SQL:

SQL (Structured Query Language) is used to perform operations on the records stored in the database, such as updating records, inserting records, deleting records, creating and modifying database tables, views, etc.

SQL is not a database system, but it is a query language.

Suppose you want to perform the queries of SQL language on the stored data in the database. You are required to install any database management system in your systems, for example, Oracle

, MySQL

, MongoDB

, PostgreSQL

, SQL Server

, DB2

, etc.

Why SQL?

Nowadays, SQL is widely used in data science and analytics. Following are the reasons which explain why it is widely used:

* The basic use of SQL for data professionals and SQL users is to insert, update, and delete the data from the relational database.
* SQL allows the data professionals and users to retrieve the data from the relational database management systems.
* It also helps them to describe the structured data.
* It allows SQL users to create, drop, and manipulate the database and its tables.
* It also helps in creating the view, stored procedure, and functions in the relational database.
* It allows you to define the data and modify that stored data in the relational database.
* It also allows SQL users to set the permissions or constraints on table columns, views, and stored procedures.

-----------------------------------------------------

Structured Query Language is a standard Database language which is used to create, maintain and retrieve the relational database. Following are some interesting facts about SQL.

* SQL is case insensitive. But it is a recommended practice to use keywords (like SELECT, UPDATE, CREATE, etc) in capital letters and use user defined things (liked table name, column name, etc) in small letters.
* We can write comments in SQL using “–” (double hyphen) at the beginning of any line.
* SQL is the programming language for relational databases (explained below) like MySQL, Oracle, Sybase, SQL Server, Postgre, etc. Other non-relational databases (also called NoSQL) databases like MongoDB, DynamoDB, etc do not use SQL

.

**What is Relational Database?**

Relational database means the data is stored as well as retrieved in the form of relations (tables). Table 1 shows the relational database with only one relation called **STUDENT** which stores **ROLL\_NO**, **NAME**, **ADDRESS**, **PHONE** and **AGE** of students.

**STUDENT**

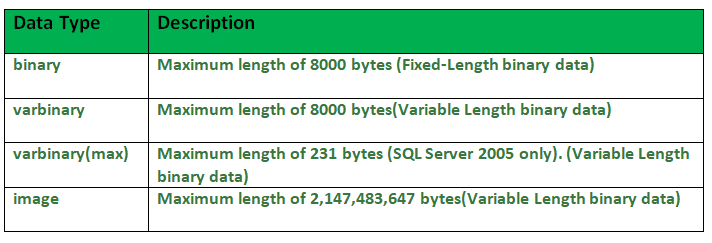
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ROLL\_NO** | **NAME** | **ADDRESS** | **PHONE** | **AGE** |
| 1 | RAM | DELHI | 9455123451 | 18 |
| 2 | RAMESH | GURGAON | 9652431543 | 18 |
| 3 | SUJIT | ROHTAK | 9156253131 | 20 |
| 4 | SURESH | DELHI | 9156768971 | 18 |

**Attribute:** Attributes are the properties that define a relation. e.g.; **ROLL\_NO**, **NAME** etc.

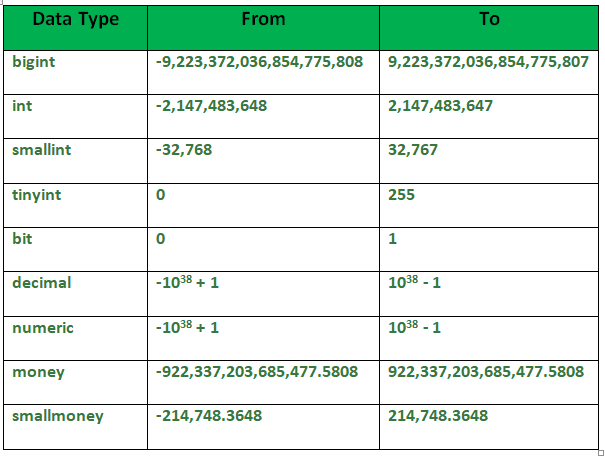
**Tuple:** Each row in the relation is known as tuple. The above relation contains 4 tuples, one of which is shown as:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | RAM | DELHI | 9455123451 | 18 |

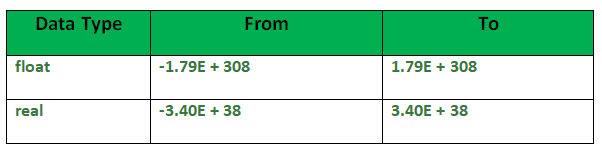
. **Binary Datatypes :**  
There are four subtypes of this datatype which are given below :



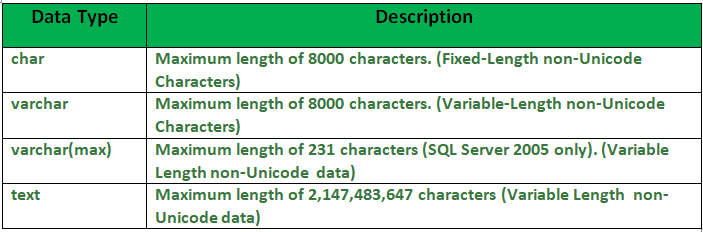
2. **Exact Numeric Datatype :**  
There are nine subtypes which are given below in the table. The table contains the range of data in a particular type.

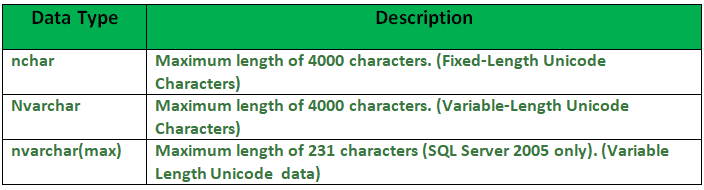


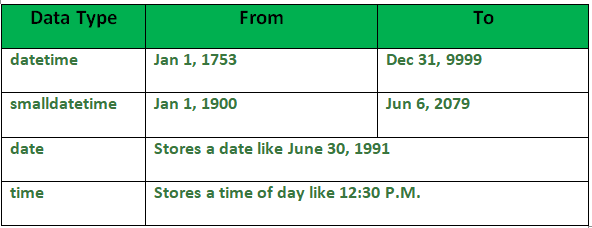
**3. Approximate Numeric Datatype :**  
The subtypes of this datatype are given in the table with the range.



**4. Character String Datatype :**  
The subtypes are given in below table –



**5. Unicode Character String Datatype :**  
The details are given in below table –  


**6. Date and Time Datatype :**  
The details are given in below table.  


1. What is the maximum value that can be stored in NUMERIC(4,2)?

A. 9999.99  
B. 99.9999  
C. 99.99  
D. 9.99

View Answer

Ans : C

Explanation: 99.99 is the maximum value that can be stored in NUMERIC(4,2)

2. Determine data type for the given column? Column Name: Price ; Description: Cost of an item in rupees and paise ; Example: 200.21

A. VARCHAR2(50)  
B. NUMBER  
C. NUMBER(5,2)  
D. NUMBER(6)

View Answer

Ans : C

Explanation: NUMBER(5,2) is the data type for the given column.

3. Determine data type for the given column? Column Name: IFSC\_Code ; Description: A 11 Character alphanumeric code that identifies a bank branch ; Example: SBIN0009044

A. VARCHAR2(50)  
B. NUMBER  
C. Char(11)  
D. NUMBER(11)

View Answer

Ans : C

Explanation: Char(11) is the data type for the given column.

4. Determine the most suitable data type for the given column? Column Name: Profile\_Image ; Description: Image of the employee

A. Clob  
B. Blob  
C. Varchar(100)  
D. None of the above

View Answer

Ans : B

Explanation: Blob is the data type for the given column.

5. The user defined data type can be created using

A. Create datatype  
B. Create data  
C. Create definetype  
D. Create type

View Answer

Ans : D

Explanation: The create type clause can be used to define new types.Syntax : create type Dollars as numeric(12,2) final; .

6. In contemporary databases, the top level of the hierarchy consists of \_\_\_\_\_\_ each of which can contain \_\_\_\_\_

A. Catalogs, schemas  
B. Schemas, catalogs  
C. Alter typeEnvironment, schemas  
D. Schemas, Environment

View Answer

Ans : A

Explanation: In contemporary databases, the top level of the hierarchy consists of Catalogs each of which can contain schemas.

7. Choose the most suitable data type in case multiple data types are possible for the column. Column Name: PIN\_Code ; Description: Six digit numeric PIN code for any address in India ; Example: 560100

A. Integer  
B. VARCHAR2(11)  
C. CHAR(11)  
D. Number(6)

View Answer

Ans : D

Explanation: Number(6) is the most suitable data type in case multiple data types are possible for the column

8. Choose the most suitable data type in case multiple data types are possible for the column. Column Name: Student\_id ; Description: Unique number assigned to every Student ; Example: 100000

A. Integer  
B. VARCHAR2(11)  
C. CHAR(11)  
D. Number(6)

View Answer

Ans : A

Explanation: Integer is the most suitable data type in case multiple data types are possible for the column

9. Choose the most suitable data type in case multiple data types are possible for the column.Column Name: Date\_Of\_Birth ; Description: Date of Birth of the employee ; Example: 1990/01/01

A. Timestamp  
B. VARCHAR2(11)  
C. Blob  
D. Date

View Answer

Ans : D

Explanation: Date is the most suitable data type in case multiple data types are possible for the column

10.  SQL allows comparison operations on the data types i.e.

A. Date  
B. Timestamp  
C. Time  
D. All of the above

View Answer

Ans : D

Explanation: SQL allows comparison operations on the all of the above data types .

