**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.

Creating the Table ‘Company:

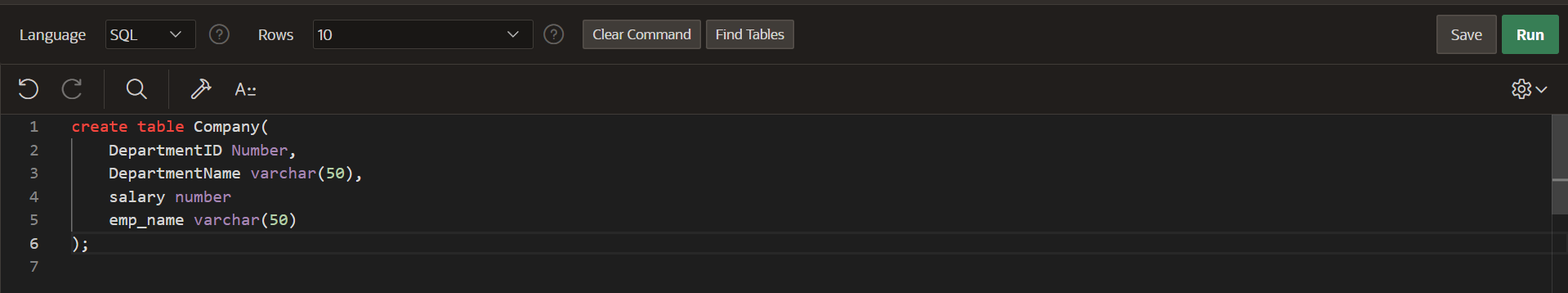
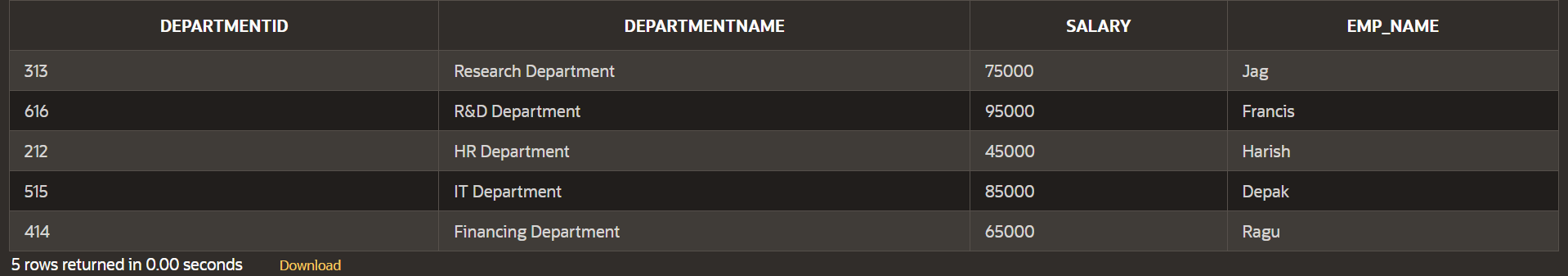


Table:



Answer:

Query:

SELECT

d.DEPARTMENTID,

d.DEPARTMENTNAME,

AVG(e.SALARY) AS AvgSalary

FROM

DEPARTMENT d

LEFT JOIN

EMPLOYEE e ON d.DEPARTMENTID = e.DEPARTMENTID

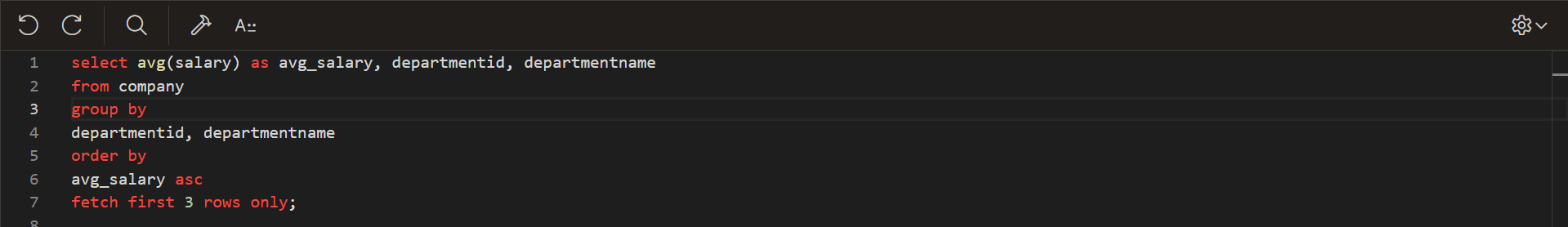
GROUP BY

d.DEPARTMENTID, d.DEPARTMENTNAME

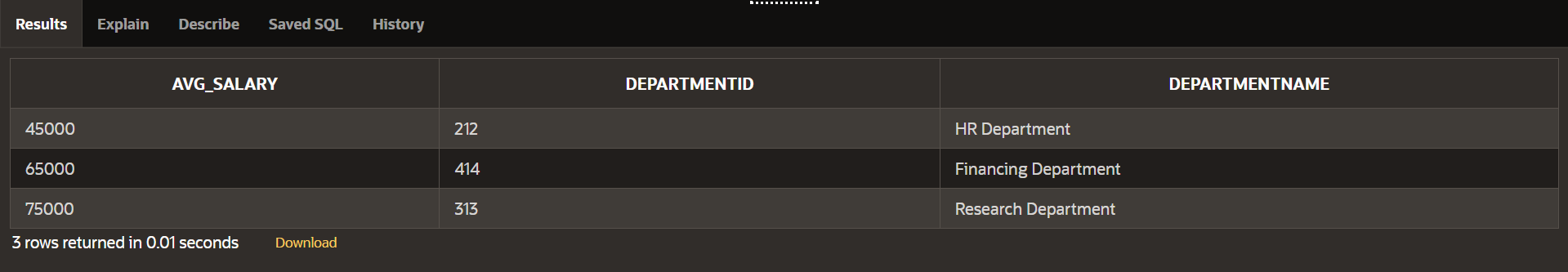
ORDER BY

AvgSalary DESC

FETCH FIRST 3 ROWS ONLY;



