Rakamin Academy

HOME CREDIT

Virtual Internship Experience

# Querying : SQL Operation

**Basic Joining (Inner, Outer, Left, Right, Full Outer), Aliasing, Grouping (Group By, Having, Rollup), Querying, Sorting, dan Filtering (SELECT, SELECT DISTINCT, WHERE, OR, AND, LIKE, LIMIT, ORDER BY, etc), SubQuerying, Set Operators (Union, Union All, Intersect, Minus), Create, Read, Update (Alter), Delete (Drop) Table**

## 1. Join (Inner, Left, Right and Full Joins)

A SQL Join statement is used to combine data or rows from two or more tables based on a common field between them. Different types of Joins are:
INNER JOIN
LEFT JOIN
RIGHT JOIN
FULL JOIN
Consider the two tables below:

| ROLL_NO | NAME | ADDRESS | PHONE | Age |
|---|---|---|---|---|
| 1 | HARSH | DELHI | xxxxxxxxxx | 18 |
| 2 | PRATIK | BIHAR | xxxxxxxxxx | 19 |
| 3 | RIYANKA | SILIGURI | xxxxxxxxxx | 20 |
| 4 | DEEP | RAMNAGAR | xxxxxxxxxx | 18 |
| 5 | SAPTARHI | KOLKATA | xxxxxxxxxx | 19 |
| 6 | DHANRAJ | BARABAJAR | xxxxxxxxxx | 20 |
| 7 | ROHIT | BALURGHAT | xxxxxxxxxx | 18 |
| 8 | NIRAJ | ALIPUR | xxxxxxxxxx | 19 |

| COURSE_ID | ROLL_NO |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 2 | 3 |
| 3 | 4 |
| 1 | 5 |
| 4 | 9 |
| 5 | 10 |
| 4 | 11 |

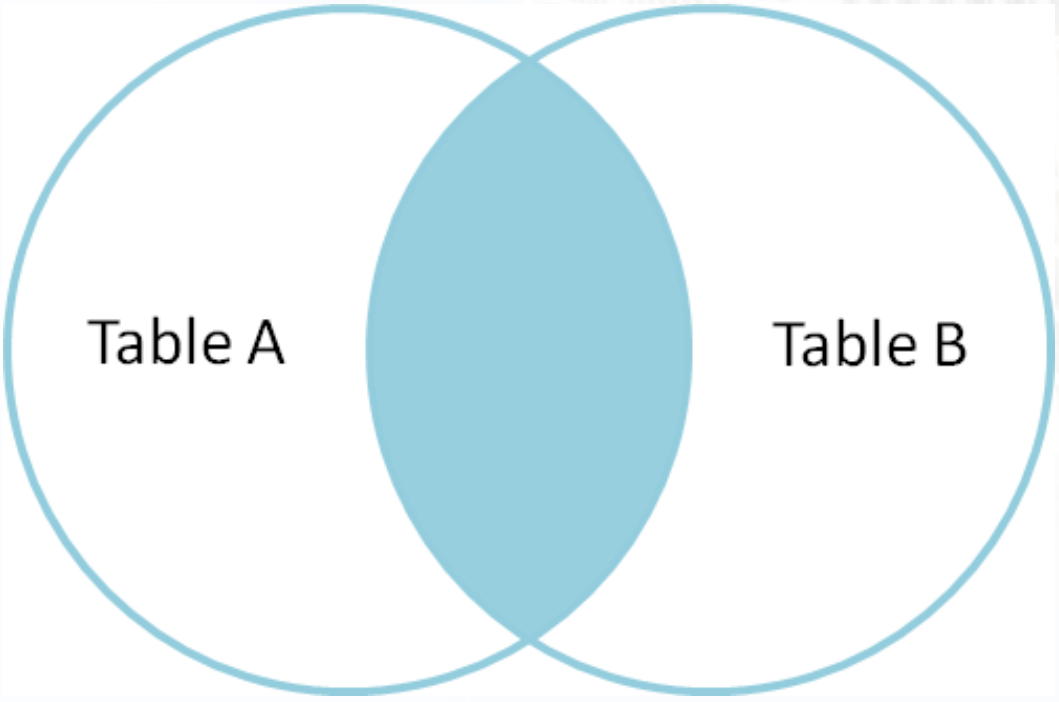
## INNER JOIN

INNER JOIN: The INNER JOIN keyword selects all rows from both the tables as long as the condition satisfies. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be same.
Syntax:

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


table1: First table.
table2: Second table
matching_column: Column common to both the tables.
```

Note: We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.




### Example Queries(INNER JOIN)

This query will show the names and age of students enrolled in different courses

```
SELECT StudentCourse.COURSE_ID, Student.NAME, Student.AGE FROM
INNER JOIN StudentCourse
ON Student.ROLL_NO = StudentCourse.ROLL_NO;
```

Output:

| COURSE_ID | NAME | Age |
|---|---|---|
| 1 | HARSH | 18 |
| 2 | PRATIK | 19 |
| 2 | RIYANKA | 20 |
| 3 | DEEP | 18 |
| 1 | SAPTARHI | 19 |

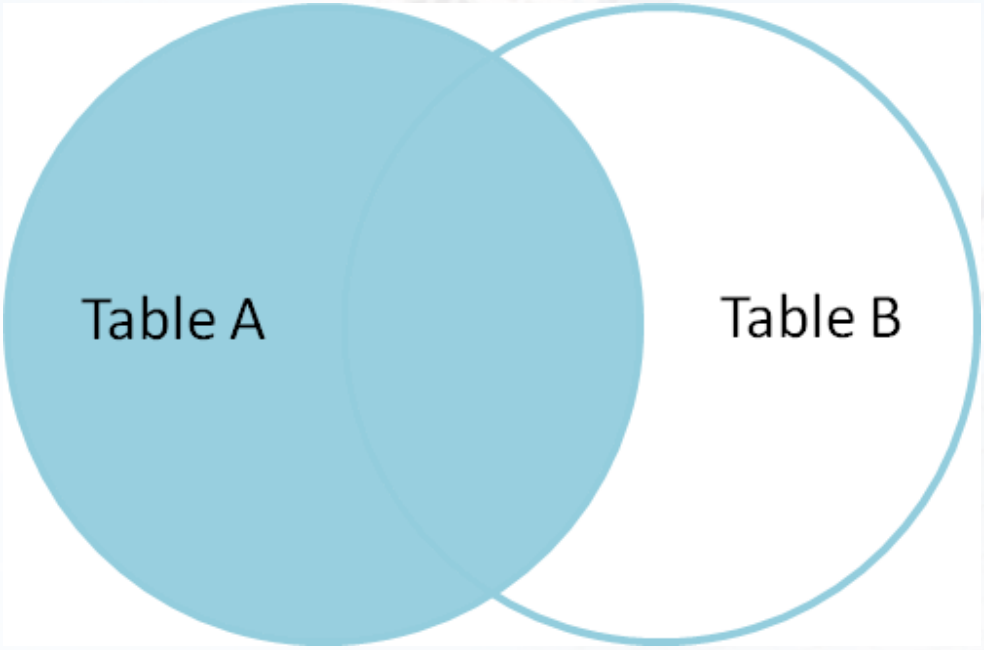
## LEFT JOIN

LEFT JOIN: This join returns all the rows of the table on the left side of the join and matching rows for the table on the right side of join. The rows for which there is no matching row on right side, the result-set will contain null. LEFT JOIN is also known as LEFT OUTER JOIN.Syntax:

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
LEFT JOIN table2
ON table1.matching_column = table2.matching_column;


table1: First table.
table2: Second table
matching_column: Column common to both the tables.
```

Note: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are same.




