**UNIT-3**

SQL: Basic SQL querying (Select and Project) using where clause

**WHERE Clause**

**WHERE** keyword is used for fetching **filtered data** in a result set. It is used to fetch data according to particular criteria. **WHERE**[keyword](https://www.geeksforgeeks.org/lateral-keyword-in-sql/)can also be used to filter data by matching patterns.

**Syntax:**

*SELECT****column1,column2****FROM table\_name WHERE column\_name operator value;*

**Parameter Explanation:**

1. **column1,column2:** fields in the table
2. **table\_name:** name of table
3. **column\_name:** name of field used for filtering the data
4. **operator:** operation to be considered for filtering
5. **value:** exact value or pattern to get related data in result

**List of Operators that Can be Used with WHERE Clause**

|  |  |
| --- | --- |
| **Operator** | **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 column |

**Query:**

CREATE TABLE Emp1(  
 EmpID INT PRIMARY KEY,  
 Name VARCHAR(50),  
 Country VARCHAR(50),  
 Age int(2),  
 mob int(10)  
);  
-- Insert some sample data into the Customers table  
INSERT INTO Emp1 (EmpID, Name,Country, Age, mob)  
VALUES (1, 'Shubham', 'India','23','738479734'),  
 (2, 'Aman ', 'Australia','21','436789555'),  
 (3, 'Naveen', 'Sri lanka','24','34873847'),  
 (4, 'Aditya', 'Austria','21','328440934'),  
 (5, 'Nishant', 'Spain','22','73248679');  
 Select \* from Emp1;

**Where Clause with Logical Operators**

To fetch records of  Employee with ages equal to 24.

**Query:**

SELECT \* FROM Emp1 WHERE Age=24;

**Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EmpID** | **Name** | **Country** | **Age** | **mobile** |
| **3** | **naveen** | **srilanka** | **24** | **34873847** |

To fetch the EmpID, Name and Country of Employees with Age greater than21.

**Query:**

SELECT EmpID, Name, Country FROM Emp1 WHERE Age > 21;

**Output:**

|  |  |  |
| --- | --- | --- |
| **EmpID** | **Name** | **Country** |
| **1** | **shubham** | **india** |
| **2** | **Naveen** | **srilanka** |
| **3** | **Nishant** | **spain** |

**Where Clause with BETWEEN Operator**

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

**Syntax:**

*SELECT column1,column2 FROM table\_name*

*WHERE column\_name BETWEEN value1 AND value2;*

**Parameter Explanation:**

1. **BETWEEN:** operator name
2. **value1 AND value2:** exact value from value1 to value2 to get related data in result set.

To fetch records of Employees where Age is between 22 and 24 (inclusive).

**Query:**

SELECT \* FROM Emp1 WHERE Age BETWEEN 22 AND 24;

**Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EmpID** | **Name** | **Country** | **Age** | **mobile** |
| **1** | **shubham** | **India** | **23** | **738479734** |
| **3** | **naveen** | **srilanka** | **24** | **34873847** |
| **5** | **Nishant** | **spain** | **22** | **73248679** |

**Where Clause with LIKE Operator**

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

**Syntax:**

*SELECT column1,column2 FROM*

*table\_name WHERE column\_name LIKE pattern;*

**Parameters Explanation:**

1. **LIKE:** operator name
2. **pattern:** exact value extracted from the pattern to get related data in the result set.

**Note**: The character(s) in the pattern is case-insensitive.

To fetch records of Employees where Name starts with the letter S.

**Query:**

SELECT \* FROM Emp1 WHERE Name LIKE 'S%';

The ‘%'(wildcard) signifies the later characters here which can be of any length and value.

**Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EmpID** | **Name** | **Country** | **Age** | **mobile** |
| **1** | **shubham** | **India** | **23** | **738479734** |

To fetch records of Employees where Name contains the pattern ‘M’.

**Query:**

SELECT \* FROM Emp1 WHERE Name LIKE '%M%';

**Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EmpID** | **Name** | **Country** | **Age** | **mobile** |
| **1** | **shubham** | **India** | **23** | **738479734** |
| **2** | **Aman** | **Australia** | **21** | **436789555** |

**Where Clause with IN Operator**

It is used to fetch the 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.

**Syntax:**

*SELECT column1,column2 FROM table\_name WHERE column\_name IN (value1,value2,..);*

**Parameters Explanation:**

1. **IN:** operator name
2. **value1,value2,..:** exact value matching the values given and get related data in the result set.

To fetch the Names of Employees where Age is 21 or 23.

**Query:**

SELECT Name FROM Emp1 WHERE Age IN (21,23);

**Output:**

|  |
| --- |
| Name |
| shubham |
| Aman |
| Aditya |

## Operators in SQL

Operators in SQL are symbols that help us to perform specific mathematical and logical computations on operands. An operator can either be unary or binary.

The unary operator operates on one operand, and the binary operator operates on two operands.

# Arithmetic Operators

We can use various Arithmetic Operators on the data stored in the tables.

Arithmetic Operators are:

**+** [Addition]

**-** [Subtraction]

**/** [Division]

**\*** [Multiplication]

**%** [Modulus]

#### **Addition** (+) :

It is used to perform **addition operation** on the data items, items include either single column or multiple columns.

**Implementation**:

SELECT employee\_id, employee\_name, salary, salary + 100

AS "salary + 100" FROM addition;

Output:

|  |  |  |  |
| --- | --- | --- | --- |
| **employee\_id** | **employee\_name** | **salary** | **Salary+100** |
| 1 | alex | 25000 | 25100 |
| 2 | rr | 55000 | 55100 |
| 3 | jpm | 52000 | 52100 |
| 4 | ggshmr | 12312 | 12412 |

Here we have done addition of 100 to each Employee’s salary i.e, addition operation on single column.

Let’s perform **addition of 2 columns**:

SELECT employee\_id, employee\_name, salary, salary + employee\_id

AS "salary + employee\_id" FROM addition;

Output:

|  |  |  |  |
| --- | --- | --- | --- |
| **employee\_id** | **employee\_name** | **salary** | **Salary+employee\_id** |
| 1 | alex | 25000 | 25001 |
| 2 | rr | 55000 | 55002 |
| 3 | jpm | 52000 | 52003 |
| 4 | ggshmr | 12312 | 12316 |

Here we have done addition of 2 columns with each other i.e, each employee’s employee\_id is added with its salary.

#### **Subtraction** (-) :

It is use to perform **subtraction operation** on the data items, items include either single column or multiple columns.

**Implementation**:

SELECT employee\_id, employee\_name, salary, salary - 100

AS "salary - 100" FROM subtraction;

Output:

|  |  |  |  |
| --- | --- | --- | --- |
| **employee\_id** | **employee\_name** | **salary** | **salary-100** |
| 12 | Finch | 15000 | 14900 |
| 22 | Peter | 25000 | 24900 |
| 32 | Warner | 5600 | 5500 |
| 42 | Watson | 90000 | 89900 |

Here we have done subtraction of 100 to each Employee’s salary i.e, subtraction operation on single column.

Let’s perform **subtraction of 2 columns**:

SELECT employee\_id, employee\_name, salary, salary - employee\_id

AS "salary - employee\_id" FROM subtraction;

Output:

|  |  |  |  |
| --- | --- | --- | --- |
| **employee\_id** | **employee\_name** | **salary** | **salary-employee\_id** |
| 12 | Finch | 15000 | 14988 |
| 22 | Peter | 25000 | 24978 |
| 32 | Warner | 5600 | 5568 |
| 42 | Watson | 90000 | 89958 |

Here we have done subtraction of 2 columns with each other i.e, each employee’s employee\_id is subtracted from its salary.

#### **Multiplication** (\*) :

It is use to perform **multiplication** of data items.

**Implementation**:

SELECT employee\_id, employee\_name, salary, salary \* 100

AS "salary \* 100" FROM addition;

Output:

|  |  |  |  |
| --- | --- | --- | --- |
| **employee\_id** | **employee\_name** | **salary** | **Salary\*100** |
| 1 | Finch | 25000 | 2500000 |
| 2 | Peter | 55000 | 5500000 |
| 3 | Warner | 52000 | 5200000 |
| 4 | Watson | 12312 | 1231200 |

Here we have done multiplication of 100 to each Employee’s salary i.e, multiplication operation on single column.

Let’s perform **multiplication of 2 columns**:

SELECT employee\_id, employee\_name, salary, salary \* employee\_id

AS "salary \* employee\_id" FROM addition;

Output:

|  |  |  |  |
| --- | --- | --- | --- |
| **employee\_id** | **employee\_name** | **salary** | **Salary\*employee\_id** |
| 1 | Finch | 25000 | 25000 |
| 2 | Peter | 55000 | 110000 |
| 3 | Warner | 52000 | 156000 |
| 4 | Watson | 12312 | 49248 |

Here we have done multiplication of 2 columns with each other i.e, each employee’s employee\_id is multiplied with its salary.

#### **Modulus** ( % ) :

It is use to get **remainder** when one data is divided by another.

**Implementation**:

SELECT employee\_id, employee\_name, salary, salary % 25000

AS "salary % 25000" FROM addition;

Output:

|  |  |  |  |
| --- | --- | --- | --- |
| **employee\_id** | **employee\_name** | **salary** | **Salary%25000** |
| 1 | Finch | 25000 | 0 |
| 2 | Peter | 55000 | 5000 |
| 3 | Warner | 52000 | 2000 |
| 4 | Watson | 12312 | 12312 |

Here we have done modulus of 100 to each Employee’s salary i.e, modulus operation on single column.

Let’s perform **modulus operation between 2 columns**:

SELECT employee\_id, employee\_name, salary, salary % employee\_id

AS "salary % employee\_id" FROM addition;

Output:

|  |  |  |  |
| --- | --- | --- | --- |
| **employee\_id** | **employee\_name** | **salary** | **Salary%25000** |
| 1 | Finch | 25000 | 0 |
| 2 | Peter | 55000 | 0 |
| 3 | Warner | 52000 | 1 |
| 4 | Watson | 12312 | 0 |

**Logical Operators**

SQL logical operators are used to test for the truth of the condition. A logical operator like the Comparison operator returns a boolean value of **TRUE**, **FALSE**,or **UNKNOWN.**In this article, we will discuss different types of Logical Operators.

