

Database Query

String Operation

The string operators in SQL are used to perform important operations such as pattern matching, concatenation, etc. where pattern matching is performed by using the wildcard characters such as '%' and '_' in concurrence with the Like operator to search for the specific patterns in strings and by the usage of concatenation operation one or more strings or columns of the tables can be combined together.

Types of Operator:

1. Concatenation Operator
2. Like Operator

String Operation

1. Concatenation Operator:

The concatenation operation is used to combine character strings, columns of a table or it can also be used for the combination of column and strings.

In the below example, the columns FIRSTNAME and LASTNAME of the table “STUDENTS” are combined with space in between the columns.

```
SELECT FIRSTNAME + ' ' + LASTNAME AS ConcatenatedName FROM STUDENTS;
```

But in Xampp server it is performed as

```
Select concat(FIRSTNAME,' ', LASTNAME ) as ConcatenatedName FROM STUDENTS;
```

String Operation

2. Like Operator

This operator is used to decide if the specific character string matches the specific pattern where the pattern can be a regular or wildcard character. While pattern matching, the regular characters should match exactly with the specific characters of the string but when we want to match the arbitrary fragments of the string, wildcard characters(%) can be used.

Example:

i. Pattern Matching

```
SELECT * FROM STUDENTS WHERE FIRSTNAME='Ram';
```

String Operation

2. Like Operator

ii. *SELECT * FROM STUDENTS WHERE FIRSTNAME LIKE 'p%';*

In the above query, the FIRSTNAME column is compared with the pattern 'p%' and then it finds the Student name which starts with 'p' as shown below

iii. *SELECT * FROM STUDENTS WHERE FIRSTNAME LIKE '%j';*

In the above query, it can be seen that the wildcard character % is used before 'j' and this will find the values which end with 'j'.

iv. *SELECT * FROM STUDENTS WHERE FIRSTNAME LIKE '%a%';*

In the above query, the values which matches the pattern in any position is found.

Ordering the Display of Tuples

- I. SQL offers the user some control over the order in which tuples in a relation are displayed. The **order by** clause causes the tuples in the result of a query to appear in sorted order.
- II. In SQL ORDER BY clause, we need to define ascending or descending order in which result needs to be sorted.
 - **ASC**: We can specify **ASC** to sort the result in ascending order
 - **DESC**: We can specify **DESC** to sort the result in descending order

Syntax:

```
1 SELECT * FROM table_name ORDER BY [column_name] ASC  
   |DESC
```

SQL Set Operation

- The set operators are available to combine information of similar type from one or more than one table.

Set Operation:

- I. **Union:** This set operator is used to combine the outputs of two or more queries into a single set of rows and columns having different records.

Syntax: `Select *from table1 union Select *from table2`

- II. **Union All :** This set operator is used to join the outputs of two or more queries into a single set of rows and columns without the removal of any duplicates.

Syntax: `Select *from table1 union all Select *from table2`

SQL Set Operation

iii. **Intersect:** This set operator is available to retrieve the information which is common in both tables. The number of columns and data type must be same in intersect set operator.

Syntax: Select *from table1 **intersect** Select *from table2

iv. **Minus:** This set operator is available to retrieve the information of one table which is not available in another table.

Syntax: Select *from table1 **minus** Select *from table2

Aggregate Functions

- An aggregate function in SQL performs a calculation on multiple values and returns a single value. [SQL](#) provides many aggregate functions that include avg, count, sum, min, max, etc. An aggregate function ignores NULL values when it performs the calculation, except for the count function.
- Various type of Aggregate Functions are:
 - I. Count()
 - II. Sum()
 - III. Avg()
 - IV. Min()
 - V. Max()

Aggregate Functions

I. Count()

- The count function returns the number of rows in the result. It does not count the null values.
- Example: Write a query to return number of rows where salary > 20000.

```
Select COUNT(*) from Employee where Salary > 20000;
```

```
Group By: Select *, COUNT(Name) from Employee Group By  
Name
```

Types –

- COUNT(*): Counts all the number of rows of the table including null.
- COUNT(COLUMN_NAME): count number of non-null values in column.
- COUNT(DISTINCT COLUMN_NAME): count number of distinct values in a column.

Aggregate Functions

ii. Sum()

- This function sums up the values in the column supplied as a parameter.
- Example: Write a query to get the total salary of employees.