Example Queries(LEFT JOIN):

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
LEFT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output:

| NAME | COURSE_ID |
|---|---|
| HARSH | 1 |
| PRATIK | 2 |
| RIYANKA | 2 |
| DEEP | 3 |
| SAPTARHI | 1 |
| DHANRAJ | *NULL* |
| ROHIT | *NULL* |
| NIRAJ | *NULL* |

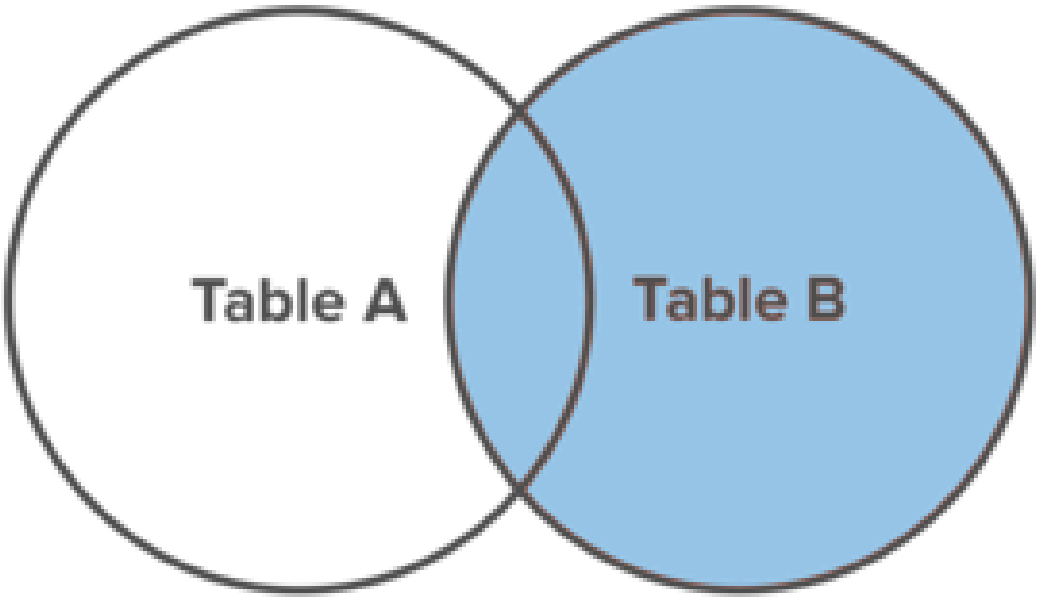
## RIGHT JOIN

RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of join. The rows for which there is no matching row on left side, the result-set will contain null. RIGHT JOIN is also known as RIGHT OUTER JOIN.Syntax:

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
RIGHT JOIN table2
ON table1.matching_column = table2.matching_column;


table1: First table.
table2: Second table
matching_column: Column common to both the tables.
```

Note: We can also use RIGHT OUTER JOIN instead of RIGHT JOIN, both are same.



Example Queries(RIGHT JOIN):

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
RIGHT JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output:

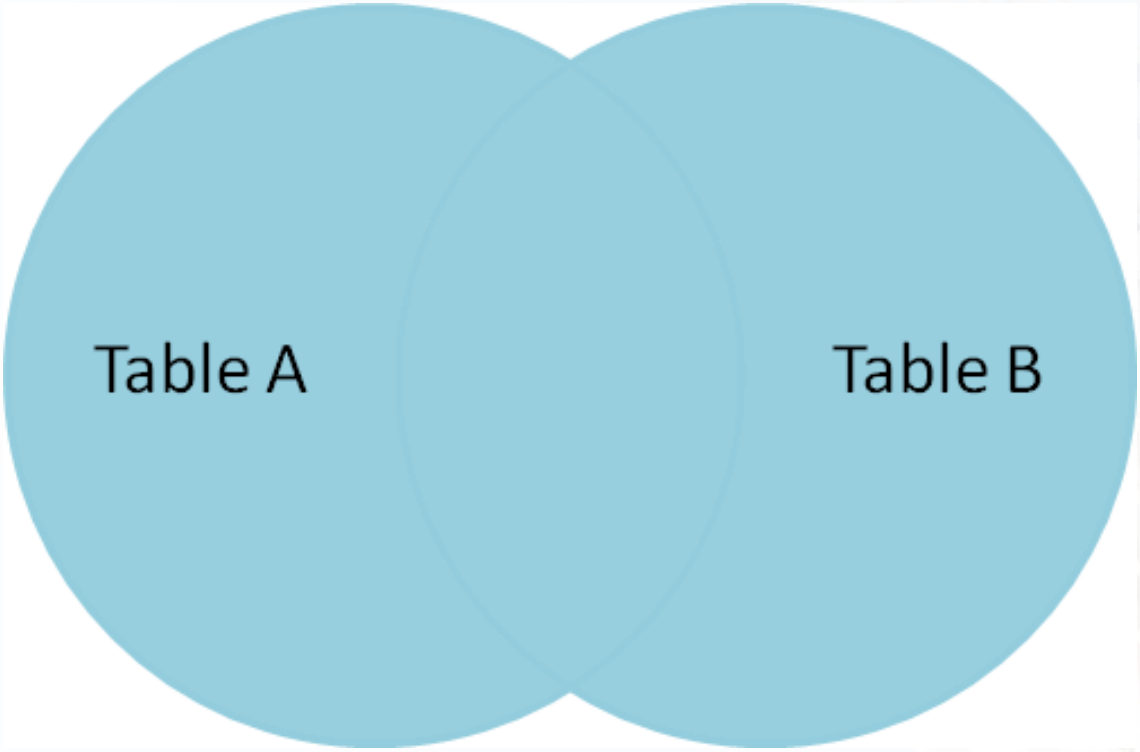
| NAME | COURSE_ID |
|---|---|
| HARSH | 1 |
| PRATIK | 2 |
| RIYANKA | 2 |
| DEEP | 3 |
| SAPTARHI | 1 |
| *NULL* | 4 |
| *NULL* | 5 |
| *NULL* | 4 |

## FULL JOIN

FULL JOIN creates the result-set by combining result of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both the tables. The rows for which there is no matching, the result-set will contain NULL values.Syntax:

```
SELECT table1.column1,table1.column2,table2.column1,....
FROM table1
FULL JOIN table2
ON table1.matching_column = table2.matching_column;


table1: First table.
table2: Second table
matching_column: Column common to both the tables.
```




Example Queries(FULL JOIN):