Result :



**Explanation :**

1. **SELECT Clause**:
   * d.DepartmentID and d.DepartmentName: These columns are selected from the Departments table.
   * AVG(e.Salary) AS AvgSalary: This calculates the average salary of employees in each department. The alias AvgSalary is given to this calculated field.
2. **FROM Clause**:
   * Departments d: This specifies the Departments table and gives it the alias d.
3. **LEFT JOIN Clause**:
   * LEFT JOIN Employees e ON d.DepartmentID = e.DepartmentID: This joins the Departments table with the Employees table based on the DepartmentID column. A LEFT JOIN is used, meaning all rows from the Departments table will be included in the result, even if there are no matching rows in the Employees table.
4. **GROUP BY Clause**:
   * d.DepartmentID, d.DepartmentName: This groups the results by department ID and department name, ensuring that the average salary is calculated for each department.
5. **ORDER BY Clause**:
   * AvgSalary DESC NULLS LAST: This orders the results by the average salary in descending order. The NULLS LAST part ensures that departments with no employees (and therefore a NULL average salary) are placed at the end of the list.
6. **FETCH FIRST 3 ROWS ONLY Clause**:
   * This limits the result set to the top 3 departments based on the sorted average salary.

**Handling Departments with No Employees**

The query uses a LEFT JOIN to include all departments, even those without any employees. Here's how it handles these cases:

* **LEFT JOIN**: When there are no employees in a department, the join results in NULL values for the columns from the Employees table. Consequently, the e.Salary column will be NULL.
* **AVG(e.Salary)**: The AVG function ignores NULL values when calculating the average. If there are no salaries to average (i.e., all e.Salary values are NULL for a department), the result of AVG(e.Salary) will be NULL.
* **ORDER BY AvgSalary DESC NULLS LAST**: Departments with a NULL average salary (i.e., no employees) are placed last in the order.
* **FETCH FIRST 3 ROWS ONLY**: Only the top 3 departments with the highest average salary (excluding departments with NULL average salary) are fetched.

**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).

Creating Table:

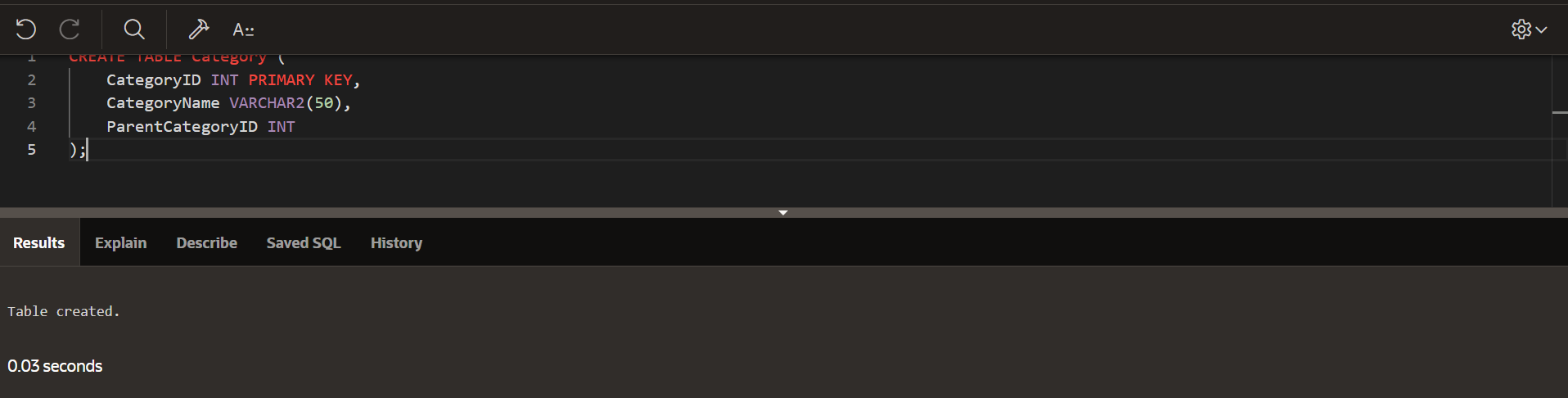
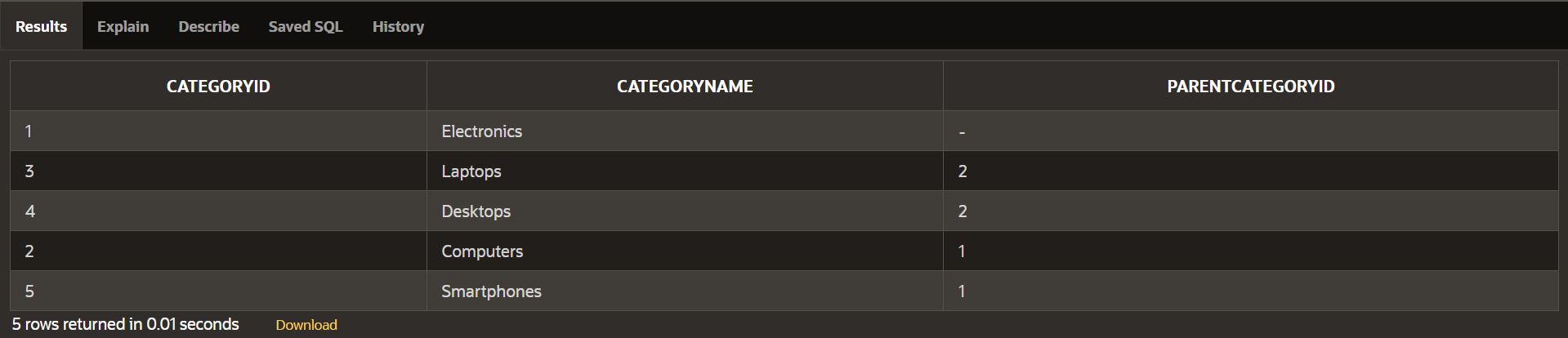


