Lab 1: Illustration of SELECT, FROM, WHERE CLAUSE

Objectives: To illustrate the usage of SELECT, FROM, and WHERE clauses in SQL (Structured Query Language)

Syntax:

SELECT column1, column2 ……. column n

FROM table_name

WHERE condition;

Query:

SELECT *

FROM employees

WHERE department = 'IT';

Output:

```
mysql> SELECT *
    -> FROM employees
    -> WHERE department = 'IT';
+-------------+-----------------+------+------------+--------+
| employee_id | name            | age  | department | salary |
+-------------+-----------------+------+------------+--------+
|           1 | Krishna Basnet  |   30 | IT         |  50000 |
|           5 | Binita Paudel   |   32 | IT         |  52000 |
|           7 | Sabitra Subedi  |   24 | IT         |  55000 |
|           9 | Khusi Ghimire   |   23 | IT         |  42000 |
+-------------+-----------------+------+------------+--------+
4 rows in set (0.01 sec)

mysql>
```

Conclusion:

The SELECT, FROM, and WHERE clauses are fundamental components of SQL queries. The SELECT clause specifies the columns to retrieve, the FROM clause specifies the table to retrieve them from, and the WHERE clause filters the results based on specified conditions.

Lab 2: Illustration of SELECTING OF ALL COLUMNS.

Objective: To illustrate the query for selecting of all columns in SQL.

Syntax:

SELECT * FROM table_name;

Query:

SELECT *

FROM employees;

Output:

```
mysql> SELECT * FROM employees;
+-------------+-----------------+------+------------+--------+
| employee_id | name            | age  | department | salary |
+-------------+-----------------+------+------------+--------+
|           1 | Krishna Basnet  |   30 | IT         |  50000 |
|           2 | Bishal Adhikari |   25 | HR         |  45000 |
|           3 | Sudan Sharma    |   35 | Finance    |  60000 |
|           4 | Suraj B.K       |   28 | Marketing  |  48000 |
|           5 | Binita Paudel   |   32 | IT         |  52000 |
|           6 | Durga Adhikari  |   20 | Accounting |  35000 |
|           7 | Sabitra Subedi  |   24 | IT         |  55000 |
|           8 | Rabindra Dhami  |   32 | Marketing  |  43000 |
|           9 | Khusi Ghimire   |   23 | IT         |  42000 |
|          10 | Bipana Paudel   |   32 | Finance    |  35000 |
+-------------+-----------------+------+------------+--------+
10 rows in set (0.00 sec)
```

Conclusion:

The SELECT * statement is used to retrieve all columns from a table in SQL. It provides a convenient way to fetch all available data from a table without explicitly specifying each column.

Lab 3: Illustration of SELECTING OF SPECIFIC COLUMNS.

Objective: To illustrate the query for selecting of specific columns in SQL.

Syntax:

SELECT column1, column2, ...column n

FROM table_name;

Query:

SELECT name, age

FROM employees;

Output:

```
mysql> SELECT name, age FROM employees;
+-----------------+------+
| name            | age  |
+-----------------+------+
| Krishna Basnet  |   30 |
| Bishal Adhikari |   25 |
| Sudan Sharma    |   35 |
| Suraj B.K       |   28 |
| Binita Paudel   |   32 |
| Durga Adhikari  |   20 |
| Sabitra Subedi  |   24 |
| Rabindra Dhami  |   32 |
| Khusi Ghimire   |   23 |
| Bipana Paudel   |   32 |
+-----------------+------+
10 rows in set (0.00 sec)
```

Conclusion:

The SELECT statement in SQL allows us to retrieve specific columns from a table by specifying their names in the SELECT clause. This approach is useful when we only need certain information from the table and helps reduce unnecessary data retrieval, thereby improving query performance.

Lab 4: Illustration of ORDER BY Clause

Objective: To illustrate ORDER BY Clause in SQL.

Syntax:

SELECT column1, column2, … column n

FROM table_name

ORDER BY column1 [ASC | DESC], column2 [ASC | DESC], ...;

Query:

SELECT name, age

FROM employees

ORDER BY age ASC;

Output:

```
mysql> SELECT name, age
    -> FROM employees
    -> ORDER BY age ASC;
+-----------------+------+
| name            | age  |
+-----------------+------+
| Durga Adhikari  |   20 |
| Khusi Ghimire   |   23 |
| Sabitra Subedi  |   24 |
| Bishal Adhikari |   25 |
| Suraj B.K       |   28 |
| Krishna Basnet  |   30 |
| Binita Paudel   |   32 |
| Rabindra Dhami  |   32 |
| Bipana Paudel   |   32 |
| Sudan Sharma    |   35 |
+-----------------+------+
10 rows in set (0.01 sec)
```

Conclusion:

The ORDER BY clause in SQL allows us to sort the result set of a SELECT query based on one or more columns. It can be used to sort data in ascending (ASC) or descending (DESC) order.

Lab 5: Illustration of Arithmetic Operators.

Objectives: To illustrate Arithmetic Operators in SQL.

Syntax:

SELECT column,…, column (/,*,%,-,+) column AS new_column

FROM table_name;

Query:

SELECT employee_id, name, salary, salary * 1.1 AS Bonus

FROM employees;

Output:

```
mysql> SELECT employee_id, name, salary, salary * 1.1 AS Bonus
    -> FROM employees;
+-------------+-----------------+--------+---------+
| employee_id | name            | salary | Bonus   |
+-------------+-----------------+--------+---------+
|           1 | Krishna Basnet  |  50000 | 55000.0 |
|           2 | Bishal Adhikari |  45000 | 49500.0 |
|           3 | Sudan Sharma    |  60000 | 66000.0 |
|           4 | Suraj B.K       |  48000 | 52800.0 |
|           5 | Binita Paudel   |  52000 | 57200.0 |
|           6 | Durga Adhikari  |  35000 | 38500.0 |
|           7 | Sabitra Subedi  |  55000 | 60500.0 |
|           8 | Rabindra Dhami  |  43000 | 47300.0 |
|           9 | Khusi Ghimire   |  42000 | 46200.0 |
|          10 | Bipana Paudel   |  35000 | 38500.0 |
+-------------+-----------------+--------+---------+
10 rows in set (0.00 sec)

mysql>
```

Conclusion:

Arithmetic operators in SQL enable us to perform mathematical calculations within SELECT queries. They are useful for performing calculations on numeric data stored in the database tables.

