**ER Diagram Question: Traffic Flow Management System (TFMS)**

Maithreyee. J

192325058

AI-ML

**CSA0564**

**Database Management Systems for Query Processing**

***ASSIGNMENT QUESTIONS***

**Question 1:**

**ER Diagram Question: Traffic Flow Management System (TFMS)**

**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.

**Entities and Attributes:**

1. **Roads:**
   * RoadID (PK)
   * RoadName
   * Length (in meters)
   * SpeedLimit (in km/h)
2. **Intersections:**
   * IntersectionID (PK)
   * IntersectionName
   * Latitude
   * Longitude
3. **Traffic Signals:**
   * SignalID (PK)
   * IntersectionID (FK)
   * SignalStatus (Green, Yellow, Red)
   * Timer (countdown to next change)
4. **Traffic Data:**
   * TrafficDataID (PK)
   * RoadID (FK)
   * Timestamp
   * Speed (average speed on the road)
   * CongestionLevel (degree of traffic congestion)

**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).

**Relationships:**

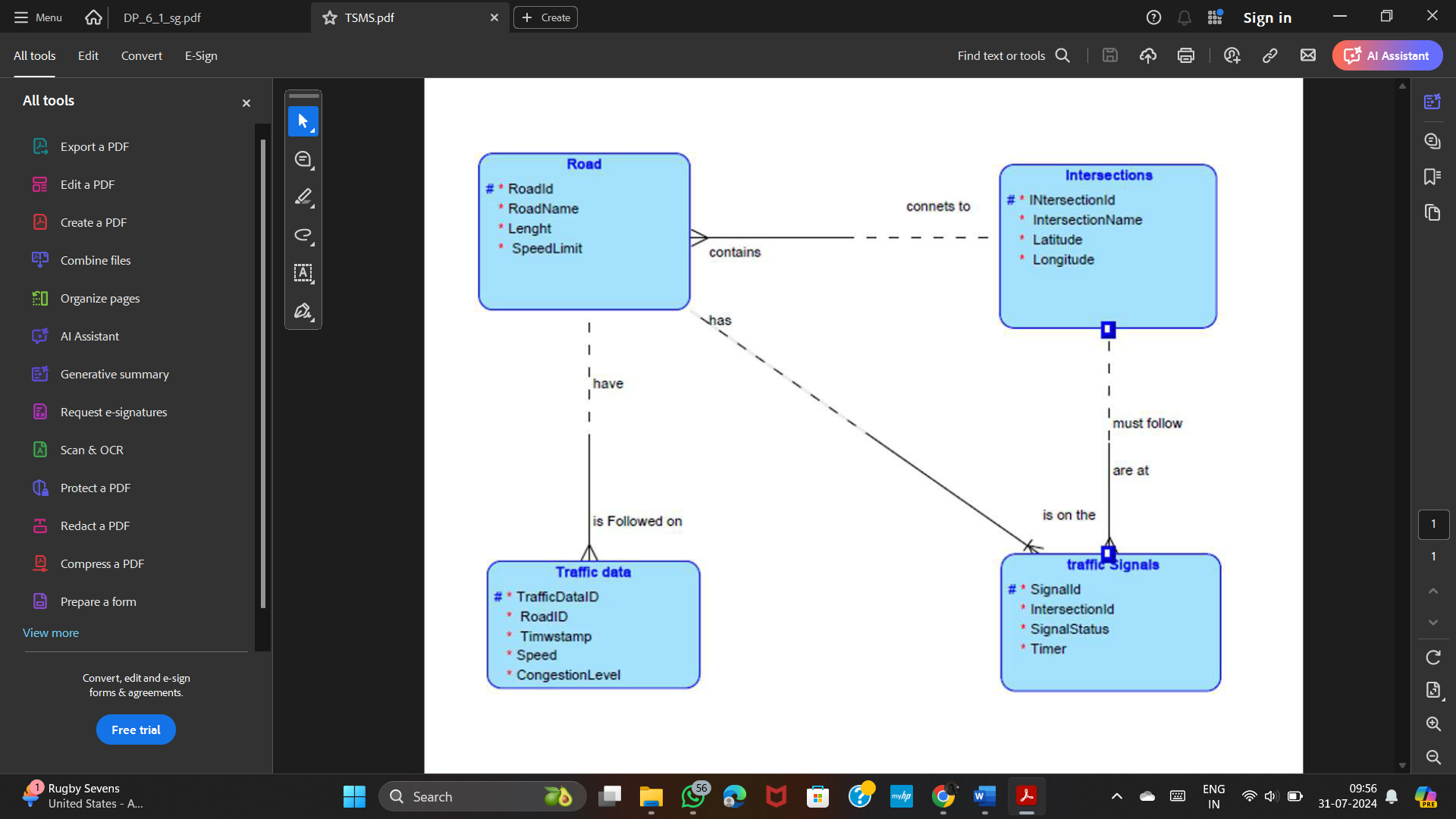
1. **Roads and Intersections**
   * A road can connect to multiple intersections (one-to-many).
   * An intersection can be connected by multiple roads (many-to-one).
2. **Intersections and Traffic Signals**
   * An intersection can have multiple traffic signals (one-to-many).
   * A traffic signal belongs to a single intersection (many-to-one).
3. **Roads and Traffic Data**
   * A road can have multiple traffic data records (one-to-many).
   * A traffic data record is associated with one road (many-to-one).

**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.

Below is the ER diagram for the Traffic Flow Management System (TFMS):



**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.

**Normalization:**

1. **1NF (First Normal Form)**:
   * Ensure that all attributes have atomic values.
   * Each entity has a unique primary key.
2. **2NF (Second Normal Form)**:
   * All non-key attributes are fully functional dependent on the primary key.
   * No partial dependency (attributes depend only on part of the composite primary key).
3. **3NF (Third Normal Form)**:
   * All attributes are functionally dependent only on the primary key.
   * No transitive dependency (non-key attributes dependent on other non-key attributes).

