MIT World Peace University

Database Management System

Assignment 4

Naman Soni Roll No. 10

Contents

1 Aim

Write suitable select commands to execute queries on the given data set.

2 Objective

- To get basic understanding of Aggregate Functions, Order By clause
- To get basic understanding of Subquery or Inner query or Nested query and Select using subquery.
- To understand the basic concept of Correlated Subquery.
- To get familiar with the basic ALL, ANY, EXISTS, SOME functionality.
- To understand basic TCL commands

3 Problem Statement

Create tables and solve given queries using Subqueries

4 Theory

4.1 *Aggregate Functions, Order By, Group By clause*

Aggregate functions, ORDER BY clause, and GROUP BY clause are essential components of SQL queries used to retrieve and manipulate data from tables.

Aggregate functions are used to perform calculations on a set of values and return a single value. The most commonly used aggregate functions are COUNT, SUM, AVG, MIN, and MAX. For example, to count the number of rows in a table, we can use the COUNT(*) function. To calculate the average salary of all employees, we can use the AVG(salary) function. Aggregate functions can be used in combination with other clauses such as GROUP BY and ORDER BY to generate more complex queries.

The ORDER BY clause is used to sort the result set in ascending or descending order based on one or more columns. By default, the ORDER BY clause sorts the result set in ascending order. To sort the result set in descending order, we can add the DESC keyword after the column name. For example, to sort the result set of a query that returns employee names and their salaries in descending order of salary, we can use the following query:

```sql
    SELECT name, salary
    FROM employees
    ORDER BY salary DESC;
```

The GROUP BY clause is used to group the result set by one or more columns. It is often used in combination with aggregate functions to calculate summary information for each group. For example, to calculate the total sales for each product category, we can use the following query:

```sql
    SELECT category, SUM(sales) as total_sales
    FROM sales
    GROUP BY category;
```

In this example, we group the sales by product category using the GROUP BY clause and calculate the sum of sales for each category using the SUM() function.

In summary, aggregate functions, ORDER BY clause, and GROUP BY clause are essential components of SQL queries. Aggregate functions allow us to perform calculations on a set of values and return a single value. ORDER BY clause is used to sort the result set, and the GROUP BY clause is used to group the result set by one or more columns. These clauses can be used in combination to generate more complex queries that extract useful information from the data in a database.

4.2 *Explain SET operations, use of SET operations in Select*

SET operations are used to combine the results of two or more SELECT statements into a single result set. The most commonly used SET operations in SQL are UNION, INTERSECT, and EXCEPT (or MINUS in some databases).

The UNION operation is used to combine the results of two or more SELECT statements into a single result set. The SELECT statements must have the same number of columns and the corresponding columns must have compatible data types. The syntax for the UNION operation is as follows:

```sql
SELECT column1 , column2 , ...
FROM table1
UNION
SELECT column1 , column2 , ...
FROM table2;
```

The result of the UNION operation is a set of distinct rows that appear in either or both result sets.

The INTERSECT operation is used to return only the common rows that appear in both result sets. The syntax for the INTERSECT operation is as follows:

```sql
    SELECT column1 , column2 , ...
    FROM table1
    INTERSECT
    SELECT column1 , column2 , ...
    FROM table2;
```

The result of the INTERSECT operation is a set of distinct rows that appear in both result sets.

The EXCEPT operation (also known as MINUS in some databases) is used to return only the rows that appear in the first result set but not in the second result set. The syntax for the EXCEPT operation is as follows:

```sql
    SELECT column1 , column2 , ...
    FROM table1
    EXCEPT
    SELECT column1 , column2 , ...
    FROM table2;
```

The result of the EXCEPT operation is a set of distinct rows that appear in the first result set but not in the second result set.

In summary, SET operations are useful for combining the results of multiple SELECT statements and generating a single result set that provides useful insights into the data in a database. The logical operators used in SET operations include UNION, INTERSECT, and EXCEPT (or MINUS). These operators are used to combine the results of two or more SELECT statements and generate a new result set based on specific criteria.

4.3 *Subqueries, use of subquery in Select*

Subqueries, also known as nested queries, are queries that are embedded within another query. A subquery is executed first, and its results are used by the outer query to perform further operations. Subqueries can be used in various clauses of a SELECT statement, such as WHERE, HAVING, and FROM.

