

Batch: B2 Roll No.: 16010122151

Experiment / assignment / tutorial No. 4

Grade: AA / AB / BB / BC / CC / CD /DD

Title: : DML – select, insert, update and delete

- 1.Group by, having clause, aggregate functions, Set Operations
- 2.Nested queries : AND,OR,NOT, IN, NOT IN, Exists, Not Exists, Between, Like, Alias, ANY,ALL,DISTINCT
3. Update
4. Delete

Objective: To perform various DML Operations and executing nested queries with various clauses.

Expected Outcome of Experiment:

CO 3: Use SQL for Relational database creation, maintenance and query processing

Books/ Journals/ Websites referred:

1. Dr. P.S. Deshpande, SQL and PL/SQL for Oracle 10g.Black book, Dreamtech Press
2. www.db-book.com
3. Korth, Silberchatz, Sudarshan : “Database Systems Concept”, 5th Edition , McGraw Hill
4. Elmasri and Navathe,”Fundamentals of database Systems”, 4th Edition PEARSON Education

Resources used: Postgres

Theory:

Select: The SQL **SELECT** statement is used to fetch the data from a database table which returns this data in the form of a result table. These result tables are called result-sets.

Syntax

The basic syntax of the SELECT statement is as follows –

SELECT column1, column2, columnN FROM table_name;

Here, column1, column2... are the fields of a table whose values you want to fetch. If you want to fetch all the fields available in the field, then you can use the following syntax.

```
SELECT * FROM table_name;
```

The following code is an example, which would fetch the ID, Name and Salary fields of the customers available in CUSTOMERS table.

```
SQL> SELECT ID, NAME, SALARY FROM CUSTOMERS;
```

Insert: The SQL **INSERT INTO** Statement is used to add new rows of data to a table in the database.

Syntax

There are two basic syntaxes of the INSERT INTO statement which are shown below.

```
INSERT INTO TABLE_NAME (column1, column2, column3,...columnN)
```

```
VALUES (value1, value2, value3,...valueN);
```

Example

The following statements would create record in the CUSTOMERS table.

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );
```

Update: The SQL **UPDATE** Query is used to modify the existing records in a table. You can use the WHERE clause with the UPDATE query to update the selected rows, otherwise all the rows would be affected.

Syntax:

The basic syntax of the UPDATE query with a WHERE clause is as follows –

```
UPDATE table_name
```

```
SET column1 = value1, column2 = value2. , columnN = valueN
```

```
WHERE [condition];
```

You can combine N number of conditions using the AND or the OR operators.

The following query will update the ADDRESS for a customer whose ID number is 6 in the table.

```
SQL> UPDATE CUSTOMERS
SET ADDRESS = 'Pune'
WHERE ID = 6;
```

Delete: The SQL DELETE Query is used to delete the existing records from a table.

You can use the WHERE clause with a DELETE query to delete the selected rows, otherwise all the records would be deleted.

Syntax

The basic syntax of the DELETE query with the WHERE clause is as follows –

```
DELETE FROM table_name
```

```
WHERE [condition];
```

The following code has a query, which will DELETE a customer, whose ID is 6.

```
SQL> DELETE FROM CUSTOMERS  
WHERE ID = 6;
```

Clauses and Operators

1. **Group by clause:** These are circumstances where we would like to apply the aggregate functions to a single set of tuples but also to a group of sets of tuples we would like to specify this wish in SQL using the group by clause. The attributes or attributes given by the group by clause are used to form groups. Tuples with the same value on all attributes in the group by clause placed in one group.

Example:.

```
Select<attribute_name,avg(<attribute_name>)>as  
<new_attribute_name>| From <table_name>  
Group by <attribute_name>
```

Example: select designation, sum(salary) as total_salary from employee group by Designation;

2. **Having clause:** A having clause is like a where clause but only applies only to groups as a whole whereas the where clause applies to the individual rows. A query can contain both where clause and a having clause. In that case

a. The where clause is applied first to the individual rows in the tables or table structures objects in the diagram pane. Only the rows that meet the conditions in the where clause are grouped.

b. The having clause is then applied to the rows in the result set that are produced by grouping. Only the groups that meet the having conditions appear in the query output.

Example:

```
select dept_no from EMPLOYEE group_by dept_no  
having avg (salary) >=all (select avg (salary)  
from EMPLOYEE group by dept_no);
```

3. **Aggregate functions:** Aggregate functions such as SUM, AVG, count, count (*), MAX and MIN generate summary values in query result sets. An aggregate functions (with the exception of count (*) processes all the selected values in a single column to produce a single result value

Example: select dept_no,count (*)
from EMPLOYEE group by dept_no;

Example: select max (salary)as maximum from EMPLOYEE;

Example: select sum (salary) as total_salary from EMPLOYEE;

Example: Select min (salary) as minsal from EMPLOYEE;

4. **Exists and Not Exists:** Subqueries introduced with exists and not queries can be used for two set theory operations: Intersection and Difference. The intersection of two sets contains all elements that belong to both of the original sets. The difference contains elements that belong to only first of the two sets.

Example:

```
Select *from DEPARTMENT
where exists(select * from PROJECT
            where DEPARTMENT.dept_no = PROJECT.dept_no) ;
```

5. **IN and Not In:** SQL allows testing tuples for membership in a relation. The “in” connective tests for set membership where the set is a collection of values produced by select clause. The “not in” connective tests for the absence of set membership. The in and not in connectives can also be used on enumerated sets.

Example:

1. Select fname, mname, lname from employee where designation In (“ceo”, “manager”, “hod”, “assistant”)
2. Select fullname from department where relationship not in(“brother”);

6. **Between:** The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates. The BETWEEN operator is inclusive. Begin and end values are included.

Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

Example:

```
SELECT * FROM Products WHERE Price BETWEEN 10 AND 20;
```

7. **LIKE:** The LIKE **operator** is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards used in conjunction with the LIKE operator:

- % - The percent sign represents zero, one, or multiple characters
- _ - The underscore represents a single character

Syntax: `SELECT column1, column2, ...`
`FROM table_name`
`WHERE columnN LIKE pattern`

Examples:

- selects all customers with a CustomerName starting with "a":

```
SELECT * FROM Customers
WHERE CustomerName LIKE 'a%';
```

- selects all customers with a CustomerName that have "r" in the second position:

```
SELECT * FROM Customers
WHERE CustomerName LIKE '_r%';
```

8. **Alias:** The use of table aliases is to rename a table in a specific SQL statement. The renaming is a temporary change and the actual table name does not change in the database. The column aliases are used to rename a table's columns for the purpose of a particular SQL query.

The basic syntax of a **table** alias is as follows.

```
SELECT column1, column2....
FROM table_name AS alias_name
WHERE [condition];
```

The basic syntax of a **column** alias is as follows.

```
SELECT column_name AS alias_name
FROM table_name
WHERE [condition];
```

Example:

```
SELECT C.ID, C.NAME, C.AGE, O.AMOUNT
FROM CUSTOMERS AS C, ORDERS AS O
WHERE C.ID = O.CUSTOMER_ID;
```

9. **Distinct:** The SELECT DISTINCT statement is used to return only distinct (different) values.

Syntax: SELECT DISTINCT *column1, column2, ...*
FROM *table_name*;

Example: SELECT DISTINCT Country FROM Customers;

10. **Set Operations:** 4 different types of SET operations, along with example:

- UNION
- UNION ALL
- INTERSECT
- MINUS

UNION Operation

UNION is used to combine the results of two or more SELECT statements. However it will eliminate duplicate rows from its resultset. In case of union, number of columns and datatype must be same in both the tables, on which UNION operation is being applied.

Query: SELECT * FROM First

UNION

SELECT * FROM Second;

UNION ALL

This operation is similar to Union. But it also shows the duplicate rows.

Query: SELECT * FROM First

UNION ALL

SELECT * FROM Second;

INTERSECT

Intersect operation is used to combine two SELECT statements, but it only returns the records which are common from both SELECT statements. In case of **Intersect** the number of columns and datatype must be same.

Query: SELECT * FROM First

INTERSECT

SELECT * FROM Second;

MINUS

The Minus operation combines results of two SELECT statements and return only those in the final result, which belongs to the first set of the result.

Query: SELECT * FROM First

MINUS

SELECT * FROM Second;

11. ANY and ALL: The ANY and ALL operators are used with a WHERE or HAVING clause. The ANY operator returns true if any of the subquery values meet the condition. The ALL operator returns true if all of the subquery values meet the condition.

ANY

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ANY
(SELECT column_name FROM table_name WHERE condition);
```