**Question 2:**

**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.

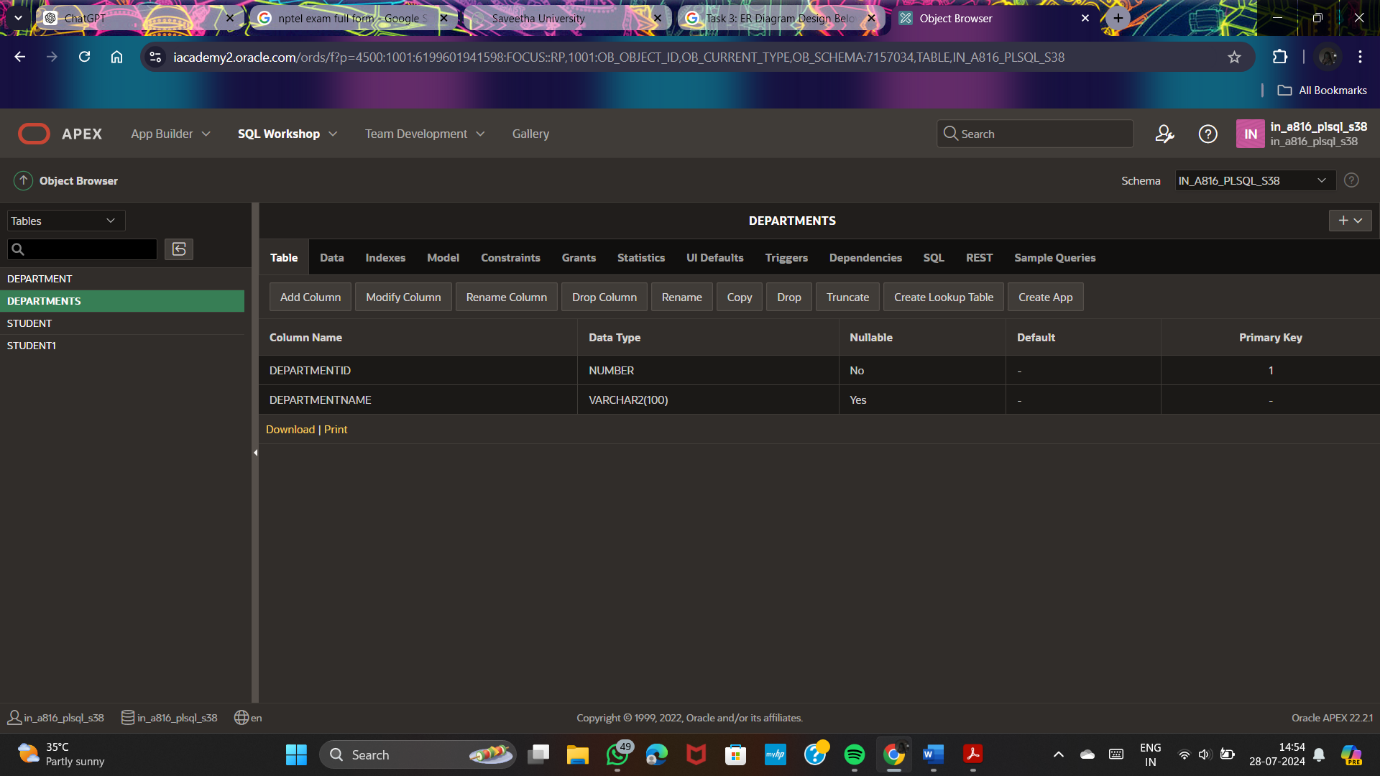
Step 1: Create Tables

CREATE TABLE Departments (

DepartmentID INT PRIMARY KEY,

DepartmentName VARCHAR2(100)

);



CREATE TABLE Employees (

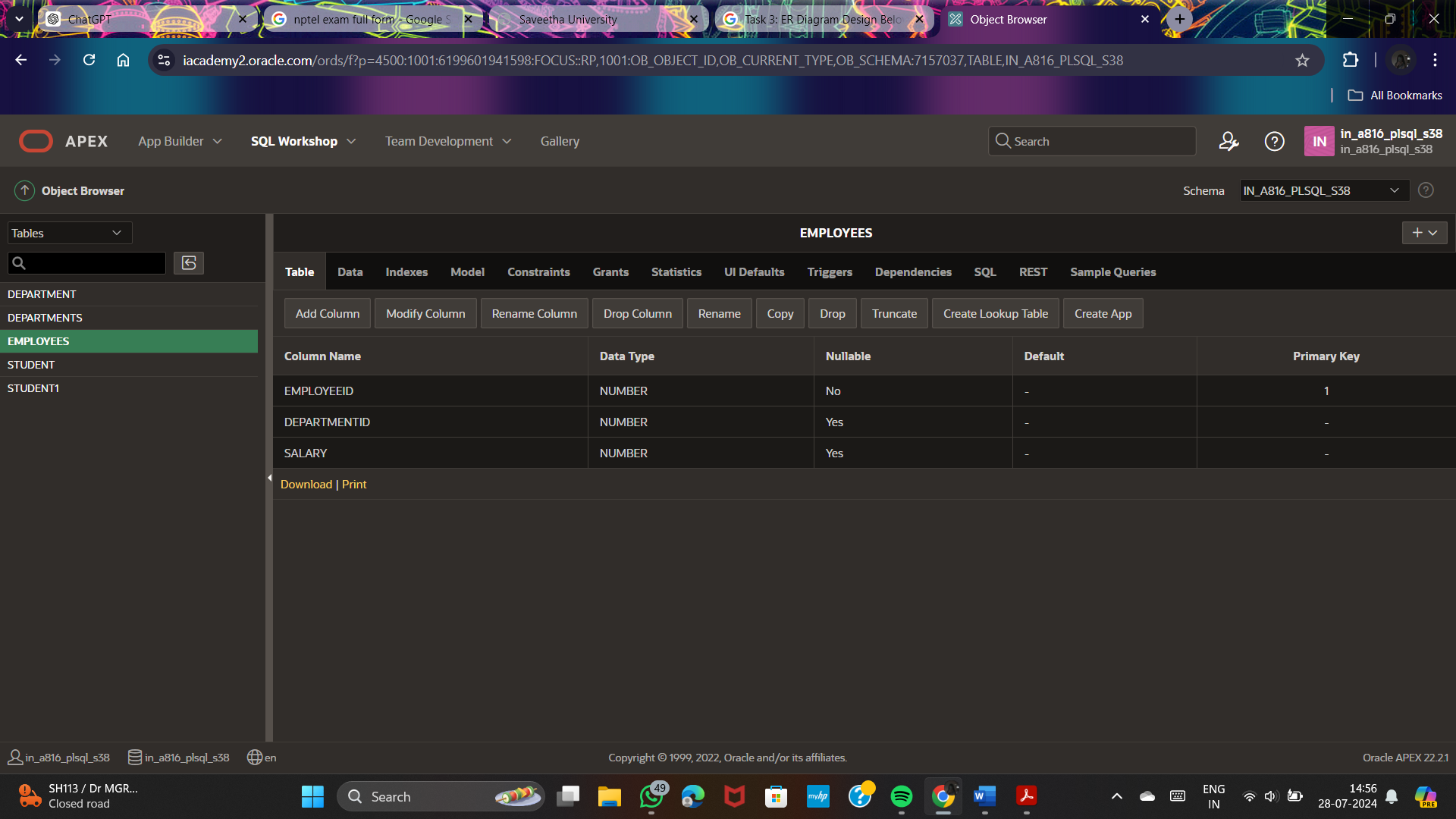
EmployeeID INT PRIMARY KEY,

DepartmentID INT,

Salary NUMBER,

FOREIGN KEY (DepartmentID) REFERENCES Departments(DepartmentID)

);



