PRACTICAL NO. 7

Title of the Exercise : Querying the Database based nesting and applying various Clauses.

OBJECTIVE (AIM) OF THE EXPERIMENT

To perform nested Queries and joining Queries using DML command.

a) Procedure:

Step no.	Details of the step
1	**Nested Queries:** Nesting of queries one within another is known as a nested queries. **Sub queries** The query within another is known as a sub query. A statement containing sub query is called parent statement. The rows returned by sub query are used by the parent statement.
2	**Types** **1. Sub queries that return several values** Sub queries can also return more than one value. Such results should be made use along with the operators in and any. **2. Multiple queries** Here more than one sub query is used. These multiple sub queries are combined by means of „and" & „or" keywords **3. Correlated sub query** A sub query is evaluated once for the entire parent statement whereas a correlated Sub query is evaluated once per row processed by the parent statement.

b) SQL Commands:

Nested Queries:

A subquery is a `SELECT` statement written within parentheses and nested inside another statement. Here's an example that looks up the IDs for grade event rows that correspond to tests (`'T'`) and uses them to select scores for those tests

```
SELECT * FROM score

WHERE event_id IN (SELECT event_id FROM grade_event WHERE category = 'T');
```

Subqueries can return different types of information:

- A scalar subquery returns a single value.
- A column subquery returns a single column of one or more values.
- A row subquery returns a single row of one or more values.
- A table subquery returns a table of one or more rows of one or more columns.

Subquery results can be tested in different ways:

- Scalar subquery results can be evaluated using relative comparison operators such as = or <.
- `IN` and `NOT IN` test whether a value is present in a set of values returned by a subquery.
- `ALL`, `ANY`, and `SOME` compare a value to the set of values returned by a subquery.
- `EXISTS` and `NOT EXISTS` test whether a subquery result is empty.

A scalar subquery is the most restrictive because it produces only a single value. But as a consequence, scalar subqueries can be used in the widest variety of contexts. They are applicable essentially anywhere that you can use a scalar operand, such as a term of an expression, as a function argument, or in the output column list. Column, row, and table subqueries that return more information cannot be used in contexts that require a single value.

Subqueries can be correlated or uncorrelated. This is a function of whether a subquery refers to and is dependent on values in the outer query.

You can use subqueries with statements other than `SELECT`. However, for statements that modify tables (`DELETE`, `INSERT`, `REPLACE`, `UPDATE`, `LOAD DATA`), MySQL enforces the restriction that the subquery cannot select from the table being modified.

In some cases, subqueries can be rewritten as joins. You might find subquery rewriting techniques useful to see whether the MySQL optimizer does a better job with a join than the equivalent subquery.

The following sections discuss the kinds of operations you can use to test subquery results, how to write correlated subqueries, and how to rewrite subqueries as joins

1) Subqueries with Relative Comparison Operators

The =, <>, >, >=, <, and <= operators perform relative-value comparisons. When used with a scalar subquery, they find all rows in the outer query that stand in particular relationship to the value returned by the subquery. For example, to identify the scores for the quiz that took place on '2012-09-23', use a scalar subquery to determine the quiz event ID and then match score table rows against that ID in the outer SELECT:

Example:

```
SELECT * FROM score
WHERE event_id =

(SELECT event_id FROM grade_event

   WHERE date = '2012-09-23' AND category = 'Q');
```

Use of scalar subqueries with relative comparison operators is handy for solving problems for which you'd be tempted to use an aggregate function in a `WHERE` clause. For example, to determine which of the presidents in the `president` table was born first, you might try this statement:
Example:

1. `SELECT * FROM president WHERE birth = MIN(birth);`

2.
   ```
   SELECT * FROM president
   WHERE birth = (SELECT MIN(birth) FROM president);
   ```

3.
   ```
   SELECT * FROM score WHERE event_id = 5
   AND score > (SELECT AVG(score) FROM score WHERE event_id = 5);
   ```