Syntax: `Select Sum(Field_Name) from table1`

```
Select SUM(salary) from Employee
```

iii. Avg()

- This function returns the average value of the numeric column that is supplied as a parameter.
- Example: Write a query to select average salary from employee table.

Syntax: `Select AVG(Field_Name) from table1`

```
Select AVG(salary) from Employee
```

Aggregate Functions

iv. **MAX()** Function

- The MAX function is used to find maximum value in the column that is supplied as a parameter. It can be used on any type of data.
- Example – Write a query to find the maximum salary in employee table

```
Select MAX(salary) from Employee
```

Nested Subqueries

- A subquery is a “query with a query”
- Subquery is query appear within WHERE or HAVING clause of other query.
- Outer query is called as main query and inner query which is written in main query is called subquery.
- Subquery in the WHERE clause: The result of the subquery is used to select some rows from main query.
- Subquery in the HAVING clause: The result of the subquery is used to select some groups from main query.
- Sub queries can be nested within other sub queries.

Nested Subqueries

Syntax:

```
SELECT select_list  
FROM table  
WHERE exp_operator  
(SELECT select_list FROM table);
```

Example: Find all employee whose salary is less than Ram's salary.

```
SELECT ename, salary  
FROM employee  
WHERE salary <  
(SELECT salary FROM employee WHERE ename='Ram');
```

Set Membership

1. IN

- The subquery forms the set membership test(IN) matches a test values returned by a subquery.
- This multiple row operator used to check that a given tuple in main query satisfy at least one values returned by a subquery.
- Inversely we can use NOT IN condition to check test value is not member of result set of inner subquery.
- Example: Find out all employees having salary not equal to any of the manager's salary he should not be manager.

```
SELECT emp_id, name, job_id, salary
FROM employee
WHERE salary NOT IN
      (SELECT salary FROM employee WHERE job_id='MAN')
AND <>'MAN';
```

Set Membership

2. ANY

- This multiple row operator used to check that a given tuple in main query satisfies given condition for at least one values returned by a subquery.
- Checking to see whether the comparison holds true for some values of subquery.
- Example: Find out all employee having salary less than any of the managers he should not be manager.

```
SELECT emp_id, Name, job_id, salary
FROM employee
WHERE salary < ANY
      (SELECT salary FROM employee WHERE job_id='MAN')
AND job_id <> 'MAN';
```


Set Membership

3. ALL

- This multiple row operator used to check that a given tuple in main query satisfies given condition for all values returned by a subquery.
- This quantified tests use one of the simple comparison operators to compare a test value to all of the values returned by a subquery, checking to see whether the comparison holds for all values.
- Example: Find out all employees having salary less than all managers and he should not be manager/

```
SELECT emp_id, name, salary
FROM employees
WHERE salary < ALL
      (SELECT salary FROM employees WHERE job_id='MAN')
AND job_id <> 'MAN';
```

This

Set Membership

4. EXISTS Clause

- The existence test(EXISTS) check whether a subquery returns any values or return nothing.
- EXISTS clause is used for testing whether a subquery has any tuples in there result set or is it empty.
- The EXISTS clause result True value if the subquery is nonempty else it return in False value.
- Example:

```
SELECT name  
FROM table2  
WHERE EXISTS  
(SELECT *FROM table1 WHERE table1.name=table2.name);
```

Test for Empty Relations

- The **exists** construct returns **true** if the argument subquery is nonempty.
- Find all customers who have a loan **and** an account at the bank.

```
SELECT cname
```

```
FROM borrower
```

```
WHERE EXISTS (SELECT *FROM depositor WHERE depositor.cname=  
borrower.cname);
```

Test for the Absence of Duplicate Tuples

- SQL includes a Boolean function for testing whether a subquery has duplicate tuples in its result. The **unique** construct returns the value **true** if the argument subquery contains no duplicate tuples.
- Syntax:

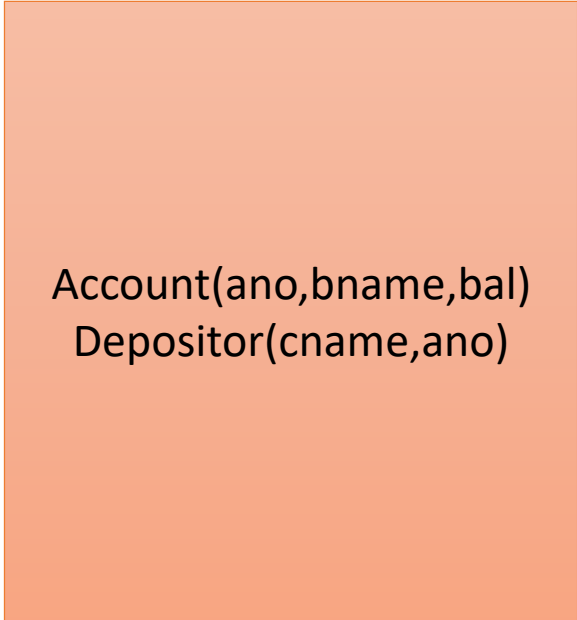
```
SELECT table.ID FROM table WHERE UNIQUE (SELECT table2.ID  
FROM table2 WHERE table.ID = table2.ID);
```

Test for the Absence of Duplicate Tuples

Example:

Find all customer who have at most one account at the 'Pokhara' branch.

```
SELECT d.cname from depositor as d
WHERE UNIQUE(SELECT DD.cname
FROM account as a, depositor as dd
WHERE d.cname=dd.cname
AND dd.ano=a.ano
AND a.bname='Pokhara');
```



Account(ano,bname,bal)
Depositor(cname,ano)

Derived Relations

- A **derived** table is a subquery nested within a FROM clause.
- Sql allows a subquery expression to be used in the **from** clause. If such an expression is used, then the result relation must be given a name, and the attributes can be renamed. We accomplish this renaming by using the **as** clause.
- **Example:**
SELECT branch_name, avg_balance FROM
(SELECT branch_name, AVG(balance))
FROM depositor
GROUP BY branch_name AS result(branch_name, avg_balance) WHERE avg_balance>120000.
- This subquery generates a relation consisting of the name of all branches and their corresponding average account balance. This temporary relation is named result, with the attributes branch_name and avg_balance.

Views

- Views in SQL are kind of virtual tables. A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database. A View can either have all the rows of a table or specific rows based on certain condition.
- **Advantages:**
 1. **Views** don't store data in a physical location.
 2. The view can be used to hide some of the columns from table.
 3. Views can provide Access Restriction, since data insertion, update and deletion is not possible with the view.

Views

- Database views are created using the **CREATE VIEW** statement. Views can be created from a single table, multiple tables or another view.

The basic **CREATE VIEW** syntax is as follows –

```
CREATE VIEW view_name AS SELECT column1, column2..... FROM  
table_name WHERE [condition];
```

Following is an example to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

```
CREATE VIEW CUSTOMERS_VIEW AS SELECT name, age FROM CUSTOMERS;
```