Lab 6: Illustration of Operator Precedence in Arithmetic expression.

Objective: To illustrate the Operator Precedence in Arithmetic expression using SQL query.

Syntax:

SELECT <operand> OPERATOR (+, -, *, /, %) <operand>

Query:

```
SELECT
   SaleID, ProductID, EmployeeID,
   QuantitySold, SaleDate, TotalSaleAmount,
   ROUND((TotalSaleAmount - (TotalSaleAmount * 0.1)) * (1 + 0.05), 2) AS NetSaleAmount
FROM  Sales;
```

Output:

```
mysql> SELECT SaleID, ProductID, EmployeeID, QuantitySold, SaleDate,TotalSaleAmount,
    ->     ROUND((TotalSaleAmount - (TotalSaleAmount * 0.1)) * (1 + 0.05), 2) AS NetSaleAmount
    -> FROM Sales;
+--------+-----------+------------+--------------+------------+-----------------+---------------+
| SaleID | ProductID | EmployeeID | QuantitySold | SaleDate   | TotalSaleAmount | NetSaleAmount |
+--------+-----------+------------+--------------+------------+-----------------+---------------+
|      1 |         1 |          1 |            2 | 2024-04-03 |         1099.98 |       1039.48 |
|      2 |         2 |          3 |            4 | 2024-04-05 |         1199.98 |       1133.98 |
|      3 |         3 |          5 |            6 | 2024-04-03 |         1049.98 |        992.23 |
|      4 |         2 |          1 |            8 | 2024-04-05 |         2399.98 |       2267.98 |
|      5 |         4 |          2 |           10 | 2024-04-03 |         1299.98 |       1228.48 |
|      6 |         9 |          4 |            3 | 2024-04-08 |          539.98 |        510.28 |
|      7 |         8 |          6 |            6 | 2024-04-04 |          739.98 |        699.28 |
```

Conclusion:

Operator precedence in arithmetic expressions ensures that mathematical operations are evaluated correctly in SQL queries. We select TotalSaleAmount and perform arithmetic operation including different operators according to the precedence of operator to find the net sale amount.

Lab 7: Illustration of aggregate functions.

Objective: To illustrate the aggregate functions in SQL

Syntax:

SELECT Aggregate_function([DISTINCT|all]column)

FROM table name

WHERE condition

Query:

```
SELECT COUNT(*) AS TotalSalesTransactions ,
SUM(TotalSaleAmount) AS TotalRevenue,
MAX(TotalSaleAmount) AS HighestSale,
AVG(TotalSaleAmount) AS AverageSaleAmount
FROM Sales;
```

Output:

```
mysql> SELECT COUNT(*) AS TotalSalesTransactions ,
    -> SUM(TotalSaleAmount) AS TotalRevenue,
    -> MAX(TotalSaleAmount) AS HighestSale,
    -> AVG(TotalSaleAmount) AS AverageSaleAmount FROM Sales;
+-----------------------+--------------+-------------+-------------------+
| TotalSalesTransactions | TotalRevenue | HighestSale | AverageSaleAmount |
+-----------------------+--------------+-------------+-------------------+
|                     13 |     34585.74 |     8969.98 |       2660.441538 |
+-----------------------+--------------+-------------+-------------------+
1 row in set (0.01 sec)
```

Conclusion:

Hence, an aggregate function used in MYSQL command. We perform SUM(TotalSaleAmount) to find the total revenue which add all sale amount from that column. MAX(TotalSaleAmount) used to find the highest sale from the column. AVG(TotalSaleAmount) is used to find the average sale from the column TotalSaleAmount.

Lab 8: Illustration of GROUP BY clause

Objectives: To illustrate the GROUP BY clause in SQL

Syntax:

SELECT column1, column2,…..,columnn

FROM table name

WHERE condition

GROUP BY expression1, expression2,…..;

Query:

SELECT ProductID, SUM(QuantitySold) AS TotalQuantitySold,

SUM(TotalSaleAmount) AS TotalRevenue

FROM Sales GROUP BY ProductID;

Output:

```
mysql> SELECT ProductID, SUM(QuantitySold) AS TotalQuantitySold,
    -> SUM(TotalSaleAmount) AS TotalRevenue
    -> FROM Sales
    -> GROUP BY ProductID;
+-----------+-------------------+--------------+
| ProductID | TotalQuantitySold | TotalRevenue |
+-----------+-------------------+--------------+
|         1 |                 2 |      1099.98 |
|         2 |                12 |      3599.96 |
|         3 |                 6 |      1049.98 |
|         4 |                10 |      1299.98 |
|         6 |                 9 |      5199.98 |
|         8 |                 6 |       739.98 |
|         9 |                 3 |       539.98 |
|        10 |                12 |      7999.98 |
|        13 |                15 |      1299.98 |
|        14 |                 8 |      1239.98 |
|        19 |                 9 |      1545.98 |
|        24 |                13 |      8969.98 |
+-----------+-------------------+--------------+
12 rows in set (0.00 sec)
```

Conclusion:

Hence, the use of GROUP BY clause was illustrated in MYSQL. By applying the clause to the sales management database, we were able to group sales data based on product, providing valuable insights into the total quantity sold and revenue generated for each product.

Lab 9: Illustration of Restricting Group Results with the HAVING Clause

Objectives: To demonstrate the usage and functionality of the HAVING clause in SQL.