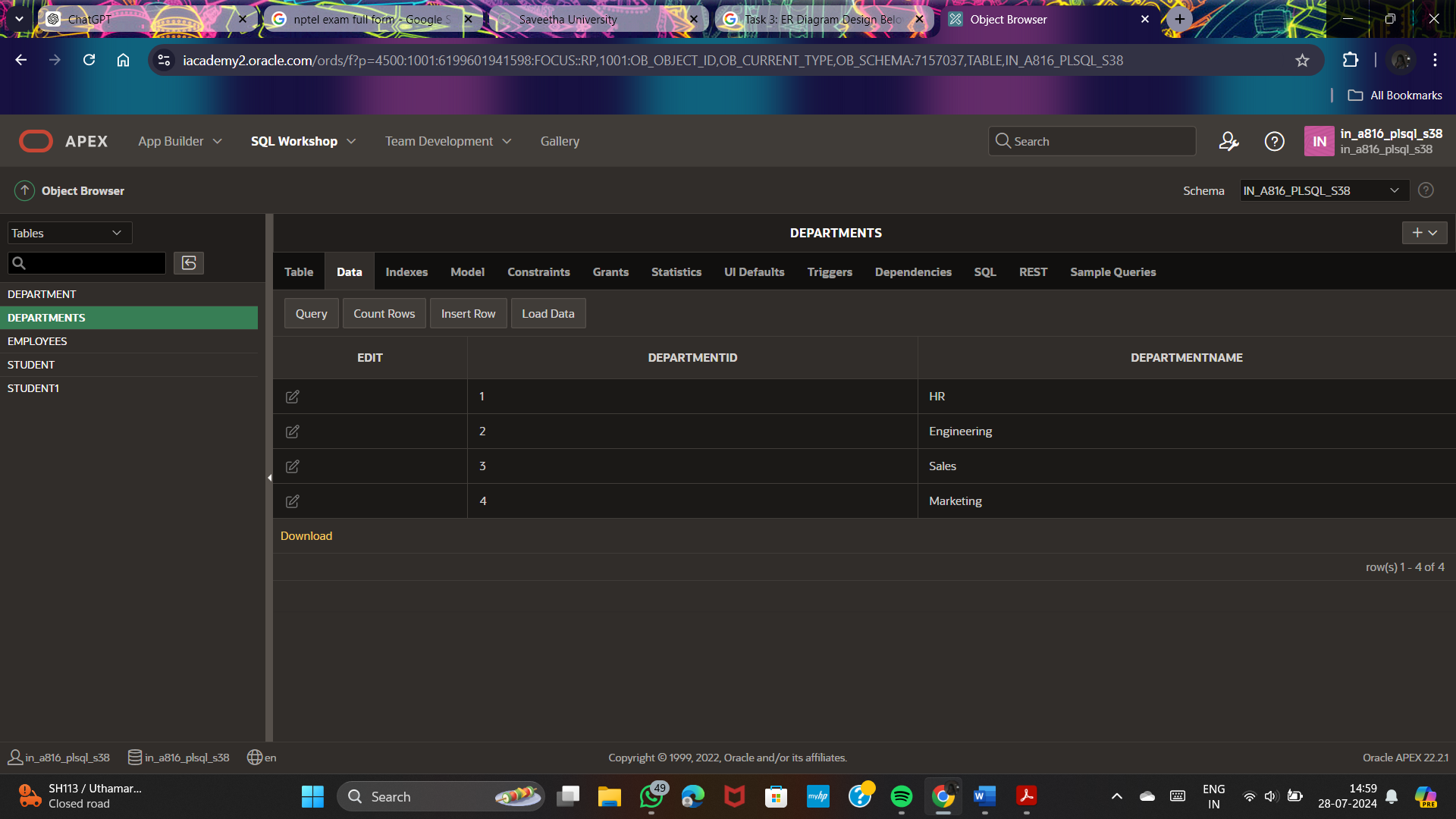
Step 2: Insert Sample Data

INSERT INTO Departments (DepartmentID, DepartmentName) VALUES (1, 'HR');

INSERT INTO Departments (DepartmentID, DepartmentName) VALUES (2, 'Engineering');

INSERT INTO Departments (DepartmentID, DepartmentName) VALUES (3, 'Sales');

INSERT INTO Departments (DepartmentID, DepartmentName) VALUES (4, 'Marketing');



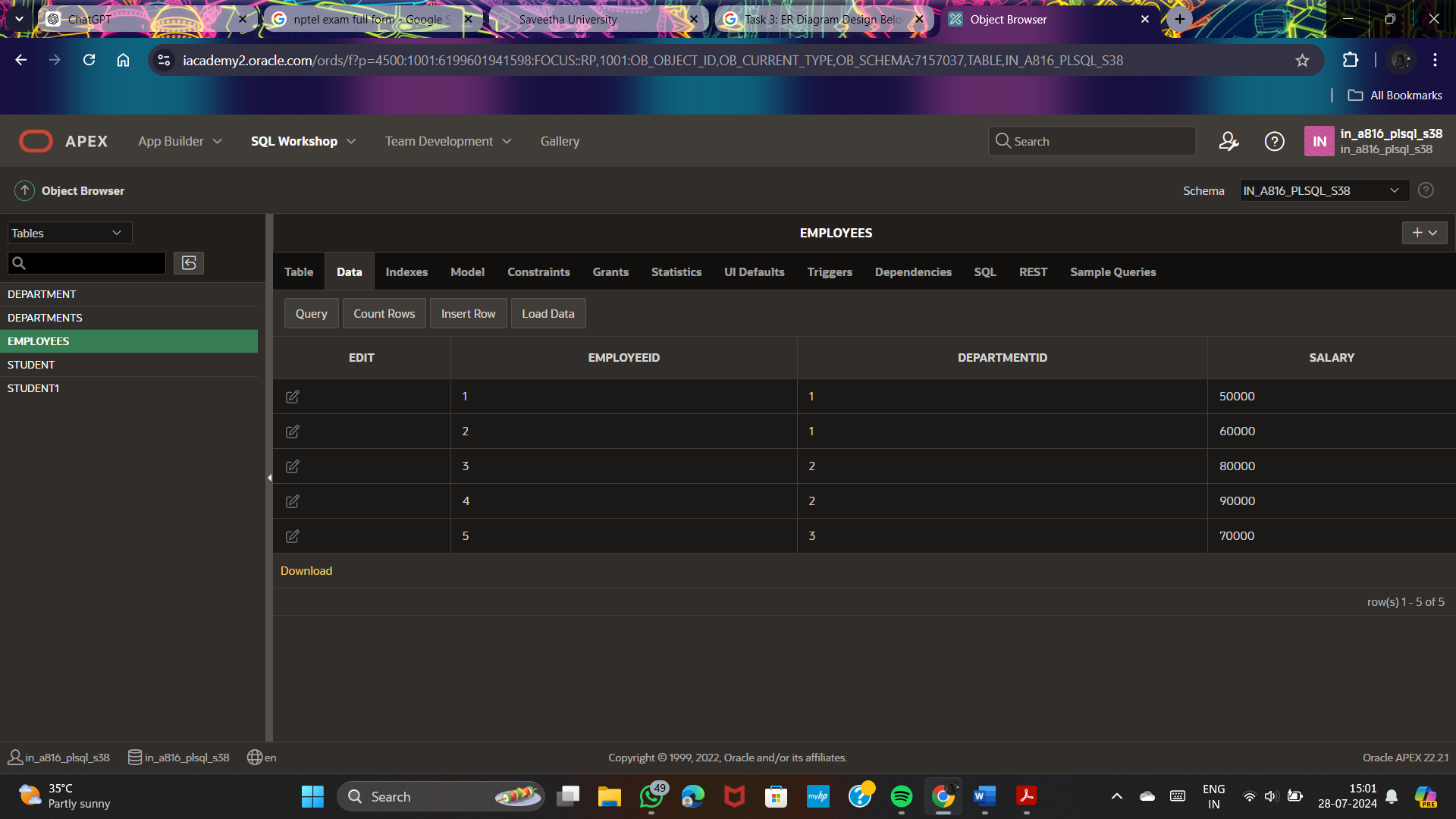
INSERT INTO Employees (EmployeeID, DepartmentID, Salary) VALUES (1, 1, 50000);

