| Experiment No.7 |
| --- |
| Nested Queries & Complex Queries |
| Date of Performance: 19/3/24 |
| Date of Submission: 26/3/24 |

**Experiment No. 7: Nested queries and Complex queries**

**Course Outcome [CSL503.3]:** Apply SQL queries ,triggers and procedures for specific module/task

**Aim**: Nested queries and Complex queries

**Theory:**

Nested queries, also known as subqueries, are queries embedded within another query, enabling more complex and context-dependent data retrieval or manipulation. Here's a brief explanation of the theory behind nested queries:

1. \*\*Subquery Purpose\*\*: Subqueries are used to perform a query within another query, allowing for dynamic filtering, calculations, or comparisons based on the results of the inner query.

2. \*\*Syntax\*\*: In SQL, a subquery is enclosed within parentheses and placed within the WHERE clause, HAVING clause, or FROM clause of an outer query. The result of the subquery is then used as a condition or value in the outer query.

3. \*\*Types of Subqueries\*\*:

- \*Correlated Subqueries\*: These subqueries are executed for each row processed by the outer query and can reference columns from the outer query. They're typically used for row-wise comparisons or filtering.

- \*Non-correlated Subqueries\*: These subqueries are independent of the outer query and execute once, returning a single result set. They're often used to retrieve values or conditions for the outer query.

4. \*\*Usage Scenarios\*\*:

- \*Filtering\*: Subqueries can filter the results of the outer query based on conditions that involve other tables or aggregated data.

- \*Aggregation\*: Subqueries can perform calculations or aggregate functions on subsets of data before being used in the outer query.

- \*Existence Checks\*: Subqueries can check for the existence of certain conditions or values within a dataset.

- \*Comparisons\*: Subqueries enable comparisons between values in different tables or based on calculated values.

5. \*\*Performance Considerations\*\*: While powerful, nested queries can impact performance, especially if the subquery is complex or executed repeatedly. Optimizing queries and ensuring appropriate indexing can mitigate performance issues.

**Code:**

CREATE TABLE Dept (

D\_id INT(10),

D\_name CHAR(50),

PRIMARY KEY (D\_id)

);

CREATE TABLE Emp (

role CHAR(30),

e\_id INT(10),

e\_name CHAR(50),

d\_id INT(10),

sal int(20),

PRIMARY KEY (e\_id),

FOREIGN KEY (d\_id) REFERENCES Dept(D\_id)

);

CREATE TABLE Sales (

S\_id INT(10),

S\_amt INT(30),

e\_id INT(10), -- Added the e\_id column

FOREIGN KEY (e\_id) REFERENCES Emp(e\_id)

);

CREATE TABLE Awards (

id INT(10),

e\_id INT(10), -- Added the e\_id column

FOREIGN KEY (e\_id) REFERENCES Emp(e\_id)

);

INSERT INTO Dept (D\_id, D\_name) VALUES

(1, 'Sales'),

(2, 'Marketing'),

(3, 'Finance');

select \*from Emp;

INSERT INTO Emp (role, e\_id, e\_name, d\_id, sal) VALUES

('Developer', 1, 'John', 1, 10000),

('Manager', 2, 'Mary', 2, 10000),

('Developer', 3, 'Bob', 1, 30000),

('Manager', 4, 'Alice', 3, 40000),

('Developer', 5, 'Tom', 1, 50000);

select \*from Emp;

INSERT INTO Sales (S\_id, S\_amt, e\_id) VALUES

(1, 1000, 1),

(2, 2000, 2),

(3, 3000, 3),

(4, 4000, 1),

(5, 5000, 5),

(6, 6000, 3),

(7, 7000, 2);

select \*from Sales;

INSERT INTO Awards (id, e\_id) VALUES

(1, 1),

(2, 3);

select \*from Awards;

-- Find the names of all the employees in sales department

select e\_name from Emp where d\_id IN (select (d\_id) from Dept where D\_name = "Sales");

-- select name of all employee who have sale amount over 4000

select e\_name, e\_id from Emp where e\_id IN (select e\_id from Sales where S\_amt > 4000);

-- find all the employee names who are handled by finance

select e\_name from Emp where d\_id = (select D\_id from Dept where D\_name = "Finance");

-- find all the employee names who have received awards

select e\_name from Emp where e\_id IN (select e\_id from Awards);

-- find all the employee names who have never received awards

select e\_name from Emp where e\_id not IN (select e\_id from Awards);