Table:



Answers:

Query:

SELECT

CategoryID,

CategoryName,

SYS\_CONNECT\_BY\_PATH(CategoryName, ' -> ') AS HierarchicalPath

FROM

Category

START WITH

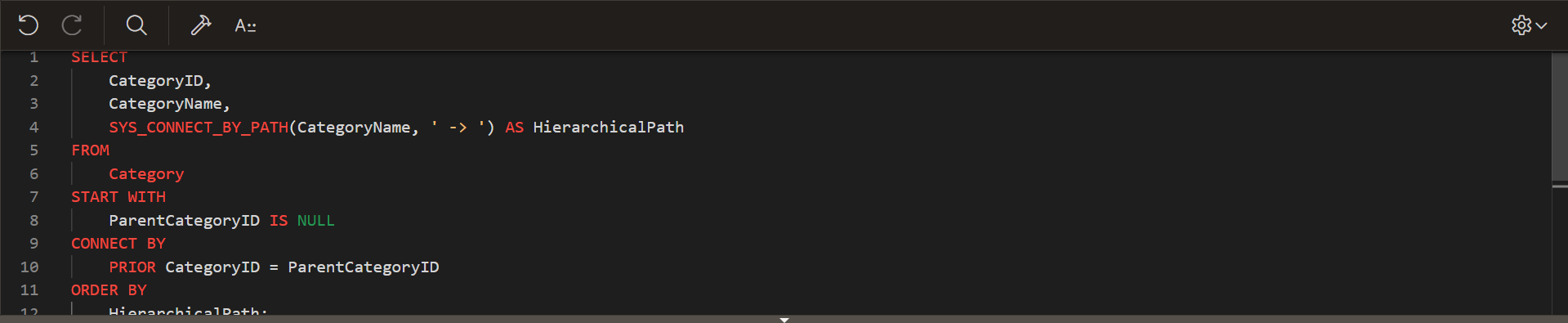
ParentCategoryID IS NULL

CONNECT BY

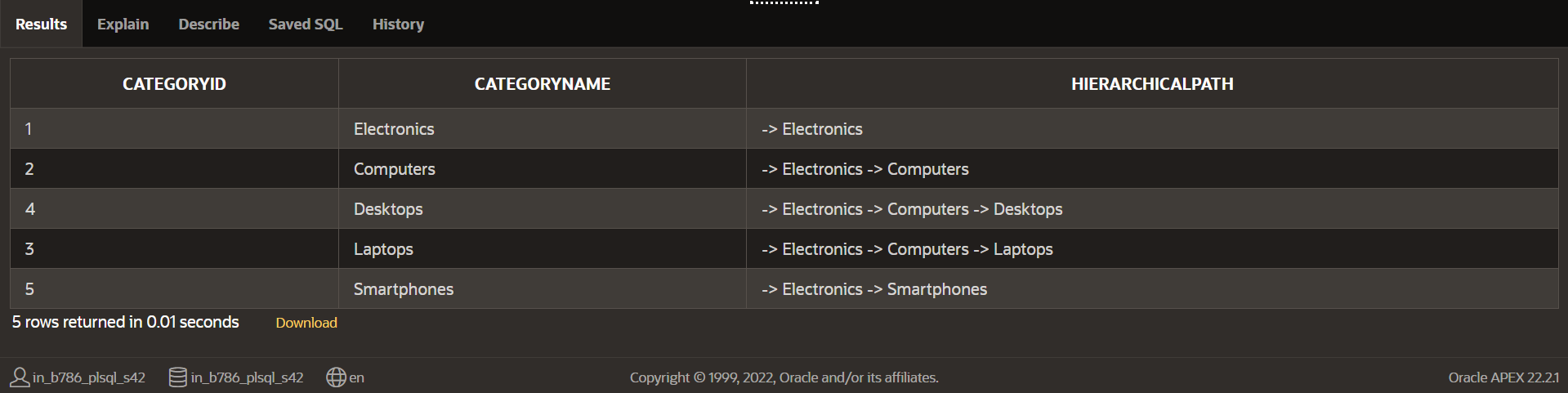
PRIOR CategoryID = ParentCategoryID

ORDER BY

HierarchicalPath;



Result :



Explanation :

**Explanation**

1. **Hierarchical Query Clauses**:
   * START WITH ParentCategoryID IS NULL specifies the root categories where ParentCategoryID is NULL.
   * CONNECT BY PRIOR CategoryID = ParentCategoryID creates the hierarchical relationship between parent and child categories.
2. **Hierarchical Path**:
   * SYS\_CONNECT\_BY\_PATH(CategoryName, ' -> ') constructs the hierarchical path string by concatenating CategoryName values, using -> as a separator.
3. **Ordering**:
   * ORDER BY HierarchicalPath sorts the results based on the constructed hierarchical 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.