Logical operators are used to combine or manipulate the conditions given in a query to retrieve or manipulate data .there are some logical operators in [SQL](https://www.geeksforgeeks.org/sql-tutorial/) like OR, AND etc.

**Types of Logical Operators in SQL**

Given below is the list of logical operators available in SQL.

|  |  |
| --- | --- |
| Operator | Meaning |
| [AND](https://www.geeksforgeeks.org/sql-and-and-or-operators/) | TRUE if both Boolean expressions are TRUE. |
| [IN](https://www.geeksforgeeks.org/sql-between-in-operator/) | TRUE if the operand is equal to one of a list of expressions. |
| [NOT](https://www.geeksforgeeks.org/sql-not-operator/) | Reverses the value of any other Boolean operator. |
| [OR](https://www.geeksforgeeks.org/sql-and-and-or-operators/) | TRUE if either Boolean expression is TRUE. |
| [LIKE](https://www.geeksforgeeks.org/sql-like/) | TRUE if the operand matches a pattern. |
| [BETWEEN](https://www.geeksforgeeks.org/sql-between-in-operator/) | TRUE if the operand is within a range. |
| [ALL](https://www.geeksforgeeks.org/sql-all-and-any/) | TRUE if all of a set of comparisons are TRUE. |
| [ANY](https://www.geeksforgeeks.org/sql-all-and-any/) | TRUE if any one of a set of comparisons is TRUE. |
| [EXISTS](https://www.geeksforgeeks.org/sql-exists/) | TRUE if a subquery contains any rows. |
| [SOME](https://www.geeksforgeeks.org/sql-some/) | TRUE if some of a set of comparisons are TRUE. |

**Example:**

In the below example, we will see how this logical operator works with the help of creating a database.

**Step 1:** Creating a Database

**In order to create a database, we need to use the CREATE operator.**

**Query**

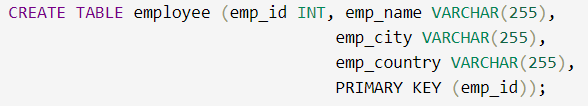
CREATE DATABASE xstream\_db;

**Step 2:**Create table employee

**In this step, we will create the table employee inside the xstream\_db database.**

**Query**

CREATE TABLE employee (emp\_id INT, emp\_name VARCHAR(255),   
 emp\_city VARCHAR(255),  
 emp\_country VARCHAR(255),  
 PRIMARY KEY (emp\_id));

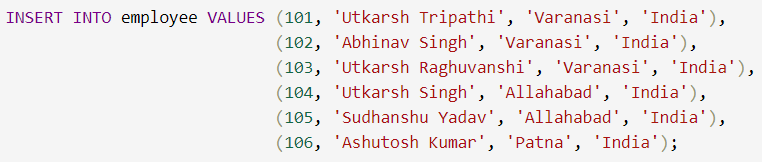


*Create Table*

**In order to insert the data inside the database, we need to use the INSERT operator.**

**Query**

INSERT INTO employee VALUES (101, 'Utkarsh Tripathi', 'Varanasi', 'India'),  
 (102, 'Abhinav Singh', 'Varanasi', 'India'),   
 (103, 'Utkarsh Raghuvanshi', 'Varanasi', 'India'),  
 (104, 'Utkarsh Singh', 'Allahabad', 'India'),  
 (105, 'Sudhanshu Yadav', 'Allahabad', 'India'),  
 (106, 'Ashutosh Kumar', 'Patna', 'India');



*Insert Value*

**Output**

|  |  |  |  |
| --- | --- | --- | --- |
| emp\_id | emp\_name | emp\_city | emp\_country |
| 101 | Utkarsh Tripathi | Varanasi | India |
| 102 | Abhinav Singh | Varanasi | India |
| 103 | Utkarsh Raghuvanshi | Varanasi | India |
| 104 | Utkarsh Singh | Allahabad | India |
| 105 | Sudhanshu Yadav | Allahabad | India |
| 106 | Ashutosh Kumar | Patna | India |

*employee Table*

**Now the given below is the list of different logical operators.**

**AND Operator**

The AND operator is used to combines two or more conditions but if it is true when all the conditions are satisfied.

**Query**

SELECT \* FROM employee WHERE emp\_city = 'Allahabad' AND emp\_country = 'India';

**Output**

|  |  |  |  |
| --- | --- | --- | --- |
| emp\_id | emp\_name | emp\_city | emp\_country |
| 104 | Utkarsh Singh | Allahabad | India |
| 105 | Sudhanshu Yadav | Allahabad | India |

**IN Operator**

It is used to remove the multiple OR conditions in [SELECT](https://www.geeksforgeeks.org/sql-select-query/), [INSERT](https://www.geeksforgeeks.org/sql-insert-statement/), [UPDATE](https://www.geeksforgeeks.org/sql-update-statement/), or [DELETE](https://www.geeksforgeeks.org/sql-delete-statement/). and We can also use NOT IN to minimize the rows in your list and any kind of duplicate entry will be retained.

**Query**

SELECT \* FROM employee WHERE emp\_city IN ('Allahabad', 'Patna');

**Output:**

|  |  |  |  |
| --- | --- | --- | --- |
| emp\_id | emp\_name | emp\_city | emp\_country |
| 104 | Utkarsh Singh | Allahabad | India |
| 105 | Sudhanshu Yadav | Allahabad | India |
| 106 | Ashutosh Kumar | Patna | India |

*Outpu*

**NOT Operator**

**Query**

SELECT \* FROM employee WHERE emp\_city NOT LIKE 'A%';

**Output:**

|  |  |  |  |
| --- | --- | --- | --- |
| *emp\_id* | *emp\_name* | *emp\_city* | *emp\_country* |
| *101* | *Utkarsh Tripathi* | *Varanasi* | *India* |
| *102* | *Abhinav Singh* | *Varanasi* | *India* |
| *103* | *Utkarsh Raghuvanshi* | *Varanasi* | *India* |
| *106* | *Ashutosh Kumar* | *Patna* | *India* |

*output*

**OR Operator**

The OR operator is used to combines two or more conditions but if it is true when one of the conditions are satisfied.

**Query**

SELECT \* FROM employee WHERE emp\_city = 'Varanasi' OR emp\_country = 'India';

**Output**

|  |  |  |  |
| --- | --- | --- | --- |
| emp\_id | emp\_name | emp\_city | emp\_country |
| 101 | Utkarsh Tripathi | Varanasi | India |
| 102 | Abhinav Singh | Varanasi | India |
| 103 | Utkarsh Raghuvanshi | Varanasi | India |
| 104 | Utkarsh Singh | Allahabad | India |
| 105 | Sudhanshu Yadav | Allahabad | India |
| 106 | Ashutosh Kumar | Patna | India |

**LIKE Operator**

In SQL, the LIKE operator is used in the [WHERE](https://www.geeksforgeeks.org/sql-where-clause/) clause to search for a specified pattern in a column.

* % – It is used for zero or more than one character.
* \_ – It is used for only one character means fixed length.

**Query**

SELECT \* FROM employee WHERE emp\_city LIKE 'P%';

|  |  |  |  |
| --- | --- | --- | --- |
| emp\_id | emp\_name | emp\_city | emp\_country |
| 106 | Ashutosh Kumar | Patna | India |

**Output**

**BETWEEN Operator**

The SQL **BETWEEN**condition allows you to easily test if an expression is within a range of values (inclusive).

**Query**

SELECT \* FROM employee WHERE emp\_id BETWEEN 101 AND 104;

**Output**

|  |  |  |  |
| --- | --- | --- | --- |
| emp\_id | emp\_name | emp\_city | emp\_country |
| 101 | Utkarsh Tripathi | Varanasi | India |
| 102 | Abhinav Singh | Varanasi | India |
| 103 | Utkarsh Raghuvanshi | Varanasi | India |
| 104 | Utkarsh Singh | Allahabad | India |

*output*

**ALL Operator**

The ALL operator returns TRUE if all of the subqueries values matches the condition.

**All operator**is used with SELECT, WHERE, HAVING statement.

**Query**

SELECT \* FROM employee WHERE emp\_id = ALL   
 (SELECT emp\_id FROM employee WHERE emp\_city = 'Varanasi');

**Output**

|  |  |  |  |
| --- | --- | --- | --- |
| emp\_id | emp\_name | emp\_city | emp\_country |
| 101 | Utkarsh Tripathi | Varanasi | India |
| 102 | Abhinav Singh | Varanasi | India |
| 103 | Utkarsh Raghuvanshi | Varanasi | India |

*output*

**ANY Operator**

The ANY operator:

* It returns a boolean value as a result
* It returns TRUE if ANY of the subquery values match the condition

**Query**

SELECT \* FROM employee WHERE emp\_id = ANY  
 (SELECT emp\_id FROM employee WHERE emp\_city = 'Varanasi');

**Output**

|  |  |  |  |
| --- | --- | --- | --- |
| emp\_id | emp\_name | emp\_city | emp\_country |
| 101 | Utkarsh Tripathi | Varanasi | India |
| 102 | Abhinav Singh | Varanasi | India |
| 103 | Utkarsh Raghuvanshi | Varanasi | India |

**EXISTS Operator**

In SQL,Exists operator is used to check whether the result of a correlated nested query is empty or not.

Exists operator is used with SELECT, UPDATE, INSERT or DELETE statement.

**Query**

SELECT emp\_name FROM employee WHERE EXISTS  
 (SELECT emp\_id FROM employee WHERE emp\_city = 'Patna');

**Output**

|  |  |  |  |
| --- | --- | --- | --- |
| emp\_id | emp\_name | emp\_city | emp\_country |
| 101 | Utkarsh Tripathi | Varanasi | India |
| 102 | Abhinav Singh | Varanasi | India |
| 103 | Utkarsh Raghuvanshi | Varanasi | India |
| 104 | Utkarsh Singh | Allahabad | India |
| 105 | Sudhanshu Yadav | Allahabad | India |
| 106 | Ashutosh Kumar | Patna | India |

**SQL Date and Time Functions**

In [SQL](https://www.geeksforgeeks.org/sql-tutorial/), dates are complicated for newbies, since while working with a database, the format of the data in the table must be matched with the input data to insert. In various scenarios instead of date, datetime (time is also involved with date) is used.