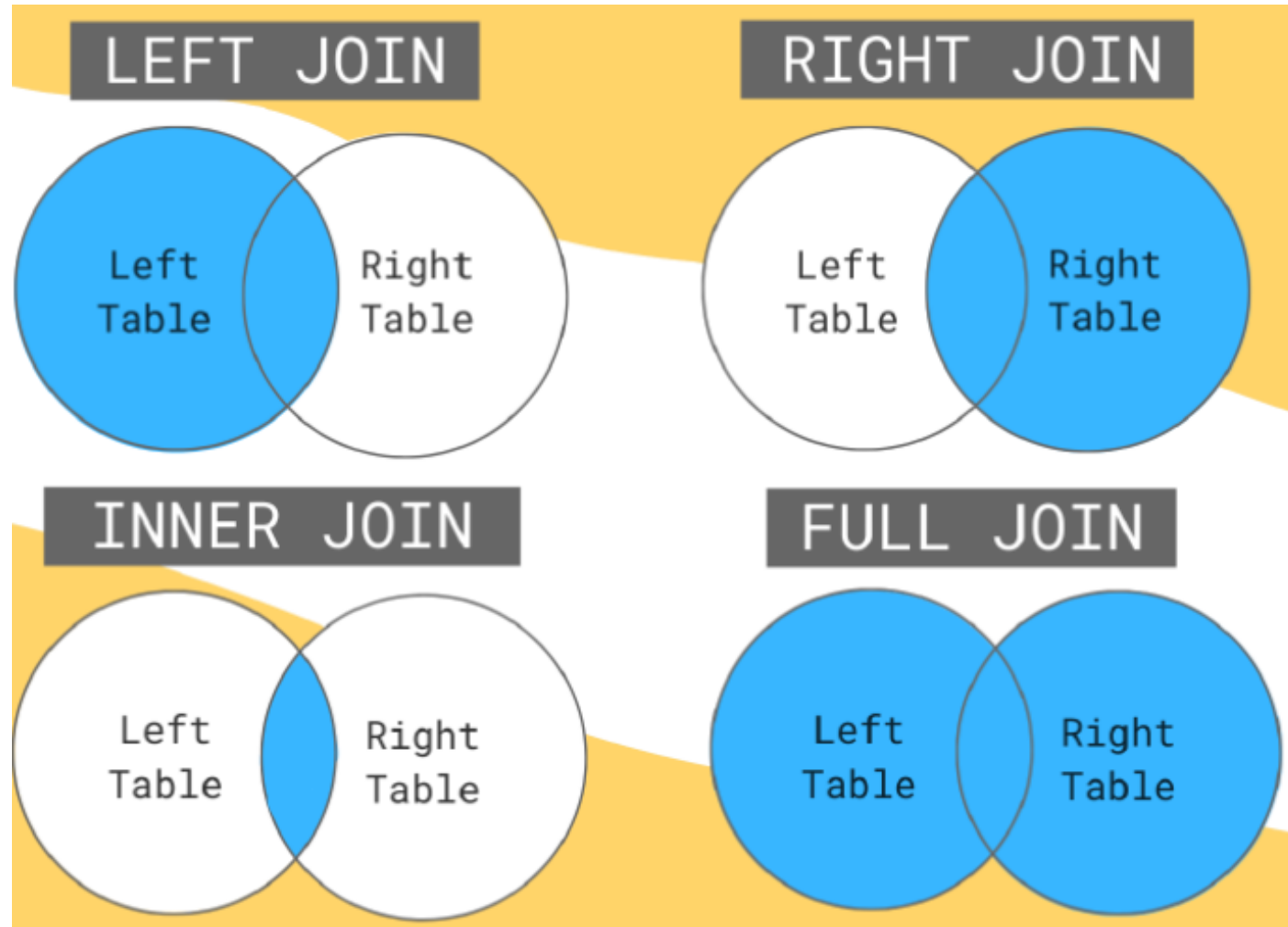
Now, you can query CUSTOMERS_VIEW in a similar way as you query an actual table. Following is an example for the same.

```
SELECT * FROM CUSTOMERS_VIEW;
```


Join

- The SQL **Joins** clause is used to combine records from two or more tables in a database. A JOIN is a means for combining fields from two tables by using values common to each.
- Types of Join
 - I. Inner Join
 - II. Left Join
 - III. Right Join
 - IV. Full Join

Join



Join

I. Inner Join

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

Syntax:

```
SELECT table1.column1, table2.column2... FROM table1 INNER JOIN  
table2 ON table1.common_field = table2.common_field;
```

Example:

```
SELECT Orders.OrderID, Customers.CustomerName  
FROM Orders  
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID;
```

Join

II. Left Join

The **LEFT JOIN** keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is Null records from the right side, if there is no match.

Syntax:

```
SELECT column_name(s)  
FROM table1  
LEFT JOIN table2  
ON table1.common_column_name = table2.common_column_name;
```

Example:

```
SELECT Customers.CustomerName, Orders.OrderID  
FROM Customers  
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID ;
```

Join

III. Right Join

The **RIGHT JOIN** keyword returns all records from the right table (table2), and the matching records from the left table (table1). The result is 0 records from the left side, if there is no match.

Syntax:

```
SELECT column_name(s)  
FROM table1  
RIGHT JOIN table2  
ON table1.common_column_name = table2.common_column_name;
```

Example:

```
SELECT Orders.OrderID, Employees.LastName, Employees.FirstName  
FROM Orders  
RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID;
```

Join

IV. Full Join

The FULL OUTER JOIN keyword returns all record when there is a match in left(table1) or right(table2) table records.

Syntax:

```
SELECT column_name(s)  
FROM table1  
FULL OUTER JOIN table2  
ON table1.common_column_name = table2.common_column_name  
WHERE condition;
```

Example:

```
SELECT Customers.CustomerName, Orders.OrderID  
FROM Customers  
FULL OUTER JOIN Orders ON Customers.CustomerID=Orders.CustomerID;
```