The use of subqueries in the WHERE clause is one of the most common applications of subqueries. In this case, the subquery is used to filter the rows returned by the outer query. For example, consider the following query:

```sql
    SELECT name , salary
    FROM employees
    WHERE salary > (SELECT AVG(salary) FROM employees);
```

In this query, the subquery '(SELECT AVG (salary) FROM employees)' calculates the average salary of all employees, and the outer query returns the names and salaries of employees whose salary is higher than the average.

Subqueries can also be used in the HAVING clause to filter groups based on a condition. For example, consider the following query:

```
    SELECT department, AVG(salary) as avg_salary
    FROM employees
    GROUP BY department
    HAVING AVG(salary) > (SELECT AVG(salary) FROM employees);
```

In this query, the subquery '(SELECT AVG (salary) FROM employees)' calculates the average salary of all employees, and the outer query returns the departments and their average salaries, but only for departments whose average salary is higher than the overall average.

4.4 *Views, Types of views*

In SQL, a view is a virtual table that is derived from one or more tables in the database. A view does not actually contain data, but rather it provides a convenient way to access and manipulate data stored in the underlying tables. Views can be used to simplify complex queries, improve security, and provide a customized view of the data to different users or applications.

There are two main types of views in SQL:

- Simple views: Simple views are based on a single table and are created using a SELECT statement that specifies the columns and rows to be included in the view. Simple views can be used to provide a subset of the data in a table, or to rename or reformat columns.

  For example, the following query creates a simple view that includes only the name and salary columns from the employees table:

```
        CREATE VIEW employee_names_and_salaries AS
        SELECT name, salary
        FROM employees;

```

- Complex views: Complex views are based on multiple tables and can include join operations, grouping, and aggregate functions. Complex views can be used to provide a customized view of the data to different users or applications.

  For example, the following query creates a complex view that joins the employees and departments tables, and calculates the average salary for each department:

```
        CREATE VIEW department_salary_averages AS
        SELECT department.name, AVG(employees.salary) as avg_salary
        FROM employees
        INNER JOIN departments ON employees.department_id = departments.id
        GROUP BY department.name;

```

  Views can be useful in many situations, such as providing a simplified view of data to users who do not have access to the underlying tables, or providing a customized view of data to different applications. Views can also improve security by allowing users to access only the data they need, and can simplify queries by providing a pre-defined structure for accessing data.

4.5 *TCL Commands*

TCL (Transaction Control Language) commands are used in SQL to manage transactions in a database. The following are some of the most commonly used TCL commands:

- **COMMIT:** Syntax: COMMIT Description: The COMMIT command is used to permanently save changes made to the database since the last COMMIT or ROLLBACK command. Once a COMMIT command is executed, the transaction is complete and cannot be undone. Example: COMMIT

- **ROLLBACK:** Syntax: ROLLBACK Description: The ROLLBACK command is used to undo any changes made to the database since the last COMMIT or ROLLBACK command. It is typically used when an error occurs during a transaction and the changes need to be undone. Example: ROLLBACK
- **SAVEPOINT:** Syntax: SAVEPOINT savepoint_name; Description: The SAVEPOINT command is used to create a savepoint in a transaction. A savepoint allows you to roll back part of a transaction without undoing the entire transaction. Example: SAVEPOINT my_savepoint;
- **ROLLBACK TO SAVEPOINT:** Syntax: ROLLBACK TO SAVEPOINT savepoint_name; Description: The ROLLBACK TO SAVEPOINT command is used to roll back a transaction to a specific savepoint that was previously created using the SAVEPOINT command. Example: ROLLBACK TO SAVEPOINT my_savepoint;
- **SET TRANSACTION:** Syntax: SET TRANSACTION [READ WRITE — READ ONLY]; Description: The SET TRANSACTION command is used to set the transaction mode to either READ WRITE or READ ONLY. READ WRITE is the default mode and allows changes to be made to the database. READ ONLY mode only allows read access to the database and cannot be used to make changes. Example: SET TRANSACTION READ ONLY

TCL commands are important in managing database transactions as they provide a way to control the outcome of a transaction and ensure data consistency.

