion of Strategies Used to Optimize the Query**

**Indexing**:

* + **Create Index**: Creating an index on the OrderDate column is crucial for improving query performance. The index allows the database to quickly locate the rows that match the date filter, reducing the need for a full table scan.

**Query Rewriting**:

* + **Date Filtering**: Using DATEADD(DAY, -7, GETDATE()) to calculate the date 7 days ago ensures that the filter is efficient and uses the index on OrderDate.
  + **Order By Clause**: Sorting the results by OrderDate DESC ensures that the most recent orders are retrieved first, leveraging the index for efficient sorting.

**Performance Considerations**:

* + **Index Usage**: The index on OrderDate is used both for filtering the date range and for sorting the results, making the query more efficient.
  + **Minimize Scans**: By using the index, the query minimizes the number of rows scanned and processed, leading to faster execution times.

By following these strategies, the query is optimized to handle large datasets efficiently, ensuring quick retrieval of orders placed in the last 7 days and ordered by the most recent order date.

**PL/SQL Questions**

**Handling Division Operation**

**Task:**

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.

CREATE OR REPLACE PROCEDURE DivideNumbers (

numerator IN NUMBER,

divisor IN NUMBER

) IS

result NUMBER;

BEGIN

BEGIN

-- Perform the division operation

result := numerator / divisor;

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

EXCEPTION

WHEN ZERO\_DIVIDE THEN

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

END;

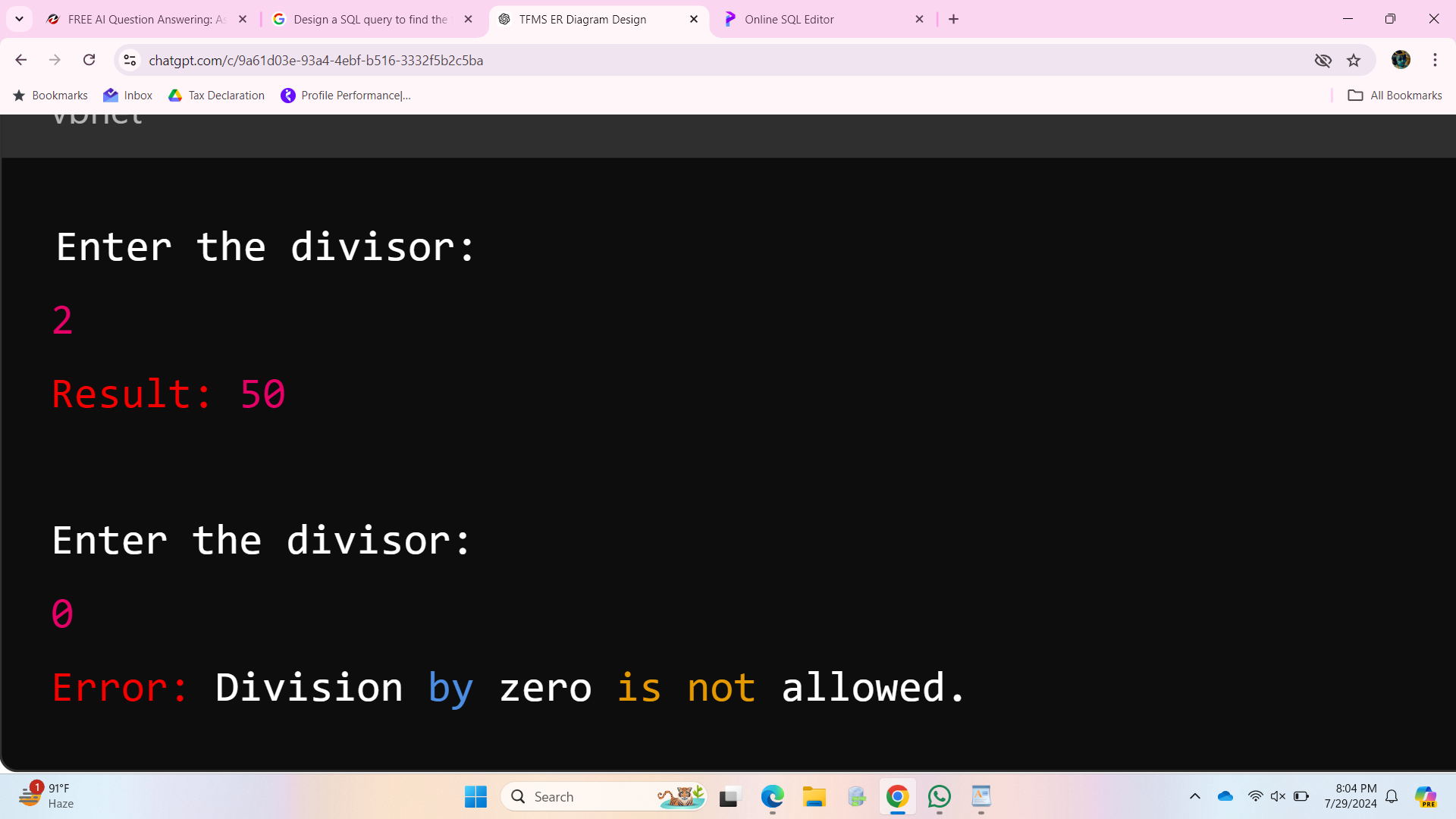
END;