```
SELECT Student.NAME,StudentCourse.COURSE_ID
FROM Student
FULL JOIN StudentCourse
ON StudentCourse.ROLL_NO = Student.ROLL_NO;
```

Output:

| NAME | COURSE_ID |
|---|---|
| HARSH | 1 |
| PRATIK | 2 |
| RIYANKA | 2 |
| DEEP | 3 |
| SAPTARHI | 1 |
| DHANRAJ | *NULL* |
| ROHIT | *NULL* |
| NIRAJ | *NULL* |
| *NULL* | 9 |
| *NULL* | 10 |
| *NULL* | 11 |

## 2. Aliasing

Aliases are the temporary names given to table or column for the purpose of a particular SQL query. It is used when name of column or table is used other than their original names, but the modified name is only temporary.

- Aliases are created to make table or column names more readable.
- The renaming is just a temporary change and table name does not change in the original database.
- Aliases are useful when table or column names are big or not very readable.
- These are preferred when there are more than one table involved in a query.

Basic Syntax:

1. For column alias:

```
SELECT column as alias_name FROM table_name;
column: fields in the table
alias_name: temporary alias name to be used in replacement of original column name
table_name: name of table
```

2. For table alias:

```
SELECT column FROM table_name as alias_name;
column: fields in the table
table_name: name of table
alias_name: temporary alias name to be used in replacement of original table name
```

**Student_Details**

| ROLL_NO | Branch | Grade |
|---|---|---|
| 1 | Information Technology | O |
| 2 | Computer Science | E |
| 3 | Computer Science | O |
| 4 | Mechanical Engineering | A |

Queries for illustrating column alias

- To fetch ROLL_NO from Student table using CODE as alias name.

```
SELECT ROLL_NO AS CODE FROM Student;
```

Output:

| CODE |
|---|
| 1 |
| 2 |
| 3 |
| 4 |

- To fetch Branch using Stream as alias name and Grade as CGPA from table Student_Details.

```
SELECT Branch AS Stream,Grade as CGPA FROM Student_Details;
```

Output:

| Stream | CGPA |
|---|---|
| Information Technology | O |
| Computer Science | E |
| Computer Science | O |
| Mechanical Engineering | A |

Queries for illustrating table alias

**Student**

| ROLL_NO | NAME | ADDRESS | PHONE | Age |
|---|---|---|---|---|
| 1 | Ram | Delhi | XXXXXXXXXX | 18 |
| 2 | RAMESH | GURGAON | XXXXXXXXXX | 18 |
| 3 | SUJIT | ROHTAK | XXXXXXXXXX | 20 |
| 4 | SURESH | Delhi | XXXXXXXXXX | 18 |

Generally table aliases are used to fetch the data from more than just single table and connect them through the field relations.

- To fetch Grade and NAME of Student with Age = 20

```
SELECT s.NAME, d.Grade FROM Student AS s, Student_Details
AS d WHERE s.Age=20 AND s.ROLL_NO=d.ROLL_NO;
```

Output:

| NAME | Grade |
|---|---|
| SUJIT | O |

## 3. Grouping (Group By, Having, Rollup)

### Group By

The GROUP BY Statement in SQL is used to arrange identical data into groups with the help of some functions. i.e if a particular column has same values in different rows then it will arrange these rows in a group.

Important Points:

- GROUP BY clause is used with the SELECT statement.
- In the query, GROUP BY clause is placed after the WHERE clause.
- In the query, GROUP BY clause is placed before ORDER BY clause if used any.

Syntax:

```
SELECT column1, function_name(column2)
FROM table_name
WHERE condition
GROUP BY column1, column2
ORDER BY column1, column2;

function_name: Name of the function used for example, SUM() , AVG().
table_name: Name of the table.
condition: Condition used.
```

Sample Table:

| SI NO | NAME | SALARY | AGE |
|---|---|---|---|
| 1 | Harsh | 2000 | 19 |
| 2 | Dhanraj | 3000 | 20 |
| 3 | Ashish | 1500 | 19 |
| 4 | Harsh | 3500 | 19 |
| 5 | Ashish | 1500 | 19 |

| SUBJECT | YEAR | NAME |
|---|---|---|
| English | 1 | Harsh |
| English | 1 | Pratik |
| English | 1 | Ramesh |
| English | 2 | Ashish |
| English | 2 | Suresh |
| Mathematics | 1 | Deepak |
| Mathematics | 1 | Sayan |

Example:

- Group By single column: Group By single column means, to place all the rows with same value of only that particular column in one group. Consider the query as shown below:

```
SELECT NAME, SUM(SALARY) FROM Employee
GROUP BY NAME;
```

The above query will produce the below output:

| NAME | SALARY |
|---|---|
| Ashish | 3000 |
| Dhanraj | 3000 |
| Harsh | 5500 |

As you can see in the above output, the rows with duplicate NAMEs are grouped under same NAME and their corresponding SALARY is the sum of the SALARY of duplicate rows. The SUM() function of SQL is used here to calculate the sum.

- Group By multiple columns: Group by multiple column is say for example, GROUP BY column1, column2. This means to place all the rows with same values of both the columns column1 and column2 in one group. Consider the below query:

```
SELECT SUBJECT, YEAR, Count(*)
FROM Student
GROUP BY SUBJECT, YEAR;
```

Output:

| SUBJECT | YEAR | Count |
|---|---|---|
| English | 1 | 3 |
| English | 2 | 2 |
| Mathematics | 1 | 2 |

As you can see in the above output the students with both same SUBJECT and YEAR are placed in same group. And those whose only SUBJECT is same but not YEAR belongs to different groups. So here we have grouped the table according to two columns or more than one column.

## Having

In MSSQL, the HAVING clause is used to apply a filter on the result of GROUP BY based on the specified condition. The conditions are Boolean type i.e. use of logical operators(AND, OR). This clause was included in SQL as the WHERE keyword failed when we use it with aggregate expressions. Having is a very generally used clause in SQL. Similar to WHERE it helps to apply conditions, but HAVING works with groups. If you wish to filter a group, the HAVING clause comes into action.

Some important points:

- Having clause is used to filter data according to the conditions provided.
- Having clause is generally used in reports of large data.
- Having clause is only used with the SELECT clause.
- The expression in the syntax can only have constants.
- In the query, ORDER BY is to be placed after the HAVING clause, if any.
- HAVING Clause implements in column operation.
- Having clause is generally used after GROUP BY.
- The GROUP BY clause is used to arrange required data into groups.

Syntax:

```
SELECT col_1, function_name(col_2)
FROM tablename
WHERE condition
GROUP BY column1, column2
HAVING Condition
ORDER BY column1, column2;
```

Here, the function_name is the name of the function used, for example, SUM(), AVG().

Example 1:

Here first we create a database name as "Company", then we will create a table named "Employee" in the database. After creating a table we will execute the query.

Step 1: Creating a database

```
CREATE DATABASE Company;
```

Step 2: To use this database

```
USE Company;
```

Step 3: Creating table

```
CREATE TABLE Employee(
   EmployeeId int,
   Name Varchar(20),
   Gender Varchar(20),
   Salary int,
   Department Varchar(20),
   Experience Varchar(20)
);
```

Add value into the table:

```
INSERT INTO Employee VALUES (1, 'Rachit', 'M', 50000, 'Engineering', '6 year')
INSERT INTO Employee VALUES (2, 'Shobit', 'M', 37000, 'HR', '3 year')
INSERT INTO Employee VALUES (3, 'Isha', 'F', 56000, 'Sales', '7 year')
INSERT INTO Employee VALUES (4, 'Devi', 'F', 43000, 'Management', '4 year')
INSERT INTO Employee VALUES (5, 'Akhil', 'M', 90000, 'Engineering', '15 year')
```

The final table is:

```
SELECT * FROM Employee;
```

Results Messages

| | EmployeeId | Name | Gender | Salary | Department | Experience |
|---|---|---|---|---|---|---|
| 1 | 1 | Rachit | M | 50000 | Engineering | 6 year |
| 2 | 2 | Shobit | M | 37000 | HR | 3 year |
| 3 | 3 | Isha | F | 56000 | Sales | 7 year |
| 4 | 4 | Devi | F | 43000 | Management | 4 year |
| 5 | 5 | Akhil | M | 90000 | Engineering | 15 year |

Step 4: Execute the query

This employee table will help us understand the HAVING Clause. It contains employee IDs, Name, Gender, department, and salary. To Know the sum of salaries, we will write the query:

```
SELECT Department, sum(Salary) as Salary
FROM employee
GROUP BY department;
```

Here is the result,

Results Messages

| | Department | Salary |
|---|---|---|
| 1 | Engineering | 140000 |
| 2 | HR | 37000 |
| 3 | Management | 43000 |
| 4 | Sales | 56000 |

Now if we need to display the departments where the sum of salaries is 50,000 or more. In this condition, we will use HAVING Clause.

```
SELECT Department, sum(Salary) as Salary
FROM employee
GROUP BY department
HAVING SUM(Salary) >= 50000;
```

Results Messages

| | Department | Salary |
|---|---|---|
| 1 | Engineering | 140000 |
| 2 | Sales | 56000 |

Example 2:

Suppose, a teacher wants to announce the toppers in class. For this, she decides to reward every student who scored more than 95%. We need to group the database by name and their percentage and find out who scored more than 95% in that year. So for this first, we create a database name as "School", and then we will create a table named "Student" in the database. After creating a table we will execute the query.

Step 1: Creating a database

```
CREATE DATABASE School;
```

Step 2: To use this database

```
USE School;
```

Step 3: Creating table

```
CREATE TABLE Student(
   student Varchar(20),
   percentage int
);
```

Add value into the table:

```
INSERT INTO Student VALUES ('Isha Patel', 98)
INSERT INTO Student VALUES ('Harsh Das', 94)
INSERT INTO Student VALUES ('Rachit Sha', 93)
INSERT INTO Student VALUES ('Sumedha', 98)
INSERT INTO Student VALUES ('Rahat Ali', 98)
```

The final table is:

```
SELECT * FROM Student;
```

Results Messages

| | student | percentage |
|---|---|---|
| 1 | Isha Patel | 98 |
| 2 | Harsh Das | 94 |
| 3 | Rachit Sha | 93 |
| 4 | Sumedha | 98 |
| 5 | Rahat Ali | 98 |

Step 4: Execute Query

```
SELECT student, percentage
FROM Student
GROUP BY student, percentage
HAVING percentage > 95;
```

Here, three students named Isha, Sumedha, Rahat Ali have scored more than 95 %.

.

Results Messages

| | student | percentage |
|---|---|---|
| 1 | Isha Patel | 98 |
| 2 | Rahat Ali | 98 |
| 3 | Sumedha | 98 |

Further, we can also filter rows on multiple values using the HAVING clause. The HAVING clause also permits filtering rows using more than one aggregate condition.

```
SELECT student
FROM Student
WHERE percentage > 90
GROUP BY student, percentage
HAVING SUM(percentage) < 1000 AND AVG(percentage) > 95;
```

This query returns the students who have more percentage than 95 and the sum of percentage is less than 1000.

Results Messages

| | student |
|---|---|
| 1 | Isha Patel |
| 2 | Rahat Ali |
| 3 | Sumedha |

## RollUp

The PostgreSQL ROLLUP belongs to the GROUP BY clause that provides a short cut for defining multiple grouping sets. Multiple columns grouped together forms a group set.

Unlike the CUBE subclause, ROLLUP does not yield all possible grouping sets based on the specified columns. It just makes a subset of those. The ROLLUP presupposes a hierarchy between the input columns and yields all grouping sets that make sense only if the hierarchy is considered. That's why ROLLUP is usually used to generate the subtotals and the grand total for reports.

Syntax:

```
SELECT
    column1,
    column2,
    column3,
    aggregate(column4)
FROM
    table_name
GROUP BY
    ROLLUP (column1, column2, column3);
```

To better understand the concept let's create a new table and proceed to the examples.
To create a sample table use the below command:

```
CREATE TABLE geeksforgeeks_courses(
    course_name VARCHAR NOT NULL,
    segment VARCHAR NOT NULL,
    quantity INT NOT NULL,
    PRIMARY KEY (course_name, segment)
);
```

Now insert some data into it using the below command:

```
INSERT INTO geeksforgeeks_courses(course_name, segment, quantity)
VALUES
    ('Data Structure in Python', 'Premium', 100),
    ('Algorithm Design in Python', 'Basic', 200),
    ('Data Structure in Java', 'Premium', 100),
    ('Algorithm Design in Java', 'Basic', 300);
```

Now that our table is set let's look into examples.

Example 1:
The following query uses the ROLLUP subclause to find the number of products sold by course_name(subtotal) and by all course_name and segments (total) as follows:

```
SELECT
    course_name,
    segment,
    SUM (quantity)
FROM
    geeksforgeeks_courses
GROUP BY
    ROLLUP (course_name, segment)
ORDER BY
    course_name,
    segment;
```

Output:

```
postgres=# SELECT
postgres-#     course_name,
postgres-#     segment,
postgres-#     SUM (quantity)
postgres-# FROM
postgres-#     geeksforgeeks_courses
postgres-# GROUP BY
postgres-#     ROLLUP (course_name, segment)
postgres-# ORDER BY
postgres-#     course_name,
postgres-#     segment;
        course_name         | segment | sum
----------------------------+---------+-----
 Algorithm Design in Java   | Basic   | 300
 Algorithm Design in Java   |         | 300
 Algorithm Design in Python | Basic   | 200
 Algorithm Design in Python |         | 200
 Data Structure in Java     | Premium | 100
 Data Structure in Java     |         | 100
 Data Structure in Python   | Premium | 100
 Data Structure in Python   |         | 100
                            |         | 700
(9 rows)
```

Example 2:
The following statement performs a partial ROLL UP as follows:

```
SELECT
    segment,
    course_name,
    SUM (quantity)
FROM
    geeksforgeeks_courses
GROUP BY
    segment,
    ROLLUP (course_name)
ORDER BY
    segment,
    course_name;
```

Output:

```
postgres=# SELECT
postgres-#     segment,
postgres-#     course_name,
postgres-#     SUM (quantity)
postgres-# FROM
postgres-#     geeksforgeeks_courses
postgres-# GROUP BY
postgres-#     segment,
postgres-#     ROLLUP (course_name)
postgres-# ORDER BY
postgres-#     segment,
postgres-#     course_name;
 segment |        course_name          | sum
---------+-----------------------------+-----
 Basic   | Algorithm Design in Java    | 300
 Basic   | Algorithm Design in Python  | 200
 Basic   |                             | 500
 Premium | Data Structure in Java      | 100
 Premium | Data Structure in Python    | 100
 Premium |                             | 200
(6 rows)
```

## 4. Querying, Sorting, dan Filtering (SELECT, SELECT DISTINCT, WHERE, OR, AND, LIKE, LIMIT, ORDER BY, etc)

### Select Querying

Select is the most commonly used statement in SQL. The SELECT Statement in SQL is used to retrieve or fetch data from a database. We can fetch either the entire table or according to some specified rules. The data returned is stored in a result table. This result table is also called result-set.

With the SELECT clause of a SELECT command statement, we specify the columns that we want to be displayed in the query result and, optionally, which column headings we prefer to see above the result table.

The select clause is the first clause and is one of the last clauses of the select statement that the database server evaluates. The reason for this is that before we can determine what to include in the final result set, we need to know all of the possible columns that could be included in the final result set.

Sample Table:

**Student**

| ROLL_NO | NAME | ADDRESS | PHONE | Age |
|---|---|---|---|---|
| 1 | Ram | Delhi | XXXXXXXXXX | 18 |
| 2 | RAMESH | GURGAON | XXXXXXXXXX | 18 |
| 3 | SUJIT | ROHTAK | XXXXXXXXXX | 20 |
| 4 | SURESH | Delhi | XXXXXXXXXX | 18 |

Basic Syntax:

```
SELECT column1,column2 FROM table_name
column1 , column2: names of the fields of the table
table_name: from where we want to fetch
```

This query will return all the rows in the table with fields column1 , column2.

- To fetch the entire table or all the fields in the table:

```
SELECT * FROM table_name;
```

- Query to fetch the fields ROLL_NO, NAME, AGE from the table Student:

```
SELECT ROLL_NO, NAME, AGE FROM Student;
```

Output:

| ROLL_NO | NAME | Age |
|---|---|---|
| 1 | Ram | 18 |
| 2 | RAMESH | 18 |
| 3 | SUJIT | 20 |
| 4 | SURESH | 18 |

## Select Distinct

The distinct keyword is used in conjunction with select keyword. It is helpful when there is a need of avoiding duplicate values present in any specific columns/table. When we use distinct keyword only the unique values are fetched.

Syntax :

```
SELECT DISTINCT column1, column2
FROM table_name
```

column1, column2 : Names of the fields of the table.
table_name : Table from where we want to fetch the records.

This query will return all the unique combinations of rows in the table with fields column1, column2.

NOTE: If distinct keyword is used with multiple columns, the distinct combination is displayed in the result set.

Table – Student

| ROLL_NO | NAME | ADDRESS | PHONE | AGE |
|---|---|---|---|---|
| 1 | RAM | DELHI | XXXXXXXXXX | 18 |
| 2 | RAMESH | GURGAON | XXXXXXXXXX | 18 |
| 3 | SUJIT | ROHTAK | XXXXXXXXXX | 20 |
| 4 | SURESH | DELHI | XXXXXXXXXX | 18 |
| 3 | SUJIT | ROHTAK | XXXXXXXXXX | 20 |
| 2 | RAMESH | GURGAON | XXXXXXXXXX | 18 |

Queries

- To fetch unique names from the NAME field –

```
SELECT DISTINCT NAME
FROM Student;
```

Output:

| NAME |
|---|
| Ram |
| RAMESH |
| SUJIT |
| SURESH |

- To fetch a unique combination of rows from the whole table –

```
SELECT DISTINCT *
FROM Student;
```

Output:

| ROLL_NO | NAME | ADDRESS | PHONE | Age |
|---|---|---|---|---|
| 1 | Ram | Delhi | XXXXXXXXX | 18 |
| 2 | RAMESH | GURGAON | XXXXXXXXX | 18 |
| 3 | SUJIT | ROHTAK | XXXXXXXXX | 20 |
| 4 | SURESH | Delhi | XXXXXXXXX | 18 |

Note : Without the keyword distinct in both the above examples 6 records would have been fetched instead of 4, since in the original table there are 6 records with the duplicate values.

## Where

WHERE keyword is used for fetching filtered data in a result set.

- It is used to fetch data according to a particular criteria.
- WHERE keyword can also be used to filter data by matching patterns.

Basic Syntax:

SELECT column1,column2 FROM table_name WHERE column_name operator value;

```
column1 , column2: fields int the table
table_name: name of table
column_name: name of field used for filtering the data
operator: operation to be considered for filtering
value: exact value or pattern to get related data in result
```

List of operators that can be used with where clause:

| perator | description |
|---|---|
| > | Greater Than |
| >= | Greater than or Equal to |
| < | Less Than |
| <= | Less than or Equal to |
| = | Equal to |
| <> | Not Equal to |
| BETWEEN | In an inclusive Range |
| LIKE | Search for a pattern |
| IN | To specify multiple possible values for a colum |

**Student**

| ROLL_NO | NAME | ADDRESS | PHONE | Age |
|---|---|---|---|---|
| 1 | Ram | Delhi | xxxxxxxxxx | 18 |
| 2 | RAMESH | GURGAON | xxxxxxxxxx | 18 |
| 3 | SUJIT | ROHTAK | xxxxxxxxxx | 20 |
| 4 | SURESH | Delhi | xxxxxxxxxx | 18 |
| 3 | SUJIT | ROHTAK | xxxxxxxxxx | 20 |
| 2 | RAMESH | GURGAON | xxxxxxxxxx | 18 |

- To fetch record of students with age equal to 20

```
SELECT * FROM Student WHERE Age=20;
```

Output:

| ROLL_NO | NAME | ADDRESS | PHONE | Age |
|---|---|---|---|---|
| 3 | SUJIT | ROHTAK | XXXXXXXXXX | 20 |
| 3 | SUJIT | ROHTAK | XXXXXXXXXX | 20 |

- To fetch Name and Address of students with ROLL_NO greater than 3

```
SELECT ROLL_NO,NAME,ADDRESS FROM Student WHERE ROLL_NO > 3;
```

Output:

| ROLL_NO | NAME | ADDRESS |
|---|---|---|
| 4 | SURESH | Delhi |

## Between

It is used to fetch filtered data in a given range inclusive of two values.

Basic Syntax:

SELECT column1,column2 FROM table_name WHERE column_name BETWEEN value1 AND value2;

```
BETWEEN: operator name
```

value1 AND value2: exact value from value1 to value2 to get related data in result set.

Queries

- To fetch records of students where ROLL_NO is between 1 and 3 (inclusive)

```
SELECT * FROM Student WHERE ROLL_NO BETWEEN 1 AND 3;
```

Output:

| ROLL_NO | NAME | ADDRESS | PHONE | Age |
|---|---|---|---|---|
| 1 | Ram | Delhi | XXXXXXXXXX | 18 |
| 2 | RAMESH | GURGAON | XXXXXXXXXX | 18 |
| 3 | SUJIT | ROHTAK | XXXXXXXXXX | 20 |
| 3 | SUJIT | ROHTAK | XXXXXXXXXX | 20 |

## Like

It is used to fetch filtered data by searching for a particular pattern in where clause.

Basic Syntax:

SELECT column1,column2 FROM table_name WHERE column_name LIKE pattern;

```
LIKE: operator name
```

pattern: exact value extracted from the pattern to get related data in

result set.

Note: The character(s) in pattern are case sensitive.

Queries

- To fetch records of students where NAME starts with letter S.