INSERT INTO Employees (EmployeeID, DepartmentID, Salary) VALUES (2, 1, 60000);

INSERT INTO Employees (EmployeeID, DepartmentID, Salary) VALUES (3, 2, 80000);

INSERT INTO Employees (EmployeeID, DepartmentID, Salary) VALUES (4, 2, 90000);

INSERT INTO Employees (EmployeeID, DepartmentID, Salary) VALUES (5, 3, 70000);



Step 3: Write the SQL Query

SELECT

d.DepartmentID,

d.DepartmentName,

AVG(e.Salary) AS AvgSalary

FROM

Departments d

LEFT JOIN

Employees e ON d.DepartmentID = e.DepartmentID

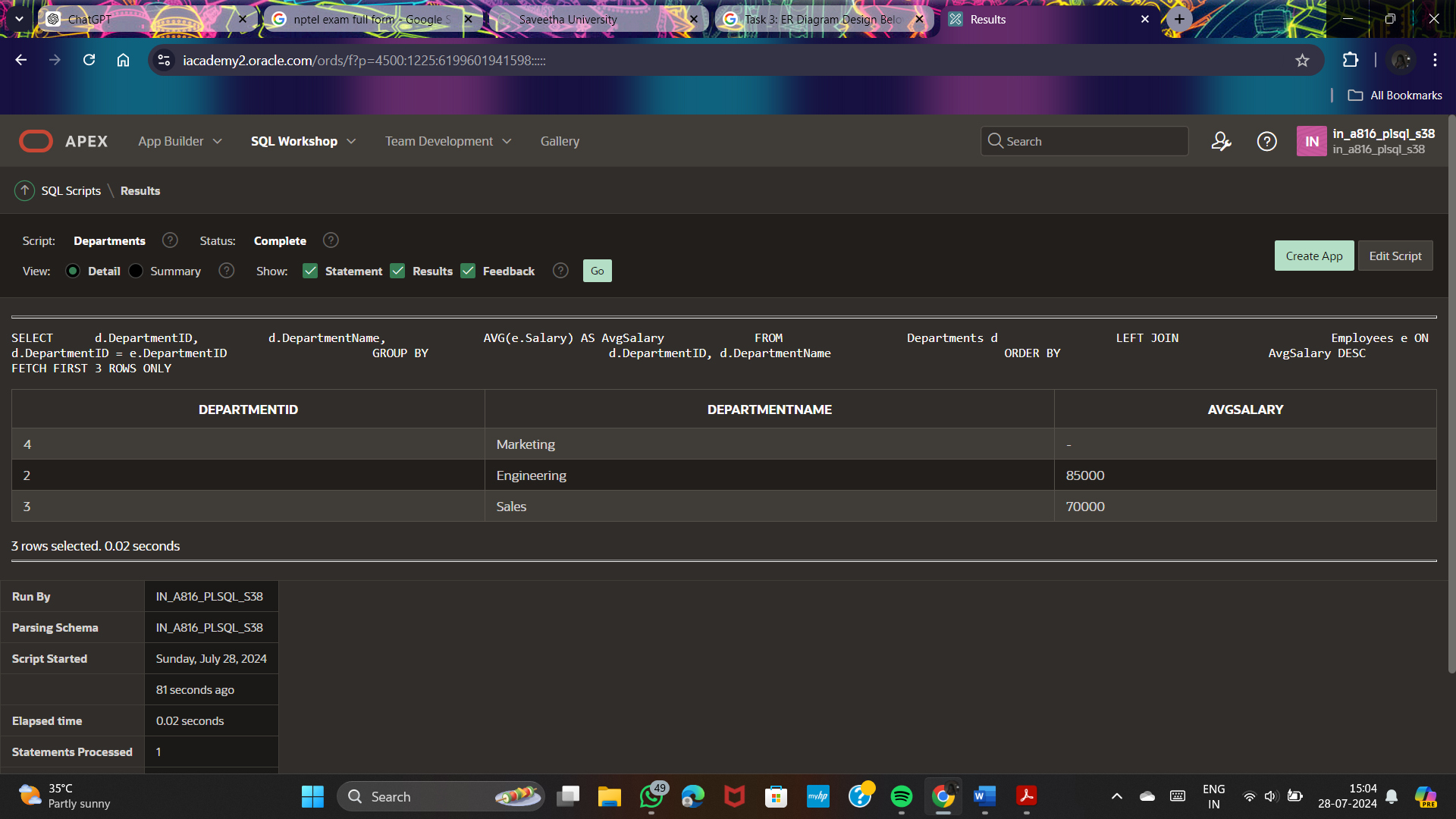
GROUP BY

d.DepartmentID, d.DepartmentName

ORDER BY

AvgSalary DESC

FETCH FIRST 3 ROWS ONLY;



**Explanation**

1.Creating Tables:

* Departments table includes DepartmentID and DepartmentName.
* Employees table includes EmployeeID, DepartmentID, and Salary. The DepartmentID is a foreign key referencing Departments.

2.Inserting Sample Data:

* Four departments are inserted into the Departments table.
* Five employees are inserted into the Employees table, with some departments having no employees.

3.SQL Query:

* LEFT JOIN: Ensures all departments are included in the result, even those without employees.
* AVG Function: Calculates the average salary for each department. Departments without employees will have an AvgSalary of NULL.
* GROUP BY: Groups results by DepartmentID and DepartmentName.
* ORDER BY: Orders results by average salary in descending order.
* FETCH FIRST 3 ROWS ONLY: Limits the result set to the top 3 departments with the highest average salaries.