If a subquery returns a single row, you can use a row constructor to compare a set of values (that is, a tuple) to the subquery result. This statement returns rows for presidents who were born in the same city and state as John Adams
Example:

```
mysql> SELECT last_name, first_name, city, state FROM president
```

```
    -> WHERE (city, state) =-> (SELECT city, state FROM president
    -> WHERE last_name = 'Adams' AND first_name = 'John');
+-----------+------------+-----------+-------+
| last_name | first_name  | city      | state |
+-----------+------------+-----------+-------+
| Adams     | John        | Braintree | MA    |
| Adams     | John Quincy | Braintree | MA    |
+-----------+------------+-----------+-------+
```

You can also use ROW(city, state) notation, which is equivalent to (city, state). Both act as row constructors.

2) IN and NOT IN Subqueries

The `IN` and `NOT IN` operators can be used when a subquery returns multiple rows to be evaluated in comparison to the outer query. They test whether a comparison value is present in a set of values. `IN` is true for rows in the outer query that match any row returned by the subquery. `NOT IN` is true for rows in the outer query that match no rows returned by the subquery. The following statements use `IN` and `NOT IN` to find those students who have absences listed in the `absence` table, and those who have perfect attendance (no absences):

```
mysql> SELECT * FROM student
    -> WHERE student_id IN (SELECT student_id FROM absence);
+-------+-----+------------+
| name  | sex | student_id |
+-------+-----+------------+
| Kyle  | M   |          3 |
| Abby  | F   |          5 |

| Peter | M   |         10 |
| Will  | M   |         17 |
| Avery | F   |         20 |
+-------+-----+------------+
mysql> SELECT * FROM student
    -> WHERE student_id NOT IN (SELECT student_id FROM absence);
+-----------+-----+------------+
| name      | sex | student_id |
+-----------+-----+------------+
| Megan     | F   |          1 |
| Joseph    | M   |          2 |
| Katie     | F   |          4 |
| Nathan    | M   |          6 |
| Liesl     | F   |          7 |
+----------------+------------------+------+
```

IN and NOT IN also work for subqueries that return multiple columns. In other words, you can use them with table subqueries. In this case, use a row constructor to specify the comparison values to test against each column:

```
mysql> SELECT last_name, first_name, city, state FROM president
    -> WHERE (city, state) IN
    -> (SELECT city, state FROM president
    -> WHERE last_name = 'Roosevelt');
+----------------+------------------+----------------+----------+
| last_name | first_name  | city      | state |
+----------------+------------------+----------------+----------+
| Roosevelt | Theodore    | New York  | NY    |
| Roosevelt | Franklin D. | Hyde Park | NY    |
+----------------+------------------+----------------+----------+
```

IN and NOT IN actually are synonyms for = ANY and <> ALL, which are covered in the next section.

3) ALL, ANY, and SOME Subqueries

The ALL and ANY operators are used in conjunction with a relative comparison operator to test the result of a column subquery. They test whether the comparison value stands in particular relationship to all or some of the values returned by the subquery. For example, <= ALL is true if the comparison value is less than or equal to every value that the subquery returns, whereas <= ANY is true if the comparison value is less than or equal to any value that the subquery returns. SOME is a synonym for ANY.

This statement determines which president was born first by selecting the row with a birth date less than or equal to all the birth dates in the president table (only the earliest date satisfies this condition):

```
mysql> SELECT last_name, first_name, birth FROM president
    -> WHERE birth <= ALL (SELECT birth FROM president);
+   ------------+---- ------------+---- ------------+----
| last_name  | first_name | birth      |
+   ------------+---- ------------+---- ------------+----
| Washington | George     | 1732-02-22 |
+   ------------+---- ------------+---- ------------+----
```

Less usefully, the following statement returns all rows because every date is less than or equal to at least one other date (itself):

```
mysql> SELECT last_name, first_name, birth FROM president
    -> WHERE birth <= ANY (SELECT birth FROM president);
+   ------------+---- ----------------+--- ------------+---
| last_name  | first_name     | birth      |
+   ------------+---- ----------------+--- ------------+---
| Washington | George         | 1732-02-22 |
| Adams      | John           | 1735-10-30 |
| Jefferson  | Thomas         | 1743-04-13 |
| Madison    | James          | 1751-03-16 |
| Monroe     | James          | 1758-04-28 |
```