Creating the table:

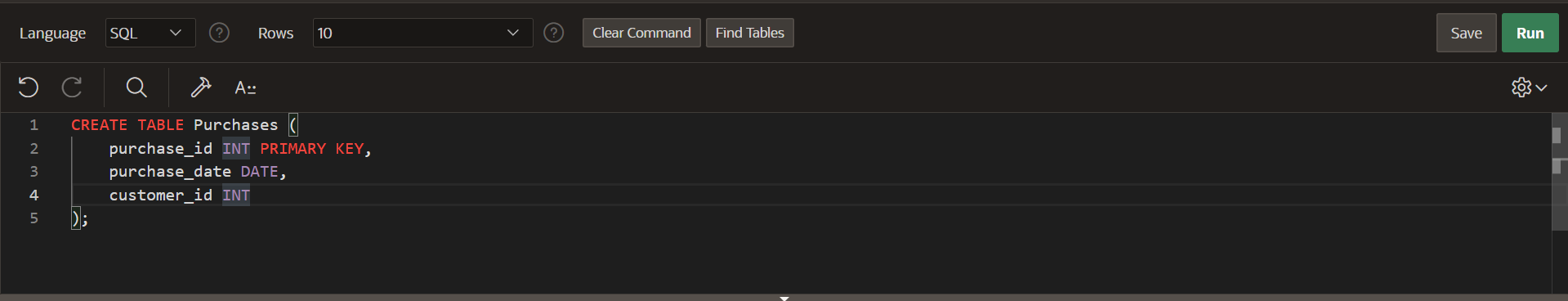
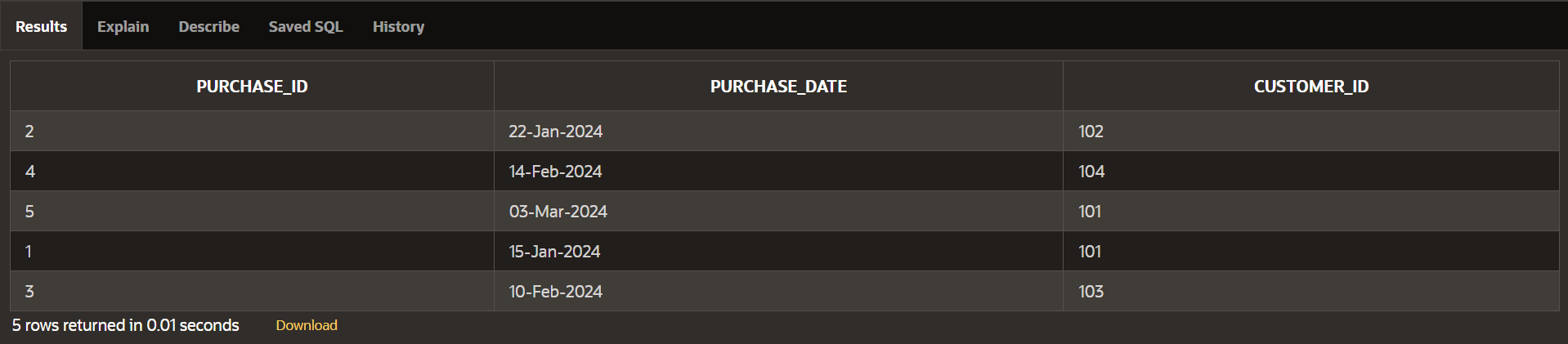
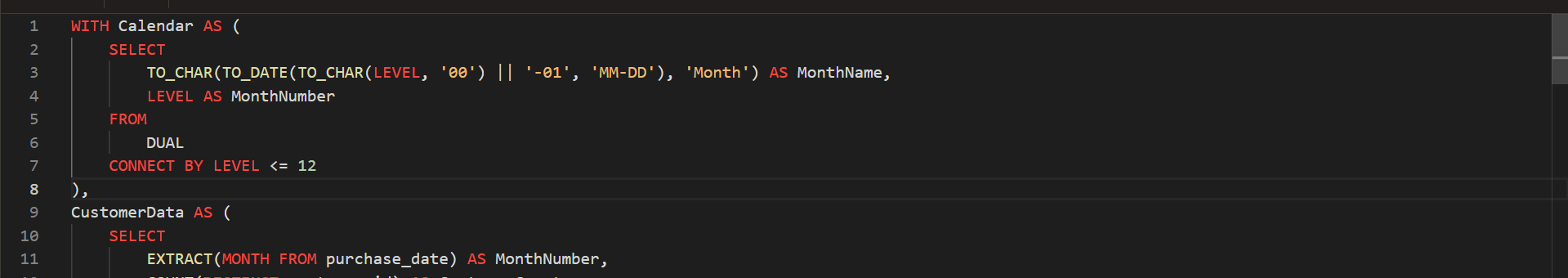