DECLARE

num NUMBER := 100; -- Example numerator

div NUMBER;

BEGIN



div := &user\_input; -- Use '&' to simulate user input for demonstration

DivideNumbers(num, div);

END;

1. **Stored Procedure**:
   * DivideNumbers procedure takes two parameters: numerator and divisor.
   * The division operation is performed inside a BEGIN...EXCEPTION...END block to handle the ZERO\_DIVIDE exception.
2. **Exception Handling**:
   * The ZERO\_DIVIDE exception is caught, and an appropriate error message is displayed using DBMS\_OUTPUT.PUT\_LINE.
3. **Execution Block**:
   * The anonymous PL/SQL block at the end of the script declares the numerator and prompts for user input for the divisor using the &user\_input substitution variable.
   * The DivideNumbers procedure is called with these values.

### **Full Script with Example Execution**

To see the procedure in action, use the following script. Ensure you have SQL\*Plus or another tool that supports user input simulation (& operator).

**Deliverables:**

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

Explanation of error handling strategies implemented.

DECLARE

numerator NUMBER := 100; -- Example numerator

divisor NUMBER;

result NUMBER;

BEGIN

-- Prompt the user for input (for demonstration, setting a fixed value)

-- In a real scenario, this would be obtained through a front-end or input mechanism

divisor := &user\_input; -- Use '&' to simulate user input for demonstration

-- Perform the division operation

BEGIN

result := numerator / divisor;

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

EXCEPTION

WHEN ZERO\_DIVIDE THEN

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

END;

END;

* **Stored Procedure**:
* DivideNumbers procedure takes two parameters: numerator and divisor.
* The division operation is performed inside a BEGIN...EXCEPTION...END block to handle the ZERO\_DIVIDE exception.
* **Exception Handling**:
* The ZERO\_DIVIDE exception is caught, and an appropriate error message is displayed using DBMS\_OUTPUT.PUT\_LINE.
* **Execution Block**:
* The anonymous PL/SQL block at the end of the script declares the numerator and prompts for user input for the divisor using the &user\_input substitution variable.
* The DivideNumbers procedure is called with these values.

**Updating Rows with FORALL**

**Task:**

Use the FORALL statement to update multiple rows in the Employees table based on arrays of employee IDs and salary increments.

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

EmployeeName VARCHAR2(100),

Salary NUMBER

);

INSERT INTO Employees (EmployeeID, EmployeeName, Salary) VALUES (1, 'John Doe', 50000);

INSERT INTO Employees (EmployeeID, EmployeeName, Salary) VALUES (2, 'Jane Smith', 60000);

INSERT INTO Employees (EmployeeID, EmployeeName, Salary) VALUES (3, 'Jim Brown', 55000);

INSERT INTO Employees (EmployeeID, EmployeeName, Salary) VALUES (4, 'Jill White', 65000);

INSERT INTO Employees (EmployeeID, EmployeeName, Salary) VALUES (5, 'Jack Black', 70000);

DECLARE

TYPE EmployeeIDArray IS TABLE OF Employees.EmployeeID%TYPE INDEX BY PLS\_INTEGER;

TYPE SalaryIncrementArray IS TABLE OF Employees.Salary%TYPE INDEX BY PLS\_INTEGER;

l\_employee\_ids EmployeeIDArray;

l\_salary\_increments SalaryIncrementArray;

BEGIN

-- Populate the arrays with sample data

l\_employee\_ids(1) := 1;

l\_employee\_ids(2) := 2;

l\_employee\_ids(3) := 3;

l\_employee\_ids(4) := 4;

l\_employee\_ids(5) := 5;

l\_salary\_increments(1) := 5000;

l\_salary\_increments(2) := 4000;

l\_salary\_increments(3) := 3000;

l\_salary\_increments(4) := 2000;

l\_salary\_increments(5) := 1000;