DDL:

  [CREATE](https://www.w3schools.in/mysql/php-mysql-create/) - to create a database and its objects like (table, index, views, store procedure, function, and triggers)

 ALTER - alters the structure of the existing database

 DROP - delete objects from the database

 TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed

 COMMENT - add comments to the data dictionary

 RENAME - rename an object

|  |  |
| --- | --- |
| CREATE | Used for creating database objects like a database and a database table. |
| ALTER | Used for modifying and renaming elements of an existing database table. |
| DROP | Used for removing an entire database or a database table. |
| TRUNCATE | Used to remove all the records from a database table. |
| COMMENT | Used to write comments within SQL queries. |

CREATE DATABASE DatabaseName;  
CREATE TABLE TableName (Column1 Datatype1, Column2 Datatype2,…,ColumnNDatatypeN);

For example, we will create a table named ‘Emp’ with some fields and similar data types that are valid in MySQL and respectively a database named ‘EmpDB’ in MySQL server using the queries below:

Example:

CREATE DATABASE EmpDB;

CREATE TABLE empdb.Emp (Emp\_ID INT PRIMARY KEY AUTO\_INCREMENT, Emp\_Name VARCHAR(255), Emp\_City VARCHAR(255), Emp\_AdmDate DATE NOT NULL);

Assignment:

Crate table with student and columns id,name,age,address

Alter:

Syntax:

ALTER TABLE TableName ADD ColumnNameData\_Type;  
ALTER TABLE TableName DROPColumnName;  
ALTER TABLE TableName MODIFY COLUMNColumnNameData\_Type;

Example:

ALTER TABLE empdb.Emp ADD Emp\_Contact INT NOT NULL;

ALTER TABLE empdb.Emp DROP Emp\_Contact;

select \* from empdb.emp;

ALTER TABLE empdb.Emp MODIFY COLUMN Emp\_AdmDate Year;

ALTER TABLE table\_name RENAME COLUMN old\_column\_name TO new\_column\_name;

Assignment:alter student table

Drop:

Syntax:

**Code:**

DROP TABLE TableName;

Example:

DROP Table Emp;

Assignment :drop student table:

Assignment:create another table and insert records and truncate records using below command

Truncate:

Syntax:

TRUNCATE TABLE TableName;

Example:

TRUNCATE TABLE microdb.users;

#### 5. COMMENT Command

/\*selectingall columns from table Employee existing in the database: \*/

SELECT \* FROM microdb.users;

DML Commands:

DML is short name of **Data Manipulation Language** which deals with data manipulation and includes most common SQL statements such SELECT, INSERT, UPDATE, DELETE, etc., and it is used to store, modify, retrieve, delete and update data in a database.

* [SELECT](https://www.w3schools.in/mysql/php-mysql-select/) - retrieve data from a database
* [INSERT](https://www.w3schools.in/mysql/php-mysql-insert/) - insert data into a table
* [UPDATE](https://www.w3schools.in/mysql/php-mysql-update/) - updates existing data within a table
* [DELETE](https://www.w3schools.in/mysql/php-mysql-delete/) - Delete all records from a database table

#### 1. INSERT

It is used to insert or add new rows or records in the existing table.

**Syntax:**

Insert into <table\_name> values(<value1>,<value2>,<value3>…….,<valuen>);

Where,

* **table name:** The name of the table In which the data needs to be inserted.
* **values:** values for each column of the table.

create table students (roll\_no int,student\_name varchar(150),course varchar(150));

insert into students values(1,'ashish','java');

Insert into students values(2,’rahul’,’C++’);

select \* from students;

select student\_name from students;

select roll\_no, student\_name, course from students where roll\_no=3;

update:

Update<table\_name> set <column\_name>=value where <condition>;

Below is the query of the update statement:

update students set roll\_no=roll\_no+10 where student\_name='ashish';

update students12 set student\_name='aman' where roll\_no=2;

select \* from students;

4. Delete

Delete statement is used to delete rows of the table based on the specified conditions.

Syntax:

delete from <table\_name> where <condition>;

table\_name: Name of the table from which the data needs to be deleted.

condition: Condition based on which the data is to be deleted.

**.**

select \* from students;

Output:

Output-1.7

delete from students where roll\_no=11;

select \* from students;

Output:

Output-1.8

delete from students where student\_name= 'divya';

Output:

Output-1.9

delete from students12 where course='Arch';

**. DELETE :**   
DELETE is a [DML(Data Manipulation Language)](https://www.geeksforgeeks.org/difference-between-ddl-and-dml-in-dbms/) command and is used when we specify the row(tuple) that we want to remove or delete from the table or relation. The DELETE command can contain a WHERE clause. If the **WHERE** clause is used with the DELETE command then it removes or deletes only those rows(tuple) that satisfy the condition otherwise by default it removes all the tuples(rows) from the table.  Remember that DELETE logs the row deletions.

**Syntax of DELETE command :**

DELETE FROM TableName

WHERE condition;

**2. TRUNCATE :**   
TRUNCATE is a [DDL(Data Definition Language)](https://www.geeksforgeeks.org/difference-between-ddl-and-dml-in-dbms/) command and is used to delete all the rows or tuples from a table. Unlike the DELETE command, the TRUNCATE command does not contain a WHERE clause. In the TRUNCATE command, the transaction log for each deleted data page is not recorded. Unlike the DELETE command, the TRUNCATE command is fast. We cannot roll back the data after using the TRUNCATE command.

**Syntax of TRUNCATE command:-**

TRUNCATE TABLE TableName;

DQL COMMAND:

Select:

**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 |

### **DCL (Data Control Language):**

DCL includes commands such as GRANT and REVOKE which mainly deal with the rights, permissions, and other controls of the database system.

List of  DCL commands:

* [**GRANT:**](https://www.geeksforgeeks.org/mysql-grant-revoke-privileges/)This commandgives users access privileges to the database.
* [**REVOKE:**](https://www.geeksforgeeks.org/difference-between-grant-and-revoke/)This command withdraws the user’s access privileges given by using the GRANT command.

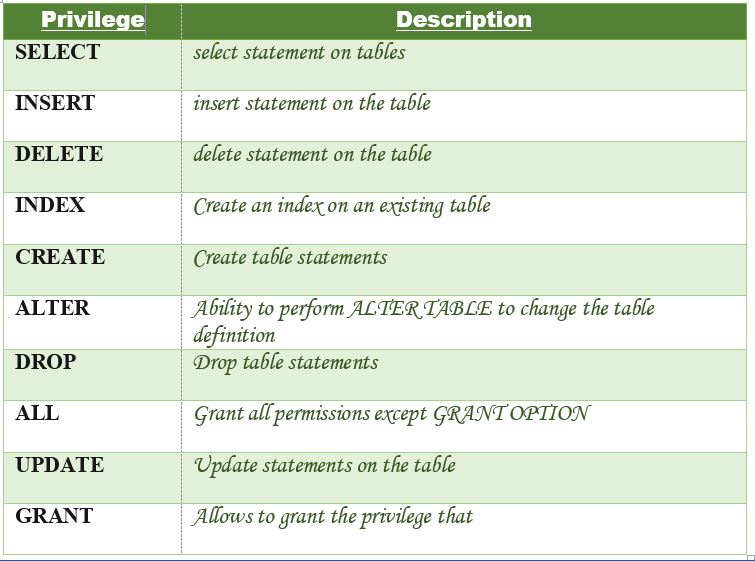
Grant:

**Syntax:**

GRANT privileges\_names ON object TO user;

**Parameters Used**:

* **privileges\_name**: These are the access rights or privileges granted to the user.
* **object:**It is the name of the database object to which permissions are being granted. In the case of granting privileges on a table, this would be the table name.
* **user:**It is the name of the user to whom the privileges would be granted.

**Privileges**:  
The privileges that can be granted to the users are listed below along with description:  


Let us now learn about different ways of granting privileges to the users:

1. **Granting SELECT Privilege to a User in a Table**: To grant Select Privilege to a table named “users” where User Name is Amit, the following GRANT statement should be executed.

GRANT SELECT ON Users TO'Amit'@'localhost;

1. **Granting more than one Privilege to a User in a Table**: To grant multiple Privileges to a user named “Amit” in a table “users”, the following GRANT statement should be executed.

GRANT SELECT, INSERT, DELETE, UPDATE ON Users TO 'Amit'@'localhost;

1. **Granting All the Privilege to a User in a Table**: To Grant all the privileges to a user named “Amit” in a table “users”, the following Grant statement should be executed.

GRANT ALL ON Users TO 'Amit'@'localhost;

1. **Granting a Privilege to all Users in a Table**: To Grant a specific privilege to all the users in a table “users”, the following Grant statement should be executed.

GRANT SELECT ON Users TO '\*'@'localhost;

In the above example the “\*” symbol is used to grant select permission to all the users of the table “users”.