For storing a date or a date and time value in a database,**MySQL**offers the following data types:

|  |  |
| --- | --- |
| **DATE** | format YYYY-MM-DD |
| **DATETIME** | format: YYYY-MM-DD HH:MI: SS |
| **TIMESTAMP** | format: YYYY-MM-DD HH:MI: SS |
| **YEAR** | format YYYY or YY |

Now, come to some popular functions in SQL date functions.

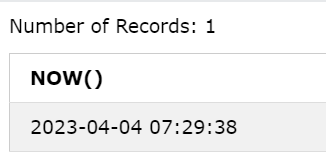
**NOW()**

Returns the current date and time.

**Query:**

SELECT NOW();

**Output:**



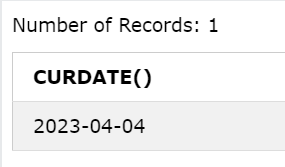
**CURDATE()**

 Returns the current date.

**Query:**

SELECT CURDATE();

**Output:**



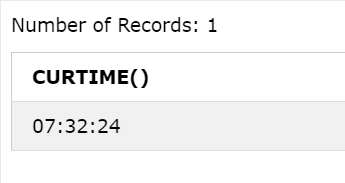
**CURTIME()**

Returns the current time.

**Query:**

SELECT CURTIME();

**Output:**



### **DATE()**

Extracts the date part of a date or date/time expression. Example: For the below table named ‘Test’

|  |  |  |
| --- | --- | --- |
| **Id** | **Name** | **BirthTime** |
| 4120 | Pratik | 1996-09-26 16:44:15.581 |

**Query:**

SELECT Name, DATE(BirthTime)   
AS BirthDate FROM Test;

**Output:**

|  |  |
| --- | --- |
| **Name** | **BirthDate** |
| Pratik | 1996-09-26 |

### **EXTRACT()**

Returns a single part of a date/time.

**Syntax**

*EXTRACT(unit FROM date);*

Several units can be considered but only some are used such as **MICROSECOND, SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER, YEAR, etc.** And ‘date’ is a valid date expression. Example: For the below table named ‘Test’

|  |  |  |
| --- | --- | --- |
| Id | Name | BirthTime |
| 4120 | Pratik | 1996-09-26 16:44:15:581 |

#### **Query:**

SELECT Name, Extract(DAY FROM   
BirthTime) AS BirthDay FROM Test;

**Output:**

|  |  |
| --- | --- |
| Name | Birthday |
| Pratik | 26 |

**Query:**

SELECT Name, Extract(YEAR FROM BirthTime)  
AS BirthYear FROM Test;

**Output:**

|  |  |
| --- | --- |
| Name | BirthYear |
| Pratik | 1996 |

**Query:**

SELECT Name, Extract(SECOND FROM   
BirthTime) AS BirthSecond FROM Test;

**Output:**

|  |  |
| --- | --- |
| Name | BirthSecond |
| Pratik | 581 |

### ****DATE\_ADD()****

 Adds a specified time interval to a date.

**Syntax:**

*DATE\_ADD(date, INTERVAL expr type);*

Where,  date – valid date expression, and expr is the number of intervals we want to add. and type can be one of the following: MICROSECOND, SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER, YEAR, etc. Example: For the below table named ‘Test’

|  |  |  |
| --- | --- | --- |
| **Id** | **Name** | **BirthTime** |
| 4120 | Pratik | 1996-09-26 16:44:15.581 |

#### **Query:**

SELECT Name, DATE\_ADD(BirthTime, INTERVAL   
1 YEAR) AS BirthTimeModified FROM Test;

**Output:**

|  |  |
| --- | --- |
| **Name** | **BirthTimeModified** |
| Pratik | 1997-09-26 16:44:15.581 |

**Query:**

SELECT Name, DATE\_ADD(BirthTime,   
INTERVAL 30 DAY) AS BirthDayModified FROM Test;

**Output:**

|  |  |
| --- | --- |
| **Name** | **BirthDayModified** |
| Pratik | 1996-10-26 16:44:15.581 |

**Query:**

SELECT Name, DATE\_ADD(BirthTime, INTERVAL  
 4 HOUR) AS BirthHourModified FROM Test;

**Output:**

|  |  |
| --- | --- |
| **Name** | **BirthSecond** |
| Pratik | 1996-10-26 20:44:15.581 |

### **DATE\_SUB()**

 Subtracts a specified time interval from a date. The syntax for DATE\_SUB is the same as DATE\_ADD just the difference is that DATE\_SUB is used to subtract a given interval of date.

### **DATEDIFF()**

Returns the number of days between two dates.

**Syntax:**

*DATEDIFF(interval,date1, date2);*

*interval – minute/hour/month/year,etc*

*date1 & date2- date/time expression*

#### **Query:**

SELECT DATEDIFF(month,'2017-01-13','2017-01-03') AS DateDiff;

**Output:**

|  |
| --- |
| **DateDiff** |
| 10 |

### **DATE\_FORMAT()**

 Displays date/time data in different formats.

**Syntax:**

*DATE\_FORMAT(date,format);*

**Numeric Functions:**

**Numeric Functions** are used to perform operations on numbers and return numbers. Following are the numeric functions defined in SQL:

**ABS():** It returns the absolute value of a number.

**Syntax:** SELECT ABS(-243.5);

**Output:**243.5

SQL> SELECT ABS(-10);

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

| ABS(10)

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

| 10

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

**CEIL():** It returns the smallest integer value that is greater than or equal to a number.

**Syntax:** SELECT CEIL(25.75);

**Output:**26

**FLOOR():** It returns the largest integer value that is less than or equal to a number.

**Syntax:** SELECT FLOOR(25.75);

**Output:**25

**GREATEST():** It returns the greatest value in a list of expressions.

**Syntax:** SELECT GREATEST(30, 2, 36, 81, 125);

**Output:**125

**LEAST():** It returns the smallest value in a list of expressions.

**Syntax:** SELECT LEAST(30, 2, 36, 81, 125);

**Output:**2

**LN():** It returns the natural logarithm of a number.

**Syntax:** SELECT LN(2);

**Output:**0.6931471805599453

**MOD():** It returns the remainder (aka. modulus) of n divided by m.

**Syntax:** SELECT MOD(18, 4);

**Output:**2

**POWER(m, n):** It returns m raised to the nth power.

**Syntax:** SELECT POWER(4, 2);

**Output:**16

**SIGN():** It returns a value indicating the sign of a number.  A return value of 1 means positive; 0 means negative.

**Syntax:** SELECT SIGN(255.5);

**Output:**1

**SQRT():** It returns the square root of a number.

**Syntax:** SELECT SQRT(25);

**Output:**5

**TRUNCATE():** This doesn’t work for SQL Server. It returns 7.53635 truncated to n places right of the decimal point.

**Syntax:** SELECT TRUNC(7.53635, 2);

**Output:**7.53

# String functions

**String functions** are used to perform an operation on input string and return an output string. Following are the string functions defined in SQL:

1. **ASCII():** This function is used to find the ASCII value of a character.

**Syntax:** SELECT ascii('t');  
**Output:** 116

1. **CHAR\_LENGTH():** Doesn’t work for SQL Server. Use LEN() for SQL Server. This function is used to find the length of a word.

**Syntax:** SELECT length('Hello!');  
**Output:** 6

1. **CHARACTER\_LENGTH():** Doesn’t work for SQL Server. Use LEN() for SQL Server. This function is used to find the length of a line.

**Syntax:** SELECT CHARACTER\_LENGTH('geeks for geeks');  
**Output:** 15

1. **CONCAT():** This function is used to add two words or strings.

**Syntax:** SELECT 'Geeks' || ' ' || 'forGeeks' FROM dual;  
**Output:** ‘GeeksforGeeks’

1. **CONCAT():** This function is used to add two words or strings with a symbol as concatenating symbol.

**Syntax:** SELECT CONCAT('\_', 'geeks', 'for', 'geeks');  
**Output:** geeks\_for\_geeks

6**.INSTR():** This function is used to find the occurrence of an alphabet.

**Syntax:** INSTR('geeks for geeks', 'e');  
**Output:** 2 (the first occurrence of ‘e’)

**Syntax:** INSTR('geeks for geeks', 'e', 1, 2 );  
**Output:** 3 (the second occurrence of ‘e’)

**7.LCASE():** This function is used to convert the given string into lower case.

**Syntax:** LCASE ("GeeksFor Geeks To Learn");  
**Output:** geeksforgeeks to learn

8.**LENGTH():** This function is used to find the length of a word.

**Syntax:** LENGTH('GeeksForGeeks');  
**Output:** 13

9**.LOWER():** This function is used to convert the upper case string into lower case.

**Syntax:** SELECT LOWER('GEEKSFORGEEKS.ORG');  
**Output:** geeksforgeeks.org

10**.LPAD():** This function is used to make the given string of the given size by adding the given symbol.

**Syntax:** LPAD('geeks', 8, '0');  
**Output:**  
000geeks

11**.LTRIM():** This function is used to cut the given sub string from the original string.

**Syntax:** LTRIM('123123geeks', '123');  
**Output:** geeks

12**.REPEAT():** This function is used to write the given string again and again till the number of times mentioned.

**Syntax:** SELECT REPEAT('geeks', 2);  
**Output:** geeksgeeks

13**.REPLACE():** This function is used to cut the given string by removing the given sub string.

**Syntax:** REPLACE('123NBKRIST123', '123');  
**Output:** NBKRIST

14.**REVERSE():** This function is used to reverse a string.

**Syntax:** SELECT REVERSE(RAJI');  
**Output:** ‘IJAR’

**15.RPAD():** This function is used to make the given string as long as the given size by adding the given symbol on the right.

**Syntax:** RPAD('geeks', 8, '0');  
**Output:** ‘geeks000’

16**.RTRIM():** This function is used to cut the given sub string from the original string.