**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.

**SQL Query Using Recursive CTE**

Assume we have a **Categories** table with the following structure:

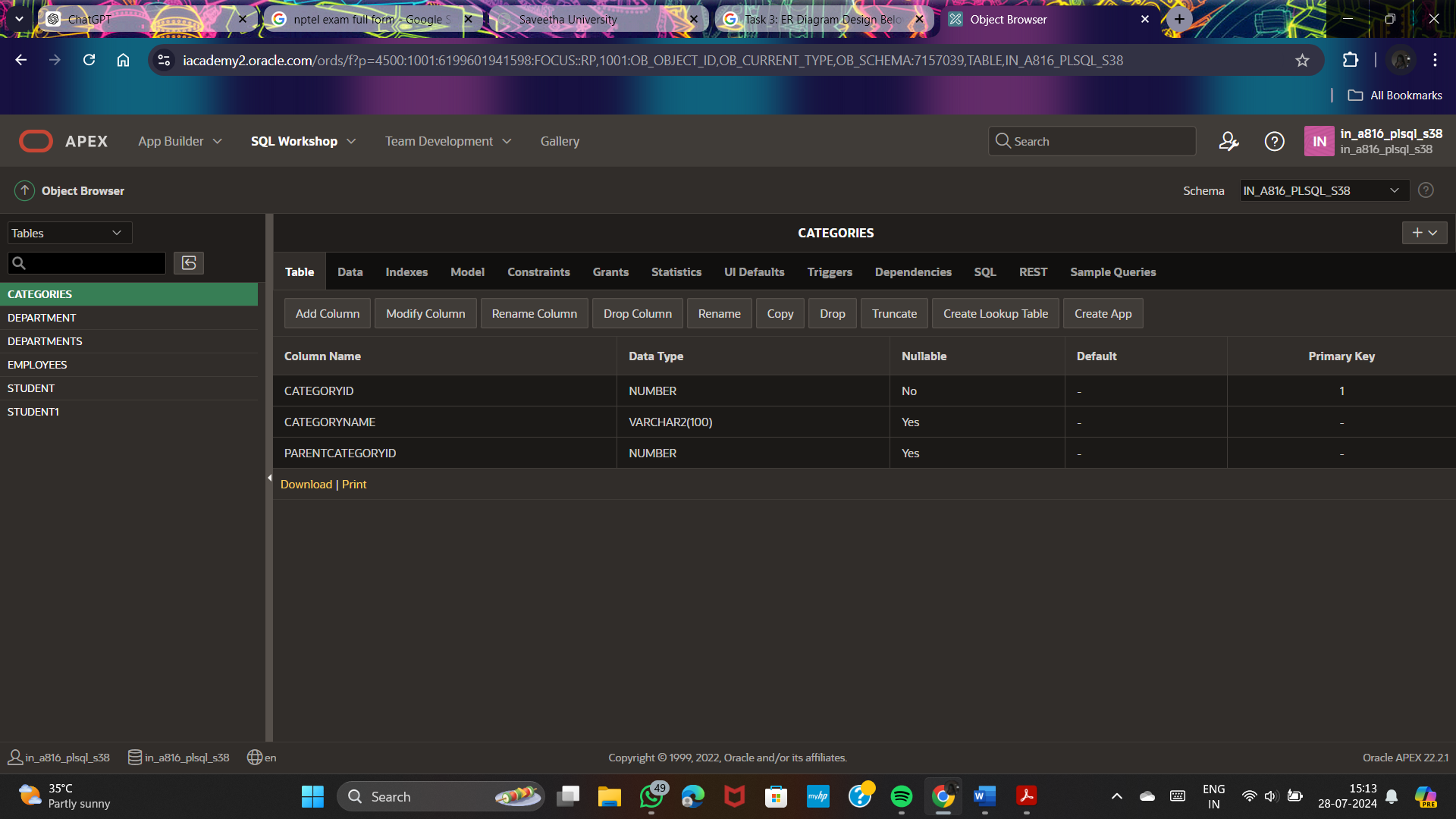
CREATE TABLE Categories (

CategoryID INT PRIMARY KEY,

CategoryName VARCHAR2(100),

ParentCategoryID INT

);



**SQL Query to Retrieve Hierarchical Category Paths:**

WITH RECURSIVE CategoryHierarchy AS (

SELECT

CategoryID,

CategoryName,

ParentCategoryID,

CategoryName AS Path

FROM

Categories

WHERE

ParentCategoryID IS NULL

UNION ALL

SELECT

c.CategoryID,

c.CategoryName,

c.ParentCategoryID,

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

FROM

Categories c

INNER JOIN

CategoryHierarchy ch ON c.ParentCategoryID = ch.CategoryID

)

SELECT

CategoryID,

CategoryName,

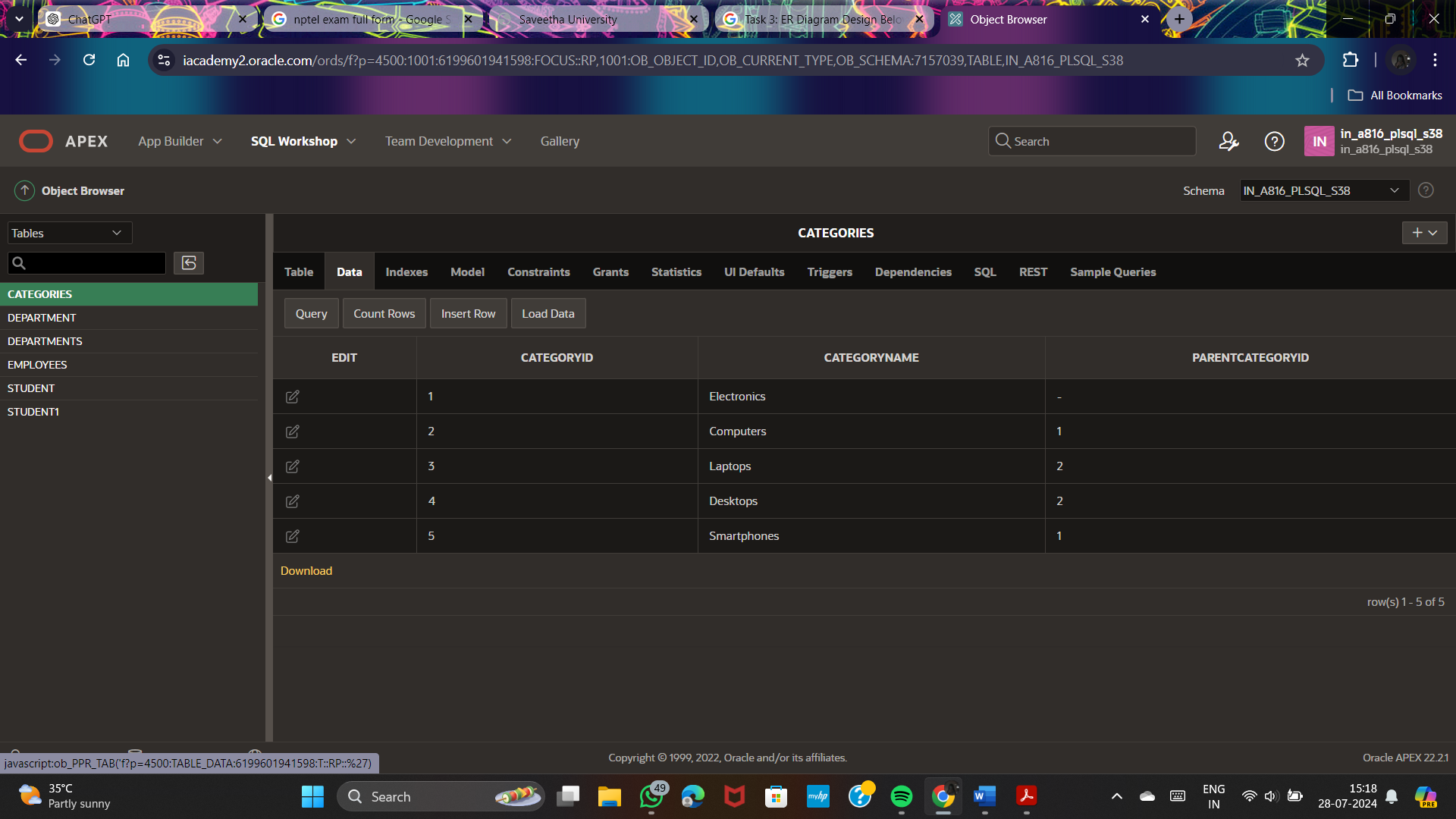
Path

FROM

CategoryHierarchy

ORDER BY

Path;