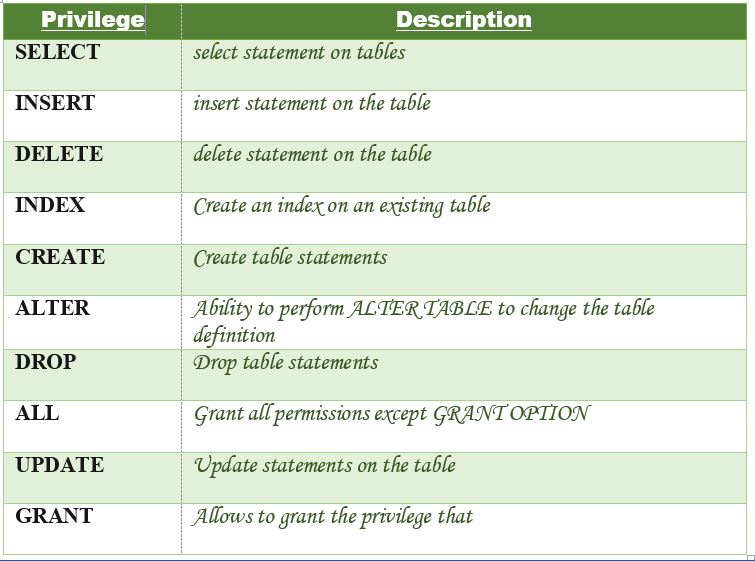
The Revoke statement is used to revoke some or all of the privileges which have been granted to a user in the past.

**Syntax:**

REVOKE privileges ON object FROM user;

**Parameters Used**:

* **object:**It is the name of the database object from which permissions are being revoked. In the case of revoking privileges from a table, this would be the table name.
* **user:**It is the name of the user from whom the privileges are being revoked.

**Privileges**  
Privileges can be of the following values:  


Different ways of revoking privileges from a user:

1. **Revoking SELECT Privilege to a User in a Table**: To revoke Select Privilege to a table named “users” where User Name is Amit, the following revoke statement should be executed.

REVOKE SELECT ON users TO 'Amit'@localhost';

1. **Revoking more than Privilege to a User in a Table**: To revoke multiple Privileges to a user named “Amit” in a table “users”, the following revoke statement should be executed.

REVOKE SELECT, INSERT, DELETE, UPDATE ON Users TO 'Amit'@'localhost;

1. **Revoking All the Privilege to a User in a Table**: To revoke all the privileges to a user named “Amit” in a table “users”, the following revoke statement should be executed.

REVOKE ALL ON Users TO 'Amit'@'localhost;

1. **Revoking a Privilege to all Users in a Table**: To Revoke a specific privilege to all the users in a table “users”, the following revoke statement should be executed.

REVOKE SELECT ON Users TO '\*'@'localhost;

# **TCL Commands in SQL**

* In SQL, TCL stands for **Transaction control language**.
* A single unit of work in a database is formed after the consecutive execution of commands is known as a transaction.
* There are certain commands present in SQL known as TCL commands that help the user manage the transactions that take place in a database.
* **COMMIT. ROLLBACK** and **SAVEPOINT** are the most commonly used TCL commands in SQL.

Now let us take a deeper dive into the TCL commands of SQL with the help of examples. All the queries in the examples will be written using the MySQL database.

### **1. COMMIT**

COMMIT command in SQL is used to save all the transaction-related changes permanently to the disk. Whenever DDL commands such as INSERT, UPDATE and DELETE are used, the changes made by these commands are permanent only after closing the current session. So before closing the session, one can easily roll back the changes made by the DDL commands. Hence, if we want the changes to be saved permanently to the disk without closing the session, we will use the commit command.

**Syntax:**

1. **COMMIT**;

**Example:**

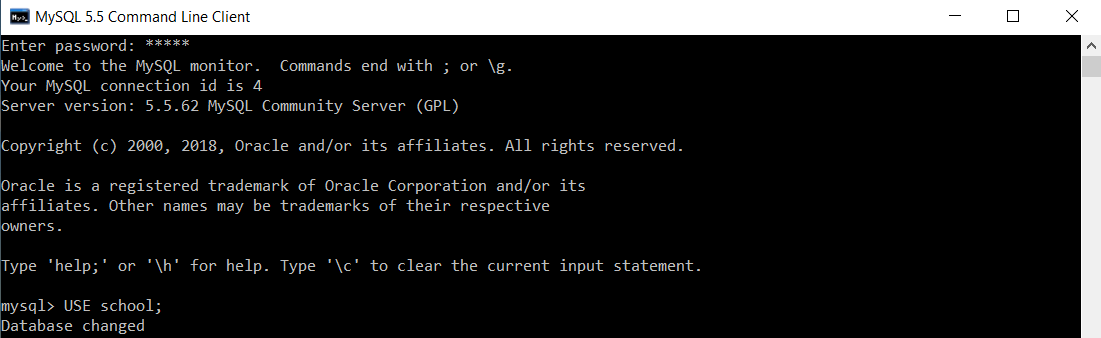
33.9M

735

Exception Handling in Java - Jav

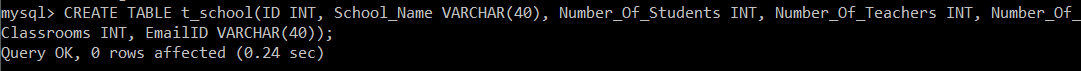
We will select an existing database, i.e., school.

1. mysql> USE school;



To create a table named t\_school, we will execute the following query:

1. mysql> **CREATE** **TABLE** t\_school(ID **INT**, School\_Name **VARCHAR**(40), Number\_Of\_Students **INT**, Number\_Of\_Teachers **INT**, Number\_Of\_Classrooms **INT**, EmailID **VARCHAR**(40));



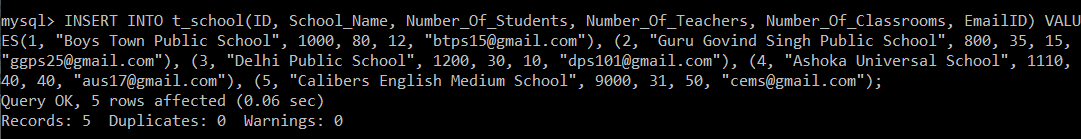
BEGIN / START TRANSACTION command is used to start the transaction.

1. mysql> START **TRANSACTION**;

TCL Commands in SQL

Now, we will execute the following query to insert multiple records at the same time in the t\_school table.

1. mysql> **INSERT** **INTO** t\_school(ID, School\_Name, Number\_Of\_Students, Number\_Of\_Teachers, Number\_Of\_Classrooms, EmailID) **VALUES**(1, "Boys Town Public School", 1000, 80, 12, "btps15@gmail.com"), (2, "Guru Govind Singh Public School", 800, 35, 15, "ggps25@gmail.com"), (3, "Delhi Public School", 1200, 30, 10, "dps101@gmail.com"), (4, "Ashoka Universal School", 1110, 40, 40, "aus17@gmail.com"), (5, "Calibers English Medium School", 9000, 31, 50, "cems@gmail.com");



We will now execute the SELECT query to verify the execution of the INSERT INTO query executed above.

1. mysql> **SELECT** \***FROM** t\_school;

After executing the SELECT query on the t\_school table, you will get the following output:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **School\_Name** | **Number\_Of\_Students** | **Number\_Of\_Teachers** | **Number\_Of\_Classrooms** | **EmailID** |
| 1 | Boys Town Public School | 1000 | 80 | 12 | btps15@gmail.com |
| 2 | Guru Govind Singh Public School | 800 | 35 | 15 | ggps25@gmail.com |
| 3 | Delhi Public School | 1200 | 30 | 10 | dps101@gmail.com |
| 4 | Ashoka Universal School | 1110 | 40 | 40 | aus17@gmail.com |
| 5 | Calibers English Medium School | 9000 | 31 | 50 | cems@gmail.com |

The output of the SELECT query shows that all the records are inserted successfully.

We will execute the COMMIT command to save the results of the operations carried on the t\_school table.

1. mysql> **COMMIT**;

TCL Commands in SQL

Autocommit is by default enabled in MySQL. To turn it off, we will set the value of autocommit as 0.

1. mysql> **SET** autocommit = 0;

TCL Commands in SQL

MySQL, by default, commits every query the user executes. But if the user wishes to commit only the specific queries instead of committing every query, then turning off the autocommit is useful.

### **2. SAVEPOINT**

We can divide the database operations into parts. For example, we can consider all the insert related queries that we will execute consecutively as one part of the transaction and the delete command as the other part of the transaction. Using the SAVEPOINT command in SQL, we can save these different parts of the same transaction using different names. **For example**, we can save all the insert related queries with the savepoint named INS. To save all the insert related queries in one savepoint, we have to execute the SAVEPOINT query followed by the savepoint name after finishing the insert command execution.

**Syntax:**

1. SAVEPOINT savepoint\_name;

### **3. ROLLBACK**

While carrying a transaction, we must create savepoints to save different parts of the transaction. According to the user's changing requirements, he/she can roll back the transaction to different savepoints. Consider a scenario: We have initiated a transaction followed by the table creation and record insertion into the table. After inserting records, we have created a savepoint INS. Then we executed a delete query, but later we thought that mistakenly we had removed the useful record. Therefore in such situations, we have an option of rolling back our transaction. In this case, we have to roll back our transaction using the ROLLBACK command to the savepoint INS, which we have created before executing the DELETE query.