**Syntax:** RTRIM('geeksxyxzyyy', 'xyz');  
**Output:** ‘geeks’

**17.STRCMP():** This function is used to compare 2 strings.

* + If string1 and string2 are the same, the STRCMP function will return 0.
  + If string1 is smaller than string2, the STRCMP function will return -1.
  + If string1 is larger than string2, the STRCMP function will return 1.

**Syntax:** SELECT STRCMP('google.com', 'geeksforgeeks.com');  
**Output:** -1

18**.SUBSTR():** This function is used to find a sub string from the a string from the given position.

**Syntax:**SUBSTR('WELCOME', 2, 5);  
**Output:** ‘ELCOM’

19**.SUBSTRING():** This function is used to find an alphabet from the mentioned size and the given string.

**Syntax:** SELECT SUBSTR('HELLO, 2, 1);  
**Output:** ‘G’

## Implementation of Integrity Constraints in SQL

Integrity Constraints are an essential part of SQL **[databases](https://www.studysmarter.co.uk/explanations/computer-science/databases/)**, as they ensure the accuracy and consistency of data stored. They define rules for maintaining data integrity and protect the database from data issues.

There are several types of Integrity **[Constraints in SQL](https://www.studysmarter.co.uk/explanations/computer-science/databases/constraints-in-sql/)**,

### Domain Integrity Constraints in SQL

Domain Integrity Constraints define the permissible values for a given column. By applying these constraints, you can restrict the data entered into a specific column, ensuring consistent data values across your database.

Some commonly used domain integrity constraints include:

* Data type – The column must contain values of a specific data type
* Data format – The format of the values in a column must follow a defined pattern
* Range – The values must fall within a specified range
* Enumeration – The values in the column can only be taken from a predefined set of values

## For example, if you have a table containing information about employees' salaries, you might enforce a domain integrity constraint on the "salary" column to ensure that only numeric values within a specific range are entered.

### Entity Integrity Constraints in SQL

Entity Integrity Constraints involve uniquely identifying the rows in a database table, such that there are no duplicate or null values in a primary key column. A primary key is a unique column in a table that uniquely identifies every row in the table. This constraint helps maintain the uniqueness and integrity of data by preventing the existence of duplicate rows.

## For example in a table storing customer information, a unique identification number (‘customer\_id’) can be assigned as the primary key to uniquely identify every customer.

## 3.Referential Integrity Constraint in SQL

Referential Integrity Constraint ensures that relationships between tables are maintained consistently. It is enforced by using foreign keys, which are columns in a table that refer to a primary key in another table. The foreign key helps to maintain the referential integrity between two related tables by making sure that changes in one table's primary key are reflected in the corresponding foreign key in another table.

There are two main rules to uphold when it comes to Referential Integrity Constraints:

* If a primary key value is updated or deleted, the corresponding foreign key values in the related table must be updated or deleted as well.
* Any new foreign key value added to the related table must have a corresponding primary key value in the other table.
* CREATE TABLE OrderDetails (
* OrderID INT NOT NULL,
* ProductID INT NOT NULL,
* Quantity INT NOT NULL,
* Email\_id VARCHAR(20) UNIQUE,
* PRIMARY KEY (OrderID, ProductID),
* FOREIGN KEY (OrderID) REFERENCES Orders(OrderID),
* FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
* );
* CREATE TABLE TABLE ProductDetails
* ProductID INT NOT NULL,
* OrderID INT NOT NULL,
* ProductName VARCHAR(20),
* ProductPrice INT
* );

|  |  |  |  |
| --- | --- | --- | --- |
| **OrderID** | **ProductID** | **Quantity** | **Email\_id** |
| 240911 | 100 | **10** | **abc@gmail.com** |
| 240912 | 121 | **5** | **def@gmail.com** |
| 240913 | 122 | **50** | **ghi@gmail.com** |
| 240914 | 123 | **40** | **jkl@gmail.com** |

## OrderDetails

|  |  |  |  |
| --- | --- | --- | --- |
| **ProductID** | **OrderID** | **ProductName** | **ProductPrice** |
| **100** | 240911 | **Mobile** | **10000** |
| **121** | 240912 | **Laptop** | **20000** |
| **122** | 240913 | **Gun** | **500** |
| **123** | 240914 | **Ball** | **100** |

# Nested Queries

One of the most powerful features of SQL is nested queries. A nested query is a query that has another query embedded within it; the embedded query is called a subquery. When writing a query, we sometimes need to express a condition that refers to a table that must itself be computed. The query used to compute this subsidiary table is a subquery and appears as part of the main query. A subquery typically appears within the WHERE clause of a query. Subqueries can sometimes appear in the FROM clause or the HAVING clause

Nested queries are a way to perform complex queries by embedding one query within another. The outer query can apply some conditions on the results of the inner query. Let use **STUDENT, COURSE, STUDENT\_COURSE** tables for understanding nested queries.

**STUDENT**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S\_ID** | **S\_NAME** | **S\_ADDRESS** | **S\_PHONE** | **S\_AGE** |
| S1 | RAM | DELHI | 9455123451 | 18 |
| S2 | RAMESH | GURGAON | 9652431543 | 18 |
| S3 | SUJIT | ROHTAK | 9156253131 | 20 |
| S4 | SURESH | DELHI | 9156768971 | 18 |

**COURSE**

|  |  |
| --- | --- |
| **C\_ID** | **C\_NAME** |
| C1 | DSA |
| C2 | Programming |
| C3 | DBMS |

**STUDENT\_COURSE**

|  |  |
| --- | --- |
| **S\_ID** | **C\_ID** |
| S1 | C1 |
| S1 | C3 |
| S2 | C1 |
| S3 | C2 |
| S4 | C2 |
| S4 | C3 |

  There are mainly **two types** of nested queries:

* **Independent Nested Queries:** In independent nested queries, query execution starts from innermost query to outermost queries. The execution of inner query is independent of outer query, but the result of inner query is used in execution of outer query. Various operators like IN, NOT IN, ANY, ALL etc are used in writing independent nested queries.

**IN:** If we want to find out **S\_ID** who are enrolled in **C\_NAME** ‘DSA’ or ‘DBMS’, we can write it with the help of independent nested query and IN operator. From **COURSE** table, we can find out **C\_ID**for **C\_NAME** ‘DSA’ or DBMS’ and we can use these **C\_ID**s for finding **S\_ID**s from **STUDENT\_COURSE** TABLE.

* **STEP 1:** Finding **C\_ID** for **C\_NAME** =’DSA’ or ‘DBMS’ Select **C\_ID** from **COURSE** where **C\_NAME** = ‘DSA’ or **C\_NAME** = ‘DBMS’
* **STEP 2:** Using **C\_ID** of step 1 for finding **S\_ID**
* Select **S\_ID** from **STUDENT\_COURSE** where **C\_ID** IN (SELECT **C\_ID** from **COURSE** where **C\_NAME** = ‘DSA’ or **C\_NAME**=’DBMS’);
* The inner query will return a set with members C1 and C3 and outer query will return those **S\_ID**s for which **C\_ID** is equal to any member of set (C1 and C3 in this case). So, it will return S1, S2 and S4.
* **Note:** If we want to find out names of **STUDENT**s who have either enrolled in ‘DSA’ or ‘DBMS’, it can be done as: Select S\_NAME from **STUDENT** where **S\_ID** IN
* (Select **S\_ID** from **STUDENT\_COURSE** where **C\_ID** IN(SELECT **C\_ID** from **COURSE** where **C\_NAME**=’DSA’ or **C\_NAME**=’DBMS’));

**NOT IN:**If we want to find out **S\_ID**s of **STUDENT**s who have neither enrolled in ‘DSA’ nor in ‘DBMS’, it can be done as:

Select **S\_ID** from **STUDENT** where **S\_ID** NOT IN

(Select **S\_ID** from **STUDENT\_COURSE** where **C\_ID** IN (SELECT **C\_ID** from **COURSE** where **C\_NAME**=’DSA’ or **C\_NAME**=’DBMS’));

* The innermost query will return a set with members C1 and C3. Second inner query will return those **S\_ID**s for which **C\_ID** is equal to any member of set (C1 and C3 in this case) which are S1, S2 and S4. The outermost query will return those **S\_ID**s where **S\_ID** is not a member of set (S1, S2 and S4). So it will return S3.
* **Co-related Nested Queries:** In co-related nested queries, the output of inner query depends on the row which is being currently executed in outer query.
* e.g.; If we want to find out **S\_NAME** of **STUDENT**s who are enrolled in **C\_ID** ‘C1’, it can be done with the help of co-related nested query as: Select S\_NAME from **STUDENT** S where EXISTS ( select \* from **STUDENT\_COURSE** SC where S.**S\_ID**=SC.**S\_ID** and SC.**C\_ID**=’C1’);
* For each row of **STUDENT** S, it will find the rows from **STUDENT\_COURSE** where S.**S\_ID** = SC.**S\_ID** and SC.**C\_ID**=’C1’.
* If for a **S\_ID** from **STUDENT** S, atleast a row exists in **STUDENT\_COURSE** SC with **C\_ID**=’C1’, then inner query will return true and corresponding **S\_ID** will be returned as output.

Example IN SQL Code:

SELECT StudentName

FROM Students

WHERE StudentID IN (SELECT StudentID FROM Grades WHERE Subject = ‘Mathematics’ AND Score > 90);

**Grade**

|  |  |  |  |
| --- | --- | --- | --- |
| **S\_ID** | **Subject** | **Score** | **Grade** |
| **S1** | **Math** | **90** | **A+** |
| **S2** | **Hindi** | **80** | **B** |
| **S3** | **Math** | **95** | **A+** |
| **S4** | **Tel** | **95** | **A+** |
| **S5** | **Math** | **96** | **A+** |

* **Co-related Nested Queries:** In co-related nested queries, the output of inner query depends on the row which is being currently executed in outer query. e.g.; If we want to find out **S\_NAME** of **STUDENT**s who are enrolled in **C\_ID** ‘C1’, it can be done with the help of co-related nested query as:

**EXISTS:**

* Select S\_NAME from **STUDENT** S where EXISTS ( select\*from **STUDENT\_COURSE** SC where S.**S\_ID**=SC.**S\_ID** and SC.**C\_ID**=’C1’
* For each row of **STUDENT** S, it will find the rows from **STUDENT\_COURSE** where S.**S\_ID** = SC.**S\_ID** and SC.**C\_ID**=’C1’. If for a **S\_ID** from **STUDENT** S, atleast a row exists in **STUDENT\_COURSE** SC with **C\_ID**=’C1’, then inner query will return true and corresponding **S\_ID** will be returned as output.