**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.

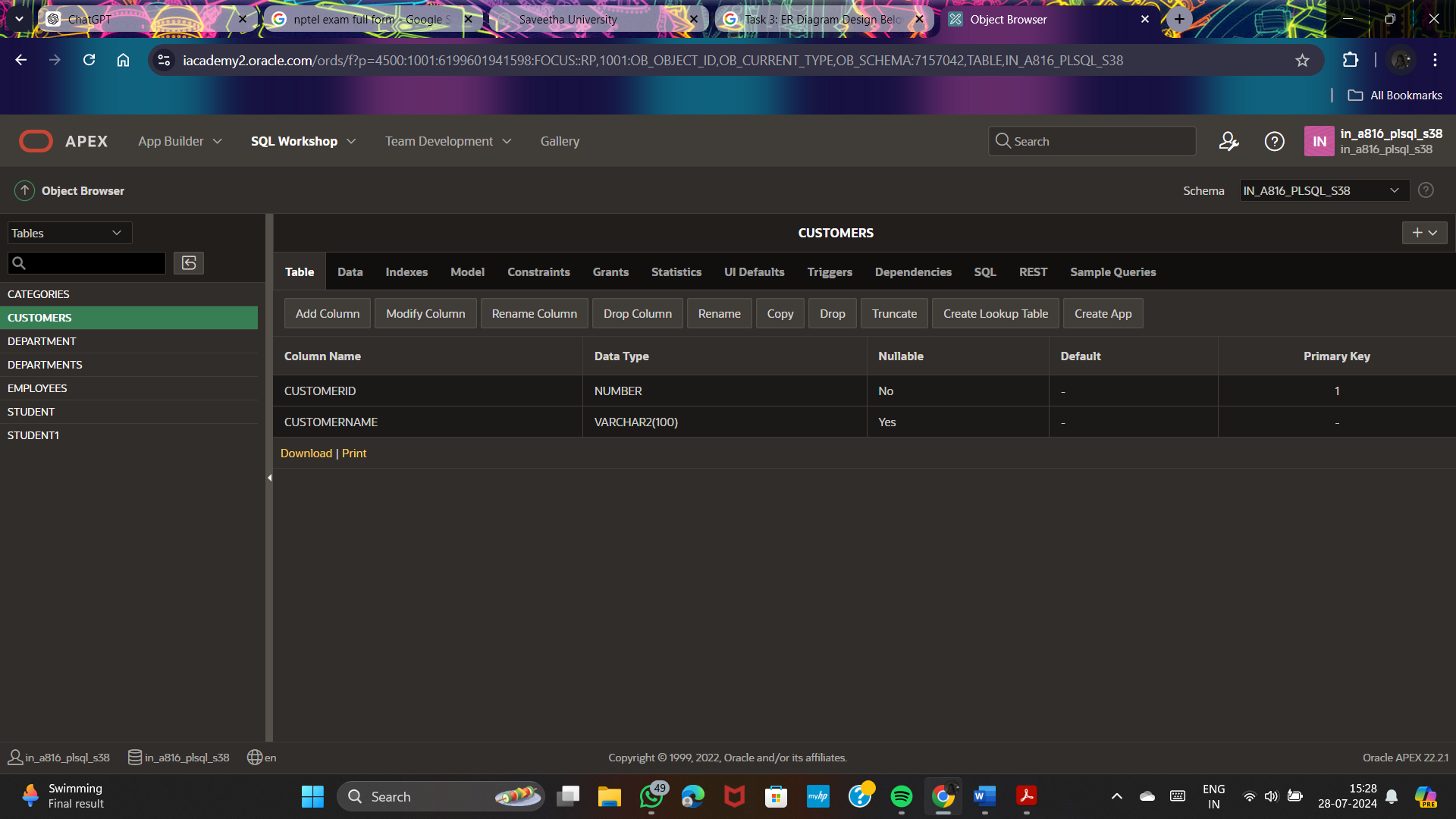
**Customers Table:**

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

CustomerName VARCHAR2(100)

);