Example: The following SQL statement returns TRUE and lists the productnames if it finds ANY records in the OrderDetails table that quantity = 10:

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);
```

ALL

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ALL
(SELECT column_name FROM table_name WHERE condition);
```

Example: The following SQL statement returns TRUE and lists the product names if ALL the records in the OrderDetails table has quantity = 10:

```
SELECT ProductName
FROM Products
WHERE ProductID = ALL (SELECT ProductID FROM OrderDetails WHERE Quantity = 10);
```

Implementation details

- Simple question based on your application, queries and screen shots for each type:

```
CREATE TABLE BANK(Emp_Id int,Emp_Name varchar(20),Cust_Id int, Cust_Name varchar(20),
Branch_Id int)
```

```
INSERT INTO BANK values(1,'abc',100,'Ronak',1100);
```

```
INSERT INTO BANK values(2,'def',101,'Hyder',1101);
```

INSERT INTO BANK values(3,'hij',102,'Nikhil',1102);

INSERT INTO BANK values(4,'klm',103,'Chinmay',1102);

INSERT INTO BANK values(5,'nop',104,'Sarvesh',1103);

SELECT*from BANK

Data Output		Explain	Messages	Notifications	
	emp_id integer	emp_name character varying (20)	cust_id integer	cust_name character varying (20)	branch_id integer
1	1	abc	100	Ronak	1100
2	2	def	101	Hyder	1101
3	3	hij	102	Nikhil	1102
4	4	klm	103	Chinmay	1102
5	5	nop	104	Sarvesh	1103

UPDATE BANK

SET emplocation='ind'

WHERE empid=2

SELECT*from BANK

Data Output		Explain	Messages	Notifications	
	emp_id integer	emp_name character varying (20)	cust_id integer	cust_name character varying (20)	branch_id integer
1	1	abc	100	Ronak	1100
2	2	def	101	Hyder	1101
3	3	hij	102	Nikhil	1102
4	5	nop	104	Sarvesh	1103
5	4	klm	103	Chinmay	1104

DELETE from BANK



WHERE Emp_Id=5;

BETWEEN Operator:

Query Editor Query History

```
1 CREATE TABLE Employ (Emp_id INT PRIMARY KEY, Emp_name VARCHAR (100) NOT NULL );
2 CREATE TABLE Custmurr (Cust_id INT PRIMARY KEY, Cust_name VARCHAR (100) NOT NULL);
3 INSERT INTO Employ(Emp_id, Emp_name) VALUES (200, 'Hyder'), (201, 'Ronak'), (202, 'Nikhil'), (203, 'Adi');
4 INSERT INTO Custmurr (Cust_id, Cust_name) VALUES (1, 'Hyder'), (2, 'Ronak'), (3, 'Nikhil'), (4, 'Aditya');
5 SELECT * FROM Custmurr
6 WHERE Cust_id BETWEEN 3 and 4
7
8
9
10
```