-- Sekect all developers who earn more than all managers

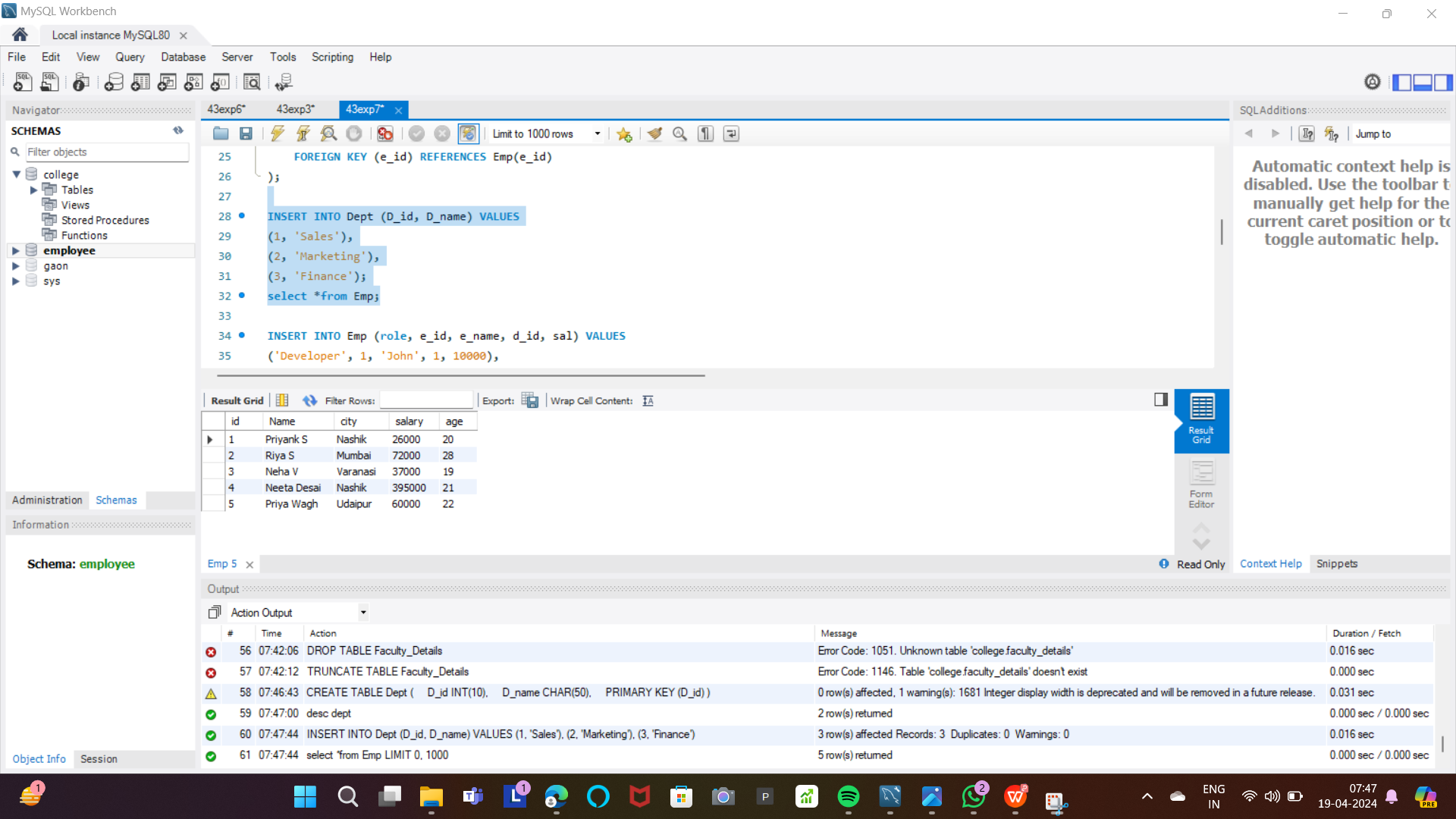
select \* from Emp where role = "Developer" and Sal > all (select Sal from Emp where role = "Manager");

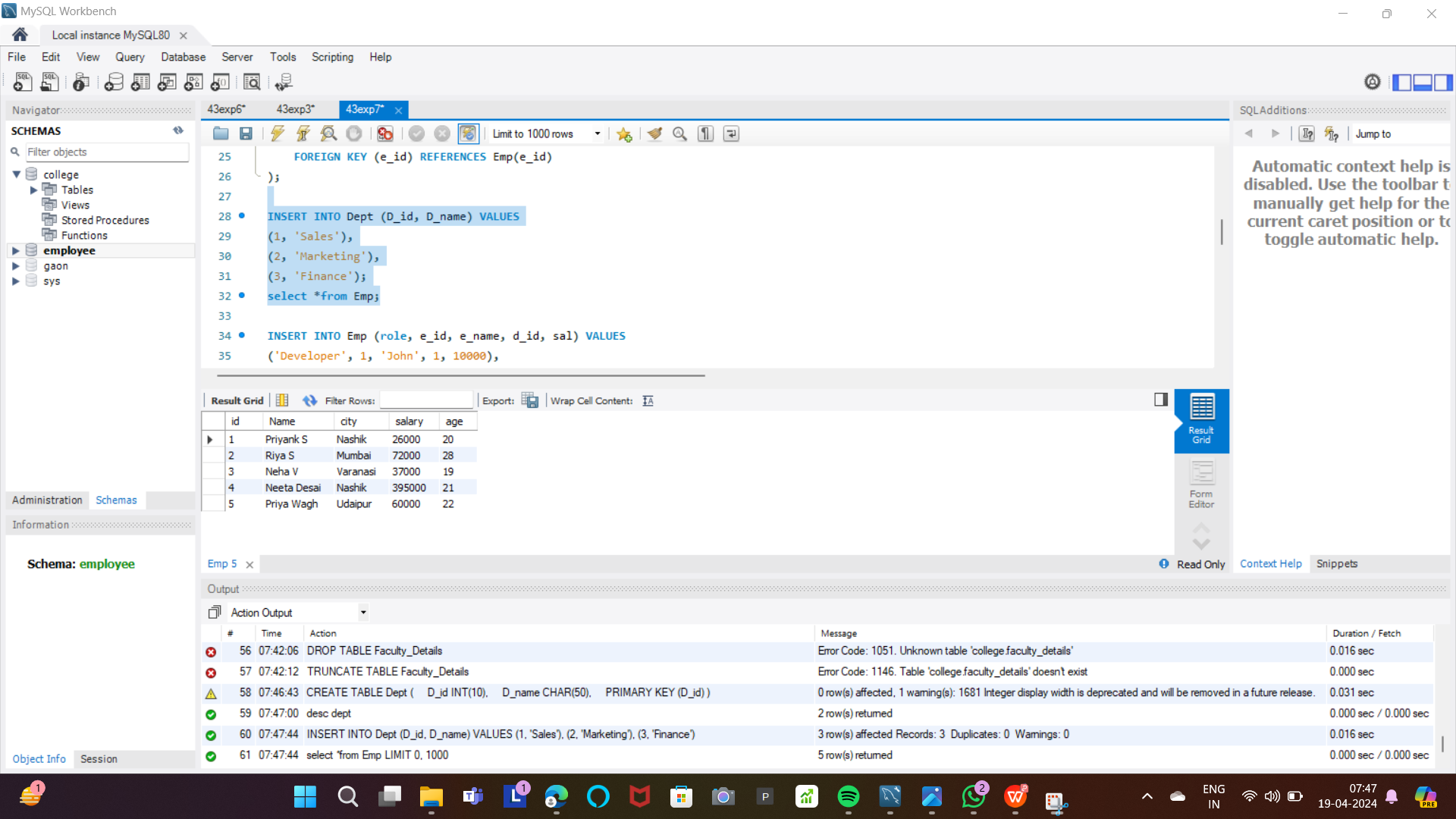
-- Sekect all developers who earn more than any managers