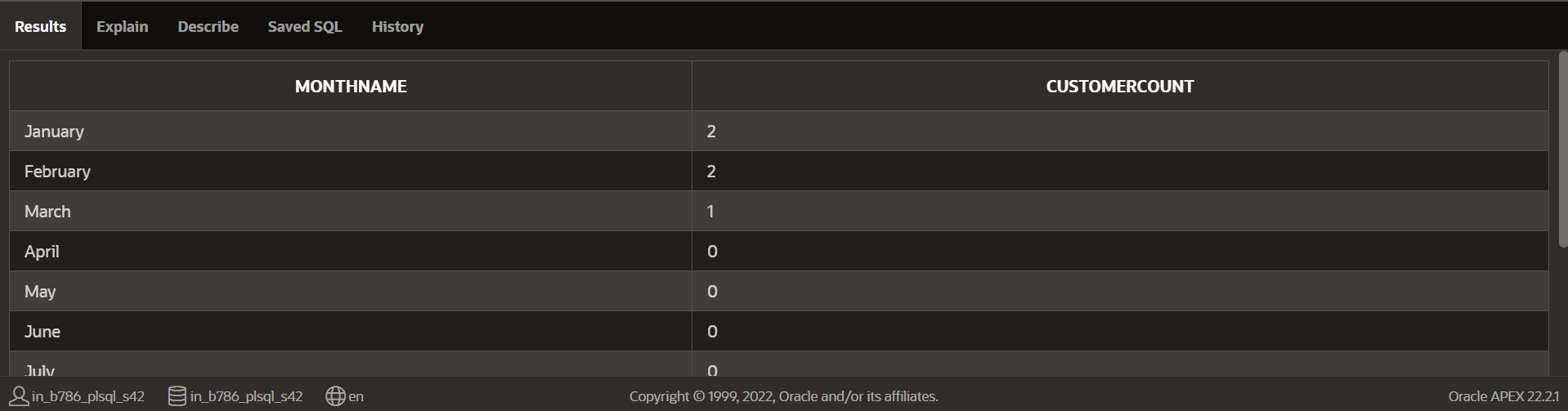
Table:



Query :



Results :



**Explanation**

1. **Calendar Table**: The Calendar CTE (Common Table Expression) generates a list of all months in the current year using DUAL and CONNECT BY LEVEL. The TO\_DATE and TO\_CHAR functions convert the level (1 to 12) into month names.
2. **Customer Data Aggregation**: The CustomerData CTE aggregates the purchase data by extracting the month from the purchase\_date and counting distinct customer\_ids for each month in the current year.
3. **Join and Handle Zero Counts**: The main query performs a LEFT JOIN between the Calendar and CustomerData CTEs. This ensures that every month from the calendar table is included, and if there are no purchases for a particular month, the CustomerCount is set to 0 using the NVL function (which replaces NULL with 0).

**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.

Creating a table :