Data Output Explain Messages Notifications

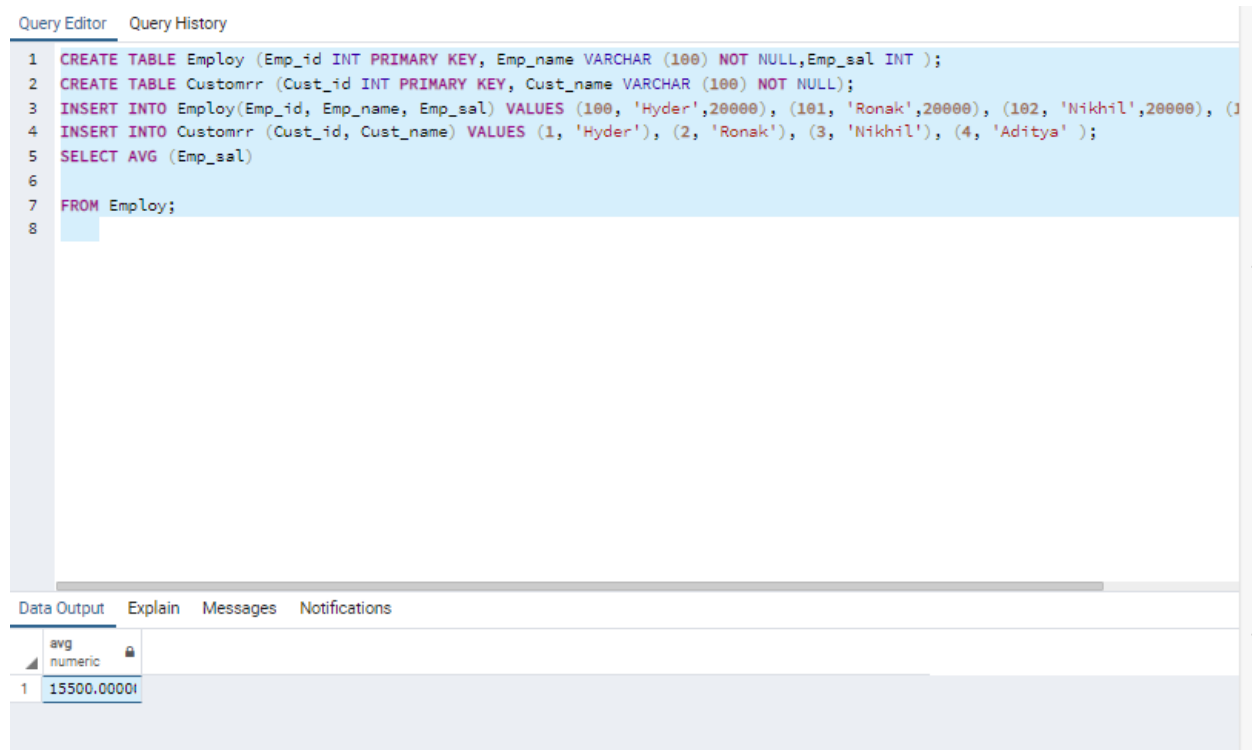
	 cust_id [PK] integer	 cust_name character varying (100)
1	3	Nikhil
2	4	Aditya

Aggregate Functions:

Average:-

```
CREATE TABLE Employ (Emp_id INT PRIMARY KEY, Emp_name VARCHAR (100) NOT
NULL,Emp_sal INT );
CREATE TABLE Customrr (Cust_id INT PRIMARY KEY, Cust_name VARCHAR (100) NOT
NULL);
INSERT INTO Employ(Emp_id, Emp_name, Emp_sal) VALUES (100, 'Hyder',20000), (101,
'Ronak',20000), (102, 'Nikhil',20000), (103, 'Adi',2000);
INSERT INTO Customrr (Cust_id, Cust_name) VALUES (1, 'Hyder'), (2, 'Ronak'), (3, 'Nikhil'), (4,
'Aditya' );
SELECT AVG (Emp_sal)
FROM Employ;
```

Output:



The screenshot shows a database query editor with a 'Query Editor' tab. The SQL code is as follows:

```
1 CREATE TABLE Employ (Emp_id INT PRIMARY KEY, Emp_name VARCHAR (100) NOT NULL,Emp_sal INT );
2 CREATE TABLE Customrr (Cust_id INT PRIMARY KEY, Cust_name VARCHAR (100) NOT NULL);
3 INSERT INTO Employ(Emp_id, Emp_name, Emp_sal) VALUES (100, 'Hyder',20000), (101, 'Ronak',20000), (102, 'Nikhil',20000), (103, 'Aditya',2000);
4 INSERT INTO Customrr (Cust_id, Cust_name) VALUES (1, 'Hyder'), (2, 'Ronak'), (3, 'Nikhil'), (4, 'Aditya' );
5 SELECT AVG (Emp_sal)
6
7 FROM Employ;
8
```