**Syntax:**

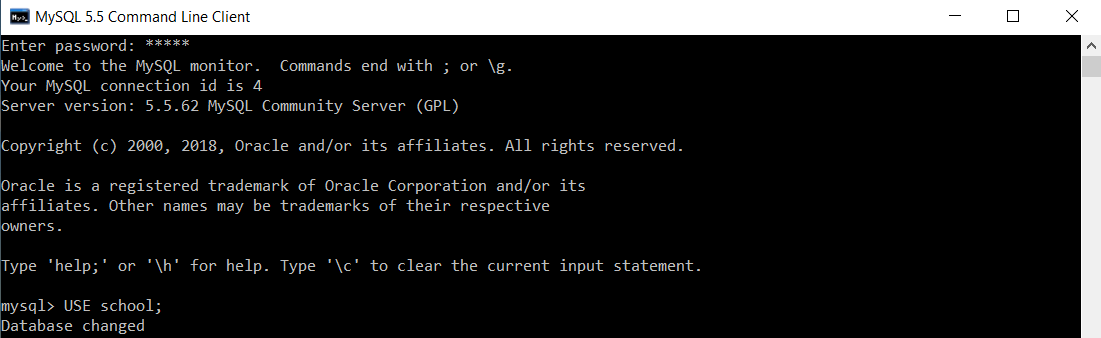
1. **ROLLBACK** **TO** savepoint\_name;

**Examples to understand the SAVEPOINT and ROLLBACK commands:**

**Example 1:**

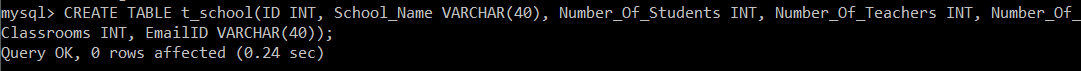
We will select an existing database, i.e., school.

1. mysql> USE school;



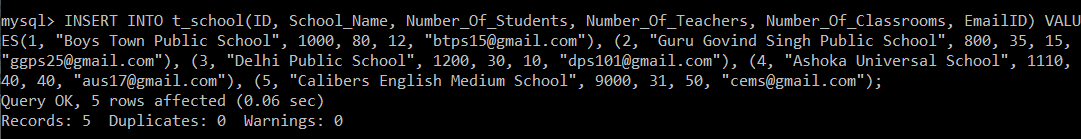
To create a table named t\_school, we will execute the following query:

1. mysql> **CREATE** **TABLE** t\_school(ID **INT**, School\_Name **VARCHAR**(40), Number\_Of\_Students **INT**, Number\_Of\_Teachers **INT**, Number\_Of\_Classrooms **INT**, EmailID **VARCHAR**(40));



Now, we will execute the following query to insert multiple records at the same time in the t\_school table.

1. mysql> **INSERT** **INTO** t\_school(ID, School\_Name, Number\_Of\_Students, Number\_Of\_Teachers, Number\_Of\_Classrooms, EmailID) **VALUES**(1, "Boys Town Public School", 1000, 80, 12, "btps15@gmail.com"), (2, "Guru Govind Singh Public School", 800, 35, 15, "ggps25@gmail.com"), (3, "Delhi Public School", 1200, 30, 10, "dps101@gmail.com"), (4, "Ashoka Universal School", 1110, 40, 40, "aus17@gmail.com"), (5, "Calibers English Medium School", 9000, 31, 50, "cems@gmail.com");



We will now execute the SELECT query to verify the execution of the INSERT INTO query executed above.

1. mysql> **SELECT** \***FROM** t\_school;

After executing the SELECT query on the t\_school table, you will get the following output:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **School\_Name** | **Number\_Of\_Students** | **Number\_Of\_Teachers** | **Number\_Of\_Classrooms** | **EmailID** |
| 1 | Boys Town Public School 1000 | 80 | 12 | btps15@gmail.com |  |
| 2 | Guru Govind Singh Public School | 800 | 35 | 15 | ggps25@gmail.com |
| 3 | Delhi Public School | 1200 | 30 | 10 | dps101@gmail.com |
| 4 | Ashoka Universal School | 1110 | 40 | 40 | aus17@gmail.com |
| 5 | Calibers English Medium School | 9000 | 31 | 50 | cems@gmail.com |

The output of the SELECT query shows that all the records are inserted successfully.

BEGIN / START TRANSACTION command is used to start the transaction.

1. mysql> START **TRANSACTION**;

TCL Commands in SQL

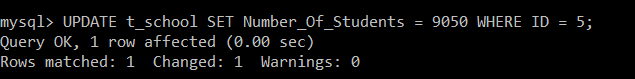
As we know, the SAVEPOINT command in SQL is used to save the different parts of the same transaction using different names. Consider till this point as one part of our transaction. We will save this part using a savepoint named Insertion.

1. mysql> SAVEPOINT Insertion;

TCL Commands in SQL

Now, we will execute the update command on the t\_school table to set the Number\_Of\_Students as 9050 for the record with ID 5.

1. mysql> **UPDATE** t\_school **SET** Number\_Of\_Students = 9050 **WHERE** ID = 5;



To verify that the record with ID 5 now has the Number\_Of\_Students as 9050, we will execute the SELECT query.

1. mysql> **SELECT** \***FROM** t\_school;

After executing the SELECT query on the t\_school table, you will get the following output:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **School\_Name** | **Number\_Of\_Students** | **Number\_Of\_Teachers** | **Number\_Of\_Classrooms** | **EmailID** |
| 1 | Boys Town Public School | 1000 | 80 | 12 | btps15@gmail.com |
| 2 | Guru Govind Singh Public School | 800 | 35 | 15 | ggps25@gmail.com |
| 3 | Delhi Public School | 1200 | 30 | 10 | dps101@gmail.com |
| 4 | Ashoka Universal School | 1110 | 40 | 40 | aus17@gmail.com |
| 5 | Calibers English Medium School | 9050 | 31 | 50 | cems@gmail.com |

The output of the SELECT query shows that the record with ID 5 is updated successfully.

Consider the update operation as one part of our transaction. We will save this part using a savepoint named Updation.

1. mysql> SAVEPOINT Updation;

TCL Commands in SQL

Suddenly, our requirement changed, and we realized that we had updated a record that was not supposed to be. In such a scenario, we need to roll back our transaction to the savepoint, which was created prior to the execution of the UPDATE command.

1. mysql> **ROLLBACK** **TO** Insertion;

TCL Commands in SQL

We didn't need the updation carried on the record. Hence, we have rolled back to the savepoint named Insertion.

For confirming that we have got the same t\_school table that we had before carrying out the updation operation, we will again execute the SELECT query.

1. mysql> **SELECT** \***FROM** t\_school;

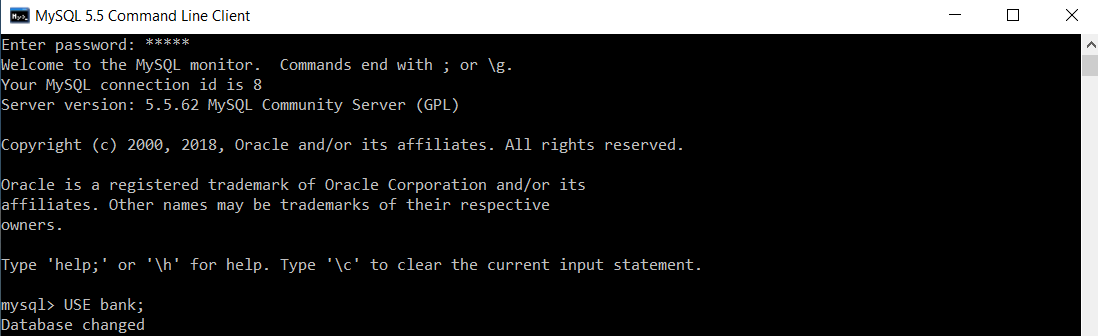
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **School\_Name** | **Number\_Of\_Students** | **Number\_Of\_Teachers** | **Number\_Of\_Classrooms** | **EmailID** |
| 1 | Boys Town Public School | 1000 | 80 | 12 | btps15@gmail.comm |
| 2 | Guru Govind Singh Public School | 800 | 35 | 15 | ggps25@gmail.comm |
| 3 | Delhi Public School | 1200 | 30 | 10 | dps101@gmail.comm |
| 4 | Ashoka Universal School | 1110 | 40 | 40 | aus17@gmail.comm |
| 5 | Calibers English Medium School | 9000 | 31 | 50 | cems@gmail.com |

The SELECT query output confirms that the transaction is now successfully rolled back to the savepoint 'Insertion'.

**Example 2:**

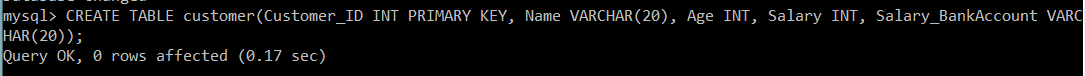
We will select an existing database, i.e., bank.