**Purchases Table**:

CREATE TABLE Purchases (

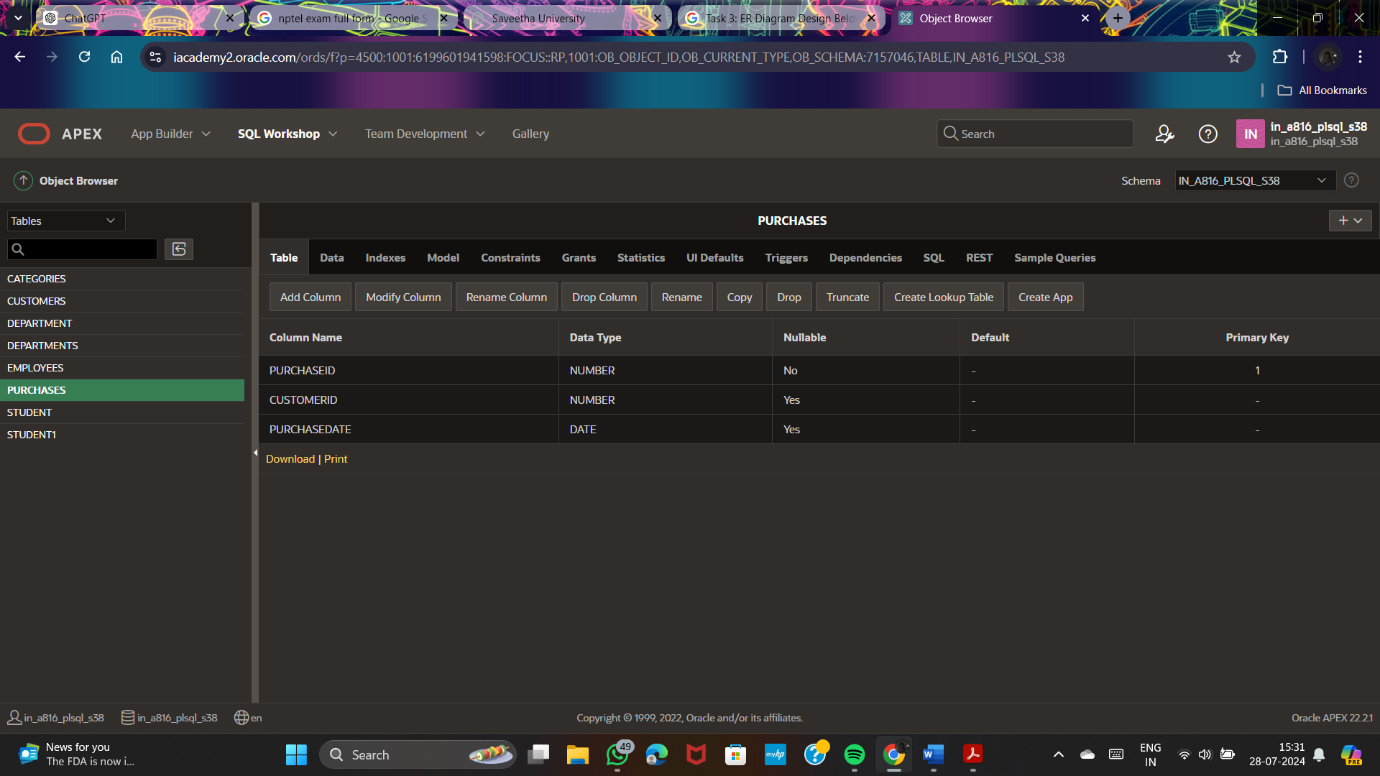
PurchaseID INT PRIMARY KEY,

CustomerID INT,

PurchaseDate DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);



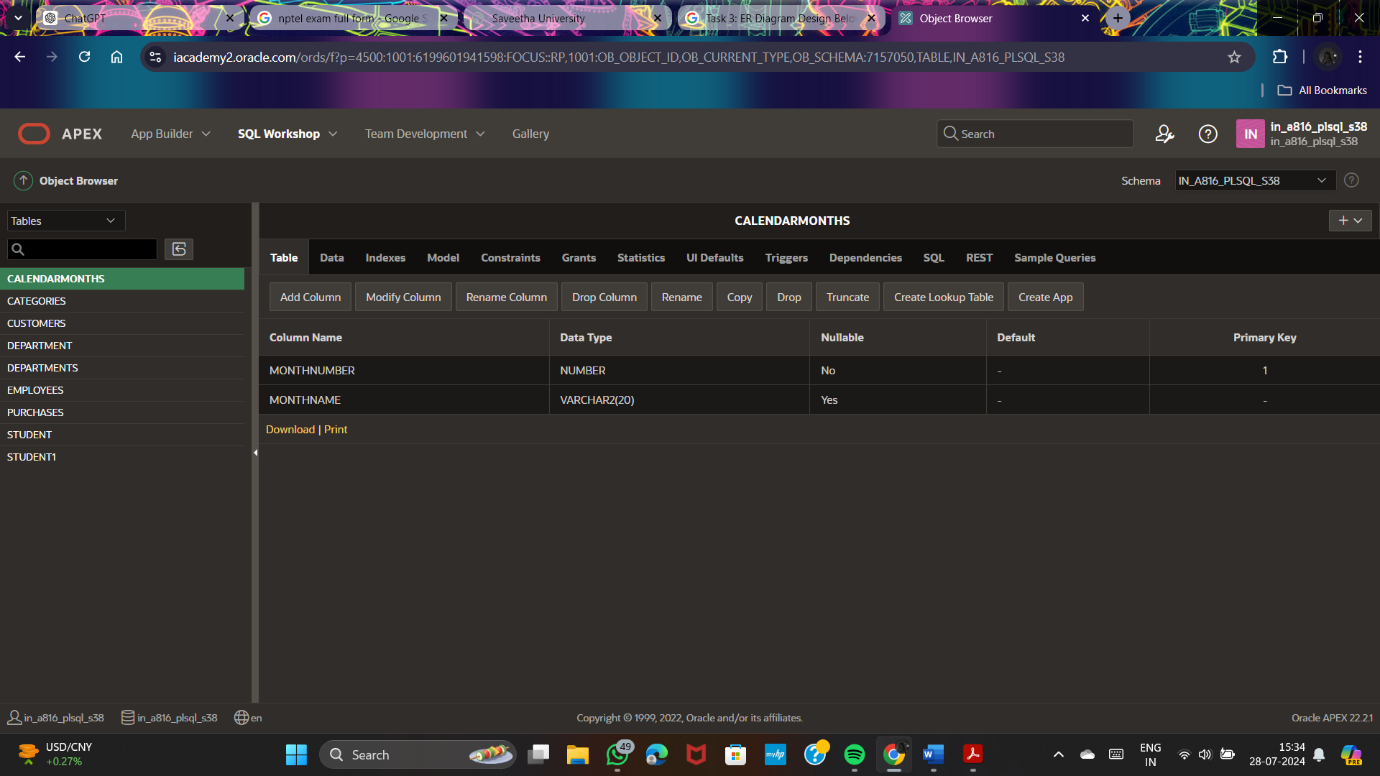
**Create a Calendar Table for Months:**

CREATE TABLE CalendarMonths (

MonthNumber INT PRIMARY KEY,

MonthName VARCHAR2(20)

);



INSERT INTO CalendarMonths (MonthNumber, MonthName) VALUES (1, 'January');

INSERT INTO CalendarMonths (MonthNumber, MonthName) VALUES (2, 'February');

INSERT INTO CalendarMonths (MonthNumber, MonthName) VALUES (3, 'March');

INSERT INTO CalendarMonths (MonthNumber, MonthName) VALUES (4, 'April');

INSERT INTO CalendarMonths (MonthNumber, MonthName) VALUES (5, 'May');

INSERT INTO CalendarMonths (MonthNumber, MonthName) VALUES (6, 'June');

INSERT INTO CalendarMonths (MonthNumber, MonthName) VALUES (7, 'July');

INSERT INTO CalendarMonths (MonthNumber, MonthName) VALUES (8, 'August');

INSERT INTO CalendarMonths (MonthNumber, MonthName) VALUES (9, 'September');

INSERT INTO CalendarMonths (MonthNumber, MonthName) VALUES (10, 'October');

INSERT INTO CalendarMonths (MonthNumber, MonthName) VALUES (11, 'November');

INSERT INTO CalendarMonths (MonthNumber, MonthName) VALUES (12, 'December');



WITH MonthlyPurchases AS (

SELECT

TO\_CHAR(PurchaseDate, 'MM') AS MonthNumber,

TO\_CHAR(PurchaseDate, 'Month') AS MonthName,

CustomerID

FROM

Purchases

WHERE

EXTRACT(YEAR FROM PurchaseDate) = EXTRACT(YEAR FROM SYSDATE)

GROUP BY

TO\_CHAR(PurchaseDate, 'MM'), TO\_CHAR(PurchaseDate, 'Month'), CustomerID

),

CustomerCounts AS (

SELECT

MonthNumber,

MonthName,

COUNT(DISTINCT CustomerID) AS CustomerCount

FROM

MonthlyPurchases

GROUP BY

MonthNumber, MonthName

)

SELECT

cm.MonthName,

NVL(cc.CustomerCount, 0) AS CustomerCount

FROM

CalendarMonths cm

LEFT JOIN

CustomerCounts cc ON cm.MonthNumber = cc.MonthNumber

ORDER BY

cm.MonthNumber;