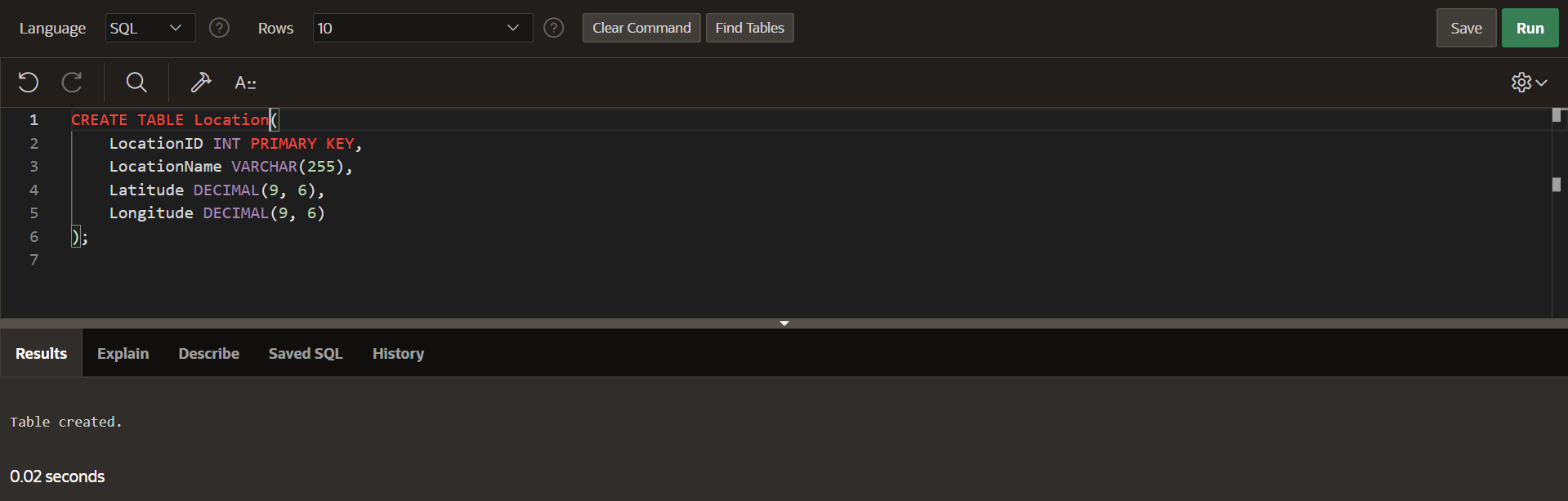
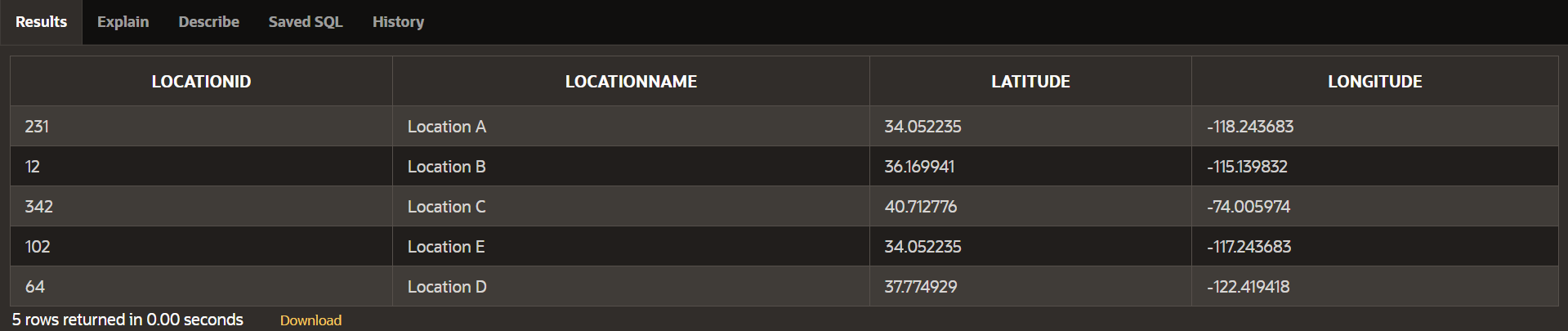
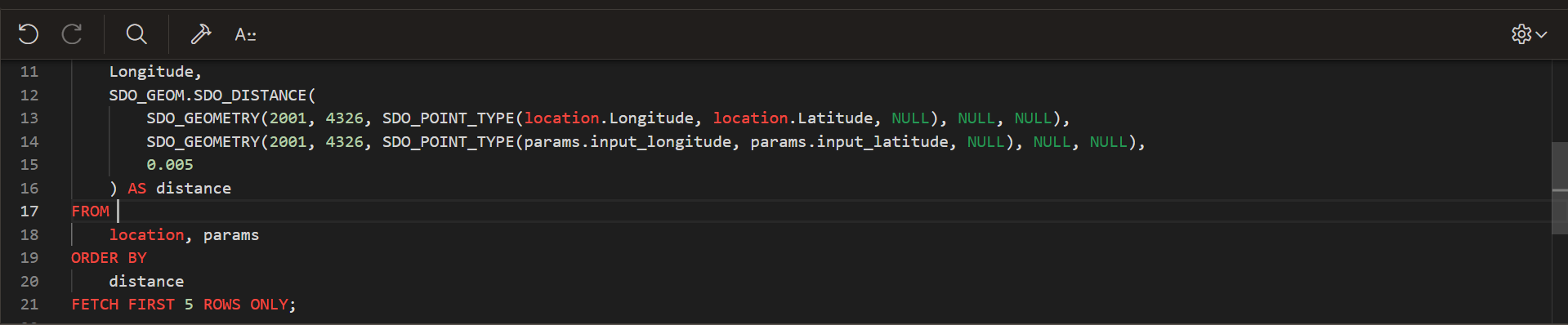
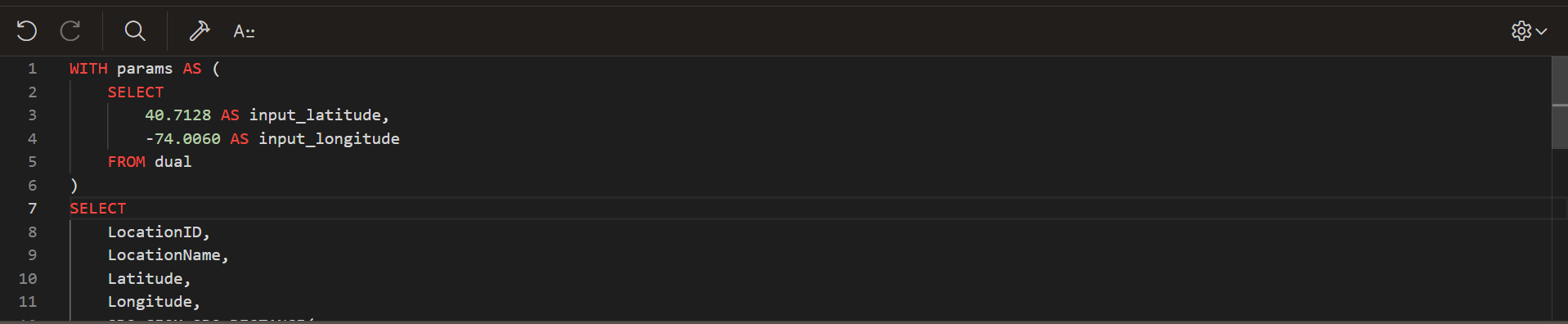


Table :