**EXAMPLE IN SQL CODE:**

SELECT StudentName

FROM Students

WHERE StudentID IN (

SELECT StudentID

FROM Grades

WHERE Subject = ‘Mathematics’ AND Score > 90

);

Inner Query Output

|  |
| --- |
| S\_ID |
| S3 |
| S5 |

### Output:

|  |
| --- |
| Sname |
| SUJIT |
| HARI |

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](https://www.geeksforgeeks.org/sql-where-clause/) clause,[HAVING](https://www.geeksforgeeks.org/having-vs-where-clause/) 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 when the subquery doesn’t have any**co-relation** with the **main query**, when there is a co-relation the parser takes the decision **on the fly**on which query to execute on **precedence** and uses the output of the subquery accordingly.
* Subquery must be enclosed in parentheses.
* Subqueries are on the right side of the comparison operator.
* Use single-row operators with single row 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 |

* **Insert Query Example:**

Table1: Student1

|  |  |  |  |
| --- | --- | --- | --- |
| **NAME** | **ROLL\_NO** | **LOCATION** | **PHONE\_NUMBER** |
| Ram | 101 | chennai | 9988773344 |
| Raju | 102 | coimbatore | 9090909090 |
| Ravi | 103 | salem | 8989898989 |

Table2: Student2

|  |  |  |  |
| --- | --- | --- | --- |
| **NAME** | **ROLL\_NO** | **LOCATION** | **PHONE\_NUMBER** |
| Raj | 111 | chennai | 8787878787 |
| Sai | 112 | mumbai | 6565656565 |
| Sri | 113 | coimbatore | 7878787878 |

* **To insert Student2 into Student1 table:**

INSERT INTO Student1   
SELECT \* FROM Student2;

* Output:

|  |  |  |  |
| --- | --- | --- | --- |
| **NAME** | **ROLL\_NO** | **LOCATION** | **PHONE\_NUMBER** |
| Ram | 101 | chennai | 9988773344 |
| Raju | 102 | coimbatore | 9090909090 |
| Ravi | 103 | salem | 8989898989 |
| Raj | 111 | chennai | 8787878787 |
| Sai | 112 | mumbai | 6565656565 |
| Sri | 113 | coimbatore | 7878787878 |

# GROUPING (GROUP BY)

The GROUP BY Statement in [SQL](https://www.geeksforgeeks.org/sql-tutorial)is used to arrange identical data into groups with the help of some functions. i.e. if a particular column has the same values in different rows then it will arrange these rows in a group.

### Features

* GROUP BY clause is used with the [SELECT](https://www.geeksforgeeks.org/sql-select-query)statement.
* In the query, the GROUP BY clause is placed after the [WHERE](https://www.geeksforgeeks.org/sql-where-clause)clause.
* In the query, the GROUP BY clause is placed before the [ORDER](https://www.geeksforgeeks.org/sql-order-by)BY clause if used.
* In the query, the Group BY clause is placed before the Having clause.
* Place condition in the [having clause](https://www.geeksforgeeks.org/sql-having-clause-with-examples).

**Syntax**:

*SELECT column1, function\_name(column2)*

*FROM table\_name*

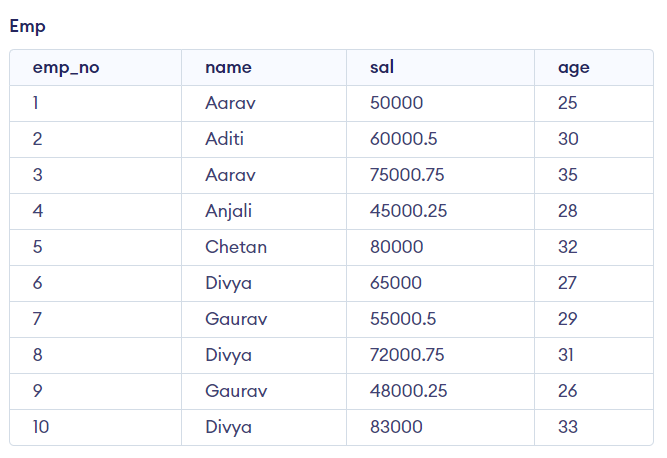
*WHERE condition*

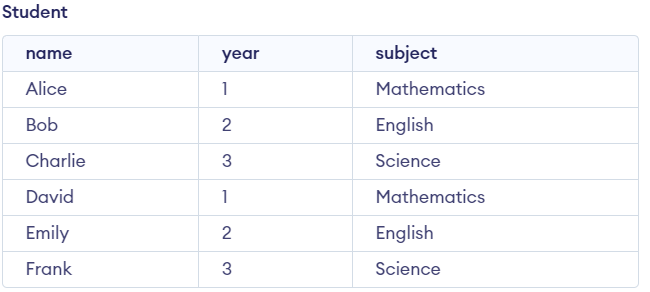
*GROUP BY column1, column2*

*ORDER BY column1, column2;*

**Explanation:**

1. **function\_name**: Name of the function used for example, SUM() , AVG().
2. **table\_name**: Name of the table.
3. **condition**: Condition used.





**Group By single column**

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

**Query:**

SELECT name, SUM(sal) FROM emp

GROUP BY name;

The above query will produce the below output:

|  |  |
| --- | --- |
| name | SUM(sal) |
| Aarav | 125000.75 |
| Aditi | 60000.5 |
| Anjali | 45000.25 |
| Chetan | 80000 |
| Divya | 220000.75 |
| Gaurav | 103000.75 |

*Output*

As you can see in the above output, the rows with duplicate NAMEs are grouped under the 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. The NAMES that are added are Aarav, Divya and Gaurav.

**Group By Multiple Columns**

 Group by multiple columns is say, for example, **GROUP BY column1, column2**. This means placing all the rows with the same values of columns **column 1** and **column 2** in one group. Consider the below query:

**Query:**

SELECT SUBJECT, YEAR, Count(\*)

FROM Student

GROUP BY SUBJECT, YEAR;

**Output:**

|  |  |  |
| --- | --- | --- |
| Subject | Year | Count(\*) |
| English | 2 | 2 |
| Mathematics | 1 | 2 |
| Science | 3 | 2 |

*Output*

**Output**: As you can see in the above output the students with both the same SUBJECT and YEAR are placed in the same group. And those whose only SUBJECT is the same but not YEAR belong to different groups. So here we have grouped the table according to two columns or more than one column. The Grouped subject and years are **(English,2)** , **(Mathematics,1)** and **(Science,3)**. The above mentioned all groups and years are repeated **twice**.

**HAVING Clause in GROUP BY Clause**

We know that the WHERE clause is used to place conditions on columns but what if we want to place conditions on groups? This is where the HAVING clause comes into use. We can use the HAVING clause to place conditions to decide which group will be part of the final result set. Also, we can not use aggregate functions like SUM(), COUNT(), etc. with the WHERE clause. So we have to use the HAVING clause if we want to use any of these functions in the conditions.

**Syntax**:

*SELECT column1, function\_name(column2)*

*FROM table\_name*

*WHERE condition*

*GROUP BY column1, column2*

*HAVING condition*

*ORDER BY column1, column2;*

**Explanation:**

1. **function\_name**: Name of the function used for example, SUM() , AVG().
2. **table\_name**: Name of the table.
3. **condition**: Condition used.

**Example**:

SELECT NAME, SUM(sal) FROM Emp

GROUP BY name

HAVING SUM(sal)>50000;

**Output**:

|  |  |
| --- | --- |
| name | SUM(sal) |
| Aarav | 125000.75 |
| Aditi | 60000.5 |
| Anjali | 45000.25 |
| Chetan | 80000 |
| Divya | 220000.75 |
| Gaurav | 103000.75 |

*Output*

# Aggregate functions

**SQL Aggregate functions** are functions where the values of multiple rows are grouped as input on certain criteria to form a single value result of more significant meaning.

It is used to summarize data, by combining multiple values to form a single result.

SQL Aggregate functions are mostly used with the GROUP BY clause of the SELECT statement.**Various Aggregate Functions**

1. Count()
2. Sum()
3. Avg()
4. Min()
5. Max()

## Aggregate Functions in SQL

Below is the list of SQL aggregate functions, with examples

### ****Count():****

* ***Count(\*):*** Returns the total number of records .i.e 6.
* ***Count(salary):*** Return the number of Non-Null values over the column salary. i.e 5.
* ***Count(Distinct Salary):*** Return the number of distinct Non-Null values over the column salary .i.e 5.

### ****Sum():****

* ***sum(salary):*** Sum all Non-Null values of Column salary i.e., 3120.
* ***sum(Distinct salary):***Sum of all distinct Non-Null values i.e., 3120..

### ****Avg():****

* ***Avg(salary)*** = Sum(salary) / count(salary) = 3120 / 5 = 624
* ***Avg(Distinct salary)*** = sum(Distinct salary) / Count(Distinct Salary) = 3120 / 5 = 624

### ****Min():****

* ***Min(salary):***Minimum value in the salary column except NULL i.e., 403.

### Max():

* ***Max(salary):***Maximum value in the salary i.e., 802.

examples:

|  |  |  |
| --- | --- | --- |
| **Id** | **Name** | **Salary** |
| 1 | A | 802 |
| 2 | B | 403 |
| 3 | C | 604 |
| 4 | D | 705 |
| 5 | E | 606 |
| 6 | F | NULL |

#### Queries

--Count the number of employees  
**SELECT COUNT**(\*) **AS** TotalEmployees **FROM** Employee;  
  
-- Calculate the total salary  
**SELECT** **SUM**(Salary) **AS** TotalSalary **FROM** Employee;  
  
-- Find the average salary  
**SELECT AVG**(Salary) **AS** AverageSalary **FROM** Employee;  
  
-- Get the highest salary  
**SELECT MAX(**Salary) **AS** HighestSalary **FROM** Employee;  
  
-- Determine the lowest salary  
**SELECT MIN**(Salary) **AS** LowestSalary **FROM** Employee;

#### Output

**TotalEmployees**  
6  
**TotalSalary**  
3120  
**AverageSalary**  
624  
**HighestSalary**  
802  
**LowestSalary**  
403

# Ordering (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. It is very useful to present data in a structured manner.