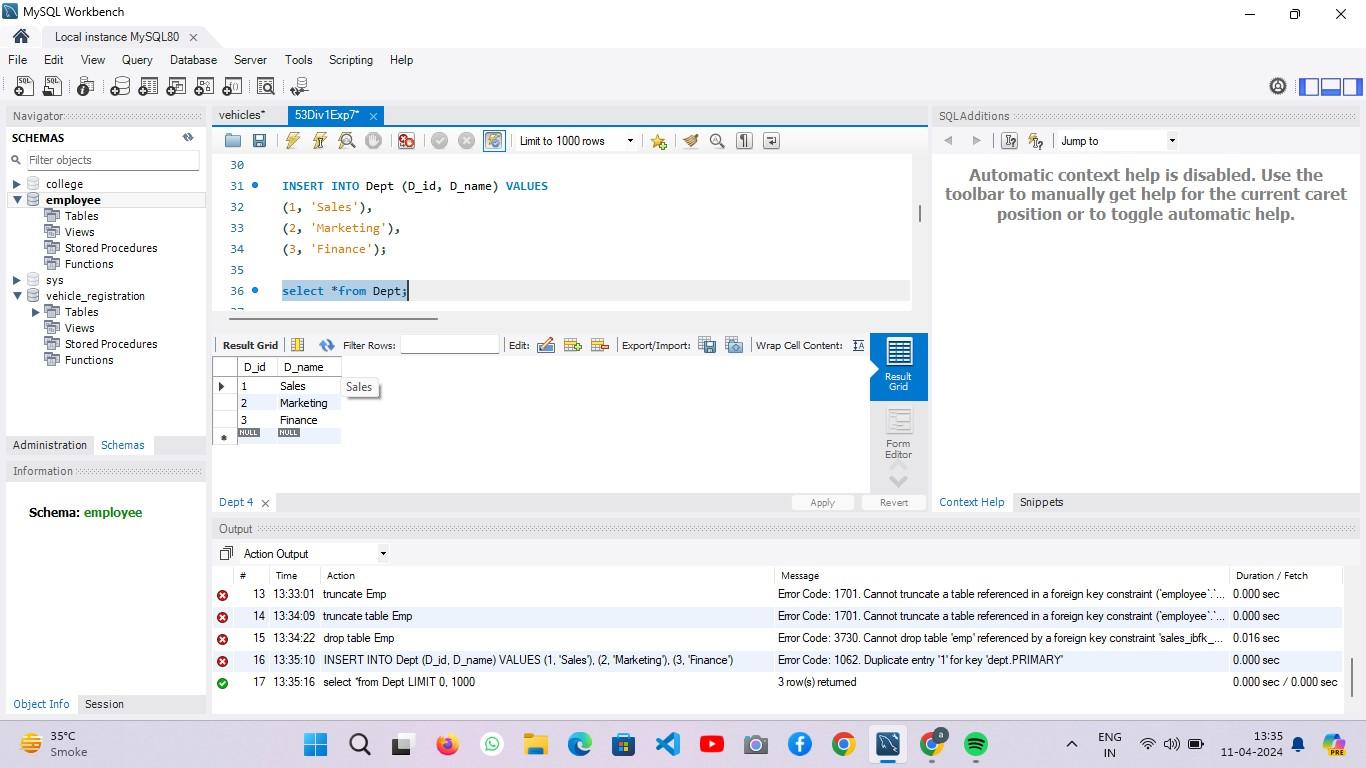
select \* from Emp where role = "Developer" and Sal > any (select Sal from Emp where role = "Manager")