**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.

**Locations Table**:

CREATE TABLE Locations (

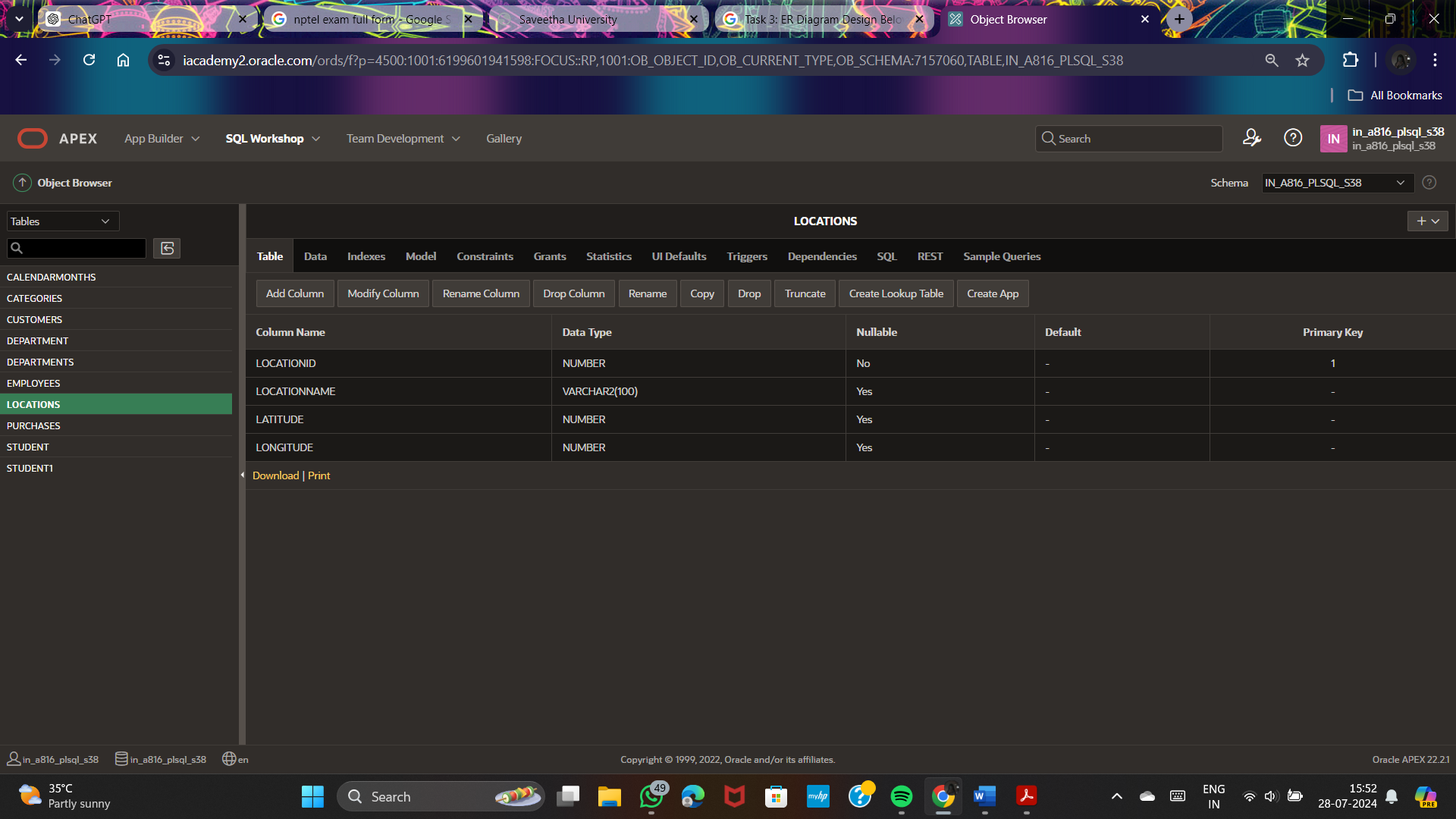
LocationID INT PRIMARY KEY,

LocationName VARCHAR2(100),

Latitude NUMBER,

Longitude NUMBER

);



**SQL Query:**

DECLARE

v\_Latitude NUMBER := 40.7128 (e.g., New York City)

v\_Longitude NUMBER := -74.0060 point (e.g., New York City)

BEGIN

SELECT

LocationID,

LocationName,

Latitude,

Longitude,

(6371 \* ACOS(

COS(RADIANS(v\_Latitude)) \* COS(RADIANS(Latitude)) \*

COS(RADIANS(Longitude) - RADIANS(v\_Longitude)) +

SIN(RADIANS(v\_Latitude)) \* SIN(RADIANS(Latitude))

)) AS Distance

FROM

Locations

ORDER BY

Distance

FETCH FIRST 5 ROWS ONLY;

END;

**SQL Query Breakdown**

1. **Given Latitude and Longitude**:
   * Replace **v\_Latitude** and **v\_Longitude** with the coordinates of the point you want to find the closest locations to.
2. **SELECT Clause**:
   * **LocationID**, **LocationName**, **Latitude**, **Longitude**: Basic information about each location.
   * Haversine formula to calculate the distance from the given point to each location.
3. **Distance Calculation**:
   * **6371** is the Earth's radius in kilometers.
   * **RADIANS** function converts degrees to radians.
   * **COS**, **SIN**, and **ACOS** are trigonometric functions used in the Haversine formula.
4. **ORDER BY Clause**:
   * Orders the results by the calculated **Distance** in ascending order.
5. **FETCH FIRST 5 ROWS ONLY**:
   * Limits the result set to the top 5 closest locations.

**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.

**Orders Table**:

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

OrderDate DATE,

TotalAmount NUMBER

);

**Optimized SQL Query:**

SELECT

OrderID,

CustomerID,

OrderDate,

TotalAmount

FROM

Orders

WHERE

OrderDate >= TRUNC(SYSDATE) - 7

ORDER BY

OrderDate DESC;

**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.

**1. Creating the Table**

Let's create a table to store the dividend and divisor values.

CREATE TABLE division\_data (

id NUMBER PRIMARY KEY,

dividend NUMBER,

divisor NUMBER

);

INSERT INTO division\_data (id, dividend, divisor) VALUES (1, 100, 10);

INSERT INTO division\_data (id, dividend, divisor) VALUES (2, 100, 0); -- This will cause a division by zero error

INSERT INTO division\_data (id, dividend, divisor) VALUES (3, 100, 5);

COMMIT;

**2. PL/SQL Block with Exception Handling**

DECLARE

v\_dividend division\_data.dividend%TYPE;

v\_divisor division\_data.divisor%TYPE;

v\_result NUMBER;

BEGIN

SELECT dividend, divisor INTO v\_dividend, v\_divisor

FROM division\_data

WHERE id = &id;

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

DBMS\_OUTPUT.PUT\_LINE('Result: ' || v\_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;

3. Explanation of Error Handling Strategies

ZERO\_DIVIDE Exception: This exception is specifically handled to catch division by zero errors. When this exception is caught, a user-friendly error message is displayed: Error: Division by zero is not allowed..

OTHERS Exception: This is a generic exception handler that catches any other unexpected errors. It outputs a message with the error details using SQLERRM to help with debugging and understanding what went wrong: An unexpected error occurred: ' || SQLERRM.

**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.