SQL ORDER BY default mode is sorting data into ascending order. To sort data in descending order use the **DESC keyword** with ORDER BY clause.

**Syntax**

The syntax to use ORDER BY clause in SQL is:

*SELECT \* FROM table\_name ORDER BY column\_name ASC | DESC*

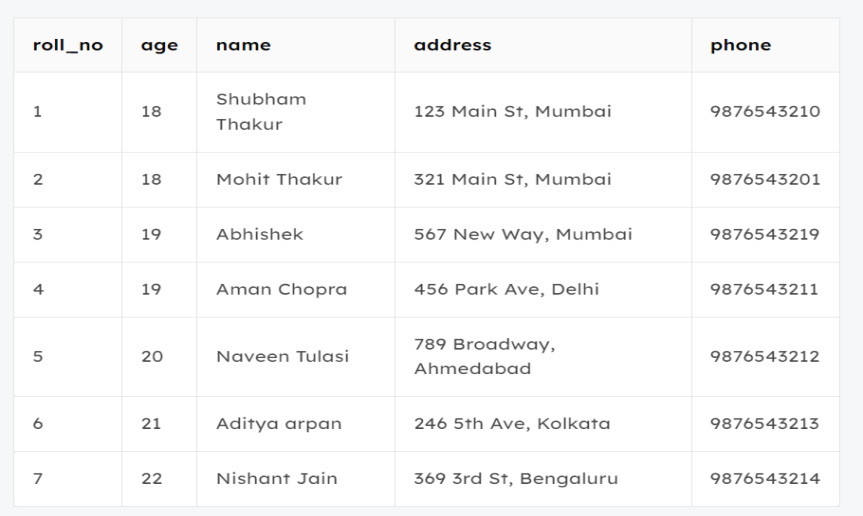
**Key Terms:**

* **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.

**SQL ORDER BY Clause Examples**

Let’s look at some examples of the SQL ORDER BY clause to understand it’s working in SQL.

We will use the following table in examples.



*Student\_Table*

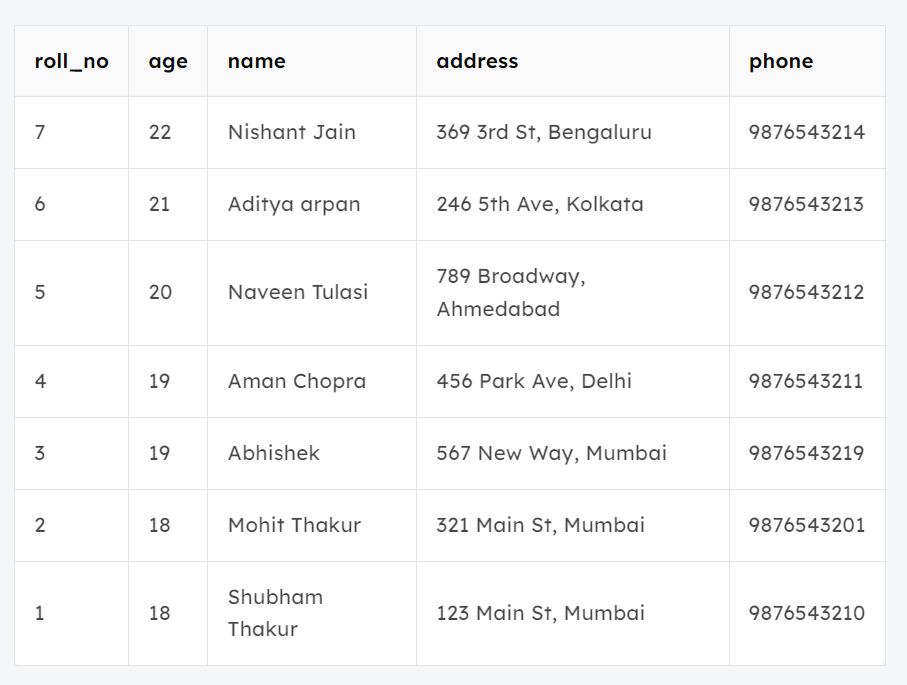
### ****Sort According To a Single Column using ORDER BY Clause Example****

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** students **ORDER BY** ROLL\_NO **DESC**;

**Output:**



In the above example, if we want to sort in ascending order we have to use ASC in place of DESC.

### ****Sort According To Multiple Columns using ORDER BY Clause Example****

To sort according to multiple columns, separate the names of columns by the (,) operator.

**Syntax:**

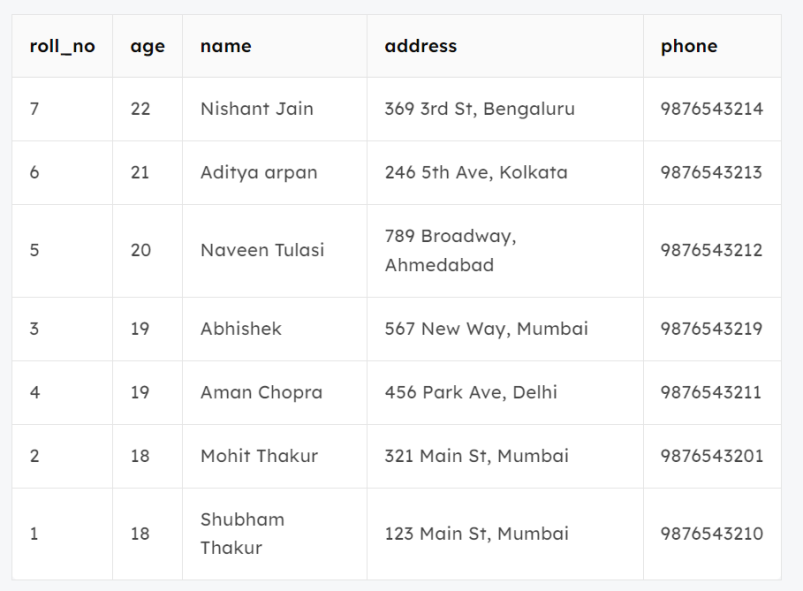
*SELECT \* FROM table\_name ORDER BY column1 ASC|DESC , column2 ASC|DESC*

In this example, we will fetch all data from the table Student and then sort the result in descending order first according to the column **age**. and then in ascending order according to the column **name**.

**Query:**

**SELECT** \* **FROM** students **ORDER BY** age **DESC** , name **ASC**;

**Output:**



*Sort\_Multiple\_Column*

In the above output, we can see that first the result is sorted in descending order according to Age. There are multiple rows of having the same Age. Now, sorting further this result-set according to name will sort the rows with the same Age according to name in ascending order.

## Implementation of different types of JOIN

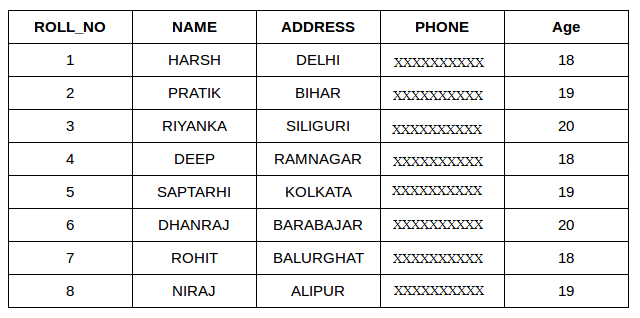
SQL JOIN clause is used to query and access data from multiple tables by establishing logical relationships between them. It can access data from multiple tables simultaneously using common key values shared across different tables.

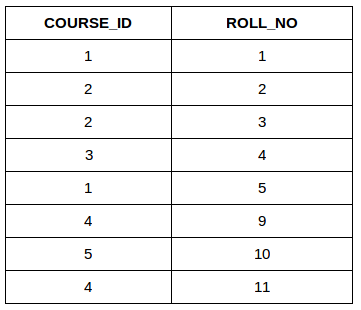
We can use SQL JOIN with multiple tables. It can also be paired with other clauses, the most popular use will be using JOIN with **[WHERE clause](https://www.geeksforgeeks.org/sql-where-clause)**to filter data retrieval.

## SQL JOIN Example

Consider the two tables below as follows:

**Student:**

**StudentCourse** :



Both these tables are connected by one common key (column) i.e **ROLL\_NO**.

We can perform a JOIN operation using the given SQL query:

**SELECT** s.roll\_no, s.name, s.address, s.phone, s.age, sc.course\_id  
**FROM** Student s  
**JOIN** StudentCourse sc **ON** s.roll\_no = sc.roll\_no;

**Output:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ROLL\_NO | NAME | ADDRESS | PHONE | AGE | COURSE\_ID |
| 1 | HARSH | DELHI | XXXXXXXXXX | 18 | 1 |
| 2 | PRATIK | BIHAR | XXXXXXXXXX | 19 | 2 |
| 3 | RIYANKA | SILGURI | XXXXXXXXXX | 20 | 2 |
| 4 | DEEP | RAMNAGAR | XXXXXXXXXX | 18 | 3 |
| 5 | SAPTARHI | KOLKATA | XXXXXXXXXX | 19 | 1 |

## Types of JOIN in SQL

There are many types of Joins in SQL. Depending on the use case, you can use different type of SQL JOIN clause.

Here are the frequently used SQL JOIN types:

**Table of Content**

* INNER JOIN
* LEFT JOIN
* RIGHT JOIN
* FULL JOIN
* Natural join

## **SQL INNER JOIN**

The **[INNER JOIN](https://www.geeksforgeeks.org/sql-inner-join)** keyword selects all rows from both the tables as long as the condition is satisfied. 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 the same.