Syntax:

SELECT column1, column2, …, columnn

FROM table name

WHERE condition

GROUP BY expression1, expression2, …

HAVING condition;

Query:

SELECT ProductID, SUM(QuantitySold) AS TotalQuantitySold,

SUM(TotalSaleAmount) AS TotalRevenue

FROM Sales

GROUP BY ProductID

HAVING TotalQuantitySold > 3;

Output:

```
mysql> SELECT ProductID, SUM(QuantitySold) AS TotalQuantitySold,
    -> SUM(TotalSaleAmount) AS TotalRevenue
    -> FROM Sales
    -> GROUP BY ProductID
    -> HAVING TotalQuantitySold > 3;
+-----------+--------------------+--------------+
| ProductID | TotalQuantitySold | TotalRevenue |
+-----------+--------------------+--------------+
|         2 |                 12 |      3599.96 |
|         3 |                  6 |      1049.98 |
|         4 |                 10 |      1299.98 |
|         6 |                  9 |      5199.98 |
|         8 |                  6 |       739.98 |
|        10 |                 12 |      7999.98 |
|        13 |                 15 |      1299.98 |
|        14 |                  8 |      1239.98 |
|        19 |                  9 |      1545.98 |
|        24 |                 13 |      8969.98 |
+-----------+--------------------+--------------+
10 rows in set (0.01 sec)
```

Conclusion:

Hence, by applying the HAVING clause to the sales management database, we restricted the grouped results to only include products with a total quantity sold greater than 3.

Lab 10: Illustration of defining a NULL value

Objectives: To illustrate the concept of NULL values in SQL databases.

Syntax:

SELECT column_name

FROM table_name

WHERE column _name IS NULL;

Query:

SELECT *

FROM customers

WHERE Email IS NULL;

Output:

```
mysql> SELECT * FROM customers WHERE Email IS NULL;
+------------+-----------+----------+-------+-------------+
| CustomerID | FirstName | LastName | Email | PhoneNumber |
+------------+-----------+----------+-------+-------------+
|          9 | Sudan     | Subedi   | NULL  | 9864356754  |
|         10 | Bindu     | Chalise  | NULL  | 9876567654  |
+------------+-----------+----------+-------+-------------+
2 rows in set (0.00 sec)
```

Conclusion:

Hence, NULL values were defined and tuples having NULL values were accessed. We inserted a customer record with a NULL Email. Then, we used the IS NULL comparison operator in the query to retrieve customer records where the Email is NULL.

Lab 11: Illustration of using Column Aliases

Objectives: To demonstrate the usage of column aliases in SQL.

Syntax:

SELECT column_name AS column_alias_name

FROM table_name AS table_alias_name

Query:

SELECT SaleID, ProductID,

TotalSaleAmount AS Revenue

FROM Sales

WHERE TotalSaleAmount >1500.00;

Output:

```
mysql>
mysql> SELECT SaleID, ProductID, TotalSaleAmount AS Revenue FROM Sales
    -> WHERE TotalSaleAmount >1500.00;
+--------+-----------+---------+
| SaleID | ProductID | Revenue |
+--------+-----------+---------+
|      4 |         2 | 2399.98 |
|      8 |         6 | 5199.98 |
|      9 |        10 | 7999.98 |
|     12 |        19 | 1545.98 |
|     13 |        24 | 8969.98 |
+--------+-----------+---------+
5 rows in set (0.00 sec)
```

Conclusion:

Hence, column can be renamed using aliases via AS keyword. We used the AS keyword to provide a more descriptive name, "Revenue", for the calculated column representing the total sale amount.

Lab 12: Illustration of using Concatenation Operator

Objectives: To demonstrate the usage of the concatenation operator in SQL.

Syntax:

SELECT CONCAT (expression1 , ‘ ’ , expression2, …)

FROM table_name

Query:

SELECT CONCAT (FirstName, ‘ ’, LastName) as FullName

FROM new_employees;

Output:

```
mysql> SELECT CONCAT(FirstName,' ', LastName) as FullName
    -> FROM new_employees;
+----------------+
| FullName       |
+----------------+
| Rohan Gurung   |
| Sabita Basnet  |
| Ashish Devkota |
| Deeya Poudel   |
| Roshni Sharma  |
| Manoj Sharma   |
+----------------+
6 rows in set (0.00 sec)
```

Conclusion:

Hence, we illustrated the usage of the concatenation operator (CONCAT) in SQL. We concatenated the first name and last name of employees with a space in between and provided a meaningful alias, "FullName", for the concatenated column.

Lab 13: Illustration of using Literal Character Strings

Objectives: To demonstrate the usage of literal character strings in SQL.

Syntax:

Sequence of characters that are enclosed in single or double string:

<'CHARACTER_STRING'>

<"CHARACTER_STRING">

Query:

```
SELECT DepartmentName, 'Active' AS Status
FROM Departments;
```

Output:

```
mysql> SELECT DepartmentName, 'Active' AS Status FROM Departments;
+-----------------+--------+
| DepartmentName  | Status |
+-----------------+--------+
| Electronics     | Active |
| Clothing        | Active |
| Hardware        | Active |
| Home Appliances | Active |
| toys            | Active |
| Sporting Goods  | Active |
+-----------------+--------+
6 rows in set (0.00 sec)
```

Conclusion:

Hence, the literal character string was implemented. We selected all departments from the Departments table and provided a literal character string, 'Active', as a static column value with the alias "Status".

Lab 14: Illustration of Displaying Distinct Rows

Objectives: To demonstrate the usage of the DISTINCT keyword in SQL.

Syntax:

SELECT DISTINCT column1, column2, ……..columnn

FROM table name

Query:

SELECT DISTINCT *

FROM sales;

Output:

```
mysql> SELECT DISTINCT EmployeeID FROM sales;
+------------+
| EmployeeID |
+------------+
|          1 |
|          2 |
|          3 |
|          4 |
|          5 |
|          6 |
+------------+
6 rows in set (0.00 sec)
```

Conclusion:

Hence, we selected all distinct EmployeeID from the Sales table. The DISTINCT keyword eliminates duplicate rows from the result set, ensuring that only unique values are returned.