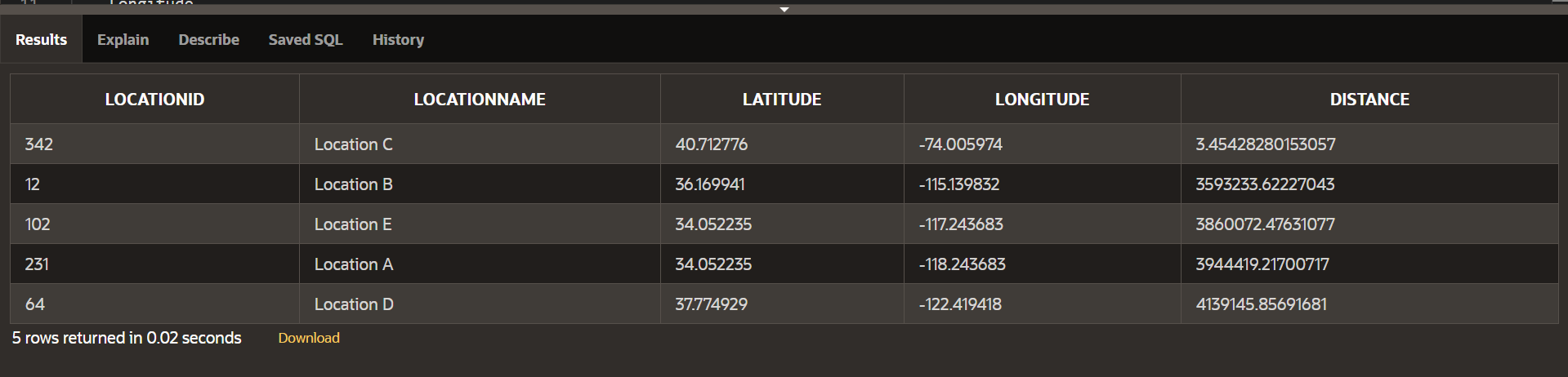


Answer :

Query:



Result:



**Explanation of the Approach**

1. **GivenPoint CTE**: This common table expression defines the given point's latitude and longitude.
2. **Haversine Formula**: The DistanceCalculation CTE calculates the great-circle distance between the given point and each location using the Haversine formula. The formula used is: distance=3959×arccos(cos(radians(lat1))×cos(radians(lat2))×cos(radians(lon2)−radians(lon1))+sin(radians(lat1))×sin(radians(lat2)))
3. **Order and Limit**: The final query orders the results by distance and limits the output to the closest 5 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.

Creating a Table :

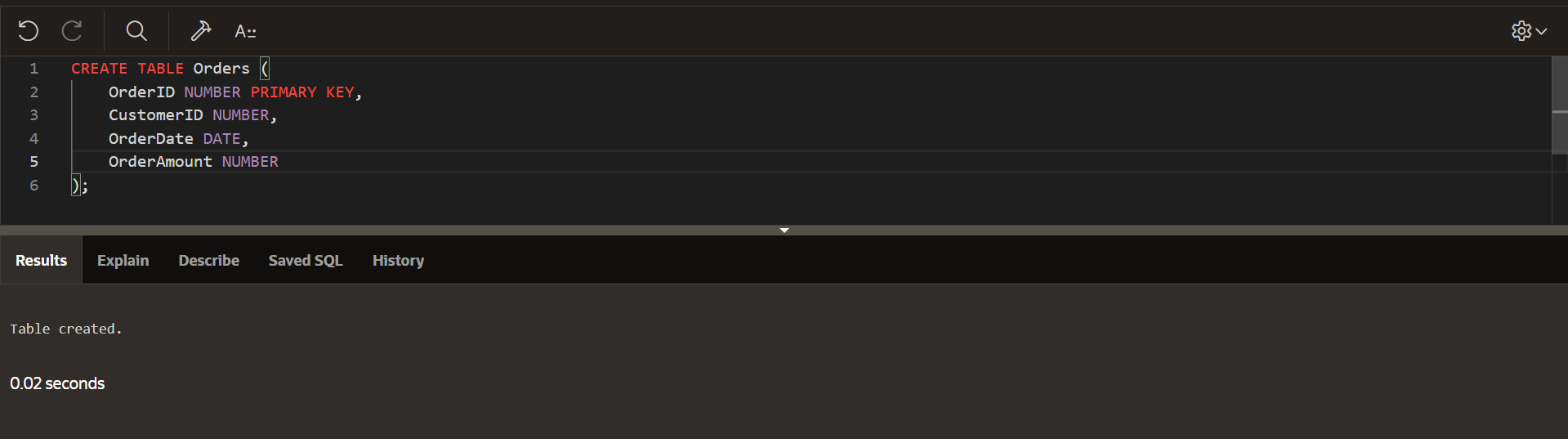
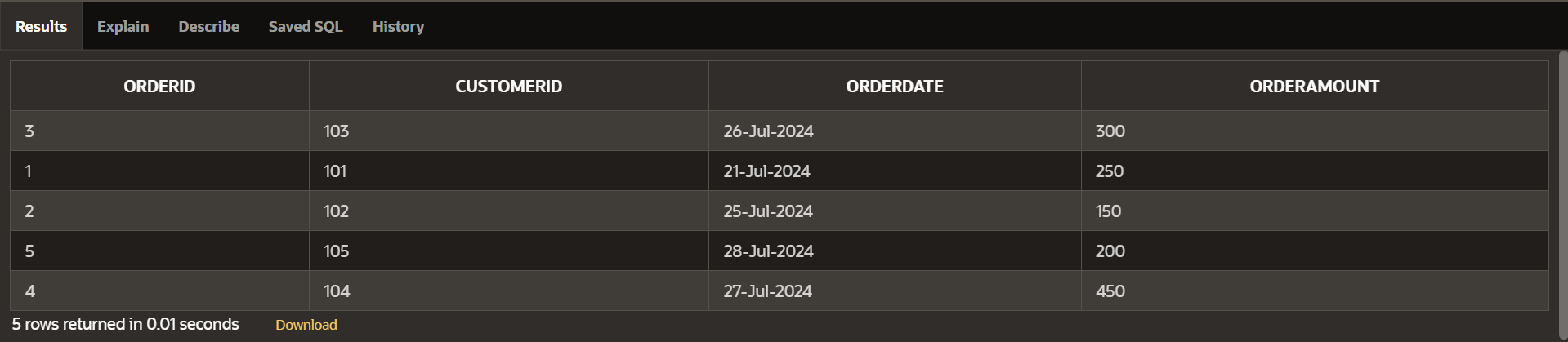
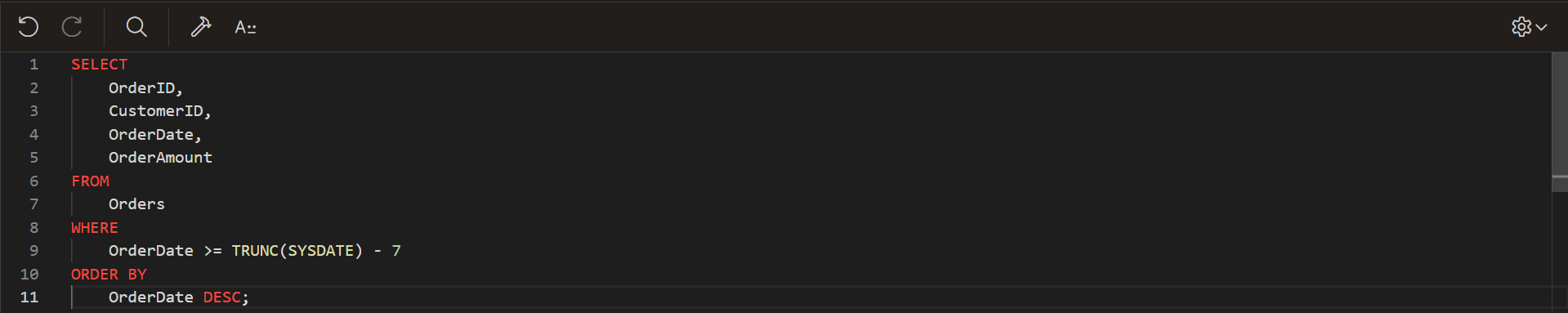