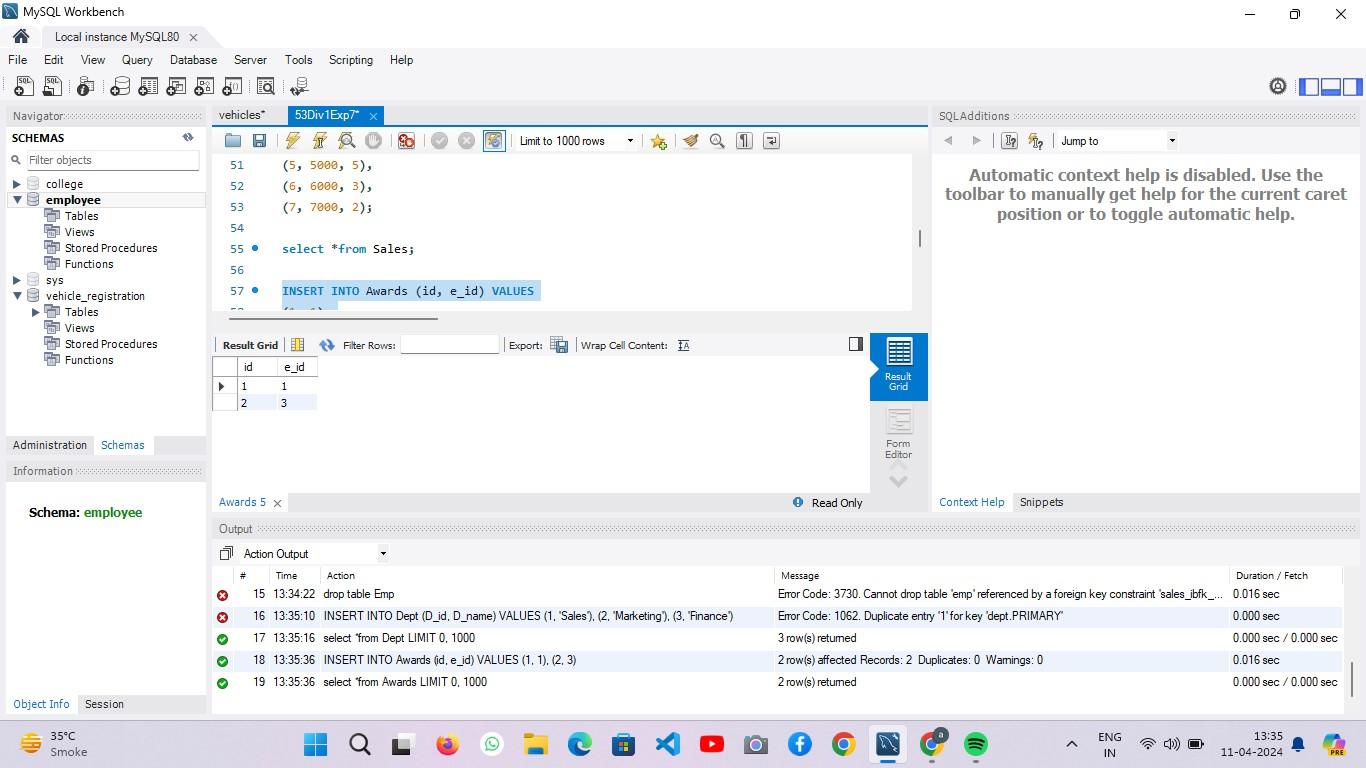
**Output:**

**Create and insert values of the all tables :**

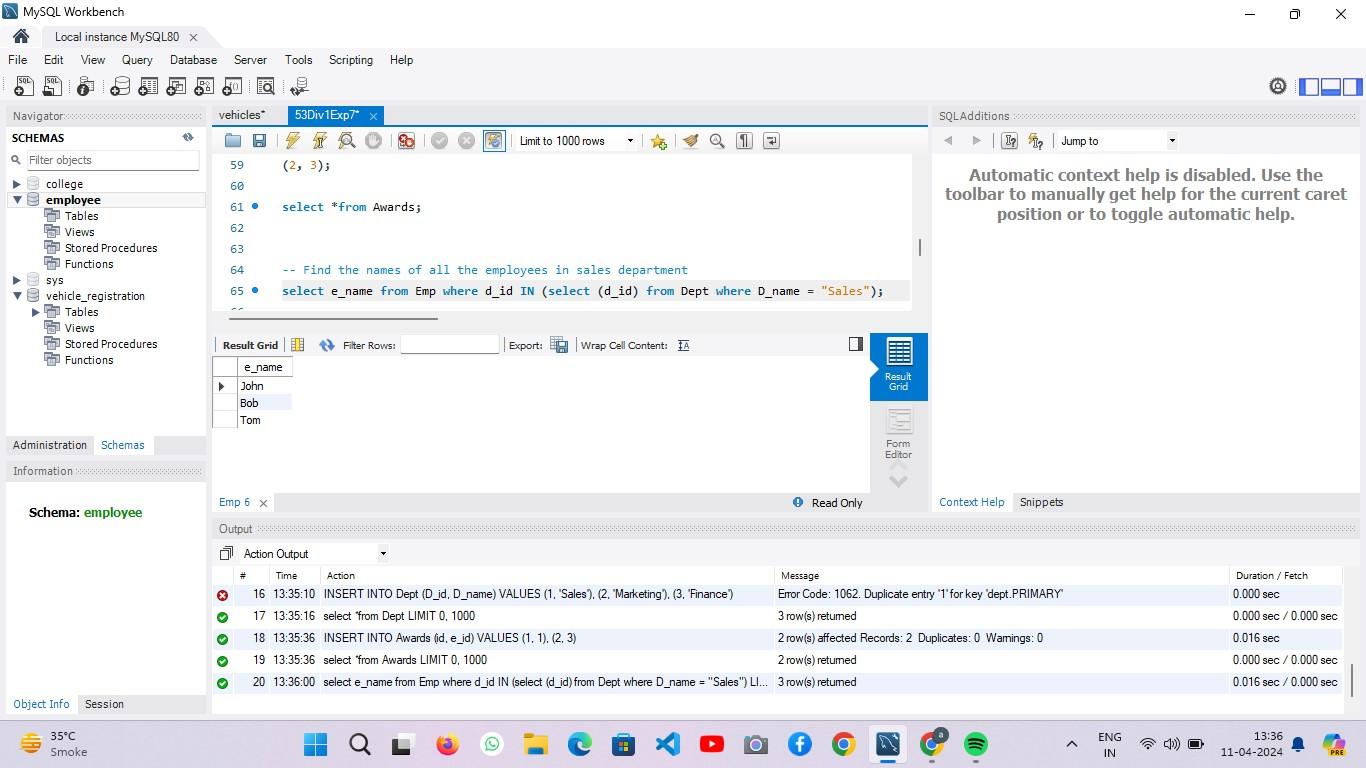