```
SELECT * FROM Student WHERE NAME LIKE 'S%';
```

The '%'(wildcard) signifies the later characters here which can be of any length and value.More about wildcards will be discussed in the later set.

Output:

| ROLL_NO | NAME | ADDRES | PHONE | Age |
|---|---|---|---|---|
| 3 | SUJIT | ROHTAK | XXXXXXXXXX | 20 |
| 4 | SURESH | Delhi | XXXXXXXXXX | 18 |

## IN operator

It is used to fetch filtered data same as fetched by '=' operator just the difference is that here we can specify multiple values for which we can get the result set.

Basic Syntax:

SELECT column1,column2 FROM table_name WHERE column_name IN (value1,value2,..);

```
IN: operator name
```

value1,value2,...: exact value matching the values given and get related data in result set.

Queries

- To fetch NAME and ADDRESS of students where Age is 18 or 20

```
SELECT NAME,ADDRESS FROM Student WHERE Age IN (18,20);
```

Output:

| NAME | ADDRESS |
|---|---|
| Ram | Delhi |
| RAMESH | GURGAON |
| SUJIT | ROHTAK |
| SURESH | Delhi |
| SUJIT | ROHTAK |
| RAMESH | GURGAON |

## OR

This operator displays the records where either one of the conditions condition1 and condition2 evaluates to True. That is, either condition1 is True or condition2 is True.

Syntax:

```
SELECT * FROM table_name WHERE condition1 OR condition2 OR... conditionN;

table_name: name of the table
condition1,2,..N : first condition, second condition and so on
```

Now, we consider a table database to demonstrate OR operators with multiple cases:

**Student**

| ROLL_NO | NAME | ADDRESS | PHONE | Age |
|---|---|---|---|---|
| 1 | Ram | Delhi | xxxxxxxxxx | 18 |
| 2 | RAMESH | GURGAON | xxxxxxxxxx | 18 |
| 3 | SUJIT | ROHTAK | xxxxxxxxxx | 20 |
| 4 | SURESH | Delhi | xxxxxxxxxx | 18 |
| 3 | SUJIT | ROHTAK | xxxxxxxxxx | 20 |
| 2 | RAMESH | GURGAON | xxxxxxxxxx | 18 |

To fetch all the records from the Student table where NAME is Ram or NAME is SUJIT.

Query:

```
SELECT * FROM Student WHERE NAME = 'Ram' OR NAME = 'SUJIT';
```

Output:

| ROLL_NO | NAME | ADDRESS | PHONE | Age |
|---|---|---|---|---|
| 1 | Ram | Delhi | XXXXXXXXXX | 18 |
| 3 | SUJIT | ROHTAK | XXXXXXXXXX | 20 |
| 3 | SUJIT | ROHTAK | XXXXXXXXXX | 20 |

## AND

This operator displays only those records where both the conditions condition1 and condition2 evaluates to True.

Syntax:

```
SELECT * FROM table_name WHERE condition1 AND condition2 and ...conditionN;

table_name: name of the table
condition1,2,..N : first condition, second condition and so on
```

Now, we consider a table database to demonstrate OR operators with multiple cases:

**Student**

| ROLL_NO | NAME | ADDRESS | PHONE | Age |
|---|---|---|---|---|
| 1 | Ram | Delhi | xxxxxxxxxx | 18 |
| 2 | RAMESH | GURGAON | xxxxxxxxxx | 18 |
| 3 | SUJIT | ROHTAK | xxxxxxxxxx | 20 |
| 4 | SURESH | Delhi | xxxxxxxxxx | 18 |
| 3 | SUJIT | ROHTAK | xxxxxxxxxx | 20 |
| 2 | RAMESH | GURGAON | xxxxxxxxxx | 18 |

If suppose we want to fetch all the records from the Student table where Age is 18 and ADDRESS is Delhi. then the query will be:

Query:

```
SELECT * FROM Student WHERE Age = 18 AND ADDRESS = 'Delhi';
```

Output:

| ROLL_NO | NAME | ADDRESS | PHONE | Age |
|---|---|---|---|---|
| 1 | Ram | Delhi | XXXXXXXXXX | 18 |
| 4 | SURESH | Delhi | XXXXXXXXXX | 18 |

## Limit

If there are a large number of tuples satisfying the query conditions, it might be resourceful to view only a handful of them at a time.

- The LIMIT clause is used to set an upper limit on the number of tuples returned by SQL.
- It is important to note that this clause is not supported by all SQL versions.
- The LIMIT clause can also be specified using the SQL 2008 OFFSET/FETCH FIRST clauses.
- The limit/offset expressions must be a non-negative integer.

Example:

Say we have a relation, Student.

Student Table:

| RollNo | Name | Grade |
|---|---|---|
| 12001 | Aditya | 9 |
| 12002 | Sahil | 6 |
| 12003 | Hema | 8 |
| 12004 | Robin | 9 |
| 12005 | Sita | 7 |
| 12006 | Anne | 10 |
| 12007 | Yusuf | 7 |
| 12008 | Alex | 5 |

Queries

```
SELECT *
FROM Student
LIMIT 5;
```

Output:

| | | |
|---|---|---|
| 12001 | Aditya | 9 |
| 12002 | Sahil | 6 |
| 12003 | Hema | 8 |
| 12004 | Robin | 9 |
| 12005 | Sita | 7 |

## Order By

The ORDER BY statement in SQL is used to sort the fetched data in either ascending or descending according to one or more columns.

By default ORDER BY sorts the data in ascending order.
We can use the keyword DESC to sort the data in descending order and the keyword ASC to sort in ascending order.
Sort according to one column:
To sort in ascending or descending order we can use the keywords ASC or DESC respectively.
Syntax:

```
SELECT * FROM table_name ORDER BY column_name ASC|DESC
//Where
table_name: name of the table.
column_name: name of the column according to which the data is needed to be arranged.
ASC: to sort the data in ascending order.
DESC: to sort the data in descending order.
| : use either ASC or DESC to sort in ascending or descending order//
```

Sort according to multiple columns:
To sort in ascending or descending order we can use the keywords ASC or DESC respectively. To sort according to multiple columns, separate the names of columns by the (,) operator.
Syntax:

```
SELECT * FROM table_name ORDER BY column1 ASC|DESC , column2 ASC|DESC
```

Output:

| ROLL_NO | NAME | ADDRESS | PHONE | Age |
|---|---|---|---|---|
| 1 | HARSH | DELHI | XXXXXXXXXX | 18 |
| 2 | PRATIK | BIHAR | XXXXXXXXXX | 19 |
| 3 | RIYANKA | SILIGURI | XXXXXXXXXX | 20 |
| 4 | DEEP | RAMNAGAR | XXXXXXXXXX | 18 |
| 5 | SAPTARHI | KOLKATA | XXXXXXXXXX | 19 |
| 6 | DHANRAJ | BARABAJAR | XXXXXXXXXX | 20 |
| 7 | ROHIT | BALURGHAT | XXXXXXXXXX | 18 |
| 8 | NIRAJ | ALIPUR | XXXXXXXXXX | 19 |

Now consider the above database table and find the results of different queries.

Sort according to a single column:

In this example, we will fetch all data from the table Student and sort the result in descending order according to the column ROLL_NO.

Query:

```
SELECT * FROM Student ORDER BY ROLL_NO DESC;
```

Output:

| ROLL_ | NAME | ADDRESS | PHONE | Age |
|---|---|---|---|---|
| 8 | NIRAJ | ALIPUR | XXXXXXXXXX | 19 |
| 7 | ROHIT | BALURGHAT | XXXXXXXXXX | 18 |
| 6 | DHANRAJ | BARABAJAR | XXXXXXXXXX | 20 |
| 5 | SAPTARHI | KOLKATA | XXXXXXXXXX | 19 |
| 4 | DEEP | RAMNAGAR | XXXXXXXXXX | 18 |
| 3 | RIYANKA | SILIGURI | XXXXXXXXXX | 20 |
| 2 | PRATIK | BIHAR | XXXXXXXXXX | 19 |
| 1 | HARSH | DELHI | XXXXXXXXXX | 18 |

## 5. SubQuerying

In SQL a Subquery can be simply defined as a query within another query. In other words we can say that a Subquery is a query that is embedded in WHERE clause of another SQL query.

Important rules for Subqueries:

- You can place the Subquery in a number of SQL clauses: WHERE clause, HAVING clause, FROM clause.
- Subqueries can be used with SELECT, UPDATE, INSERT, DELETE statements along with expression operator. It could be equality operator or comparison operator such as =, >, =, <= and Like operator.
- A subquery is a query within another query. The outer query is called as main query and inner query is called as subquery.
- The subquery generally executes first, and its output is used to complete the query condition for the main or outer query.
- Subquery must be enclosed in parentheses.
- Subqueries are on the right side of the comparison operator.
- ORDER BY command cannot be used in a Subquery. GROUPBY command can be used to perform same function as ORDER BY command.
- Use single-row operators with singlerow Subqueries. Use multiple-row operators with multiple-row Subqueries.

Syntax:

There is not any general syntax for Subqueries. However, Subqueries are seen to be used most frequently with SELECT statement as shown below:

```
SELECT column_name
FROM table_name
WHERE column_name expression operator
    ( SELECT COLUMN_NAME  from TABLE_NAME   WHERE ... );
```

Sample Table:

DATABASE

| NAME | ROLL_NO | LOCATION | PHONE_NUMBER |
|---|---|---|---|
| Ram | 101 | Chennai | 9988775566 |
| Raj | 102 | Coimbatore | 8877665544 |
| Sasi | 103 | Madurai | 7766553344 |
| Ravi | 104 | Salem | 8989898989 |
| Sumathi | 105 | Kanchipuram | 8989856868 |

STUDENT

| NAME | ROLL_NO | SECTION |
|---|---|---|
| Ravi | 104 | A |
| Sumathi | 105 | B |
| Raj | 102 | A |

Sample Queries:

To display NAME, LOCATION, PHONE_NUMBER of the students from DATABASE table whose section is A.

```
Select NAME, LOCATION, PHONE_NUMBER from DATABASE
WHERE ROLL_NO IN
(SELECT ROLL_NO from STUDENT where SECTION='A');
```

Explanation : First subquery executes " SELECT ROLL_NO from STUDENT where SECTION='A' " returns ROLL_NO from STUDENT table whose SECTION is 'A'.Then outer-query executes it and return the NAME, LOCATION, PHONE_NUMBER from the DATABASE table of the student whose ROLL_NO is returned from inner subquery.

Output:

| NAME | ROLL_NO | LOCATION | PHONE_NUMBER |
|---|---|---|---|
| Ravi | 104 | Salem | 8989898989 |
| Raj | 102 | Coimbatore | 8877665544 |

## 6. Set Operators (Union, Union All, Intersect, Minus)

### Union

The Union Clause is used to combine two separate select statements and produce the result set as a union of both the select statements.

NOTE:

1. The fields to be used in both the select statements must be in same order, same number and same data type.
2. The Union clause produces distinct values in the result set, to fetch the duplicate values too UNION ALL must be used instead of just UNION.

Basic Syntax:

```
SELECT column_name(s) FROM table1 UNION SELECT column_name(s) FROM table2;

Resultant set consists of distinct values.
```

```
SELECT column_name(s) FROM table1 UNION ALL SELECT column_name(s) FROM table2;

Resultant set consists of duplicate values too.
```

**Student**

| ROLL_NO | NAME | ADDRESS | PHONE | Age |
|---|---|---|---|---|
| 1 | Ram | Delhi | xxxxxxxxxx | 18 |
| 2 | RAMESH | GURGAON | xxxxxxxxxx | 18 |
| 3 | SUJIT | ROHTAK | xxxxxxxxxx | 20 |
| 4 | SURESH | Delhi | xxxxxxxxxx | 18 |
| 3 | SUJIT | ROHTAK | xxxxxxxxxx | 20 |
| 2 | RAMESH | GURGAON | xxxxxxxxxx | 18 |

Student_Details

| ROLL_NO | Branch | Grade |
|---|---|---|
| 1 | Information Technology | O |
| 2 | Computer Science | E |
| 3 | Computer Science | O |
| 4 | Mechanical Engineering | A |

Queries

To fetch ROLL_NO , NAME from Student table WHERE ROLL_NO is greater than 3 and ROLL_NO , Branch from Student_Details table WHERE ROLL_NO is less than 3 , including duplicate values and finally sorting the data by ROLL_NO.