When ALL, ANY, or SOME are used with the = comparison operator, the subquery can be a table subquery. In this case, you test return rows using a row constructor to provide the comparison values.

```
mysql> SELECT last_name, first_name, city, state FROM president
```

```
    -> WHERE (city, state) = ANY
    -> (SELECT city, state FROM president
    -> WHERE last_name = 'Roosevelt');
+   -----------+---  --------------+---  -----------+---  ------+---
| last_name | first_name  | city      | state |
+   -----------+---  --------------+---  -----------+---  ------+---
| Roosevelt | Theodore    | New York  | NY    |
| Roosevelt | Franklin D. | Hyde Park | NY    |
+   -----------+---  --------------+---  -----------+---  ------+---
```

As mentioned in the previous section, IN and NOT IN are shorthand for = ANY and <> ALL. That is, IN means "equal to any of the rows returned by the subquery" and NOT IN means "unequal to all rows returned by the subquery."

EXISTS and NOT EXISTS Subqueries

The EXISTS and NOT EXISTS operators merely test whether a subquery returns any rows. If it does, EXISTS is true and NOT EXISTS is false. The following statements show some trivial examples of these subqueries. The first returns 0 if the absence table is empty, the second returns 1:

```
SELECT EXISTS (SELECT * FROM absence);
SELECT NOT EXISTS (SELECT * FROM
absence);
```

EXISTS and NOT EXISTS actually are much more commonly used in correlated subqueries.

With EXISTS and NOT EXISTS, the subquery uses * as the output column list. There's no need to name columns explicitly, because the subquery is assessed as true or false based on whether it returns any rows, not based on the particular values that the rows might contain. You can actually write pretty much anything for the subquery column selection list, but if you want to make it explicit that you're returning a true value when the subquery succeeds, you might write it as SELECT 1 rather than SELECT *.

4) Correlated Subqueries

Subqueries can be uncorrelated or correlated:

- An uncorrelated subquery contains no references to values from the outer query, so it could be executed by itself as a separate statement. For example, the subquery in the following statement is uncorrelated because it refers only to the table t1 and not to t2:

  ```
  SELECT j FROM t2 WHERE j IN (SELECT i FROM t1);
  ```

- A correlated subquery does contain references to values from the outer query, and thus is dependent on it. Due to this linkage, a correlated subquery cannot be executed by itself as a separate statement. For example, the subquery in the following statement is true for each value of column `j` in `t2` that matches a column `i` value in `t1`:

  ```
  SELECT j FROM t2 WHERE (SELECT i FROM t1 WHERE i = j);
  ```

The following `EXISTS` subquery identifies matches between the tables—that is, values that are present in both. The statement selects students who have at least one absence listed in the `absence` table:

```
SELECT student_id, name FROM student WHERE EXISTS

(SELECT * FROM absence WHERE absence.student_id = student.student_id);
```

`NOT EXISTS` identifies nonmatches—values in one table that are not present in the other. This statement selects students who have no absences:

```
SELECT student_id, name FROM student WHERE NOT EXISTS

(SELECT * FROM absence WHERE absence.student_id = student.student_id);
```

5) Subqueries in the FROM Clause

Subqueries can be used in the `FROM` clause to generate values. In this case, the result of the subquery acts like a table. A subquery in the `FROM` clause can participate in joins, its values can be tested in the`WHERE` clause, and so forth. With this type of subquery, you must provide a table alias to give the subquery result a name:

```
mysql> SELECT * FROM (SELECT 1, 2) AS t1 INNER JOIN (SELECT 3, 4) AS t2;

+---+---+---+---+

| 1 | 2 | 3 | 4 |

+---+---+---+---+

| 1 | 2 | 3 | 4 |

+---+---+---+---+
```

c) Queries:

1. Get the names of employees who have a below average salary?

```
SQL> SELECT EMP_NAME

  2  FROM EMP
  3  WHERE SALARY <
  4  ( SELECT AVG( SALARY ) FROM EMP );
```

```
   --------------
EMP_NAME

Jone
s
Robe
rts
Jarv
is
Flet
cher
Evan
s
```

2. Get the names of employees who have a greater than average salary?

```
SQL> SELECT EMP_NAME

  2  FROM EMP
  3  WHERE SALARY >
  4  ( SELECT AVG( SALARY ) FROM EMP );

   --------------
EMP_NAME

B
r
o
w
n
G
r
e
e
n
```

3. Get the names of employees who are paid exactly the avarage salary ?

```
SQL> SELECT EMP_NAME

  2  FROM EMP
  3  WHERE SALARY =
  4  ( SELECT AVG( SALARY ) FROM EMP );

no rows selected
```

4. Get the names of Employee is the most highly paid ?

```
SQL> SELECT EMP_NAME

  2  FROM EMP
  3  WHERE SALARY =
  4  ( SELECT MAX( SALARY ) FROM EMP );

   --------------
EMP_NAME

B
r
o
w
n
```

```
G
r
e
e
n
```

5. **Get the names of Employee is the least paid?**

```
SQL> SELECT EMP_NAME

  2  FROM EMP
  3  WHERE SALARY =( SELECT MIN( SALARY ) FROM EMP ); EMP_NAME
Evans

   --------------
```

6. **Get the names of Employee whose salary exceeds the lowest salary by more than 4,000 ?**

```
SQL> SELECT EMP_NAME

  2  FROM EMP
  3  WHERE SALARY >
  4  ( SELECT (4000 + MIN( SALARY ) ) AS BOUNDARY
  5    FROM

EMP );
   --------------
EMP_NAME

Rob
ert
s
Jar
vis
Bro
wn
Gre
en
```

7. **Get the names of employees for the departments that have only one employee**

```
SQL> SELECT DEPT_NAME FROM DEPT

  2  WHERE 1 =
  3  ( SELECT COUNT(*) FROM EMP
  4    WHERE EMP.DEPT_NO =

DEPT.DEPT_NO ); DEPT_NAME
   --------------
Transport
```

8. **Get the deadlines of projects that have more than 3 employees working on them**

```
SQL> SELECT DEADLINE FROM PROJ

  2  WHERE 3 <
  3  ( SELECT COUNT(*) FROM ALLOC
  4    WHERE ALLOC.PROJ_NO =

PROJ.PROJ_NO ); DEADLINE
   -------------
```

```
01-JAN-09
```

9. Get the names of employees who work in departments such that their salary is less than 10% of their department’s budget

```
SQL> SELECT EMP_NAME FROM EMP

  2  WHERE SALARY <
  3  ( SELECT BUDGET/10 FROM DEPT
  4    WHERE EMP.DEPT_NO =

DEPT.DEPT_NO ); EMP_NAME
   --------------
Evans
```

10. Get the details of the projects not worked on by employee E2.

```
SQL> SELECT * FROM PROJ

  2  WHERE PROJ_NO IN
  3   (SELECT PROJ_NO FROM ALLOC
  4    WHERE EMP_NO = 'E2');
  5    PR START_DAT DEADLINE
   --- ---------- ----------
P2 21-JAN-05 30-SEP-07
```

11. Get the names of employees not allocated to projects.

```
SQL> SELECT EMP_NAME FROM EMP

  2  WHERE EMP_NO  NOT IN
  3  ( SELECT EMP_NO FROM

ALLOC ); EMP_NAME
   --------------
Rob
ert
s
Jar
vis
```

12. Get the details of all employees who are not departmental managers.

```
SQL> SELECT * FROM EMP

  2  WHERE EMP_NO  NOT IN
  3  ( SELECT MANAGER_NO FROM DEPT );

EM EMP_NAME       SALARY M DE

   --- -------------- ---------- -   ---

E9 Fletcher        12000 S D1

E4 Evans           11000 S D5

E6 Green           38500 ? D2
```