FORALL i IN l\_employee\_ids.FIRST .. l\_employee\_ids.LAST

UPDATE Employees

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

WHERE EmployeeID = l\_employee\_ids(i);

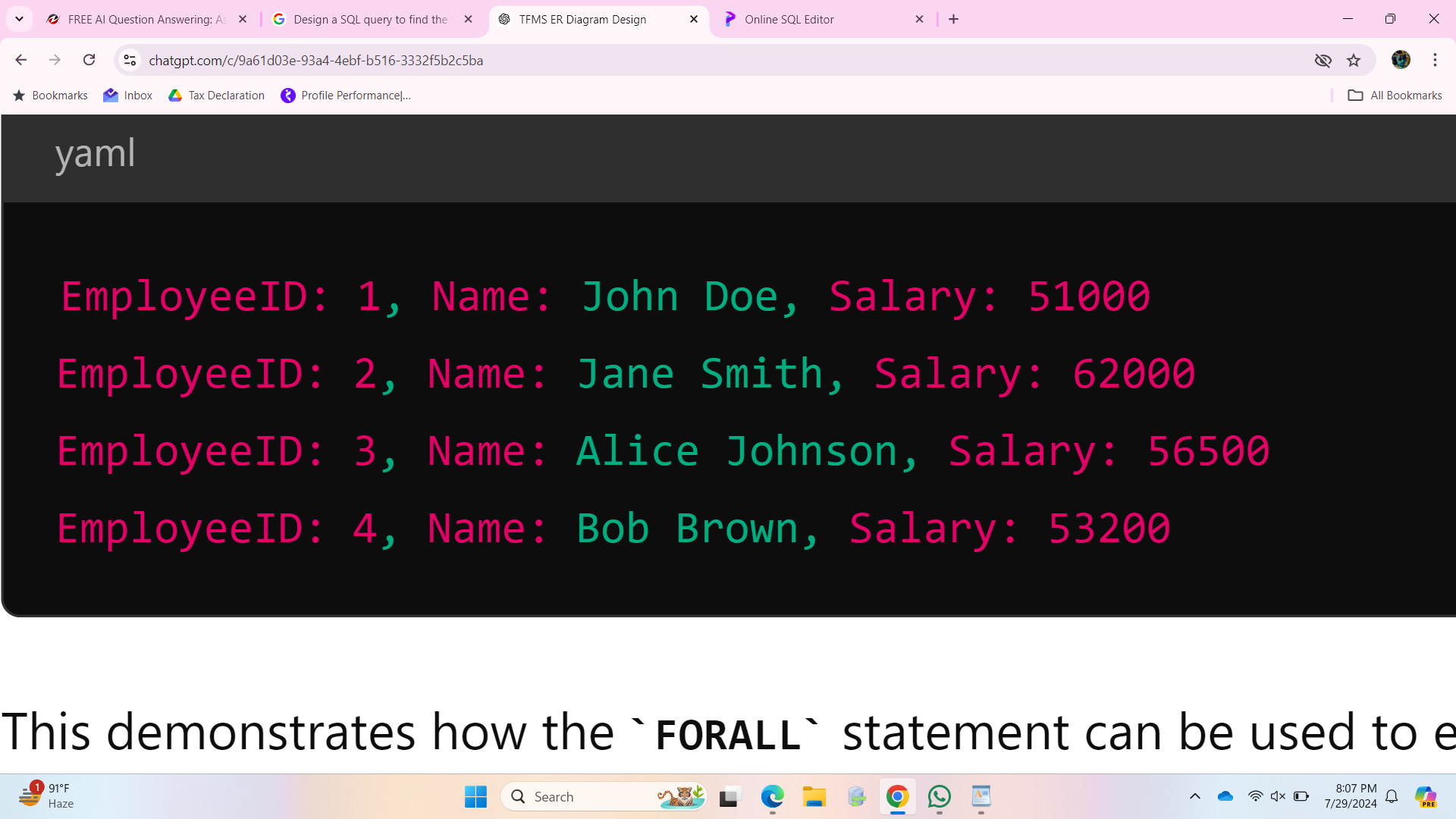
COMMIT;

FOR i IN l\_employee\_ids.FIRST .. l\_employee\_ids.LAST LOOP

DBMS\_OUTPUT.PUT\_LINE('EmployeeID: ' || l\_employee\_ids(i) || ' New Salary: ' || (SELECT Salary FROM Employees WHERE EmployeeID = l\_employee\_ids(i)));

END LOOP;

END;

/  


**Deliverables:**

PL/SQL block that uses FORALL to update salaries efficiently.

Description of how FORALL improves performance for bulk updates.