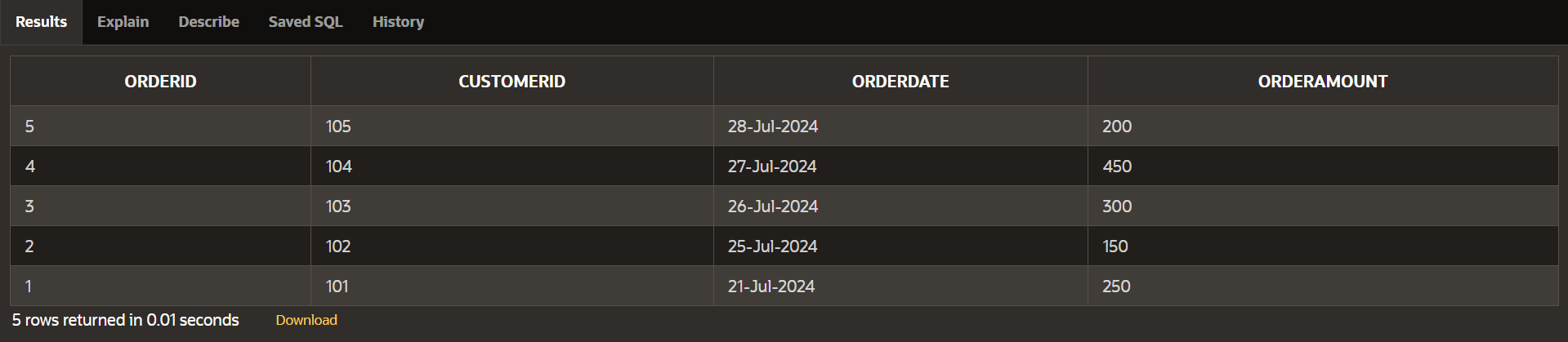
Table :



Query :



Result :



**Explanation**

1. **Indexing**:
   * Create an index on the OrderDate column to speed up retrieval of recent orders. This is crucial for performance when filtering and sorting by OrderDate.
2. **Date Calculation**:
   * Use TRUNC(SYSDATE) - 7 to calculate the date 7 days ago. TRUNC(SYSDATE) ensures that the comparison is done at the start of the day, which can help with consistency in date comparisons.
3. **Order By Clause**:
   * Sort by OrderDate DESC to get the most recent orders first. Since the index is on OrderDate, sorting will be more efficient.

**Additional Optimization Techniques**

1. **Partitioning**:
   * If the Orders table is very large, consider partitioning the table by date. This can improve query performance by allowing the database to scan only relevant partitions.
2. **Query Hints**:
   * In some databases, you can use query hints to guide the optimizer. For example, in Oracle, you might use /\*+ INDEX(Orders idx\_order\_date) \*/ to ensure the index is used.
3. **Materialized Views**:
   * For frequently accessed queries, consider using materialized views to store precomputed results. This can reduce the load on the database for repeated queries.
4. **Caching**:
   * Implement caching strategies for the results of this query if the data does not change frequently. This can significantly reduce the load on the database.
5. **Monitoring and Analyzing Execution Plans**:
   * Use database tools to analyze the execution plan of your query. Look for full table scans or other inefficiencies and adjust indexing or rewriting the query accordingly.