1. mysql> USE bank;



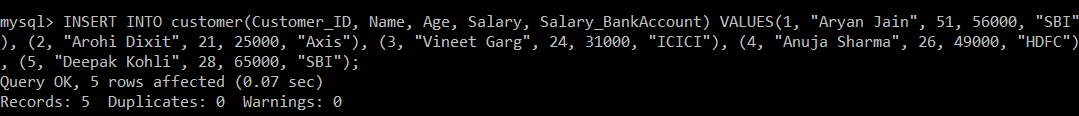
To create a table named customer, we will execute the following query:

1. mysql> **CREATE** **TABLE** customer(Customer\_ID **INT** **PRIMARY** **KEY**, **Name** **VARCHAR**(20), Age **INT**, Salary **INT**, Salary\_BankAccount **VARCHAR**(20));



Now, we will execute the following query to insert multiple records at the same time in the customer table.

1. mysql> **INSERT** **INTO** customer(Customer\_ID, **Name**, Age, Salary, Salary\_BankAccount) **VALUES**(1, "Aryan Jain", 51, 56000, "SBI"), (2, "Arohi Dixit", 21, 25000, "Axis"), (3, "Vineet Garg", 24, 31000, "ICICI"), (4, "Anuja Sharma", 26, 49000, "HDFC"), (5, "Deepak Kohli", 28, 65000, "SBI");



We will now execute the SELECT query to verify the execution of the INSERT INTO query executed above.

1. mysql> **SELECT** \***FROM** customer;

After executing the SELECT query on the t\_school table, you will get the following output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Customer\_ID** | **Name** | **Age** | **Salary** | **Salary\_BankAccount** |
| 1 | Aryan Jain | 51 | 56000 | SBI |
| 2 | Arohi Dixit | 21 | 25000 | Axis |
| 3 | Vineet Garg | 24 | 31000 | ICICI |
| 4 | Anuja Sharma | 26 | 49000 | HDFC |
| 5 | Deepak Kohli | 28 | 65000 | SBI |

The output of the SELECT query shows that all the records are inserted successfully.

BEGIN / START TRANSACTION command is used to start the transaction.

1. mysql> START **TRANSACTION**;

TCL Commands in SQL

As we know, the SAVEPOINT command in SQL is used to save the different parts of the same transaction using different names. Consider till this point as one part of our transaction. We will save this part using a savepoint named Insertion.

1. mysql> SAVEPOINT Insertion;

TCL Commands in SQL

We will execute the delete command on the customer table to remove the record with ID 5.

1. mysql> **DELETE** **FROM** customer **WHERE** Customer\_ID = 5;

TCL Commands in SQL

We will execute the SELECT query to verify that the record with ID 5 has been removed.

1. mysql> **SELECT** \***FROM** customer;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Customer\_ID** | **Name** | **Age** | **Salary** | **Salary\_BankAccount** |
| 1 | Aryan Jain | 51 | 56000 | SBI |
| 2 | Arohi Dixit | 21 | 25000 | Axis |
| 3 | Vineet Garg | 24 | 31000 | ICICI |
| 4 | Anuja Sharma | 26 | 49000 | HDFC |

The output of the SELECT query shows that the record with ID 5 is removed successfully.

Consider the delete operation as one part of our transaction. We will save this part using a savepoint named Deletion.

1. mysql> SAVEPOINT Deletion;

TCL Commands in SQL

Suddenly, our requirement changed, and we realized that we had deleted a record that was not supposed to be. In such a scenario, we need to roll back our transaction to the savepoint, which was created prior to the execution of the DELETE command.

1. mysql> **ROLLBACK** **TO** Insertion;

We didn't need the deletion carried on the record. Hence, we have rolled back to the savepoint named Insertion.

For confirming that we have got the same customer table that we had before carrying out the deletion operation, we will again execute the SELECT query.

1. mysql> **SELECT** \***FROM** customer;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Customer\_ID** | **Name** | **Age** | **Salary** | **Salary\_BankAccount** |
| 1 | Aryan Jain | 51 | 56000 | SBI |
| 2 | Arohi Dixit | 21 | 25000 | Axis |
| 3 | Vineet Garg | 24 | 31000 | ICICI |
| 4 | Anuja Sharma | 26 | 49000 | HDFC |
| 5 | Deepak Kohli | 28 | 65000 | SBI |

The SELECT query output confirms that the transaction is now successfully rolled back to the savepoint 'Insertion'.

# SQL | Constraints

Constraints are the rules that we can apply on the type of data in a table. That is, we can specify the limit on the type of data that can be stored in a particular column in a table using constraints.

The available constraints in SQL are: 

* **NOT NULL**: This constraint tells that we cannot store a null value in a column. That is, if a column is specified as NOT NULL then we will not be able to store null in this particular column any more.
* **UNIQUE**: This constraint when specified with a column, tells that all the values in the column must be unique. That is, the values in any row of a column must not be repeated.
* **PRIMARY KEY**: A primary key is a field which can uniquely identify each row in a table. And this constraint is used to specify a field in a table as primary key.
* **FOREIGN KEY**: A Foreign key is a field which can uniquely identify each row in a another table. And this constraint is used to specify a field as Foreign key.
* **CHECK**: This constraint helps to validate the values of a column to meet a particular condition. That is, it helps to ensure that the value stored in a column meets a specific condition.
* **DEFAULT**: This constraint specifies a default value for the column when no value is specified by the user.

**How to specify constraints?**   
We can specify constraints at the time of creating the table using CREATE TABLE statement. We can also specify the constraints after creating a table using ALTER TABLE statement.

**Syntax**:   
Below is the syntax to create constraints using CREATE TABLE statement at the time of creating the table. 

CREATE TABLE sample\_table

(

column1 data\_type(size) constraint\_name,

column2 data\_type(size) constraint\_name,

column3 data\_type(size) constraint\_name,

....

);

**sample\_table**: Name of the table to be created.

**data\_type**: Type of data that can be stored in the field.

**constraint\_name**: Name of the constraint. for example- NOT NULL, UNIQUE, PRIMARY KEY etc.

Let us see each of the constraint in detail. 

**1. NOT NULL –**   
If we specify a field in a table to be NOT NULL. Then the field will never accept null value. That is, you will be not allowed to insert a new row in the table without specifying any value to this field.   
For example, the below query creates a table Student with the fields ID and NAME as NOT NULL. That is, we are bound to specify values for these two fields every time we wish to insert a new row. 

CREATE TABLE Student

(

ID int(6) NOT NULL,

NAME varchar(10) NOT NULL,

ADDRESS varchar(20)

);

**2. UNIQUE** **–**  
This constraint helps to uniquely identify each row in the table. i.e. for a particular column, all the rows should have unique values. We can have more than one UNIQUE columns in a table.   
For example, the below query creates a table Student where the field ID is specified as UNIQUE. i.e, no two students can have the same ID. [Unique constraint in detail.](https://www.geeksforgeeks.org/sql-unique-constraint/) 

CREATE TABLE Student

(

ID int(6) NOT NULL UNIQUE,

NAME varchar(10),

ADDRESS varchar(20)

);

**3. PRIMARY KEY –**   
Primary Key is a field which uniquely identifies each row in the table. If a field in a table as primary key, then the field will not be able to contain NULL values as well as all the rows should have unique values for this field. So, in other words we can say that this is combination of NOT NULL and UNIQUE constraints.   
A table can have only one field as primary key. Below query will create a table named Student and specifies the field ID as primary key. 

CREATE TABLE Student

(

ID int(6) NOT NULL UNIQUE,

NAME varchar(10),

ADDRESS varchar(20),

PRIMARY KEY(ID)

);

1. **FOREIGN KEY –**   
   Foreign Key is a field in a table which uniquely identifies each row of a another table. That is, this field points to primary key of another table. This usually creates a kind of link between the tables.   
   Consider the two tables as shown below:

**CREATE** **TABLE** Persons (

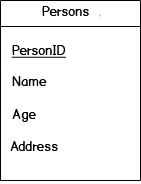
PersonID int AUTO\_INCREMENT **PRIMARY** **KEY**,

Name **VARCHAR**(20) NOT **NULL**,

Age int,

Address **VARCHAR**(100)

);



The column **PersonID** is a primary key of “Persons” table. This means that the values in **PersonID** column uniquely identify the rows in the table. The following statement display all the data in “Persons” table.

> SELECT \* FROM persons;

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

| PersonID | Name | age | Address |

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

| 101 | Alex | 25 | 819 Saint Francis Way |

| 102 | Emily | 15 | 171 Jarvisville Road Michigan |

| 103 | Jean | 35 | 188 Clay Street Indiana |

| 104 | Bob | 40 | 285 Java Lane Missouri |

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

###### **Foreign Key**

A foreign key is a simple mechanism to ensure referential integrity between data in different tables. In other words, the foreign key forces a table to be linked to the data of another table. In the following example, “Orders” table is linked to “Persons” table by **PersonID**.  
   
**Example of Foreign Key:**  
Let’s assume that each person has made orders. To store the orders, you can create a new table named “Orders”:

**CREATE** **TABLE** Orders (

OrderID int AUTO\_INCREMENT **PRIMARY** **KEY**,

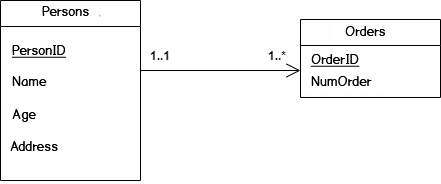
NumOrder int NOT **NULL**,

PersonID int,

**FOREIGN** **KEY** (PersonID) **REFERENCES** Persons(PersonID)

);

The column “PersonID” is a foreign key that refers to the column “PersonID” in the table “Persons”. We used the “Foreign Key” constraint to establish this relationship.