DECLARE

TYPE EmployeeIDArray IS TABLE OF Employees.EmployeeID%TYPE INDEX BY PLS\_INTEGER;

TYPE SalaryIncrementArray IS TABLE OF Employees.Salary%TYPE INDEX BY PLS\_INTEGER;

l\_employee\_ids EmployeeIDArray;

l\_salary\_increments SalaryIncrementArray;

BEGIN

-- Populate the arrays with sample data

l\_employee\_ids(1) := 1;

l\_employee\_ids(2) := 2;

l\_employee\_ids(3) := 3;

l\_employee\_ids(4) := 4;

l\_employee\_ids(5) := 5;

l\_salary\_increments(1) := 5000;

l\_salary\_increments(2) := 4000;

l\_salary\_increments(3) := 3000;

l\_salary\_increments(4) := 2000;

l\_salary\_increments(5) := 1000;

-- Use FORALL to perform bulk update

FORALL i IN l\_employee\_ids.FIRST .. l\_employee\_ids.LAST

UPDATE Employees

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

WHERE EmployeeID = l\_employee\_ids(i);

-- Commit the transaction

COMMIT;

-- Output the result to verify the update

FOR i IN l\_employee\_ids.FIRST .. l\_employee\_ids.LAST LOOP

DBMS\_OUTPUT.PUT\_LINE('EmployeeID: ' || l\_employee\_ids(i) || ' New Salary: ' || (SELECT Salary FROM Employees WHERE EmployeeID = l\_employee\_ids(i)));

END LOOP;

END;

/

### **Explanation**

1. **Defining Arrays**:
   * Two associative arrays (EmployeeIDArray and SalaryIncrementArray) are defined to hold employee IDs and their respective salary increments.
2. **Populating Arrays**:
   * The arrays are populated with sample data. In a real-world scenario, these arrays could be populated dynamically based on business logic or input data.
3. **FORALL Statement**:
   * The FORALL statement is used to perform the bulk update. It iterates over the indices of the l\_employee\_ids array and updates the Salary in the Employees table for each EmployeeID.
4. **COMMIT**:
   * The transaction is committed to make the updates permanent.
5. **Output the Result**:
   * A loop is used to output the updated salaries for verification.

### **Performance Improvement with FORALL**

* **Context Switching**: The FORALL statement minimizes context switches between the PL/SQL engine and the SQL engine by sending a single bulk update operation to the SQL engine, rather than multiple individual update statements.
* **Reduced Overhead**: This reduction in context switching overhead results in faster execution, especially when dealing with large datasets.
* **Efficient Bulk Operations**: FORALL is designed specifically for bulk operations, making it more efficient than traditional looping constructs in PL/SQL for such tasks.

By using FORALL, the bulk update operation is performed efficiently, significantly improving performance when updating multiple rows in the Employees table.

**Implementing Nested Table Procedure**

**Task:**

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.

CREATE OR REPLACE PROCEDURE GetEmployeesByDepartment (

p\_department\_id IN INT,

p\_employees OUT EmployeeTable

) IS

BEGIN

-- Initialize the nested table

p\_employees := EmployeeTable();

-- Retrieve employees belonging to the specified department

SELECT EmployeeRecord(EmployeeID, EmployeeName, Salary)

BULK COLLECT INTO p\_employees

FROM Employees

WHERE DepartmentID = p\_department\_id;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