Lab 15: Illustration of Displaying Table Structures

Objectives: To demonstrate how to display the structure of database tables in SQL.

Syntax:

DESCRIBE table_name;

OR

DESC table_name;

Query:

DESCRIBE employees;

DESC customers;

Output:

```
mysql> describe employees;
+-------------+-------------+------+-----+---------+-------+
| Field       | Type        | Null | Key | Default | Extra |
+-------------+-------------+------+-----+---------+-------+
| employee_id | int         | NO   | PRI | NULL    |       |
| name        | varchar(50) | YES  |     | NULL    |       |
| age         | int         | YES  |     | NULL    |       |
| department  | varchar(20) | YES  |     | NULL    |       |
| salary      | int         | YES  |     | NULL    |       |
+-------------+-------------+------+-----+---------+-------+
5 rows in set (0.02 sec)

mysql> desc customers;
+-------------+--------------+------+-----+---------+----------------+
| Field       | Type         | Null | Key | Default | Extra          |
+-------------+--------------+------+-----+---------+----------------+
| CustomerID  | int          | NO   | PRI | NULL    | auto_increment |
| FirstName   | varchar(255) | NO   |     | NULL    |                |
| LastName    | varchar(255) | NO   |     | NULL    |                |
| Email       | varchar(255) | YES  |     | NULL    |                |
| PhoneNumber | varchar(25)  | YES  |     | NULL    |                |
+-------------+--------------+------+-----+---------+----------------+
5 rows in set (0.00 sec)
```

Conclusion:

We illustrated how to display the structure of database tables in SQL using the DESC command. By executing DESC followed by the table name, we can view the details of each column in the specified table, including the column name, data type, and any constraints.

Lab 16: Illustration of Using BETWEEN Operator

Objectives: To demonstrate the usage of the BETWEEN operator in SQL.

Syntax:

SELECT columns

FROM <table name>

WHERE <column_name> BETWEEN value1 AND value2;

Query:

SELECT *

FROM Employees

WHERE Salary BETWEEN 50000 AND 60000;

Output:

```
mysql> SELECT * FROM Employees WHERE Salary BETWEEN 50000 AND 60000;
+-------------+----------------+------+------------+--------+
| employee_id | name           | age  | department | salary |
+-------------+----------------+------+------------+--------+
|           1 | Krishna Basnet |   30 | IT         |  50000 |
|           3 | Sudan Sharma   |   35 | Finance    |  60000 |
|           5 | Binita Paudel  |   32 | IT         |  52000 |
|           7 | Sabitra Subedi |   24 | IT         |  55000 |
+-------------+----------------+------+------------+--------+
4 rows in set (0.00 sec)
```

Conclusion:

Hence, the BETWEEN operator was implemented as WHERE condition to find the records between certain two values for an attribute.

Lab 17: Illustration of Using IN Operator

Objectives: To demonstrate the usage of the IN operator in SQL.

Syntax:

SELECT columns

FROM <table name>

WHERE <column_name> IN(value1,value2,…..,valuen);

Query:

SELECT * FROM new_Employees WHERE DepartmentID IN

(SELECT DepartmentID FROM Departments WHERE DepartmentName IN ('Electronics', 'Clothing'));

Output:

```
mysql> SELECT * FROM new_Employees WHERE DepartmentID IN
    -> (SELECT DepartmentID FROM Departments WHERE DepartmentName IN ('Electronics', 'Clothing'));
+------------+-----------+----------+--------------+--------------------------+-------------+------------+
| EmployeeID | FirstName | LastName | DepartmentID | Email                    | PhoneNumber | HireDate   |
+------------+-----------+----------+--------------+--------------------------+-------------+------------+
|          1 | Rohan     | Gurung   |            1 | rohan.gurung@example.com | 9834836482  | 2023-01-10 |
|          2 | Sabita    | Basnet   |            2 | sabi.basnet@example.com  | 9857442364  | 2023-02-15 |
+------------+-----------+----------+--------------+--------------------------+-------------+------------+
2 rows in set (0.00 sec)
```

Conclusion:

Hence, we illustrated the usage of the IN operator in SQL. We selected employees from the Employees table whose department IDs belong to the Sales or HR departments using the IN operator.

Lab 18: Illustration of Using LIKE Operator

Objectives: To demonstrate the usage of the LIKE operator in SQL.

Syntax:

SELECT columns

FROM <table name>

WHERE <column_name> LIKE pattern;

Query:

SELECT EmployeeID, FirstName, LastName, DepartmentID

FROM new_employees WHERE LastName LIKE ‘S%’

Output:

```
mysql> SELECT EmployeeID, FirstName, LastName,DepartmentID
    -> FROM new_employees WHERE lastname LIKE 'S%';
+------------+-----------+----------+--------------+
| EmployeeID | FirstName | LastName | DepartmentID |
+------------+-----------+----------+--------------+
|          5 | Roshni    | Sharma   |            5 |
|          6 | Manoj     | Sharma   |            6 |
+------------+-----------+----------+--------------+
2 rows in set (0.00 sec)
```

Conclusion:

Hence, we illustrated the usage of the LIKE operator in SQL. we selected employees from the new_Employees table whose last names start with the letter 'S' using the LIKE operator with the pattern 'S%'.

Lab 19: Illustration of Using AND Operator

Objectives: To demonstrate the usage of the AND operator in SQL.