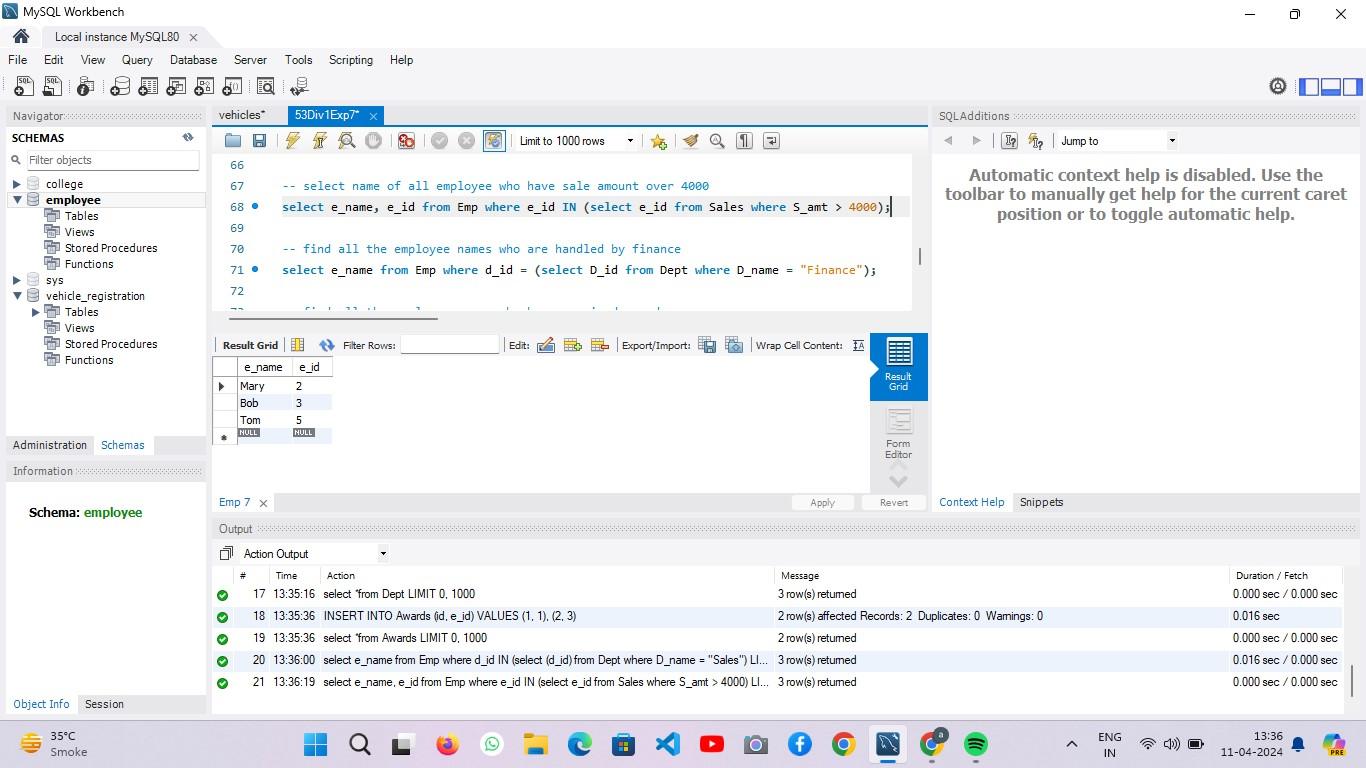




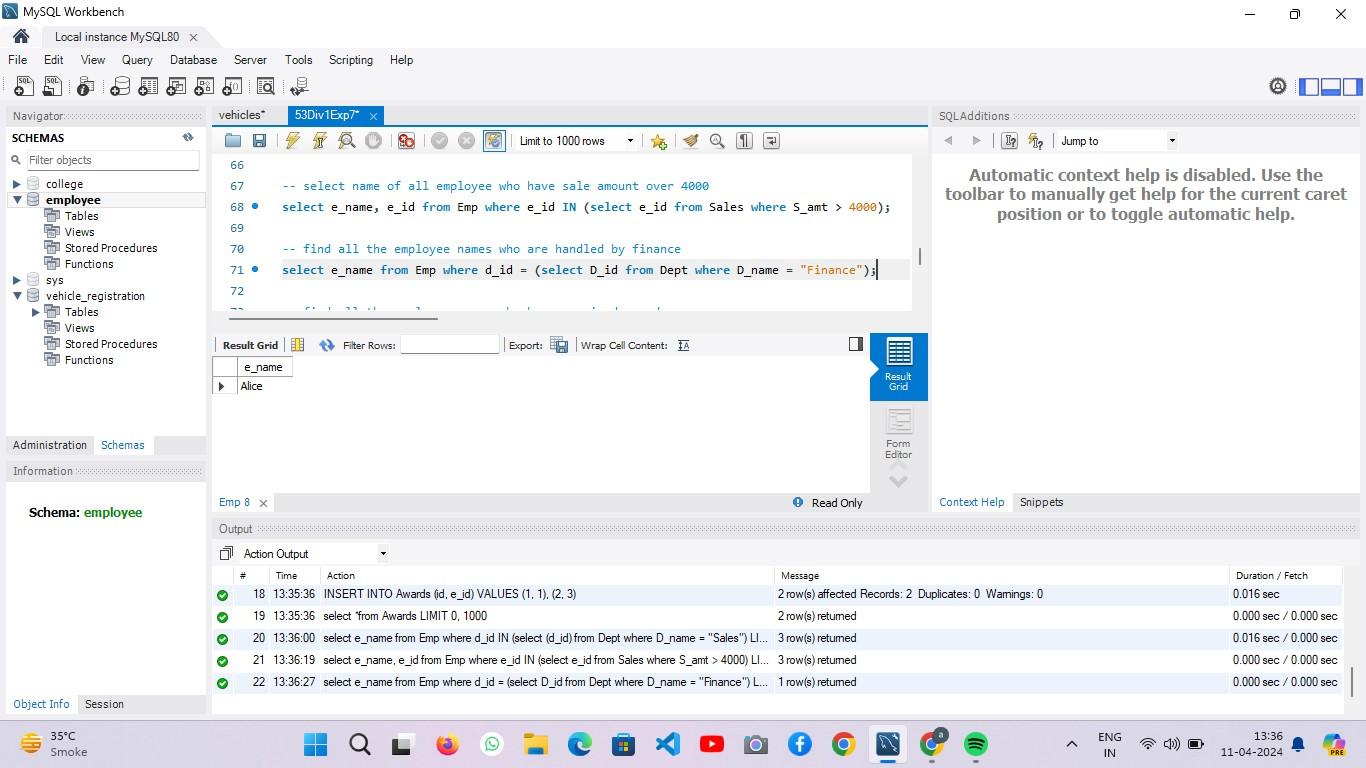
1. Find the names of all the employees in