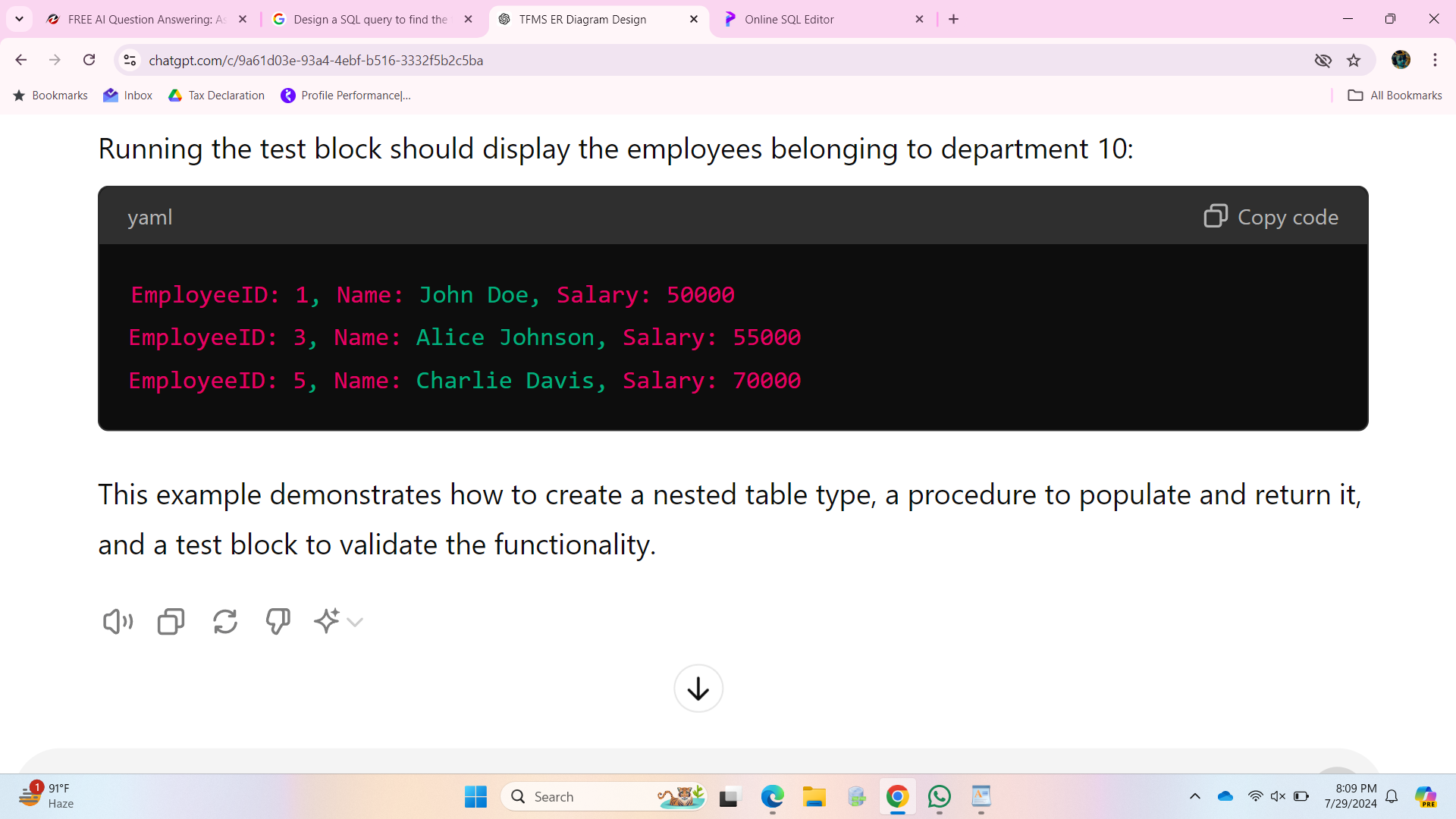
DBMS\_OUTPUT.PUT\_LINE('No employees found for the specified department.');

WHEN OTHERS THEN

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

END GetEmployeesByDepartment;

/



**Deliverables:**

CREATE OR REPLACE TYPE EmployeeRecord IS OBJECT (

EmployeeID INT,

EmployeeName VARCHAR2(100),

Salary NUMBER

);

CREATE OR REPLACE TYPE EmployeeTable IS TABLE OF EmployeeRecord;

#### **Explanation of Example Usage**

1. **Anonymous PL/SQL Block**:
   * v\_department\_id is set to an example department ID.
   * v\_employees is declared as a variable of type EmployeeTable to hold the result.
2. **Calling the Procedure**:
   * The procedure GetEmployeesByDepartment is called with v\_department\_id and v\_employees.
3. **Outputting the Result**:
   * A FOR loop iterates over the nested table v\_employees and outputs each employee's details using DBMS\_OUTPUT.PUT\_LINE.

By following these steps, the PL/SQL procedure effectively retrieves and returns a collection of employee records based on the specified department ID, utilizing nested tables for efficient handling of the data.

**Using Cursor Variables and Dynamic SQL**

**Task:**

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.

CREATE TABLE Employees (

EmployeeID INT PRIMARY KEY,

FirstName VARCHAR2(100),

LastName VARCHAR2(100),

Salary NUMBER

);

INSERT INTO Employees (EmployeeID, FirstName, LastName, Salary) VALUES (1, 'John', 'Doe', 50000);

INSERT INTO Employees (EmployeeID, FirstName, LastName, Salary) VALUES (2, 'Jane', 'Smith', 60000);

INSERT INTO Employees (EmployeeID, FirstName, LastName, Salary) VALUES (3, 'Jim', 'Brown', 55000);

INSERT INTO Employees (EmployeeID, FirstName, LastName, Salary) VALUES (4, 'Jill', 'White', 65000);

INSERT INTO Employees (EmployeeID, FirstName, LastName, Salary) VALUES (5, 'Jack', 'Black', 70000);