5 Input

Given Database from the Problem Statement for the Assignment for our batch. (A1 PA10)

6 Output

```
MariaDB [dbms_lab]> -- QUERIES
MariaDB [dbms_lab]>
MariaDB [dbms_lab]> -- 1. Display the Passenger email ,Flight_no,Source and Destination
Airport Names for all flights MariaDB [dbms_lab]> -- booked
MariaDB [dbms_lab]>
MariaDB [dbms_lab]> select b.email , b.flight_no , f.place_from , f.place_to from
flight_booking
as b inner join flights as f where b.flight_no = f.flight_no;
+----------------+-----------+------------+-----------+
| email          | flight_no  | place_from | place_to  |
+----------------+-----------+------------+-----------+
| love@gmail.com | 12345     | Mumbai     | London    |
| joe@gmail.com  | 23456     | London     | Pune      |
| beck@gmail.com | 67890     | Pune       | Bangalore |
+----------------+-----------+------------+-----------+
3 rows in set (0.001 sec)
MariaDB [dbms_lab]>
MariaDB [dbms_lab]> -- 2.
MariaDB [dbms_lab]> -- Display the flight and passenger details for the flights booked
having Departure Date between
MariaDB [dbms_lab]> -- 23-08-2021 and 25-08-2021
MariaDB [dbms_lab]>
MariaDB [dbms_lab]> select * from flights as f, passenger as p, flight_booking as b
where b.
email = p.email and b.flight_no = f.flight_no and departure_date between "2021-07-27"
and " 2021-07-28";
+--
----------+------------+----------+----------------+----------------+-------------+--------

| flight_no | place_from | place_to | departure_date | departure_time | arrival_date |
arrival_time | reg_no | email | first_name | surname | email | flight_no | no_seats |
+--
----------+------------+-----------+----------------+----------------+-------------+--------
| 67890 | Pune | Bangalore | 2021 -07 -27 | 12:12:12 | 2021 -07 -27 |
16:59:56 | 221 | beck@gmail.com | Gwen | Beck | beck@gmail.com | 67890 |6|
| 23456 | London | Pune | 2021 -07 -27 | 12:12:12 | 2021 -07 -28 | 22:59:56 | 333 |
joe@gmail.com | Joe | Goldberg | joe@gmail.com |
23456 |2|
| 12345 | Mumbai | London | 2021 -07 -27 | 12:12:12 | 2021 -07 -28 |
23:59:56 | 111 | love@gmail.com | Love | Quinn | love@gmail.com | 12345 |6|
+--
----------+------------+-----------+----------------+----------------+-------------+--------
3 rows in set (0.004 sec)
MariaDB [dbms_lab]>
MariaDB [dbms_lab]> -- 3.
MariaDB [dbms_lab]> -- Display the top 5 airplanes that participated in Flights from
Mumbai to London based on the
MariaDB [dbms_lab]> -- airplane capacity MariaDB [dbms_lab]>
MariaDB [dbms_lab]> select * from airplane as a, flights as f where a.reg_no = f.reg_no
and f .place_from = "Mumbai" and f.place_to = "London" order by a.capacity desc limit 5;
+--
-------+----------+----------+--------------+----------+-----------+----------+-----------
| reg_no | model_no | capacity | name | flight_no | place_from | place_to |
departure_date | departure_time | arrival_date | arrival_time | reg_no | +--
--
----+----------+----------+--------------+----------+-----------+----------+-----------
| 111 | 7 | 180 | Qatar Airways | 12345 | Mumbai | London | 2021 -07 -27 | 12:12:12 |
2021 -07 -28 | 23:59:56 | 111 |
+--
------+----------+----------+--------------+----------+-----------+----------+-----------
1 row in set (0.002 sec)
MariaDB [dbms_lab]>
MariaDB [dbms_lab]> -- 4.Display the passenger first names who have booked the no_of
seats smaller than the average
MariaDB [dbms_lab]> -- number of seats booked by all passengers for the arrival airport:
New Delhi
MariaDB [dbms_lab]>
MariaDB [dbms_lab]> select * from passenger as p, flight_booking as b, flights as f