Syntax:

SELECT column1,column2,…

FROM <table name>

WHERE condition1 AND condition2 AND …;

Query:

SELECT *

FROM Employees

WHERE Salary BETWEEN 50000 AND 60000 AND DepartmentID = ‘IT’;

Output:

```
mysql> SELECT * FROM Employees WHERE Salary BETWEEN 50000 AND 60000 AND DEPARTMENT = 'IT';
+-------------+-----------------+------+------------+--------+
| employee_id | name            | age  | department | salary |
+-------------+-----------------+------+------------+--------+
|           1 | Krishna Basnet  |   30 | IT         |  50000 |
|           5 | Binita Paudel   |   32 | IT         |  52000 |
|           7 | Sabitra Subedi  |   24 | IT         |  55000 |
+-------------+-----------------+------+------------+--------+
3 rows in set (0.00 sec)
```

Conclusion:

Hence, we illustrated the usage of the AND operator in SQL. We selected employees from the Employees table who meet two conditions: their salary is between $50,000 and $60,000, and they belong to department (Department = ‘IT’).

Lab 20: Illustration of Using OR Operator

Objectives: To demonstrate the usage of the OR operator in SQL.

Syntax:

SELECT column1, column2,….

FROM <table name>

WHERE condition1 OR condition2 OR …

Query:

SELECT *

FROM Employees

WHERE Salary > 50000 OR age > 30;

Output:

```
mysql> SELECT * FROM Employees WHERE Salary > 50000 OR age > 30;
+-------------+---------------+------+------------+--------+
| employee_id | name          | age  | department | salary |
+-------------+---------------+------+------------+--------+
|           3 | Sudan Sharma  |   35 | Finance    |  60000 |
|           5 | Binita Paudel |   32 | IT         |  52000 |
|           7 | Sabitra Subedi|   24 | IT         |  55000 |
|           8 | Rabindra Dhami|   32 | Marketing  |  43000 |
|          10 | Bipana Paudel |   32 | Finance    |  35000 |
+-------------+---------------+------+------------+--------+
5 rows in set (0.00 sec)
```

Conclusion:

Hence, we illustrated the usage of the OR operator in SQL. We selected employees from the Employees table who meet one of two conditions: their salary is above $50,000, or their age is above 30.

Lab 21: Illustration of Using NOT Operator

Objectives: To demonstrate the usage of the NOT operator in SQL.

Syntax:

SELECT column1, column2,…

FROM <table name>

WHERE NOT condition;

Query:

SELECT *

FROM Employees

WHERE NOT Salary > 45000;

Output:

```
mysql> SELECT * FROM Employees WHERE NOT Salary > 45000;
+-------------+-----------------+------+------------+--------+
| employee_id | name            | age  | department | salary |
+-------------+-----------------+------+------------+--------+
|           2 | Bishal Adhikari |   25 | HR         |  45000 |
|           6 | Durga Adhikari  |   20 | Accounting |  35000 |
|           8 | Rabindra Dhami  |   32 | Marketing  |  43000 |
|           9 | Khusi Ghimire   |   23 | IT         |  42000 |
|          10 | Bipana Paudel   |   32 | Finance    |  35000 |
+-------------+-----------------+------+------------+--------+
5 rows in set (0.00 sec)
```

Conclusion:

Hence, we illustrated the usage of the NOT operator in SQL. We selected employees from the Employees table whose salary is not above $45,000 using the NOT operator with the condition Salary > 45000.

Lab 22: Illustration of Subquery

Objectives: To demonstrate the usage of subqueries in SQL.

Syntax:

SELECT column1, column2,…

FROM <table name>

WHERE <column_name> Comparison Operator, Relational Operator ALL| ANY| SOME| IN

(SELECT column1, column2,… FROM <table name> WHERE inner_condition);

Query:

SELECT *

FROM Employees

WHERE Salary > (SELECT AVG(Salary) FROM Employees);

Output:

```
mysql> SELECT * FROM Employees WHERE Salary > (SELECT AVG(Salary) FROM Employees);
+-------------+----------------+------+------------+--------+
| employee_id | name           | age  | department | salary |
+-------------+----------------+------+------------+--------+
|           1 | Krishna Basnet |   30 | IT         |  50000 |
|           3 | Sudan Sharma   |   35 | Finance    |  60000 |
|           4 | Suraj B.K      |   28 | Marketing  |  48000 |
|           5 | Binita Paudel  |   32 | IT         |  52000 |
|           7 | Sabitra Subedi |   24 | IT         |  55000 |
+-------------+----------------+------+------------+--------+
5 rows in set (0.01 sec)
```

Conclusion:

Hence, we illustrated the usage of subqueries in SQL. we selected employees from the Employees table whose salary is greater than the average salary of all employees. The subquery (SELECT AVG(Salary) FROM Employees) calculates the average salary of all employees, and the outer query selects employees with a salary greater than this average.

Lab 23: Illustration of CROSS JOIN

Objectives: To demonstrate the usage of CROSS JOIN in SQL

Syntax:

SELECT column_name_list

FROM <table_name1> CROSS JOIN <table_name2>

Query:

SELECT * FROM customers CROSS JOIN salestocustomers;