The following statement display all the data in “Orders” table.

> SELECT \* FROM orders;

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

| OrderID | NumOrder | PersonID |

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

| 55 | 00001 | 101 |

| 56 | 00002 | 101 |

| 57 | 00003 | 102 |

| 58 | 00004 | 104 |

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

**(i) CHECK –**   
Using the CHECK constraint we can specify a condition for a field, which should be satisfied at the time of entering values for this field.   
For example, the below query creates a table Student and specifies the condition for the field AGE as (AGE >= 18 ). That is, the user will not be allowed to enter any record in the table with AGE < 18. [Check constraint in detail](https://www.geeksforgeeks.org/sql-check-constraint/) 

CREATE TABLE Student

(

ID int(6) NOT NULL,

NAME varchar(10) NOT NULL,

AGE int NOT NULL CHECK (AGE >= 18)

);

**(ii) DEFAULT –**   
This constraint is used to provide a default value for the fields. That is, if at the time of entering new records in the table if the user does not specify any value for these fields then the default value will be assigned to them.   
For example, the below query will create a table named Student and specify the default value for the field AGE as 18. 

CREATE TABLE Student

(

ID int(6) NOT NULL,

NAME varchar(10) NOT NULL,

AGE int DEFAULT 18

);

## What are SQL operators?

SQL operators are reserved keywords used in the WHERE clause of a [SQL statement](https://www.edureka.co/blog/sql-commands) to perform arithmetic, logical and comparison operations. Operators act as conjunctions in SQL statements to fulfill multiple conditions in a statement.

* [Arithmetic operator](https://www.geeksforgeeks.org/sql-arithmetic-operators/)
* Comparison operator
* [Logical operator](https://www.geeksforgeeks.org/sql-and-and-or-operators/)
* [Arithmetic operator](https://www.geeksforgeeks.org/sql-arithmetic-operators/):
* Arithmetic operators
* Comparison operators
* Logical operators
* Operators used to negate conditions

## SQL Arithmetic Operators

Assume **'variable a'** holds 10 and **'variable b'** holds 20, then −

[Show Examples](https://www.tutorialspoint.com/sql/sql-arithmetic-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + (Addition) | Adds values on either side of the operator. | a + b will give 30 |
| - (Subtraction) | Subtracts right hand operand from left hand operand. | a - b will give -10 |
| \* (Multiplication) | Multiplies values on either side of the operator. | a \* b will give 200 |
| / (Division) | Divides left hand operand by right hand operand. | b / a will give 2 |
| % (Modulus) | Divides left hand operand by right hand operand and returns remainder. | b % a will give 0 |

## SQL Comparison Operators

Assume **'variable a'** holds 10 and **'variable b'** holds 20, then −

[Show Examples](https://www.tutorialspoint.com/sql/sql-comparison-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (a = b) is not true. |
| != | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (a != b) is true. |
| <> | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (a <> b) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (a > b) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (a < b) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (a >= b) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (a <= b) is true. |
| !< | Checks if the value of left operand is not less than the value of right operand, if yes then condition becomes true. | (a !< b) is false. |
| !> | Checks if the value of left operand is not greater than the value of right operand, if yes then condition becomes true. | (a !> b) is true. |

## SQL Logical Operators

Here is a list of all the logical operators available in SQL.

[Show Examples](https://www.tutorialspoint.com/sql/sql-logical-operators.htm)

|  |  |
| --- | --- |
| **Sr.No.** | **Operator & Description** |
| 1 | **ALL**  The ALL operator is used to compare a value to all values in another value set. |
| 2 | **AND**  The AND operator allows the existence of multiple conditions in an SQL  statement's WHERE clause. |
| 3 | **ANY**  The ANY operator is used to compare a value to any applicable value in  the list as per the condition. |
| 4 | **BETWEEN**  The BETWEEN operator is used to search for values that are within a set of values,  given the minimum value and the maximum value. |
| 5 | **EXISTS**  The EXISTS operator is used to search for the presence of a row in a  specified table that meets a certain criterion. |
| 6 | **IN**  The IN operator is used to compare a value to a list of literal values  that have been specified. |
| 7 | **LIKE**  The LIKE operator is used to compare a value to similar values using wildcard  operators. |
| 8 | **NOT**  The NOT operator reverses the meaning of the logical operator with which it is used.  Eg: NOT EXISTS, NOT BETWEEN, NOT IN, etc. **This is a negate operator.** |
| 9 | **OR**  The OR operator is used to combine multiple conditions in an SQL statement's  WHERE clause. |
| 10 | **IS NULL**  The NULL operator is used to compare a value with a NULL value. |
| 11 | **UNIQUE**  The UNIQUE operator searches every row of a specified table for  uniqueness (no duplicates). |

Arthimatic Examples:

## Example 1

SQL> select 10+ 20;

## Output

+--------+

| 10+ 20 |

+--------+

| 30 |

+--------+

1 row in set (0.00 sec)

## Example 2

SQL> select 10 \* 20;

## Output

+---------+

| 10 \* 20 |

+---------+

| 200 |

+---------+

1 row in set (0.00 sec)

## Example 3

SQL> select 10 / 5;

## Output

+--------+

| 10 / 5 |

+--------+

| 2.0000 |

+--------+

1 row in set (0.03 sec)

## Example 4

SQL> select 12 % 5;

## Output

+---------+

| 12 % 5 |

+---------+

| 2 |

+---------+

1 row in set (0.00 sec)

## SQL Comparison Operators:

* SQL Equal (=) Operator
* SQL Not Equal (!= or <>) Operator
* SQL Greater Than (>) Operator
* SQL Less Than (<) Operator
* SQL Greater Than or Equal To (>=) Operator
* SQL Less Than or Equal To (<=) Operator
* SQL Not Less Than (!<) Operator
* SQL Not Greater Than (!>) Operator

Before we proceed to check each operator try to create “**EmployeeDetails**” table by using below script in SQL database

create table EmployeeDetails(empid int, empname varchar(50),designation varchar(50),salary int,Location varchar(50))

insert into EmployeeDetails

values(1,'suresh','software engineer',25000,'chennai'),

(2,'rohini','AEO',15000,'chennai'),

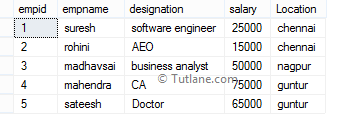
(3,'madhavsai','business analyst',50000,'nagpur'),

(4,'mahendra','CA',75000,'guntur'),

(5,'sateesh','Doctor',65000,'guntur')

select \* from EmployeeDetails

Once we run above SQL script our table “**EmployeeDetails**” will create and result will be like as shown below



Now we will learn each comparison operator in SQL with proper examples

SQL Equal (=) Operator

In SQL, the **equal** operator is useful to check whether the given two expressions equal or not. If it’s equal, then the condition will be true and it will return matched records.

**Example**:

If we run following SQL statement for the equal operator it will return records where **empid equals to 1**.

SELECT \* FROM EmployeeDetails WHERE empid = 1

When we execute the above SQL equal operator query, we will get the result like as shown below.

Output of Equal (=) Operator Example in SQL Server

SQL Not Equal (!=) Operator

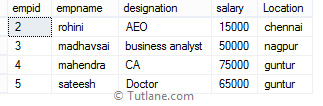
In SQL, **not equal** operator is used to check whether two expressions equal or not. If it’s not equal then the condition will be true and it will return not matched records.

**Example**:

If we run following SQL statement for not equal operator it will return a records where **empid not equals to 1**.

SELECT \* FROM EmployeeDetails WHERE empid != 1

When we execute the above SQL not equal operator query, we will get the result like as shown below.



SQL Not Equal (<>) Operator

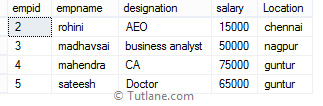
In SQL, **not equal** operator is used to check whether two expressions equal or not. If it’s not equal then condition will be true and it will return not matched records. Both **!=** and **<>** operators are not equal operators and will return same result but **!=** operator is not a ISO standard.

**Example**:

If we run following SQL statement for not equal operator it will return records where **empid not equals to 1**

SELECT \* FROM EmployeeDetails WHERE empid <> 1

When we execute the above SQL, not equal operator query we will get the result like as shown below.



SQL Greater Than (>) Operator

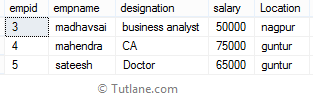
In SQL, **greater than** operator is used to check whether the left-hand operator is **higher than** the right-hand operator or not. If left-hand operator **higher than** right-hand operator then condition will be true and it will return matched records.

**Example**:

If we run following SQL statement for greater than operator it will return records where **empid greater than 2**

SELECT \* FROM EmployeeDetails WHERE empid > 2

When we execute the above SQL greater than the operator query, we will get the result like as shown below.