Below the editor, there are tabs for 'Data Output', 'Explain', 'Messages', and 'Notifications'. The 'Data Output' tab is active, showing a single row of results:

	avg numeric
1	15500.00000

Sum:-

```
CREATE TABLE Employ (Emp_id INT PRIMARY KEY, Emp_name
VARCHAR (100) NOT NULL,Emp_sal INT );
CREATE TABLE Customrr (Cust_id INT PRIMARY KEY, Cust_name
VARCHAR (100) NOT NULL);
INSERT INTO Employ(Emp_id, Emp_name, Emp_sal) VALUES (100,
'Hyder',20000), (101, 'Ronak',20000), (102, 'Nikhil',20000), (103,
'Adi',2000);
INSERT INTO Customrr (Cust_id, Cust_name) VALUES (1, 'Hyder'), (2,
'Ronak'), (3, 'Nikhil'), (4, 'Aditya' );
SELECT SUM(Emp_sal)
FROM Employ;
```

OUTPUT:-

Query EditorQuery History

```
1 CREATE TABLE Employ (Emp_id INT PRIMARY KEY, Emp_name VARCHAR (100) NOT NULL,Emp_sal INT );
2 CREATE TABLE Customrr (Cust_id INT PRIMARY KEY, Cust_name VARCHAR (100) NOT NULL);
3 INSERT INTO Employ(Emp_id, Emp_name, Emp_sal) VALUES (100, 'Hyder',20000), (101, 'Ronak',20000), (102, 'Nikhil',20000), (103, 'Aditya',20000);
4 INSERT INTO Customrr (Cust_id, Cust_name) VALUES (1, 'Hyder'), (2, 'Ronak'), (3, 'Nikhil'), (4, 'Aditya' );
5 SELECT SUM(Emp_sal)
6 FROM Employ;
```

Data OutputExplainMessagesNotifications

	sum	
	bigint	
1	62000	

Theory

Join is a combination of a Cartesian product followed by a selection process. A Join operation pairs two tuples from different relations, if and only if a given join condition is satisfied. Or JOINS are used to retrieve data from multiple tables. A JOIN is performed whenever two or more tables are joined in a SQL statement.

There are different types of Joins:

- The CROSS JOIN
- The INNER JOIN
- The LEFT OUTER JOIN
- The RIGHT OUTER JOIN
- The FULL OUTER JOIN

A **CROSS JOIN** matches every row of the first table with every row of the second table. If the input tables have x and y columns, respectively, the resulting table will have x+y columns. Because CROSS JOINS have the potential to generate extremely large tables, care must be taken to use them only when appropriate.

Ex. SELECT EMP_ID, NAME, DEPT FROM COMPANY CROSS JOIN DEPARTMENT;

A **INNER JOIN** creates a new result table by combining column values of two tables (table1 and table2) based upon the join-predicate. The query compares each row of table1 with each row of table2 to find all pairs of rows, which satisfy the join-predicate. When the join-predicate is satisfied, column values for each matched pair of rows of table1 and table2 are combined into a result row.

Ex. SELECT EMP_ID, NAME, DEPT FROM COMPANY INNER JOIN DEPARTMENT ON COMPANY.ID = DEPARTMENT.EMP_ID;

The **OUTER JOIN** is an extension of the INNER JOIN. SQL standard defines three types of OUTER JOINS: LEFT, RIGHT, and FULL and PostgreSQL supports all of these.

In case of **LEFT OUTER JOIN**, an inner join is performed first. Then, for each row in table T1 that does not satisfy the join condition with any row in table T2, a joined row is added with null values in columns of T2. Thus, the joined table always has at least one row for each row in T1.

Ex. SELECT EMP_ID, NAME, DEPT FROM COMPANY LEFT OUTER JOIN DEPARTMENT ON COMPANY.ID = DEPARTMENT.EMP_ID;

The RIGHT OUTER JOIN

First, an inner join is performed. Then, for each row in table T2 that does not satisfy the join condition with any row in table T1, a joined row is added with null values in columns of T1. This is the converse of a left join; the result table will always have a row for each row in T2.

Ex. SELECT EMP_ID, NAME, DEPT FROM COMPANY RIGHT OUTER JOIN DEPARTMENT ON COMPANY.ID = DEPARTMENT.EMP_ID;

The FULL OUTER JOIN

First, an inner join is performed. Then, for each row in table T1 that does not satisfy the join condition with any row in table T2, a joined row is added with null values in columns of T2.