Output:

```
mysql> SELECT * FROM customers CROSS JOIN salestocustomers;
+------------+-----------+----------+---------------------------+-------------+--------+------------+
| CustomerID | FirstName | LastName | Email                     | PhoneNumber | SaleID | CustomerID |
+------------+-----------+----------+---------------------------+-------------+--------+------------+
|          1 | Deepak    | Neupane  | depak@example.com         | 9864356754  |      6 |          6 |
|          1 | Deepak    | Neupane  | depak@example.com         | 9864356754  |      5 |          5 |
|          1 | Deepak    | Neupane  | depak@example.com         | 9864356754  |      4 |          4 |
|          1 | Deepak    | Neupane  | depak@example.com         | 9864356754  |      3 |          3 |
|          1 | Deepak    | Neupane  | depak@example.com         | 9864356754  |      2 |          2 |
|          1 | Deepak    | Neupane  | depak@example.com         | 9864356754  |      1 |          1 |
|          2 | Mahendra  | Sharma   | sharma.mahi@example.com   | 9876567654  |      6 |          6 |
|          2 | Mahendra  | Sharma   | sharma.mahi@example.com   | 9876567654  |      5 |          5 |
|          2 | Mahendra  | Sharma   | sharma.mahi@example.com   | 9876567654  |      4 |          4 |
|          2 | Mahendra  | Sharma   | sharma.mahi@example.com   | 9876567654  |      3 |          3 |
|          2 | Mahendra  | Sharma   | sharma.mahi@example.com   | 9876567654  |      2 |          2 |
|          2 | Mahendra  | Sharma   | sharma.mahi@example.com   | 9876567654  |      1 |          1 |
|          3 | Ishan     | Bhandari | isan.fid@example.com      | 9876532234  |      6 |          6 |
|          3 | Ishan     | Bhandari | isan.fid@example.com      | 9876532234  |      5 |          5 |
|          3 | Ishan     | Bhandari | isan.fid@example.com      | 9876532234  |      4 |          4 |
|          3 | Ishan     | Bhandari | isan.fid@example.com      | 9876532234  |      3 |          3 |
|          3 | Ishan     | Bhandari | isan.fid@example.com      | 9876532234  |      2 |          2 |
|          3 | Ishan     | Bhandari | isan.fid@example.com      | 9876532234  |      1 |          1 |
|          4 | jenisha   | Sharma   | sharma.jenny@example.com  | 9807065676  |      6 |          6 |
|          4 | jenisha   | Sharma   | sharma.jenny@example.com  | 9807065676  |      5 |          5 |
|          4 | jenisha   | Sharma   | sharma.jenny@example.com  | 9807065676  |      4 |          4 |
|          4 | jenisha   | Sharma   | sharma.jenny@example.com  | 9807065676  |      3 |          3 |
|          4 | jenisha   | Sharma   | sharma.jenny@example.com  | 9807065676  |      2 |          2 |
|          4 | jenisha   | Sharma   | sharma.jenny@example.com  | 9807065676  |      1 |          1 |
|          5 | Binita    | Khattri  | benny.homa@example.com    | 9870987890  |      6 |          6 |
|          5 | Binita    | Khattri  | benny.homa@example.com    | 9870987890  |      5 |          5 |
```

Conclusion:

Hence, we illustrated the usage of CROSS JOIN in SQL. We performed a cross join between the customers and salestocustomers, resulting in all possible combinations of customers and selestocustomers.

Lab 24: Illustration of NATURAL JOIN

Objectives: To demonstrate the usage of NATURAL JOIN in SQL.

Syntax:

SELECT column_name_list

FROM <table_name1> NATURAL JOIN <table_name2>

Query:

SELECT EmployeeID, CONCAT(FirstName, ' ', LastName) AS FullName,

DepartmentID, Email, PhoneNumber

FROM new_Employees

NATURAL JOIN Departments;

Output:

```
mysql> SELECT EmployeeID, CONCAT(FirstName, ' ', LastName) AS FullName,
    ->  DepartmentID, Email, PhoneNumber FROM new_Employees
    ->  NATURAL JOIN Departments;
+------------+----------------+--------------+---------------------------+-------------+
| EmployeeID | FullName       | DepartmentID | Email                     | PhoneNumber |
+------------+----------------+--------------+---------------------------+-------------+
|          1 | Rohan Gurung   |            1 | rohan.gurung@example.com  | 9834836482  |
|          2 | Sabita Basnet  |            2 | sabi.basnet@example.com   | 9857442364  |
|          3 | Ashish Devkota |            3 | ashish.dev@example.com    | 9765342738  |
|          4 | Deeya Poudel   |            4 | deeya.poudel@example.com  | 9876543576  |
|          5 | Roshni Sharma  |            5 | roshni.sharma@example.com | 9786854533  |
|          6 | Manoj Sharma   |            6 | manoj.sharma@example.com  | 9764098743  |
+------------+----------------+--------------+---------------------------+-------------+
6 rows in set (0.00 sec)
```

Conclusion:

Hence, natural join between two tables can be done for tables having a common attribute among the tables. We selected DepartmentID as a common attribute between new_employees table and departments table.

Lab 25: Illustration of Creating JOINS with USING Clause

Objectives: To demonstrate the usage of JOINS with the USING clause in SQL.

Syntax:

SELECT column_name_list

FROM <table_name1> INNER JOIN <table_name2>

USING (common_column_name);

Query:

SELECT EmployeeID, CONCAT(FirstName, ' ', LastName) AS FullName,

DepartmentID, Email, PhoneNumber

FROM new_Employees

JOIN Departments USING (DepartmentID);

Output:

```
mysql> SELECT EmployeeID, CONCAT(FirstName, ' ', LastName) AS FullName,
    ->  DepartmentID, Email, PhoneNumber FROM new_Employees
    -> JOIN Departments USING (DepartmentID);
+------------+----------------+--------------+---------------------------+-------------+
| EmployeeID | FullName       | DepartmentID | Email                     | PhoneNumber |
+------------+----------------+--------------+---------------------------+-------------+
|          1 | Rohan Gurung   |            1 | rohan.gurung@example.com  | 9834836482  |
|          2 | Sabita Basnet  |            2 | sabi.basnet@example.com   | 9857442364  |
|          3 | Ashish Devkota |            3 | ashish.dev@example.com    | 9765342738  |
|          4 | Deeya Poudel   |            4 | deeya.poudel@example.com  | 9876543576  |
|          5 | Roshni Sharma  |            5 | roshni.sharma@example.com | 9786854533  |
|          6 | Manoj Sharma   |            6 | manoj.sharma@example.com  | 9764098743  |
+------------+----------------+--------------+---------------------------+-------------+
6 rows in set (0.00 sec)
```

Conclusion:

Hence, we performed a join between the new_Employees and Departments tables using the DepartmentID column.

Lab 26: Illustration of Creating JOINS with ON Clause

Objectives: To demonstrate the usage of JOINS with the ON clause in SQL.