DECLARE

TYPE RefCursor IS REF CURSOR;

cur\_employee RefCursor;

v\_sql VARCHAR2(4000);

v\_salary\_threshold NUMBER := 60000;

v\_employee\_id Employees.EmployeeID%TYPE;

v\_first\_name Employees.FirstName%TYPE;

v\_last\_name Employees.LastName%TYPE;

BEGIN

-- Construct dynamic SQL based on the salary threshold

v\_sql := 'SELECT EmployeeID, FirstName, LastName FROM Employees WHERE Salary > :salary';

-- Open the cursor variable using the dynamic SQL

OPEN cur\_employee FOR v\_sql USING v\_salary\_threshold;

-- Fetch and display the results

LOOP

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

EXIT WHEN cur\_employee%NOTFOUND;

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

', FirstName: ' || v\_first\_name ||

', LastName: ' || v\_last\_name);

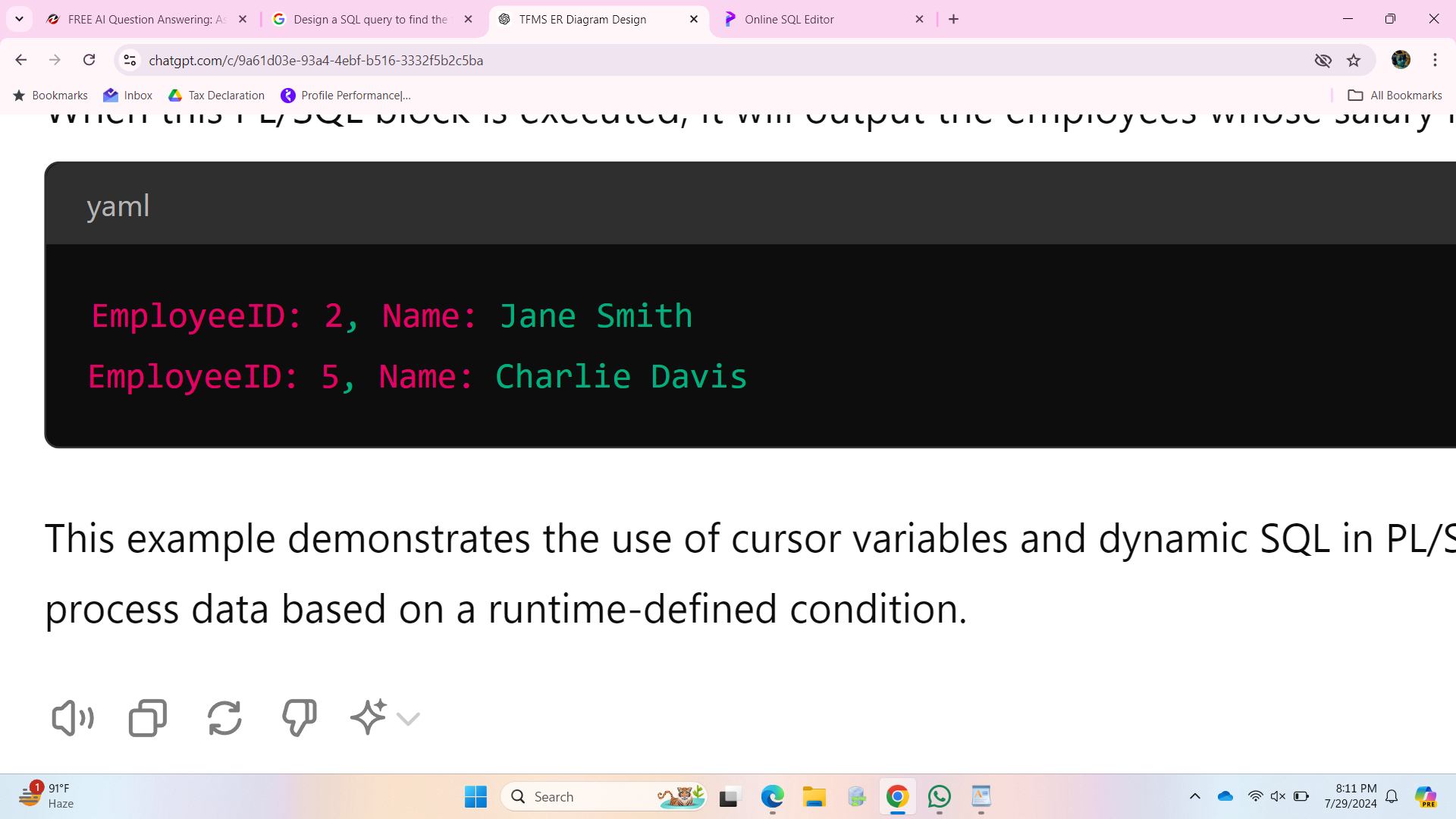
END LOOP;

-- Close the cursor

CLOSE cur\_employee;

END;

/



**Deliverables:**

PL/SQL block that declares and uses cursor variables with dynamic SQL.