where p.
email = b.email and f.flight_no = b.flight_no and f.place_to = "New Delhi" and b.
no_seats < all(select avg(no_seats) from flight_booking);
Empty set (0.002 sec)
MariaDB [dbms_lab]> MariaDB [dbms_lab]>
MariaDB [dbms_lab]> /*5.Display the surnames of passengers who have not booked a flight
from
Pune to Bangalore */
MariaDB [dbms_lab]> select surname
-> from passenger
-> where email not in(
-> select email
-> from flight_booking
-> where flight_no in (
-> select flight_no
-> from flights
-> where place_from =     Pune
-> and place_to =     Bangalore     -> )
-> );
+----------+
| surname  |
+----------+
| Goldberg |
| Quinn    |
+----------+
2 rows in set (0.003 sec)
```

```
MariaDB [dbms_lab]>
MariaDB [dbms_lab]> /*6. Display the Passenger details only if they have booked flights
on 21 st July 2021. (Use Exists)*/
MariaDB [dbms_lab]> select * -> from passenger
-> where exists (
-> select email
-> from flight_booking
-> where flight_no in(
-> select flight_no
-> from flights
-> where departure_date =   2021  -07-27    -> )
-> );
+----------------+------------+----------+
|      email     | first_name  | surname  |
+----------------+------------+----------+
| beck@gmail.com |   Gwen     |   Beck   |
| joe@gmail.com  |   Joe      | Goldberg |
| love@gmail.com |   Love     | Quinn    |
+----------------+------------+----------+
3 rows in set (0.001 sec)
MariaDB [dbms_lab]> /*--7.Display the Flight-wise total time duration of flights if the
duration is more than 8 hours (Hint : Date function ,Aggregation ,Grouping)*/
MariaDB [dbms_lab]>
MariaDB [dbms_lab]> select flight_no , timediff(f.arrival_time , f.departure_time ) from
 flights as f where timediff(f.arrival_time, f.departure_time ) > "8:00:00" group by
flight_no;
+-----------+-----------------------------------------------+
| flight_no  | timediff(f.arrival_time, f.departure_time ) |
+-----------+-----------------------------------------------+
| 12345     | 11:47:44                                      |
| 23456     | 10:47:44                                      |
+-----------+-----------------------------------------------+
2 rows in set (0.001 sec)
MariaDB [dbms_lab]>
MariaDB [dbms_lab]> /*8.Display the Airplane-wise average seating capacity for any
airline*/ MariaDB [dbms_lab]> select name,
-> avg(capacity) -> from airplane
-> group by name;
+---------------+---------------+
| name          | avg(capacity) |
+---------------+---------------+
| Air India     | 175.0000      |
| Emirates      | 155.0000      |
| Qatar Airways | 183.0000      |
+---------------+---------------+
3 rows in set (0.001 sec)
MariaDB [dbms_lab]>
MariaDB [dbms_lab]> /*9.Display the total number of flights which are booked and
travelling to London airport.*/
MariaDB [dbms_lab]> select count(b.flight_no) as total -> from flight_booking b,
-> flights f
-> where f.place_to =    London   ;
+-------+
| total |
+-------+
|   3   |
+-------+
1 row in set (0.000 sec)
MariaDB [dbms_lab]>
MariaDB [dbms_lab]> /*10. Create a view having information about flight_no ,airplane_no
, capacity.*/
MariaDB [dbms_lab]> create view flightinfo as -> select f.flight_no ,
-> a.reg_no , -> a.capacity
-> from flights f, -> airplane a
-> where a.reg_no = f.reg_no; MariaDB [dbms_lab]>
MariaDB [dbms_lab]> select * from flightinfo;
+-----------+--------+----------+
| flight_no  | reg_no | capacity |
+-----------+--------+----------+
| 12345     | 111    | 180      |
| 67890     | 221    | 150      |
| 23456     | 333    | 200      |
+-----------+--------+----------+
3 rows in set (0.001 sec)
```