SYNTAX:

SELECT column_name_list

FROM <table_name1> INNER JOIN <table_name2>

ON table1.column = table2.column;

Query:

SELECT e.EmployeeID, e.FirstName,

d.DepartmentID, e.Email, e.PhoneNumber

FROM new_Employees e

INNER JOIN Departments d ON

e.DepartmentID = d.DepartmentID;

Output:

```
mysql> SELECT e.EmployeeID, e.FirstName,
    ->  d.DepartmentID, e.Email, e.PhoneNumber FROM new_Employees e
    ->  INNER JOIN Departments d ON
    -> e.DepartmentID = d.DepartmentID;
+------------+-----------+--------------+--------------------------+-------------+
| EmployeeID | FirstName | DepartmentID | Email                    | PhoneNumber |
+------------+-----------+--------------+--------------------------+-------------+
|          1 | Rohan     |            1 | rohan.gurung@example.com | 9834836482  |
|          2 | Sabita    |            2 | sabi.basnet@example.com  | 9857442364  |
|          3 | Ashish    |            3 | ashish.dev@example.com   | 9765342738  |
|          4 | Deeya     |            4 | deeya.poudel@example.com | 9876543576  |
|          5 | Roshni    |            5 | roshni.sharma@example.com| 9786854533  |
|          6 | Manoj     |            6 | manoj.sharma@example.com | 9764098743  |
+------------+-----------+--------------+--------------------------+-------------+
6 rows in set (0.00 sec)
```

Conclusion:

Hence, we performed a join between the new_Employees and Departments tables using the DepartmentID column as the join condition specified in the ON clause.

Lab 27: Illustration of LEFT OUTER JOIN

Objectives: To demonstrate the usage of LEFT OUTER JOIN in SQL.

Syntax:

SELECT column_name_list

FROM <table_name1> LEFT OUTER JOIN <table_name2>

ON table1.column = table2.column;

Query:

SELECT e.EmployeeID, CONCAT(e.FirstName, ' ', e.LastName) AS FullName,

d.DepartmentID, e.Email, e.PhoneNumber

FROM new_Employees e

LEFT OUTER JOIN Departments d ON

e.DepartmentID = d.DepartmentID;

Output:

```
mysql> SELECT e.EmployeeID, CONCAT(e.FirstName, ' ', e.LastName) AS FullName,
    ->  d.DepartmentID, e.Email, e.PhoneNumber FROM new_Employees e
    -> LEFT OUTER JOIN Departments d ON
    -> e.DepartmentID = d.DepartmentID;
+------------+----------------+--------------+---------------------------+-------------+
| EmployeeID | FullName       | DepartmentID | Email                     | PhoneNumber |
+------------+----------------+--------------+---------------------------+-------------+
|          1 | Rohan Gurung   |            1 | rohan.gurung@example.com  | 9834836482  |
|          2 | Sabita Basnet  |            2 | sabi.basnet@example.com   | 9857442364  |
|          3 | Ashish Devkota |            3 | ashish.dev@example.com    | 9765342738  |
|          4 | Deeya Poudel   |            4 | deeya.poudel@example.com  | 9876543576  |
|          5 | Roshni Sharma  |            5 | roshni.sharma@example.com | 9786854533  |
|          6 | Manoj Sharma   |            6 | manoj.sharma@example.com  | 9764098743  |
|          7 | Hari Poudel    |         NULL | deeya.poudel@example.com  | 9876543576  |
|          8 | mandeep giri   |         NULL | roshni.sharma@example.com | 9786854533  |
|          9 | Ramesh kaphle  |         NULL | manoj.sharma@example.com  | 9764098743  |
+------------+----------------+--------------+---------------------------+-------------+
9 rows in set (0.00 sec)
```

Conclusion:

Hence, we performed a left outer join between the new_Employees and Departments tables. The LEFT OUTER JOIN retrieves all rows from the left table (new_Employees) and the matching rows from the right table (Departments), if any. If there is no match found in the right table, NULL values are included in the result set.

Lab 28: Illustration of RIGHT OUTER JOIN

Objectives: To demonstrate the usage of RIGHT OUTER JOIN in SQL.

Syntax:

SELECT column_name_list

FROM <table_name1>RIGHT OUTER JOIN <table_name2>

ON table1.column = table2.column;

Query:

SELECT e.EmployeeID, CONCAT(e.FirstName, ' ', e.LastName) AS FullName,

e.PhoneNumber, d.DepartmentID, d.DepartmentName FROM new_Employees e

RIGHT OUTER JOIN Departments d ON

e.DepartmentID = d.DepartmentID;

Output:

```
mysql> SELECT e.EmployeeID, CONCAT(e.FirstName, ' ', e.LastName) AS FullName,
    -> e.PhoneNumber, d.departmentID, d.departmentName FROM new_Employees e
    -> RIGHT OUTER JOIN Departments d ON
    -> e.DepartmentID = d.DepartmentID;
+------------+----------------+-------------+--------------+-----------------+
| EmployeeID | FullName       | PhoneNumber | departmentID | departmentName  |
+------------+----------------+-------------+--------------+-----------------+
|          1 | Rohan Gurung   | 9834836482  |            1 | Electronics     |
|          2 | Sabita Basnet  | 9857442364  |            2 | Clothing        |
|          3 | Ashish Devkota | 9765342738  |            3 | Hardware        |
|          4 | Deeya Poudel   | 9876543576  |            4 | Home Appliances |
|          5 | Roshni Sharma  | 9786854533  |            5 | toys            |
|          6 | Manoj Sharma   | 9764098743  |            6 | Sporting Goods  |
|       NULL | NULL           | NULL        |            7 | Medicine        |
|       NULL | NULL           | NULL        |            8 | Cusmetics       |
|       NULL | NULL           | NULL        |            9 | Vegetables      |
+------------+----------------+-------------+--------------+-----------------+
9 rows in set (0.00 sec)
```

Conclusion:

Hence, we performed a right outer join between the new_Employees and Departments tables. The RIGHT OUTER JOIN retrieves all rows from the right table (Departments) and the matching rows from the left table (new_Employees), if any. If there is no match found in the left table, NULL values are included in the result set.