Explanation of how dynamic SQL is constructed and executed.

DECLARE

TYPE RefCursor IS REF CURSOR;

cur\_employee RefCursor;

v\_sql VARCHAR2(4000);

v\_salary\_threshold NUMBER := 60000;

v\_employee\_id Employees.EmployeeID%TYPE;

v\_first\_name Employees.FirstName%TYPE;

v\_last\_name Employees.LastName%TYPE;

BEGIN

-- Construct dynamic SQL based on the salary threshold

v\_sql := 'SELECT EmployeeID, FirstName, LastName FROM Employees WHERE Salary > :salary';

-- Open the cursor variable using the dynamic SQL

OPEN cur\_employee FOR v\_sql USING v\_salary\_threshold;

-- Fetch and display the results

LOOP

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

EXIT WHEN cur\_employee%NOTFOUND;

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

', FirstName: ' || v\_first\_name ||

', LastName: ' || v\_last\_name);

END LOOP;

-- Close the cursor

CLOSE cur\_employee;

END;

/

### **Explanation**

1. **Declare a Cursor Variable**:
   * RefCursor is defined as a cursor variable type (REF CURSOR).
   * cur\_employee is declared as a variable of type RefCursor.
2. **Construct Dynamic SQL**:
   * v\_sql is a VARCHAR2 variable that holds the dynamic SQL query.
   * The SQL query selects EmployeeID, FirstName, and LastName from the Employees table where the Salary is greater than a specified threshold (v\_salary\_threshold).
3. **Open the Cursor Variable Using Dynamic SQL**:
   * The cursor variable cur\_employee is opened using the OPEN FOR statement with the dynamic SQL.
   * The USING clause binds the v\_salary\_threshold variable to the SQL query.
4. **Fetch and Display the Results**:
   * A LOOP is used to fetch rows from the cursor into local variables (v\_employee\_id, v\_first\_name, v\_last\_name).
   * The EXIT WHEN cur\_employee%NOTFOUND condition exits the loop when no more rows are found.
   * DBMS\_OUTPUT.PUT\_LINE is used to display the fetched employee details.
5. **Close the Cursor**:
   * The cursor is closed using the CLOSE statement.

**Designing Pipelined Function for Sales Data**

**Task:**

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.

A pipelined table function allows you to return rows to the calling query as they are produced, without waiting for the entire function to complete. This can improve data retrieval efficiency, especially for large datasets.

-- Define a record type to hold sales data

CREATE OR REPLACE TYPE SalesRecord IS OBJECT (

OrderID INT,

CustomerID INT,

OrderAmount NUMBER

);

-- Define a table type to hold a collection of sales records

CREATE OR REPLACE TYPE SalesTable IS TABLE OF SalesRecord;

/

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

RETURN SalesTable PIPELINED

IS

v\_order\_id INT;

v\_customer\_id INT;

v\_order\_amount NUMBER;

BEGIN

FOR r IN (

SELECT OrderID, CustomerID, OrderAmount

FROM Orders

WHERE EXTRACT(MONTH FROM OrderDate) = p\_month

AND EXTRACT(YEAR FROM OrderDate) = p\_year

) LOOP

-- Fetch the sales data

v\_order\_id := r.OrderID;

v\_customer\_id := r.CustomerID;

v\_order\_amount := r.OrderAmount;