the sales department.



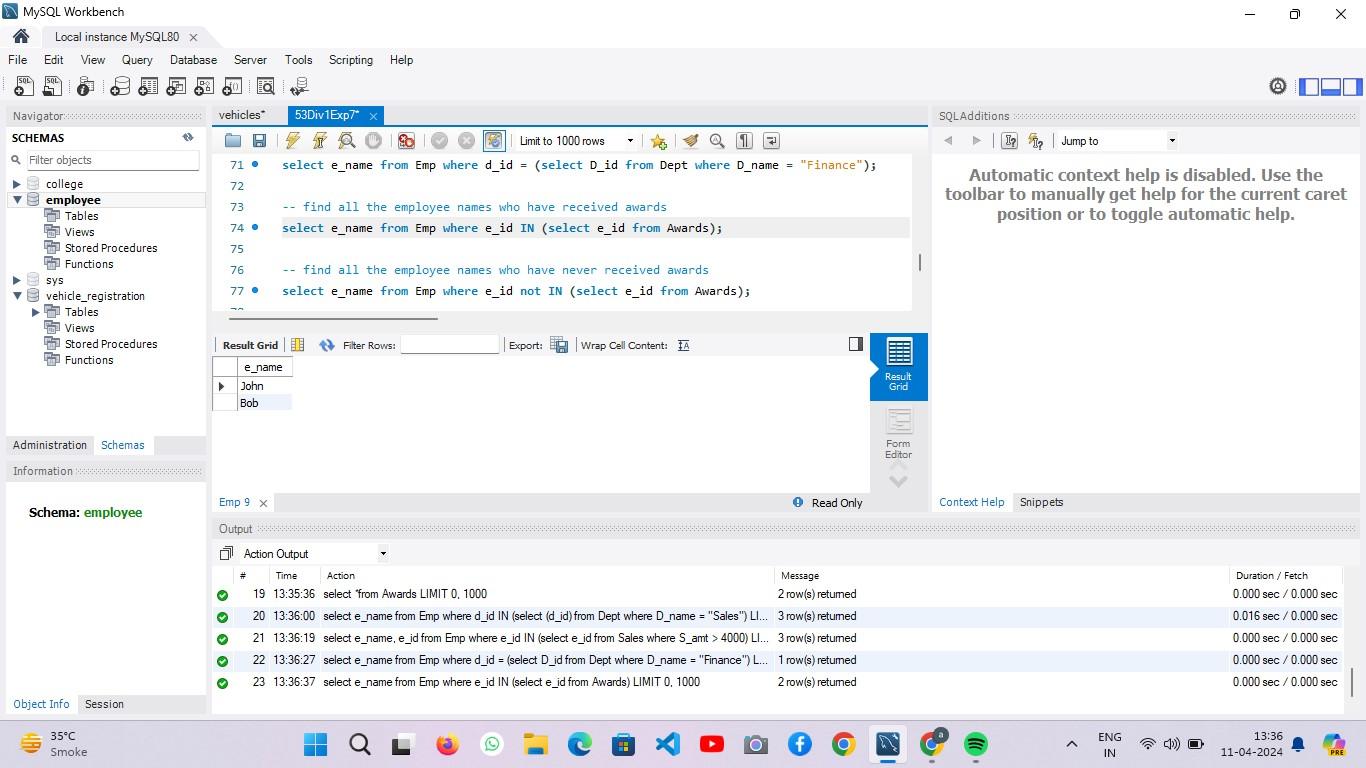
2. Select the name of all employees who have a sales amount over 4000.