Lab 29: Illustration of FULL OUTER JOIN

Objectives: To demonstrate the usage of FULL OUTER JOIN in SQL.

Syntax:

SELECT column_name_list FROM <table_name1>LEFT OUTER JOIN <table_name2> ON table1.column = table2.column UNION

SELECT column_name_list FROM <table_name1>RIGHT OUTER JOIN <table_name2>ON table1.column = table2.column;

Query:

SELECT * FROM new_employees e LEFT OUTER JOIN departments d ON e.DepartmentID = d. DepartmentID UNION

SELECT *FROM new_employees e RIGHT OUTER JOIN departments d ON e. DepartmentID = d. DepartmentID

Output:

```
mysql> SELECT *
    -> FROM new_employees e LEFT OUTER JOIN departments d
    -> ON e.DepartmentID = d.DepartmentID
    -> UNION
    -> SELECT *
    -> FROM new_employees e RIGHT OUTER JOIN departments d
    -> ON e.DepartmentID = d.DepartmentId;
+------------+-----------+----------+--------------+--------------------------+-------------+------------+--------------+-----------
------+-----------+
| EmployeeID | FirstName | LastName | DepartmentID | Email                    | PhoneNumber | HireDate   | DepartmentID | Department
Name  | ManagerID |
+------------+-----------+----------+--------------+--------------------------+-------------+------------+--------------+-----------
------+-----------+
|          1 | Rohan     | Gurung   |            1 | rohan.gurung@example.com | 9834836482  | 2023-01-10 |            1 | Electronic
s     |      NULL |
|          2 | Sabita    | Basnet   |            2 | sabi.basnet@example.com  | 9857442364  | 2023-02-15 |            2 | Clothing
      |      NULL |
|          3 | Ashish    | Devkota  |            3 | ashish.dev@example.com   | 9765342738  | 2023-01-10 |            3 | Hardware
      |      NULL |
|          4 | Deeya     | Poudel   |            4 | deeya.poudel@example.com | 9876543576  | 2023-02-15 |            4 | Home Appli
ances |      NULL |
|          5 | Roshni    | Sharma   |            5 | roshni.sharma@example.com | 9786854533  | 2023-01-10 |            5 | toys
      |      NULL |
|          6 | Manoj     | Sharma   |            6 | manoj.sharma@example.com | 9764098743  | 2023-02-15 |            6 | Sporting G
oods  |      NULL |
|          7 | Hari      | Poudel   |         NULL | deeya.poudel@example.com | 9876543576  | 2023-02-15 |         NULL | NULL
      |      NULL |
|          8 | mandeep   | giri     |         NULL | roshni.sharma@example.com | 9786854533  | 2023-01-10 |         NULL | NULL
      |      NULL |
|          9 | Ramesh    | kaphle   |         NULL | manoj.sharma@example.com | 9764098743  | 2023-02-15 |         NULL | NULL
      |      NULL |
|       NULL | NULL      | NULL     |         NULL | NULL                     | NULL        | NULL       |            7 | Medicine
      |      NULL |
```

Conclusion:

Hence, full outer join was implemented by union of left outer and right outer join in MYSQL

Lab 30: Illustration of Creating Table with Enforcement of Integrity Constraints PRIMARY KEY, NOT NULL, UNIQUE, CHECK, REFERENTIAL INTEGRITY.

Objective: To illustrate the creation of a table in a relational database system with the enforcement of various integrity constraints such as PRIMARY KEY, NOT NULL, UNIQUE, CHECK, and REFERENTIAL INTEGRITY.

Syntax:

```
CREATE TABLE <table_name>
(
column1 data_type(size) CONSTRAINT,
column2 data_type(size) CONSTRAINT,
. . . . . .
. . . . . .
columnn data_type(size) CONSTRAINT
);
```

Query:

```
CREATE TABLE Classes (
   ClassID INT NOT NULL,
   ClassName VARCHAR(50) NOT NULL,
   PRIMARY KEY (ClassID)
);

CREATE TABLE Students (
   StudentID INT NOT NULL,
   ClassID INT NOT NULL,
   FirstName VARCHAR(50) NOT NULL,
   LastName VARCHAR(50) NOT NULL,
   Email VARCHAR(100) NOT NULL UNIQUE,
   Age INT CHECK (Age >= 5 AND Age <= 18),
   PRIMARY KEY (StudentID, ClassID),
   FOREIGN KEY (ClassID) REFERENCES Classes(ClassID)
);
```

Output:

```
mysql> CREATE TABLE Students (
    ->     StudentID INT NOT NULL,
    ->     ClassID INT NOT NULL,
    ->     FirstName VARCHAR(50) NOT NULL,
    ->     LastName VARCHAR(50) NOT NULL,
    ->     Email VARCHAR(100) NOT NULL UNIQUE,
    ->     Age INT CHECK (Age >= 5 AND Age <= 18),
    ->     PRIMARY KEY (StudentID, ClassID),
    ->     FOREIGN KEY (ClassID) REFERENCES Classes(ClassID)
    -> );
Query OK, 0 rows affected (0.03 sec)

mysql> DESCRIBE Students;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| StudentID | int          | NO   | PRI | NULL    |       |
| ClassID   | int          | NO   | PRI | NULL    |       |
| FirstName | varchar(50)  | NO   |     | NULL    |       |
| LastName  | varchar(50)  | NO   |     | NULL    |       |
| Email     | varchar(100) | NO   | UNI | NULL    |       |
| Age       | int          | YES  |     | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
6 rows in set (0.00 sec)
```

Conclusion:

Hence, we successfully created a table named Students with the enforcement of integrity constraints including PRIMARY KEY, NOT NULL, CHECK, UNIQUE and REFERENTIAL INTEGRITY. These constraints ensure the accuracy, consistency, and reliability of data stored in the database, thereby maintaining data integrity