### **Syntax**:

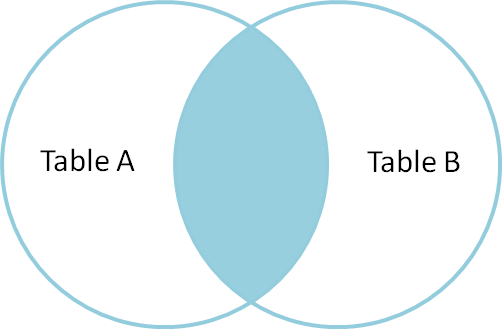
The syntax for SQL INNER JOIN is:

**SELECT** table1.column1,table1.column2,table2.column1,....  
**FROM** table1   
**INNER JOIN** table2  
**ON** table1.matching\_column = table2.matching\_column;

Here,

* **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.*



### **INNER JOIN Example**

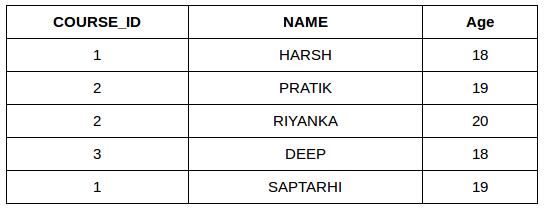
Let’s look at the example of INNER JOIN clause, and understand it’s working.

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

**Query:**

**SELECT** StudentCourse.COURSE\_ID, Student.NAME, Student.AGE **FROM** Student  
**INNER JOIN** StudentCourse  
**ON** Student.ROLL\_NO = StudentCourse.ROLL\_NO;

**Output**:



## **SQL LEFT JOIN**

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

### **Syntax:**

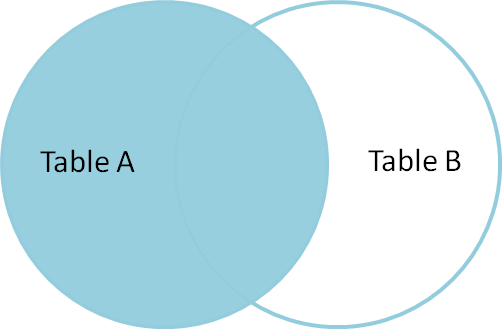
The syntax of LEFT JOIN in SQL is**:**

**SELECT** table1.column1,table1.column2,table2.column1,....  
**FROM** table1   
**LEFT JOIN** table2  
**ON** table1.matching\_column = table2.matching\_column;

Here,

* **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 the same.*

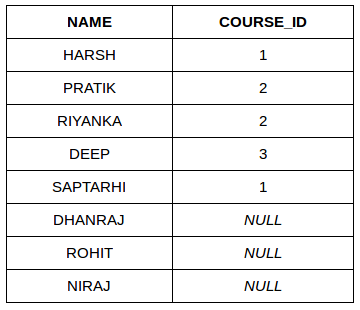


### **LEFT JOIN Example**

Let’s look at the example of LEFT JOIN clause, and understand it’s working

**SELECT** Student.NAME,StudentCourse.COURSE\_ID   
**FROM** Student  
**LEFT JOIN** StudentCourse   
**ON** StudentCourse.ROLL\_NO = Student.ROLL\_NO;

**Output**:



## **SQL RIGHT JOIN**

**[RIGHT JOIN](https://www.geeksforgeeks.org/sql-right-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 the join.It is very similar to LEFT JOIN For the rows for which there is no matching row on the left side, the result-set will contain null. RIGHT JOIN is also known as RIGHT OUTER JOIN.

### **Syntax:**

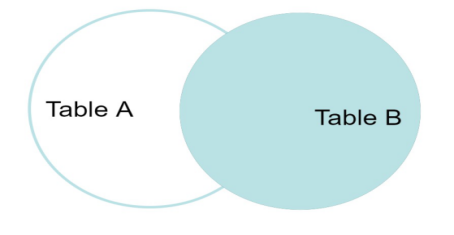
The syntax of RIGHT JOIN in SQL is:

**SELECT** table1.column1,table1.column2,table2.column1,....  
**FROM** table1   
**RIGHT JOIN** table2  
**ON** table1.matching\_column = table2.matching\_column;

Here,

* **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 the same.

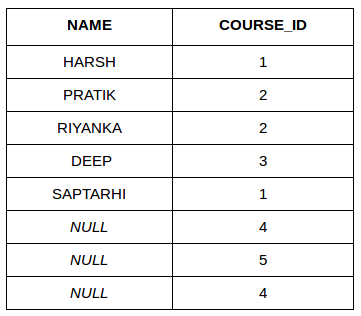


### **RIGHT JOIN Example**:

Let’s look at the example of RIGHT JOIN clause, and understand it’s working

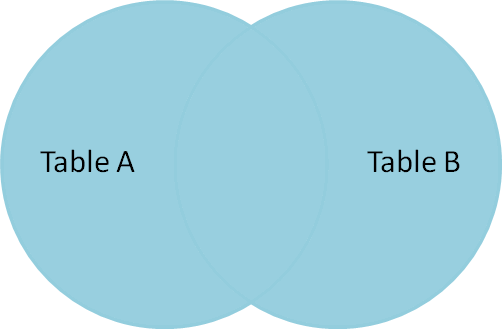
**SELECT** Student.NAME,StudentCourse.COURSE\_ID   
**FROM** Student  
**RIGHT JOIN** StudentCourse   
**ON** StudentCourse.ROLL\_NO = Student.ROLL\_NO;

**Output:**



## **SQL FULL JOIN**

**[FULL JOIN](https://www.geeksforgeeks.org/sql-full-join)** creates the result-set by combining results of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both tables. For the rows for which there is no matching, the result-set will contain NULL values.



### **Syntax**

The syntax of SQL FULL JOIN is:

**SELECT** table1.column1,table1.column2,table2.column1,....  
**FROM** table1   
**FULL JOIN** table2  
**ON** table1.matching\_column = table2.matching\_column;

Here,

* **table1**: First table.
* **table2**: Second table
* **matching\_column**: Column common to both the tables.

### **FULL JOIN Example**

Let’s look at the example of FULL JOIN clause, and understand it’s working

**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 | 4 |
| NULL | 5 |
| NULL | 4 |

## SQL Natural join (?)

Natural join can join tables based on the common columns in the tables being joined. A natural join returns all rows by matching values in common columns having same name and data type of columns and that column should be present in both tables.

Both table must have at least one common column with same column name and same data type.

The two table are joined using **Cross join**.

DBMS will look for a common column with same name and data type Tuples having exactly same values in common columns are kept in result.

### Natural join Example:

Look at the two tables below- Employee and Department

|  |  |  |
| --- | --- | --- |
| **Employee** | | |
| Emp\_id | Emp\_id | Dept\_id |
| 1 | Ram | 10 |
| 2 | Jon | 30 |
| 3 | Bob | 50 |

|  |  |
| --- | --- |
| **Department table** | |
| **Dept\_id** | **Dept\_name** |
| 10 | IT |
| 30 | HR |
| 40 | TIS |

**Problem**: Find all Employees and their respective departments.

**Solution Query**: (Employee) ? (Department)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Employee data** | | | **Department data** | |
| **Emp\_id** | **Emp\_name** | **Dept\_id** | **Dept\_id** | **Dept\_name** |
| 1 | Ram | 10 | 10 | IT |
| 2 | Jon | 30 | 30 | HR |

### Updatable and Non-updatable Views

Some views are updatable and references to them can be used to specify tables to be updated in data change statements. That is, you can use them in statements such as [UPDATE](https://dev.mysql.com/doc/refman/8.4/en/update.html" \o "15.2.17 UPDATE Statement), [DELETE](https://dev.mysql.com/doc/refman/8.4/en/delete.html" \o "15.2.2 DELETE Statement), or [INSERT](https://dev.mysql.com/doc/refman/8.4/en/insert.html" \o "15.2.7 INSERT Statement) to update the contents of the underlying table. Derived tables and common table expressions can also be specified in multiple-table [UPDATE](https://dev.mysql.com/doc/refman/8.4/en/update.html" \o "15.2.17 UPDATE Statement) and [DELETE](https://dev.mysql.com/doc/refman/8.4/en/delete.html" \o "15.2.2 DELETE Statement) statements, but can only be used for reading data to specify rows to be updated or deleted. Generally, the view references must be updatable, meaning that they may be merged and not materialized. Composite views have more complex rules.

For a view to be updatable, there must be a one-to-one relationship between the rows in the view and the rows in the underlying table. There are also certain other constructs that make a view nonupdatable. To be more specific, a view is not updatable if it contains any of the following:

* Aggregate functions or window functions ([SUM()](https://dev.mysql.com/doc/refman/8.4/en/aggregate-functions.html" \l "function_sum), [MIN()](https://dev.mysql.com/doc/refman/8.4/en/aggregate-functions.html" \l "function_min), [MAX()](https://dev.mysql.com/doc/refman/8.4/en/aggregate-functions.html" \l "function_max), [COUNT()](https://dev.mysql.com/doc/refman/8.4/en/aggregate-functions.html" \l "function_count), and so forth)
* DISTINCT
* GROUP BY
* HAVING
* [UNION](https://dev.mysql.com/doc/refman/8.4/en/union.html" \o "15.2.18 UNION Clause) or [UNION ALL](https://dev.mysql.com/doc/refman/8.4/en/union.html" \o "15.2.18 UNION Clause)
* Subquery in the select list

Non-dependent sub-queries in the select list fail for [INSERT](https://dev.mysql.com/doc/refman/8.4/en/insert.html" \o "15.2.7 INSERT Statement), but are okay for [UPDATE](https://dev.mysql.com/doc/refman/8.4/en/update.html" \o "15.2.17 UPDATE Statement), [DELETE](https://dev.mysql.com/doc/refman/8.4/en/delete.html" \o "15.2.2 DELETE Statement). For dependent sub-queries in the select list, no data change statements are permitted.