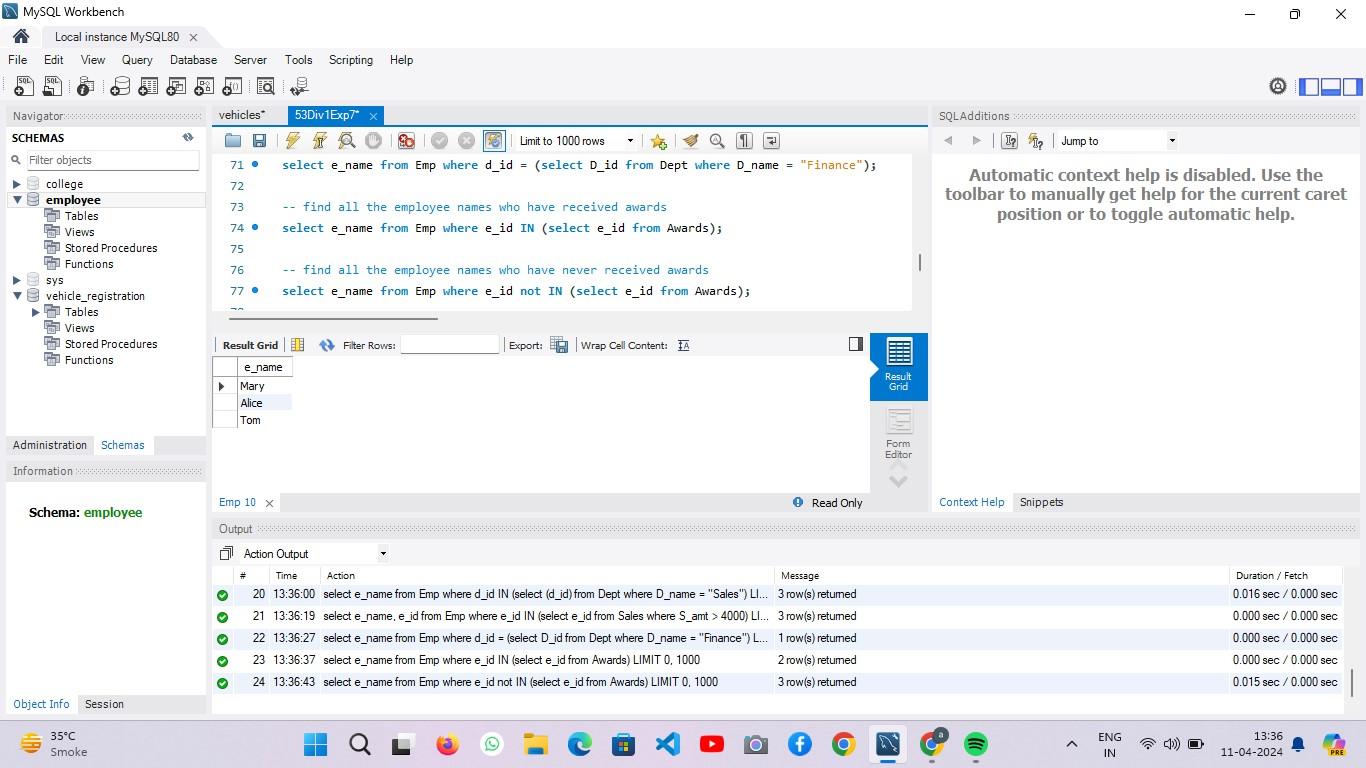
3. Find all the employee names who are handled by the finance department.