```
SELECT ROLL_NO,NAME FROM Student WHERE ROLL_NO>3
UNION ALL
SELECT ROLL_NO,Branch FROM Student_Details WHERE ROLL_NO<3
ORDER BY 1;

Note:The column names in both the select statements can be different but the
 data type must be same.And in the result set the name of column used in the first
 select statement will appear.
```

Output:

| ROLL_NO | NAME |
|---|---|
| 1 | Information Technology |
| 2 | Computer Science |
| 4 | SURESH |

## Union All

A union is used for extracting rows using the conditions specified in the query while Union All is used for extracting all the rows from a set of two tables.

Syntax :

```
query1 UNION ALL query2
```

The same conditions are applicable to Union All. The only difference between Union and Union All is that Union extracts the rows that are being specified in the query

while Union All extracts all the rows including the duplicates (repeated values) from both the queries.

## INTERSECT

As the name suggests, the intersect clause is used to provide the result of the intersection of two select statements. This implies the result contains all the rows which are common to both the SELECT statements.

Syntax :

```
SELECT column-1, column-2 ……
FROM table 1
WHERE…..

INTERSECT

SELECT column-1, column-2 ……
FROM table 2
WHERE…..
```

Example:

```
SELECT ID, Name, Bonus
FROM
table1
LEFT JOIN
table2
ON table1.ID = table2.Employee_ID

INTERSECT

SELECT ID, Name, Bonus
FROM
table1
RIGHT JOIN
table2
ON table1.ID = table2.Employee_ID;
```

Output:

| ID | Name | Bonus |
|---|---|---|
| 1 | Suresh | 20,000 |
| 3 | Kashish | 30,000 |

## Minus

The Minus Operator in SQL is used with two SELECT statements. The MINUS operator is used to subtract the result set obtained by first SELECT query from the result set obtained by second SELECT query. In simple words, we can say that MINUS operator will return only those rows which are unique in only first SELECT query and not those rows which are common to both first and second SELECT queries.

Basic syntax:

```
SELECT column1 , column2 , ... columnN
FROM table_name
WHERE condition
MINUS
SELECT column1 , column2 , ... columnN
FROM table_name
WHERE condition;

columnN: column1, column2.. are the name of columns of the table.
 Important Points:
```

- The WHERE clause is optional in the above query.
- The number of columns in both SELECT statements must be same.
- The data type of corresponding columns of both SELECT statement must be same.

Sample Tables:

Table 1

| Name | Address | Age | Grade |
|---|---|---|---|
| Harsh | Delhi | 20 | A |
| Gaurav | Jaipur | 21 | B |
| Pratik | Mumbai | 21 | A |
| Dhanraj | Kolkata | 22 | B |

Table 2

| Name | Age | Phone | Grade |
|---|---|---|---|
| Akash | 20 | xxxxxxxxxx | A |
| Dhiraj | 21 | xxxxxxxxxx | B |
| Vaibhav | 21 | xxxxxxxxxx | A |
| Dhanraj | 22 | xxxxxxxxxx | B |

Queries:

```
SELECT NAME, AGE , GRADE
FROM Table1
MINUS
SELECT NAME, AGE, GRADE
FROM Table2
```

Output:

The above query will return only those rows which are unique in 'Table1'. We can clearly see that values in the fields NAME, AGE and GRADE for the last row in both tables are same. Therefore, the output will be the first three rows from Table1. The obtained output is shown below:

| Name | Age | Grade |
|---|---|---|
| Harsh | 20 | A |
| Gaurav | 21 | B |
| Pratik | 21 | A |

## 7. Create, Read, Update (Alter), Delete (Drop) Table

### Create

There are two CREATE statements available in SQL:

1. CREATE DATABASE
2. CREATE TABLE

1. Create Database

   A Database is defined as a structured set of data. So, in SQL the very first step to store the data in a well structured manner is to create a database. The CREATE DATABASE statement is used to create a new database in SQL.

   Syntax:

```
CREATE DATABASE database_name;

database_name: name of the database.
```

2. Create Table

   We have learned above about creating databases. Now to store the data we need a table to do that. The CREATE TABLE statement is used to create a table in SQL. We know that a table comprises of rows and columns. So while creating tables we have to provide all the information to SQL about the names of the columns, type of data to be stored in columns, size of the data etc. Let us now dive into details on how to use CREATE TABLE statement to create tables in SQL.

   Syntax:

```
CREATE TABLE table_name
(
column1 data_type(size),
column2 data_type(size),
column3 data_type(size),
....
);

table_name:  name of the table.
column1 name of the first column.
data_type: Type of data we want to store in the particular column.
           For example,int for integer data.
size: Size of the data we can store in a particular column. For example if for
a column we specify the data_type as int and size as 10 then this column can store an integer
number of maximum 10 digits.
```

## Read

To read sql table into a DataFrame using only the table name, without executing any query we use read_sql_table() method in Pandas. This function does not support DBAPI connections.

read_sql_table()

Syntax : pandas.read_sql_table(table_name, con, schema=None, index_col=None, coerce_float=True, parse_dates=None, columns=None, chunksize=None)

```
CREATE TABLE table_name
(
column1 data_type(size),
column2 data_type(size),
column3 data_type(size),
....
);

table_name:  name of the table.
column1 name of the first column.
data_type: Type of data we want to store in the particular column.
           For example,int for integer data.
size: Size of the data we can store in a particular column. For example if for
a column we specify the data_type as int and size as 10 then this column can store an integer
number of maximum 10 digits.
```

## Update

The UPDATE statement in SQL is used to update the data of an existing table in database. We can update single columns as well as multiple columns using UPDATE statement as per our requirement.

Basic Syntax

```
UPDATE table_name SET column1 = value1, column2 = value2,...
WHERE condition;

table_name: name of the table
column1: name of first , second, third column....
value1: new value for first, second, third column....
condition: condition to select the rows for which the
values of columns needs to be updated.
```

NOTE: In the above query the SET statement is used to set new values to the particular column and the WHERE clause is used to select the rows for which the columns are needed to be updated. If we have not used the WHERE clause then the columns in all the rows will be updated. So the WHERE clause is used to choose the particular rows.

## Delete

The DELETE Statement in SQL is used to delete existing records from a table. We can delete a single record or multiple records depending on the condition we specify in the WHERE clause.

Basic Syntax:

```
DELETE FROM table_name WHERE some_condition;

table_name: name of the table
some_condition: condition to choose particular record.
```

Note: We can delete single as well as multiple records depending on the condition we provide in WHERE clause. If we omit the WHERE clause then all of the records will be deleted and the table will be empty.

Reference :


1. https://www.geeksforgeeks.org/sql-join-set-1-inner-left-right-and-full-joins/
2. https://www.geeksforgeeks.org/sql-aliases/
3. https://www.geeksforgeeks.org/sql-having-clause-with-examples/?ref=gcse
4. https://www.geeksforgeeks.org/sql-group-by/?ref=gcse

5. https://www.geeksforgeeks.org/postgresql-rollup/?ref=gcse
6. https://www.geeksforgeeks.org/sql-where-clause/?ref=gcse
7. https://www.geeksforgeeks.org/sql-limit-clause/?ref=gcse
8. https://www.geeksforgeeks.org/sql-order-by/?ref=gcse
9. https://www.geeksforgeeks.org/sql-subquery/?ref=gcse
10. https://www.geeksforgeeks.org/sql-intersect-except-clause/?ref=gcse
11. https://www.geeksforgeeks.org/sql-delete-statement/?ref=gcse
12. https://www.geeksforgeeks.org/sql-update-statement/?ref=gcse