* Certain joins (see additional join discussion later in this section)
* Reference to non up-datable view in the FROM clause
* Sub-query in the WHERE clause that refers to a table in the FROM clause
* Refers only to literal values (in this case, there is no underlying table to update)

If a view is not updatable, statements such [UPDATE](https://dev.mysql.com/doc/refman/8.4/en/update.html" \o "15.2.17 UPDATE Statement), [DELETE](https://dev.mysql.com/doc/refman/8.4/en/delete.html" \o "15.2.2 DELETE Statement), and [INSERT](https://dev.mysql.com/doc/refman/8.4/en/insert.html" \o "15.2.7 INSERT Statement) are illegal and are rejected. (Even if a view is updatable, it might not be possible to insert into it, as described elsewhere in this section.)

The updatability of views may be affected by the value of the [updatable\_views\_with\_limit](https://dev.mysql.com/doc/refman/8.4/en/server-system-variables.html" \l "sysvar_updatable_views_with_limit) system variable.

For the following discussion, suppose that these tables and views exist:

CREATE TABLE t1 (x INTEGER);

CREATE TABLE t2 (c INTEGER);

CREATE VIEW vmat AS SELECT SUM(x) AS s FROM t1;

CREATE VIEW vup AS SELECT \* FROM t2;

CREATE VIEW vjoin AS SELECT \* FROM vmat JOIN vup ON vmat.s=vup.c;

**[INSERT](https://dev.mysql.com/doc/refman/8.4/en/insert.html" \o "15.2.7 INSERT Statement), [UPDATE](https://dev.mysql.com/doc/refman/8.4/en/update.html" \o "15.2.17 UPDATE Statement), and [DELETE](https://dev.mysql.com/doc/refman/8.4/en/delete.html" \o "15.2.2 DELETE Statement) statements are permitted as follows:**

* [INSERT](https://dev.mysql.com/doc/refman/8.4/en/insert.html" \o "15.2.7 INSERT Statement): The insert table of an [INSERT](https://dev.mysql.com/doc/refman/8.4/en/insert.html" \o "15.2.7 INSERT Statement) statement may be a view reference that is merged. If the view is a join view, all components of the view must be updatable (not materialized). For a multiple-table updatable view, [INSERT](https://dev.mysql.com/doc/refman/8.4/en/insert.html" \o "15.2.7 INSERT Statement) can work if it inserts into a single table.

This statement is invalid because one component of the join view is non updatable:

INSERT INTO vjoin (c) VALUES (1);

This statement is valid; the view contains no materialized components:

INSERT INTO vup (c) VALUES (1);

* [UPDATE](https://dev.mysql.com/doc/refman/8.4/en/update.html" \o "15.2.17 UPDATE Statement): The table or tables to be updated in an [UPDATE](https://dev.mysql.com/doc/refman/8.4/en/update.html" \o "15.2.17 UPDATE Statement) statement may be view references that are merged. If a view is a join view, at least one component of the view must be updatable (this differs from [INSERT](https://dev.mysql.com/doc/refman/8.4/en/insert.html" \o "15.2.7 INSERT Statement)).

In a multiple-table [UPDATE](https://dev.mysql.com/doc/refman/8.4/en/update.html" \o "15.2.17 UPDATE Statement) statement, the updated table references of the statement must be base tables or updatable view references. Nonupdated table references may be materialized views or derived tables.

This statement is valid; column c is from the updatable part of the join view:

UPDATE vjoin SET c=c+1;

This statement is invalid; column x is from the non updatable part:

UPDATE vjoin SET x=x+1;

This statement is valid; the updated table reference of the multiple-table [UPDATE](https://dev.mysql.com/doc/refman/8.4/en/update.html" \o "15.2.17 UPDATE Statement) is an updatable view (vup):

UPDATE vup JOIN (SELECT SUM(x) AS s FROM t1) AS dt ON ...

SET c=c+1;

This statement is invalid; it tries to update a materialized derived table:

UPDATE vup JOIN (SELECT SUM(x) AS s FROM t1) AS dt ON ...

SET s=s+1;

* [DELETE](https://dev.mysql.com/doc/refman/8.4/en/delete.html" \o "15.2.2 DELETE Statement): The table or tables to be deleted from in a [DELETE](https://dev.mysql.com/doc/refman/8.4/en/delete.html" \o "15.2.2 DELETE Statement) statement must be merged views. Join views are not allowed (this differs from [INSERT](https://dev.mysql.com/doc/refman/8.4/en/insert.html" \o "15.2.7 INSERT Statement) and [UPDATE](https://dev.mysql.com/doc/refman/8.4/en/update.html" \o "15.2.17 UPDATE Statement)).

This statement is invalid because the view is a join view:

DELETE vjoin WHERE ...;

This statement is valid because the view is a merged (updatable) view:

DELETE vup WHERE ...;

This statement is valid because it deletes from a merged (updatable) view:

DELETE vup FROM vup JOIN (SELECT SUM(x) AS s FROM t1) AS dt ON ...;

Additional discussion and examples follow.

Earlier discussion in this section pointed out that a view is not insertable if not all columns are simple column references (for example, if it contains columns that are expressions or composite expressions). Although such a view is not insertable, it can be updatable if you update only columns that are not expressions. Consider this view:

CREATE VIEW v AS SELECT col1, 1 AS col2 FROM t;

This view is not insertable because col2 is an expression. But it is updatable if the update does not try to update col2. This update is permissible:

UPDATE v SET col1 = 0;

This update is not permissible because it attempts to update an expression column:

UPDATE v SET col2 = 0;

If a table contains an AUTO\_INCREMENT column, inserting into an insertable view on the table that does not include the AUTO\_INCREMENT column does not change the value of [LAST\_INSERT\_ID()](https://dev.mysql.com/doc/refman/8.4/en/information-functions.html" \l "function_last-insert-id), because the side effects of inserting default values into columns not part of the view should not be visible.

SQL Set Operation

The SQL Set operation is used to combine the two or more SQL SELECT statements.

## Types of Set Operation

**Set Operations**

The following standard operations on sets are also available in relational algebra: *union* (*∪*), *intersection* (*∩*), *set-difference* (*−*), and *cross-product* (*×*).

**Union:** *R∪S* returns a relation instance containing all tuples that occur in *either* relation instance *R* or relation instance *S* (or both). *R* and *S* must be *unioncompatible*, and the schema of the result is defined to be identical to the schema of *R*.

Two relation instances are said to be **union-compatible** if the following conditions hold:

**–** they have the same number of the fields, and

**–** corresponding fields, taken in order from left to right, have the same

Note that field names are not used in defining union-compatibility. For convenience, we will assume that the fields of *R ∪ S* inherit names from *R*, if the fields of *R* have names. (This assumption is implicit in defining the schema of *R ∪ S* to be identical to the schema of *R*, as stated earlier.)

**Intersection:** *R∩S* returns a relation instance containing all tuples that occur in *both R* and *S*. The relations *R* and *S* must be union-compatible, and the schema of the result is defined to be identical to the schema of *R*.

**Set-difference:** *R−S* returns a relation instance containing all tuples that occur in *R* but not in *S*. The relations *R* and *S* must be union-compatible, and the schema of the result is defined to be identical to the schema of *R*.

**Cross-product:** *R ×S* returns a relation instance whose schema contains all the fields of *R* (in the same order as they appear in *R*) followed by all the fields of *S* (in the same order as they appear in *S*). The result of *R × S* contains one tuple < *r, s>* (the concatenation of tuples *r* and *s*) for each pair of tuples *r ∈ R, s ∈ S*.

The cross-product operation is sometimes called **Cartesian product**.

### 1. Union

* The SQL Union operation is used to combine the result of two or more SQL SELECT queries.
* In the union operation, all the number of datatype and columns must be same in both the tables on which UNION operation is being applied.
* The union operation eliminates the duplicate rows from its result set.

**Syntax**

SELECT column\_name FROM table1

UNION

SELECT column\_name FROM table2;

**Example:**

**The First table**

|  |  |
| --- | --- |
| **ID** | **Name** |
| 1 | Ramu |
| 2 | Ravi |
| 3 | Rajesh |

**The Second table**

|  |  |
| --- | --- |
| ID | Name |
| 3 | Rajesh |
| 4 | Arvindh |
| 5 | Ramesh |

Union SQL query will be:

**SELECT \* FROM First   UNION  SELECT \* FROM Second;**

The resultset table will look like:

|  |  |
| --- | --- |
| **ID** | **Name** |
| 1 | Ramu |
| 2 | Ravi |
| 3 | Rajesh |
| 4 | Arvindh |
| 5 | Ramesh |

### 2. Union All

Union All operation is equal to the Union operation. It returns the set without removing duplication and sorting the data.

**Syntax:**

SELECT column\_name FROM table1

UNION ALL

SELECT column\_name FROM table2;

**Example:** Using the above First and Second table.

Union All query will be like:

**SELECT \* FROM First   UNION ALL  SELECT \* FROM Second;**

The result-set table will look like:

|  |  |
| --- | --- |
| **ID** | **Name** |
| 1 | Ramu |
| 2 | Ravi |
| 3 | Rajesh |

### 3. Intersect

* It is used to combine two SELECT statements. The Intersect operation returns the common rows from both the SELECT statements.
* In the Intersect operation, the number of datatype and columns must be the same.
* It has no duplicates and it arranges the data in ascending order by default.

**Syntax**

SELECT column\_name FROM table1

INTERSECT

SELECT column\_name FROM table2;

**Example:**

**Using the above First and Second table.**

Intersect query will be:

**SELECT \* FROM First   INTERSECT  SELECT \* FROM Second;**

The result set table will look like:

|  |  |
| --- | --- |
| **ID** | **Name** |
| 3 | Rajesh |

### 4. Minus or set-difference

* It combines the result of two SELECT statements. Minus operator is used to display the rows which are present in the first query but absent in the second query.
* It has no duplicates and data arranged in ascending order by default.

**Syntax:**

SELECT column\_name FROM table1

MINUS

SELECT column\_name FROM table2;

**Example**

**Using the above First and Second table.**

Minus query will be:

**SELECT \* FROM First   MINUS  SELECT \* FROM Second;**

The resultset table will look like:

|  |  |
| --- | --- |
| **ID** | **Name** |
| 1 | Ramu |
| 2 | Ravi |