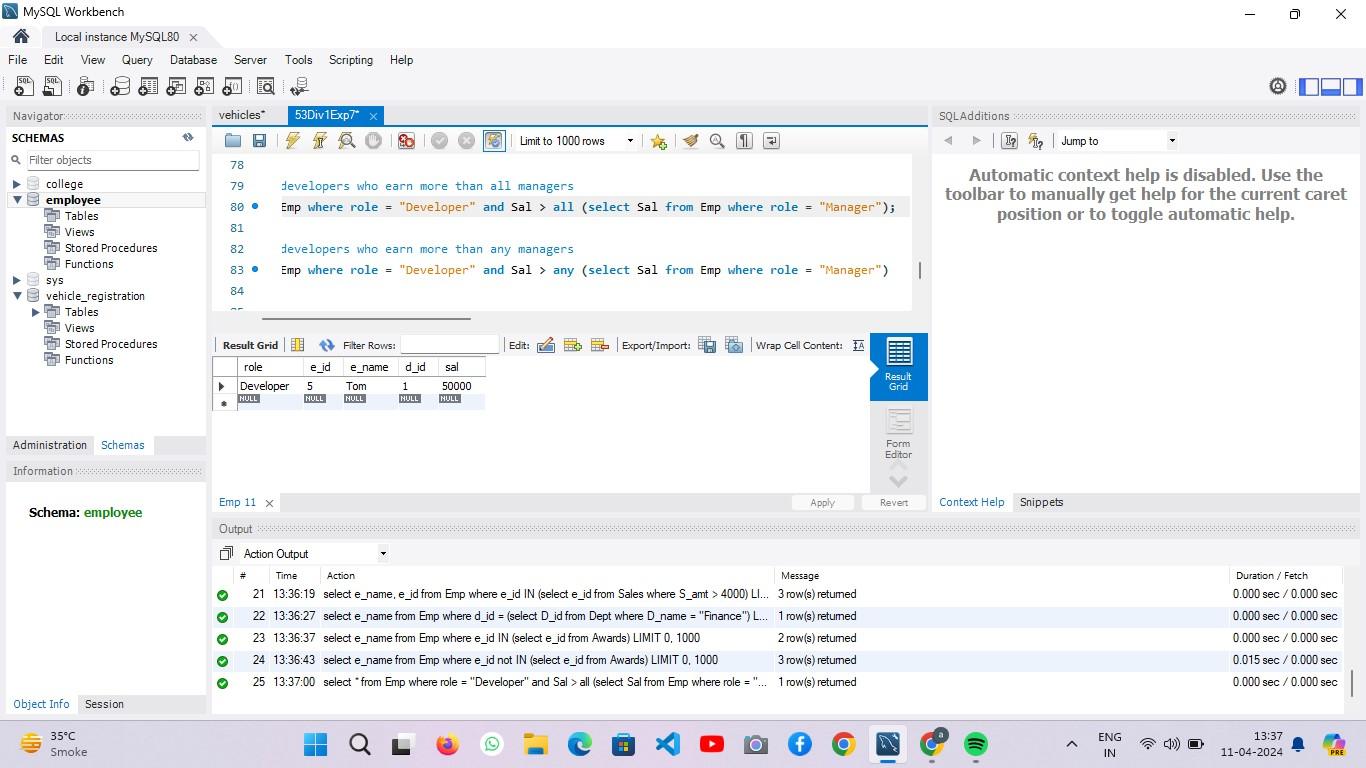
4. Retrieve the names of all employees who have received awards.

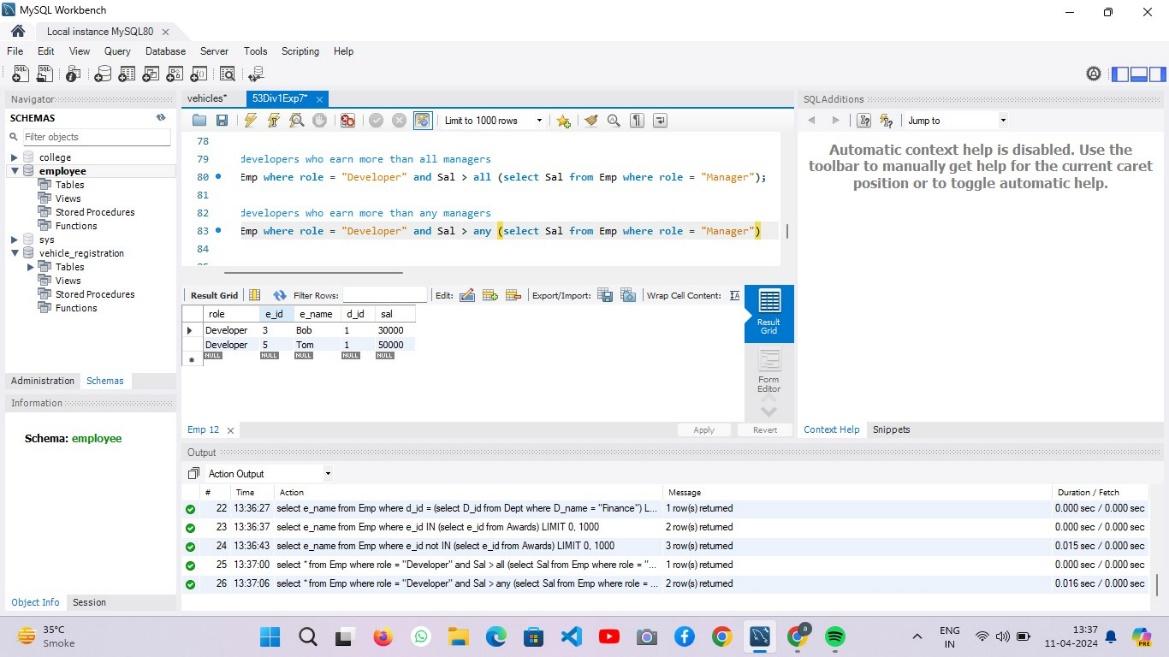


5. Get the names of all employees who have never received awards.



6. Select all developers who earn more than all managers.



7. Select all developers who earn more than any managers.

**CONCLUSION:**

After exploring nested queries and complex SQL queries, it's evident that they provide significant flexibility and power in data retrieval and manipulation tasks. Understanding how to leverage subqueries effectively enhances the ability to extract specific data subsets, perform complex calculations, and filter results based on dynamic conditions. However, it's crucial to consider performance implications and optimize queries accordingly to ensure efficient execution, highlighting the importance of balancing complexity with performance considerations in database management tasks.