13. Get the details of employees whose salary is less than that of all those employees who work on project P2.

```
SQL> SELECT * FROM EMP

  2  WHERE SALARY <ALL
  3  ( SELECT SALARY FROM EMP    NATURAL JOINALLOC
     4    WHERE PROJ_NO = 'P2' );


   EM EMP_NAME       SALARY M DE

      --- -------------- -------------   ---

   E4 Evans           11000 S D5
```

14. Get details of the department with the largest budget;

```
    SQL> SELECT * FROM
                  DEPT

 2  WHERE BUDGET >=ALL
     3  ( SELECT BUDGET FROM

   DEPT ); DE DEPT_NAME MA
      --- -------------- --- --------------
        BUDGET

   D2 Sales      E5     250000
```

Querying the Database based nesting and applying various Clauses.

Create the following Tables:

EMPLOYEE(Emp_id, EMP_name,Job_name,Manager_id,Hire_date,Salary,Deptno)

DEPARTMENT(Deptno, Dname, MGRSSN)

PROJECT(Pname,Pno,Plocation,Deptno)

Q1. Enter the following data into the Employee Table:

```
emp_id | emp_name | job_name  | manager_id | hire_date  | salary  | | dep_no

  68319 | KAYLING  | PRESIDENT |            | 1991-11-18 | 6000.00 |  |   1001
  66928 | BLAZE    | MANAGER   |      68319 | 1991-05-01 | 2750.00 |  |   3001
  67832 | CLARE    | MANAGER   |      68319 | 1991-06-09 | 2550.00 |      1001
  65646 | JONAS    | MANAGER   |      68319 | 1991-04-02 | 2957.00 |  |   2001
  67858 | SCARLET  | ANALYST   |      65646 | 1997-04-19 | 3100.00 |  |   2001
  69062 | FRANK    | ANALYST   |      65646 | 1991-12-03 | 3100.00 |      2001
  63679 | SANDRINE | CLERK     |      69062 | 1990-12-18 |  900.00 |      2001
  64989 | ADELYN   | SALESMAN  |      66928 | 1991-02-20 | 1700.00 |      3001
  65271 | WADE     | SALESMAN  |      66928 | 1991-02-22 | 1350.00 |  |   3001
  66564 | MADDEN   | SALESMAN  |      66928 | 1991-09-28 | 1350.00 |      3001
  68454 | TUCKER   | SALESMAN  |      66928 | 1991-09-08 | 1600.00 |      3001
  68736 | ADNRES   | CLERK     |      67858 | 1997-05-23 | 1200.00 |      2001
  69000 | JULIUS   | CLERK     |      66928 | 1991-12-03 | 1050.00 |      3001
  69324 | MARKER   | CLERK     |      67832 | 1992-01-23 | 1400.00 |      1001
```

Department Table

deptno	**dname**	**Citylocation**	**dCountry**
1001	Accounting	New York	United States of America,
2001	Research	Dallas	United States

3001	Sales	Chicago	United States of America
4001	Marketing	Los Angeles	United States

Project Table

Pno	Pname	PCitylocation	PCountry
111	P_1	New York	United States of America,
112	P_2	Dallas	United States
113	P_3	Chicago	United States of America
114	P_4	Denmark	northern Europe
115	P_5	Paris	France
116	P_6	Chicago	United States of America

Q1. Display sum of salaries of each job.

Q2. Display the details of employees sorting the salary in increasing order.

Q3. Display number of employees working in each department and their department name.

Q4. Display lowest paid employee details under each manager.

Q5. Show the record of employee earning salary greater than 1600 in each department.

Q6. Display the employee names and id of the department having more than five employees.

Q7. Display the employee of each department who don’t earn more than 1000$.

Q8. Find out how many employees work under the name of salesman.

Q9. Display the total employees who work as manager with increasing order of their salaries.

Q10. Find the location of department accounting and no of employees working in that department.

Q11. Display the employee of each department with highest salary employee at the top of each list.

Q12. Display the highest and lowest salary employee of each department.