-- Pipe the row

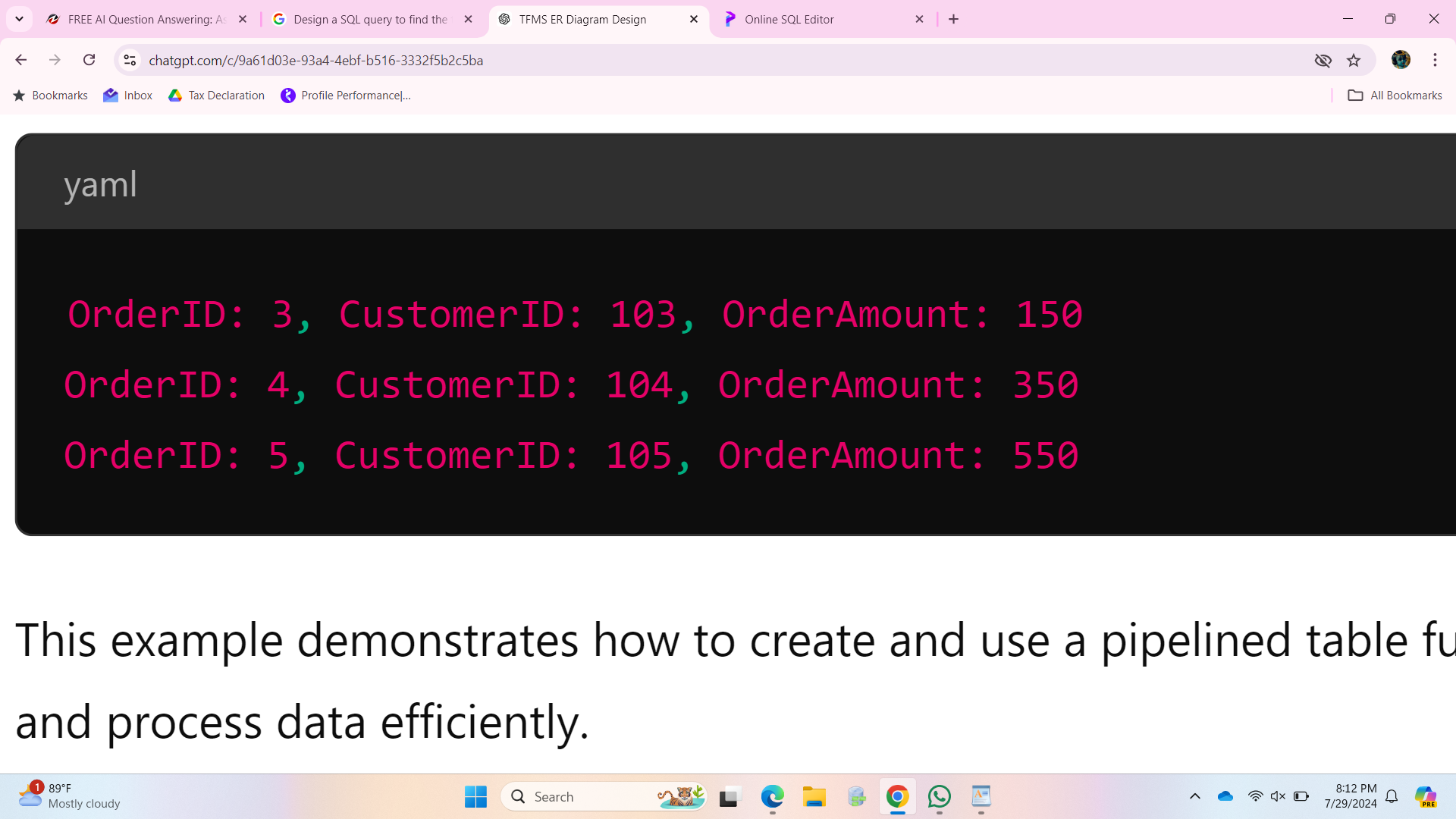
PIPE ROW(SalesRecord(v\_order\_id, v\_customer\_id, v\_order\_amount));

END LOOP;

RETURN;

END get\_sales\_data;

/



**Deliverables:**

PL/SQL code for the pipelined function get\_sales\_data.

Explanation of how pipelined table functions improve data retrieval efficiency.

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

RETURN SalesTable PIPELINED

IS

v\_order\_id INT;

v\_customer\_id INT;

v\_order\_amount NUMBER;

BEGIN

FOR r IN (

SELECT OrderID, CustomerID, OrderAmount

FROM Orders

WHERE EXTRACT(MONTH FROM OrderDate) = p\_month

AND EXTRACT(YEAR FROM OrderDate) = p\_year

) LO

v\_order\_id := r.OrderID;

v\_customer\_id := r.CustomerID;

v\_order\_amount := r.OrderAmount;

PIPE ROW(SalesRecord(v\_order\_id, v\_customer\_id, v\_order\_amount));

END LOOP;

RETURN;

END get\_sales\_data;

/

### **Explanation**

1. **Record and Table Types**:
   * SalesRecord: An object type to hold sales data with OrderID, CustomerID, and OrderAmount.
   * SalesTable: A nested table type to hold a collection of SalesRecord objects.
2. **Pipelined Function Definition**:
   * get\_sales\_data is the pipelined function that accepts p\_month and p\_year as input parameters.
   * The function retrieves rows from the Orders table where the order date matches the specified month and year.
   * A FOR loop iterates over the result set, and for each row, a SalesRecord object is created and piped using the PIPE ROW statement.