7 Conclusion

Thus, we have learned Subqueries commands thoroughly.

8 FAQ's

8.1 *Explain following types of subqueries*

- ***Single-row subquery***
- ***Multiple-row subquery***
- ***Multiple-column subquery***

Ans. In SQL, a subquery is a query that is nested inside another query, and it is used to retrieve data that will be used as a condition in the main query. A subquery can be classified into three types based on the number of rows and columns returned by the subquery: single-row subquery, multiple-row subquery, and multiple-column subquery.

Single-row subquery: A single-row subquery is a subquery that returns only one row of data. It is commonly used to retrieve a single value that will be used as a condition in the main query. The result of a single-row subquery can be compared using a single-value operator such as =, ¡, ¿, ¡=, ¿=, or ¡¿. Example:

```sql
SELECT product_name, price
FROM products
WHERE price = (SELECT MAX(price) FROM products);
```

This query returns the product name and price of the product with the highest price in the products table. The subquery returns a single value (the maximum price), which is used as a condition in the main query.

Multiple-row subquery: A multiple-row subquery is a subquery that returns multiple rows of data. It is commonly used to retrieve a set of values that will be used as a condition in the main query. The result of a multiple-row subquery can be compared using a multiple-value operator such as IN or NOT IN. Example:

```sql
SELECT product_name, price
FROM products
WHERE product_id IN (SELECT product_id FROM orders WHERE order_date = '2023-05-04');
```

This query returns the product name and price of all the products that have been ordered on the date '2023-05-04'. The subquery returns a set of values (the product IDs from the orders table), which are used as a condition in the main query.

Multiple-column subquery: A multiple-column subquery is a subquery that returns multiple columns and rows of data. It is commonly used to retrieve a set of values that will be used as a condition in the main query. The result of a multiple-column subquery can be compared using a multiple-column operator such as IN or NOT IN. Example:

```sql
SELECT customer_name
FROM customers
WHERE (customer_id, city) IN (SELECT customer_id, city FROM orders WHERE order_date = '
2023-05-04');
```

This query returns the names of all the customers who have placed an order on the date '2023-05-04' and live in the same city as the customer who placed the order. The subquery returns a set of values (the customer IDs and cities from the orders table), which are used as a condition in the main query.

8.2 *When subquery is used?*

Ans. A subquery is used when we need to retrieve data from one or more tables based on a condition that involves data from another table. Subqueries are often used in SQL to filter, sort, or group data based on some criteria.

Subqueries can be used in various ways, such as:

- **As a condition in a WHERE or HAVING clause:** Subqueries can be used to filter data based on some condition that involves data from another table. For example, we can use a subquery to retrieve all the orders placed by a specific customer:

```sql
        SELECT *
        FROM orders
        WHERE customer_id = (SELECT customer_id FROM customers WHERE customer_name = 'John Doe');


```

- **As a column expression in a SELECT statement:** Subqueries can be used to retrieve data that will be used as a column expression in the main query. For example, we can use a subquery to retrieve the total number of orders placed by each customer:

```sql
        SELECT customer_name, (SELECT COUNT(*) FROM orders WHERE orders.customer_id = customers.customer_id) AS num_orders
        FROM customers;

```

- **As a table expression in a FROM clause:** Subqueries can be used to create a virtual table that can be used in the main query. For example, we can use a subquery to retrieve all the products that have been ordered more than once:

```sql
        SELECT *
        FROM (SELECT product_id, COUNT(*) AS num_orders FROM order_details GROUP BY product_id) AS order_counts
        WHERE num_orders > 1;


```

Overall, subqueries provide a powerful way to manipulate data in SQL and are widely used in SQL queries to achieve complex operations.

8.3 *Explain SQL SubQueries with ALL, ANY, EXISTS, SOME, With UPDATE*

SQL subqueries are queries that are nested within other queries. They can be used in various ways to filter, sort, or group data based on some criteria. Here are some common types of SQL subqueries:

- **ALL:** The ALL operator is used to compare a value with all values returned by a subquery. For example, the following query retrieves all customers whose total orders are greater than the total orders of any customer from the UK:

```sql
        SELECT *
        FROM customers
        WHERE total_orders > ALL(SELECT total_orders FROM customers WHERE country = 'UK');

```

- **ANY/SOME:** The ANY or SOME operator is used to compare a value with any value returned by a subquery. For example, the following query retrieves all products that have been ordered by at least one customer:

```
        SELECT *
        FROM products
        WHERE product_id = ANY(SELECT product_id FROM order_details);


```

- **EXISTS:** The EXISTS operator is used to check if a subquery returns any rows. For example, the following query retrieves all customers who have placed at least one order:

```
        SELECT *
        FROM customers
        WHERE EXISTS(SELECT * FROM orders WHERE orders.customer_id = customers.
    customer_id);


```

- **UPDATE with subquery:** We can also use subqueries in an UPDATE statement to update values in a table based on data from another table. For example, the following query updates the order quantity of a product based on the number of units in stock:

```
        UPDATE products
        SET units_in_stock = units_in_stock - (SELECT SUM(quantity) FROM order_details
    WHERE order_details.product_id = products.product_id)


```

8.4 *How to get groupwise data from a table. What is use of Having Clause*

To get groupwise data from a table, we use the GROUP BY clause in SQL.The GROUP BY clause is used with aggregate functions like SUM, COUNT, MAX, MIN, and AVG to group the result-set by one or more columns. This helps to summarize data and generate reports based on some criteria.