SQL Less Than (<) Operator

In SQL, **less than** operator is used to check whether the left-hand operator is **lower than** the right-hand operator or not. If left-hand operator **lower than** right-hand operator then condition will be true and it will return matched records.

**Example**:

If we run following SQL statement for less than operator it will return records where **empid less than 2**

SELECT \* FROM EmployeeDetails WHERE empid < 2

When we execute the above SQL less than operator query, we will get the result like as shown below.

Output of Less than (<) Operator Example in SQL Server

SQL Greater Than or Equal To (>=) Operator

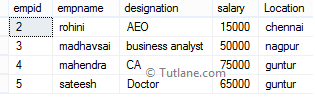
In SQL, **greater than or equal to** the operator is used to check whether the left-hand operator is **higher than or equal to** the right-hand operator or not. If left-hand operator **higher than or equal to** right-hand operator then condition will be true and it will return matched records.

**Example**:

If we run following SQL statement for greater than or equal to the operator it will return records where empid **higher than or equal to 2**

SELECT \* FROM EmployeeDetails WHERE empid >= 2

When we execute the above SQL greater than or equal to operator query, we will get the result like as shown below.



SQL Less Than or Equal To (<=) Operator

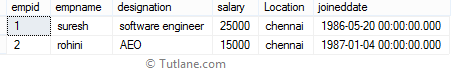
In sql, **less than or equal to** the operator is useful to check whether the left-hand operator is lower than or equal to the right-hand operator or not. If left-hand operator lower than or equal to right-hand operator then condition will be true and it will return matched records.

**Example**:

If we run following SQL statement for less than or equal to the operator it will return records where empid lower than or equal to 2

SELECT \* FROM EmployeeDetails WHERE empid <= 2

When we execute above sql less than or equal to operator query, we will get the result like as shown below.



SQL Not Less Than (!<) Operator

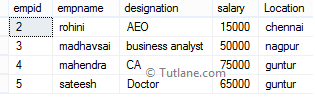
In sql, **not less than** operator is used to check whether the left-hand operator not lower than the right-hand operator or not. If left-hand operator not lower than right-hand operator then condition will be true and it will return matched records.

**Example**:

If we run following SQL statement for not less than operator it will return records where **empid not lower than 2**

SELECT \* FROM EmployeeDetails WHERE empid !< 2

When we execute the above SQL not less than operator query, we will get the result like as shown below.



SQL Not Greater Than (!>) Operator

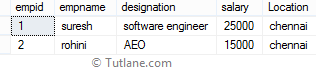
In sql, **not greater than** operator is used to check whether the left-hand operator is not higher than the right-hand operator or not. If left-hand operator not higher than right-hand operator then condition will be true and it will return matched records.

**Example**:

If we run following SQL statement for not greater than operator it will return records where **empid not higher than 2**

SELECT \* FROM EmployeeDetails WHERE empid !> 2

When we execute the above SQL not greater than operator query, we will get the result like as shown below.



SQL Logical Operators:

  [AND Operator](https://www.tutlane.com/tutorial/sql-server/sql-and-operator)

 [OR Operator](https://www.tutlane.com/tutorial/sql-server/sql-or-operator)

 [LIKE Operator](https://www.tutlane.com/tutorial/sql-server/sql-like-operator)

 [IN Operator](https://www.tutlane.com/tutorial/sql-server/sql-in-operator)

 [BETWEEN Operator](https://www.tutlane.com/tutorial/sql-server/sql-between-operator)

 [Exists Operator](https://www.tutlane.com/tutorial/sql-server/sql-exists-operator)

 [NOT Operator](https://www.tutlane.com/tutorial/sql-server/sql-not-operator)

 [SOME Operator](https://www.tutlane.com/tutorial/sql-server/sql-some-operator)

 [ALL Operator](https://www.tutlane.com/tutorial/sql-server/sql-all-operator)

 [ANY Operator](https://www.tutlane.com/tutorial/sql-server/sql-any-operator)

AND :

create table EmployeeDetails(empid int, empname varchar(50),designation varchar(50),salary int,Location varchar(50))

insert into EmployeeDetails

values(1,'suresh','software engineer',25000,'chennai'),

(2,'rohini','AEO',15000,'chennai'),

(3,'madhavsai','business analyst',50000,'nagpur'),

(4,'mahendra','CA',75000,'guntur'),

(5,'sateesh','Doctor',65000,'guntur')

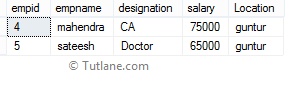
select \* from EmployeeDetails

## SQL AND Operator Example1

In the following SQL query, we are checking multiple conditions (**Location**, **Salary**) with **AND** operator. It will return records whatever it satisfies both the conditions.

SELECT \* FROM EmployeeDetails WHERE Location='guntur' AND Salary>40000

When we execute the above SQL query, we will get the result as shown below.

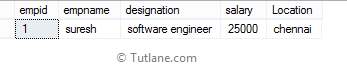


## SQL Server AND Operator Example2

In the following SQL query, we are checking **Location** and **Empname** conditions with **AND** operator. It will return records whatever it satisfies the defined conditions.

SELECT \* FROM EmployeeDetails WHERE Location='chennai' AND Empname='suresh'

When we execute the above SQL query, we will get the result as shown below.



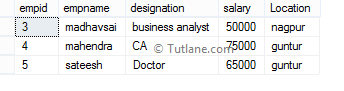
## OR:

## SQL OR Operator Example1

In the following SQL query, we are checking multiple conditions with the **OR** operator. It will return records whatever it satisfies either of one condition or both conditions.

SELECT \* FROM EmployeeDetails WHERE Location='guntur' OR Salary>40000

When we execute the above OR operator example, we will get the result as shown below.

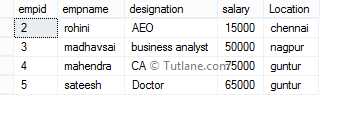


## SQL OR Operator Example2

In following SQL query we are checking multiple conditions (**empname**, **salary**) with **OR** operator. It will return the records whatever it satisfy either of one condition or both conditions.

SELECT \* FROM EmployeeDetails WHERE empname='rohini' OR Salary>40000

When we execute above OR operator example, we will get the result as shown below.

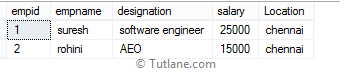


## SQL LIKE Operator Example1

The following SQL like query will return all employees whose **location** starts with character '**c**' and followed by any string of characters because here we mentioned a pattern like 'c%'. Here, the '%' is a wildcard character which we will use before or after characters to search for required matched string of characters.

SELECT \* FROM EmployeeDetails WHERE Location LIKE 'c%'

Once we execute the above SQL query our SQL like operator example result will be like as shown below.

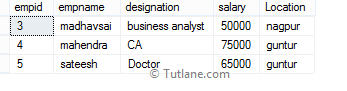


## SQL Like Operator Example2

The following SQL query will return all employees whose **location** end with character 'r' because here, we mentioned a pattern like '%r', this means it will return all the records whose **location** ends with a character 'r'.

SELECT \* FROM EmployeeDetails WHERE Location LIKE '%r'

When we execute the above SQL like query, we will get the result like as shown below.



## SQL Like Operator Example3

The following SQL query will return all the employees whose **location** contains a word called 'en' anywhere in the **location** column because we mentioned a pattern like '%en%'. This means it will check for the respective word anywhere in the column irrespective of characters in front or back.

SELECT \* FROM EmployeeDetails WHERE Location LIKE '%en%'

When we execute the above SQL like query, we will get the result like as shown below.



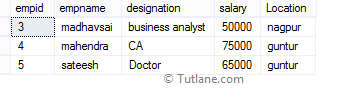
Now we will see how to use **NOT** keyword with a **LIKE** operator and will check how it will return records. Generally, if we use **NOT** keyword with **LIKE** operator it will return the records that do not contain a matching pattern.

## SQL LIKE Operator with NOT Keyword Example

The following SQL statement will return all the employees whose **location** not containing a word called '**en**', anywhere within the **location** column because we used a **NOT** keyword with **LIKE** operator and mentioned a pattern like '%en%'.

SELECT \* FROM EmployeeDetails WHERE Location NOT LIKE '%en%'

When we execute above SQL like operator with not keyword example query, we will get the result like as shown below.

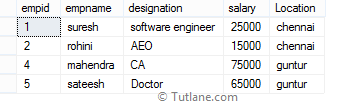


## SQL IN Operator Example

The following SQL query will return all the employees whose **location** in 'chennai', 'guntur', 'bangalore'.

SELECT \* FROM EmployeeDetails WHERE Location IN('chennai','guntur','bangalore')

When we execute the above SQL IN operator query, we will get the result like as shown below.

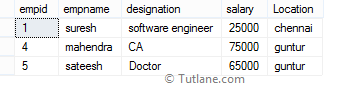


## SQL IN Operator with SubQuery Example

In the following SQL query, we will see how to use subquery with **IN** operator in the SQL server.

SELECT \* FROM EmployeeDetails WHERE empname IN(SELECT empname FROM EmployeeDetails Where empid in(1,4,5))

The above SQL query will return all the employees whose **empname** values matches with the results returned by SQL subquery. Following is the result of SQL IN operator with subquery example.