In addition, for each row of T2 that does not satisfy the join condition with any row in T1, a joined row with null values in the columns of T1 is added.

```
SELECT EMP_ID, NAME, DEPT FROM COMPANY FULL OUTER JOIN  
DEPARTMENT ON COMPANY.ID = DEPARTMENT.EMP_ID;
```

Views are pseudo-tables. That is, they are not real tables; nevertheless appear as ordinary tables to SELECT. A view can represent a subset of a real table, selecting certain columns or certain rows from an ordinary table. A view can even represent joined tables. Because views are assigned separate permissions, you can use them to restrict table access so that the users see only specific rows or columns of a table.

A view can contain all rows of a table or selected rows from one or more tables. A view can be created from one or many tables, which depends on the written PostgreSQL query to create a view.

Views, which are kind of virtual tables, allow users to do the following –

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data such that a user can only see limited data instead of complete table.
- Summarize data from various tables, which can be used to generate reports.

Since views are not ordinary tables, you may not be able to execute a DELETE, INSERT, or UPDATE statement on a view. However, you can create a RULE to correct this problem of using DELETE, INSERT or UPDATE on a view.

Syntax

```
CREATE [TEMP | TEMPORARY] VIEW view_name AS
```

```
SELECT column1, column2.....
```

```
FROM table_name
```

```
WHERE [condition];
```

Ex

```
CREATE VIEW COMPANY_VIEW AS
```

```
SELECT ID, NAME, AGE
```

```
FROM COMPANY;
```

Dropping Views

Syntax: DROP VIEW view_name;

Implementation Screenshots:

Inner Join:

BANK/postgres@PostgreSQL 12

Query Editor Query History

```
1 CREATE TABLE Empl8 ( Emp_id INT PRIMARY KEY, Emp_name VARCHAR (100) NOT NULL );
2 CREATE TABLE Custmrr ( Cust_id INT PRIMARY KEY, Cust_name VARCHAR (100) NOT NULL );
3 INSERT INTO Empl8(Emp_id, Emp_name) VALUES (100,'Hyder'),(101,'Ronak'),(102,'Nikhil'),(103,'Adi
4 INSERT INTO Custmrr(Cust_id, Cust_name) VALUES (1,'Hyder'),(2,'Ronak'),(3,'Nikhil'),(4,'Aditya'
5 SELECT
6     Emp_id,
7     Emp_name,
8     Cust_id,
9     Cust_name
10 FROM Empl8
11 INNER JOIN Custmrr ON Custmrr.Cust_name=Empl8.Emp_name
```

Data Output Explain Messages Notifications

	emp_id integer	emp_name character varying (100)	cust_id integer	cust_name character varying (100)
1	100	Hyder	1	Hyder
2	101	Ronak	2	Ronak
3	102	Nikhil	3	Nikhil
4	103	Aditya	4	Aditya

BANK/postgres@PostgreSQL 12

Query Editor Query History

```
1 CREATE TABLE Empl8 ( Emp_id INT PRIMARY KEY, Emp_name VARCHAR (100) NOT NULL );
2 CREATE TABLE Custmur ( Cust_id INT PRIMARY KEY, Cust_name VARCHAR (100) NOT NULL );
3 INSERT INTO Empl8(Emp_id, Emp_name) VALUES (100,'Hyder'),(101,'Ronak'),(102,'Nikhil'),(103,'Adi
4 INSERT INTO Custmur(Cust_id, Cust_name) VALUES (1,'Hyder'),(2,'Ronak'),(3,'Nikl'),(4,'itya');
5 SELECT
6     Emp_id,
7     Emp_name,
8     Cust_id,
9     Cust_name
10 FROM Empl8
11 INNER JOIN Custmur ON Custmur.Cust_name=Empl8.Emp_name
```

Data Output Explain Messages Notifications

	emp_id integer	emp_name character varying (100)	cust_id integer	cust_name character varying (100)
1	100	Hyder	1	Hyder
2	101	Ronak	2	Ronak

Here we have changed the employees name
Hence Nikhil and Aditya are not showing.

Conclusion: *Successfully executed the given experiment of performing various DML Operations and executing nested queries with various clauses.*