For example, let's say we have a table called "sales" with columns "region", "product", and "sales_amount". To get the total sales amount for each region and product, we can use the following query:

```
    SELECT region, product, SUM(sales_amount) as total_sales
    FROM sales
    GROUP BY region, product;
```

The above query will group the result-set by "region" and "product" columns and calculate the total sales amount for each group using the SUM () function.

Now, to filter the groups based on some condition, we can use the HAVING clause. The HAVING clause is used with the GROUP BY clause to filter groups based on some condition.

For example, if we want to find the regions and products with total sales greater than 10000, we can use the following query:

```
    SELECT region, product, SUM(sales_amount) as total_sales
    FROM sales
    GROUP BY region, product
    HAVING total_sales > 10000;
```

In this query, the HAVING clause is used to filter the groups based on the condition "total_sales ¿ 10000". The groups that do not meet this condition will be excluded from the result-set.

8.5 *What is 'having' clause and when to use it?*

The HAVING clause in SQL is used to filter groups based on a condition. It is used in conjunction with the GROUP BY clause to specify conditions on the groups generated by the GROUP BY clause.

The HAVING clause is used to restrict the groups returned by the GROUP BY clause based on some condition. The condition specified in the HAVING clause is evaluated after the GROUP BY clause has grouped the rows into groups.

The HAVING clause is similar to the WHERE clause, but it operates on groups rather than individual rows. The WHERE clause is used to filter rows based on conditions, while the HAVING clause is used to filter groups based on conditions.

For example, let's say we have a table called "sales" with columns "region", "product", and "sales_amount". To get the total sales amount for each region and product, we can use the following query:

```
SELECT region, product, SUM(sales_amount) as total_sales
FROM sales
GROUP BY region, product;
```

This query will return the total sales amount for each region and product. However, if we want to filter out the groups where the total sales amount is less than 10000, we can use the HAVING clause as follows:

```
SELECT region, product, SUM(sales_amount) as total_sales
FROM sales
GROUP BY region, product
HAVING total_sales > 10000;
```

In this example, the HAVING clause is used to filter out the groups where the total sales amount is less than 10000. The groups that do not meet this condition will be excluded from the result-set.

In summary, the HAVING clause is used to filter groups based on some condition and is used in conjunction with the GROUP BY clause. It is useful in situations where you need to perform calculations on groups and then filter the results based on some criteria.

8.6 *How to display data from View. Are the views updatable? Explain*

To display data from a view in SQL, you can simply use a SELECT statement on the view, just like you would with a table. The syntax is:

```
SELECT column1, column2, ...
FROM view_name;
```

Where 'view_name' is the name of the view, and 'column1','column2', etcåre the names of the columns you want to select.

Views are virtual tables that are created by a query, so they do not actually store data themselves. Instead, they are defined by a SELECT statement that retrieves data from one or more tables. When you query a view, the database engine executes the underlying SELECT statement and returns the result set as if it were a table.

As for whether views are updatable, the answer is: it depends. In general, views that are based on a single table that has a primary key are updatable, but views that are based on multiple tables or complex queries may not be updatable.

Views can be updatable or not based on the following conditions:

- *The SELECT statement in the view must not contain any of the following:*
  - Aggregate functions such as COUNT, AVG, MAX, MIN, and SUM
  - DISTINCT keyword
  - GROUP BY or HAVING clauses
  - UNION or UNION ALL operators
  - Subqueries
  - TOP keyword
- The view must contain all the NOT NULL columns of the base table.

- If the view contains computed columns, the view must be updatable only if the columns are updatable.
- The view must not use the READONLY keyword.

If all of these conditions are met, then the view is updatable and you can use INSERT, UPDATE, and DELETE statements to modify the data in the view. If any of these conditions are not met, the view is not updatable and any attempts to modify the data will result in an error.