Now we will see how to use **NOT** keyword with **IN** operator and will check how it will return records. Generally if we use **NOT** keyword with **IN** operator it will return all the records that does not exist in the list of values.

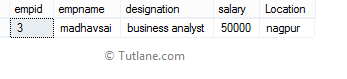
## SQL IN Operator with NOT Keyword Example

Generally, if we use **IN** operator in SQL statements that will return records whose column values matches with defined set of values. Suppose if we use **NOT** keyword with **IN** operator, then it will return data whose column values not match with the set of values.

The following SQL query will return all the records whose **location** is not in mentioned values.

SELECT \* FROM EmployeeDetails WHERE Location NOT IN('chennai','guntur','bangalore')

 Once we execute the above SQL IN operator with not keyword result, we will get the result as shown below.

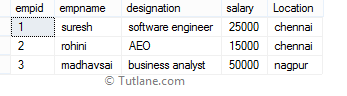


## SQL Between Operator Example

The following SQL query will return all employee's details where **empid** in between **1** and **3**.

SELECT \* FROM EmployeeDetails WHERE empid BETWEEN 1 and 3

When we execute above SQL between operator, then we will get the result like as shown below.



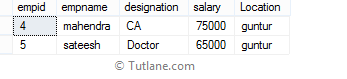
## SQL NOT BETWEEN Operator with Example

Generally, in SQL statements if we use **BETWEEN** operator then it will return a records whose value between the defined range. Suppose if we use **NOT** keyword with **BETWEEN** operator, then it will return data where the column value not in between the defined range of values.

In the following SQL query, we will see how to use **NOT** with **BETWEEN** operator in SQL. The following query will return all the employees details whose **empname** not in between **1** and **3** range.

SELECT \* FROM EmployeeDetails WHERE empid NOT BETWEEN 1 and 3

When we execute the above SQL not between operator example, we will get the result like as shown below.



Now we will see how to use **NOT** keyword with **IN** operator and will check how it will return records. Generally, if we use **NOT** keyword with **IN** operator it will return all the records that do not exist in the list of values.

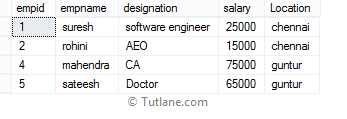
## SQL BETWEEN Operator with Character (Text) Value Example

In SQL, if we use the **BETWEEN** operator with **character string values**, then it will return all the records where the column name beginning with any letter between the character string values.

Following is the SQL query which will return all the records whose **location** name starts with any letter between 'A' and 'K'.

SELECT \* FROM EmployeeDetails WHERE Location BETWEEN 'A' AND 'K'

 When we execute the above SQL query, we will get the result for SQL between operator with a character like as shown below.



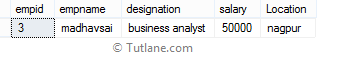
## SQL NOT BETWEEN Operator with Character (Text) Value Example

In SQL, if we use **BETWEEN** operator with **character values** it will return all the records where the column name beginning with any letter between character string values. Suppose if we use **NOT** with **BETWEEN** operator, then it will return all the records whose column name not start between string values.

The following SQL query will return all the records whose location name not begin with any letter between 'A' and 'K'.

SELECT \* FROM EmployeeDetails WHERE Location NOT BETWEEN 'A' AND 'K'

When we execute above SQL not between operator with character, then we will get the result like as shown below.



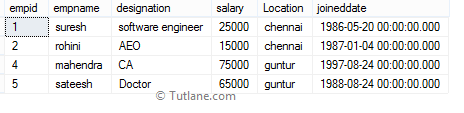
## SQL BETWEEN Operator with Date Value Example

In SQL, if we use the **BETWEEN** operator with **date values**, then it will return all the records between the defined range of date values.

The following SQL query will return all the records between defined ranges of date values ('1986-05-20', '1997-08-24').

SELECT \* FROM EmployeeDetails WHERE joineddate BETWEEN '1986-05-20' AND '1997-08-24'

When we execute the above query, we will get the result of SQL between operator with date value like as shown below.

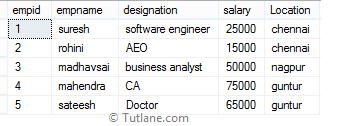


## SQL EXISTS Operator Example

Following SQL exists operator example will return an employee details who values are matching with the values returned by subquery.

SELECT \* FROM EmployeeDetails WHERE EXISTS(SELECT \* FROM EmployeeDetails WHERE empid =1)

When we execute above SQL exists operator query, we will get the result like as shown below.

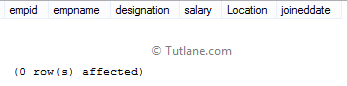


## SQL Exists Operator Example2

Following SQL exists operator query will return employee details whose values are matching with the values returned by the subquery.

SELECT \* FROM EmployeeDetails WHERE EXISTS(SELECT \* FROM EmployeeDetails WHERE empid =100)

When we execute the above SQL exists operator query, we will get the result like as shown below.



## SQL NOT EXISTS Operator

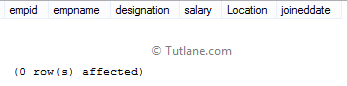
Generally, in SQL statement if we use the **EXISTS** operator, then it will return all the records whose values match with the subquery return values. Suppose if we use **NOT** with **EXISTS** operator, then it will return all the records whose values not match with the subquery values.

## SQL Not Exists Operator Example

Following SQL not exists operator example will return records whose values not exists in the subquery returned result or rows.

SELECT \* FROM EmployeeDetails WHERE NOT EXISTS(SELECT \* FROM EmployeeDetails WHERE empid =1)

When we execute the above SQL not exists operator query, we will get the result like as shown below.

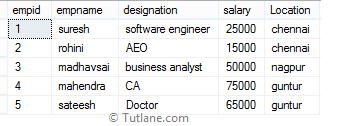


## SQL Not Exists Operator Example2

Following SQL not exists operator example will return records whose values not exists in the subquery returned result or rows.

SELECT \* FROM EmployeeDetails WHERE NOT EXISTS(SELECT \* FROM EmployeeDetails WHERE empid =100)

When we execute the above SQL not exists operator query, we will get the result like as shown below.



SQL NOT Operator:

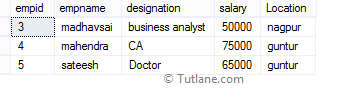
Now run following examples to check **NOT** with **LIKE**, **IN**, **EXISTS** in SQL server

## SQL NOT with LIKE Operator Example

The following SQL not operator with [like](https://www.tutlane.com/tutorial/sql-server/sql-like-operator) statement will return all the employees whose **location** not containing a word 'en' anywhere in the **location** column because we used a **NOT** keyword with a [LIKE](https://www.tutlane.com/tutorial/sql-server/sql-like-operator) operator and mentioned a pattern like '%en%'.

SELECT \* FROM EmployeeDetails WHERE Location NOT LIKE '%en%'

When we execute above SQL [like](https://www.tutlane.com/tutorial/sql-server/sql-like-operator) operator with **not** keyword example, we will get the result like as shown below.



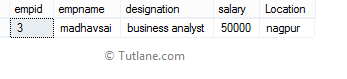
## SQL NOT with IN Operator Example

Generally, if we use [IN](https://www.tutlane.com/tutorial/sql-server/sql-in-operator) operator in SQL it will return data whose column value within the set of values. Suppose if we use **NOT** keyword with [IN](https://www.tutlane.com/tutorial/sql-server/sql-in-operator) operator, then it will return the data whose column value not in the set of values.

The following SQL statement will return all the records whose **location** column values not in the mentioned values.

SELECT \* FROM EmployeeDetails WHERE Location NOT IN('chennai','guntur','bangalore')

When we execute the above SQL [IN](https://www.tutlane.com/tutorial/sql-server/sql-in-operator) operator with **not** keyword example, we will get the result like as shown below.



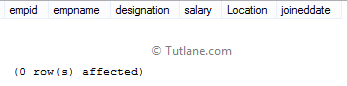
## SQL NOT with EXISTS Operator Example

In SQL, if we use [EXISTS](https://www.tutlane.com/tutorial/sql-server/sql-exists-operator) operator it will return all the records whose values match with the subquery values. Suppose if we use **NOT** with [EXISTS](https://www.tutlane.com/tutorial/sql-server/sql-exists-operator) operator, then it will return all the records whose values not matching with the subquery returned values.

The following SQL statement will return data from EmployeeDetails whose values not exist in subquery returned values.

SELECT \* FROM EmployeeDetails WHERE NOT EXISTS(SELECT \* FROM EmployeeDetails WHERE empid =1)

When we execute the above SQL **not** keyword with the [EXISTS](https://www.tutlane.com/tutorial/sql-server/sql-exists-operator) operator statement, we will get the result as shown below.



SOME Operator:

In SQL, the **SOME** operator is used to compare a value with a single column set of values returned by the subquery. The **SOME** operator in SQL must match at least one value in a subquery and that value must be preceded by [comparison operators](https://www.tutlane.com/tutorial/sql-server/sql-comparison-operators).

Generally we will use this **SOME** operator in **WHERE** clause to check whether the required column values are matching with the set of values returned by subquery or not.

## Syntax of SQL SOME Operator

Following is the syntax of using a **some** operator in the SQL server.

SELECT column1,column2 FROM tablename WHERE column1 = SOME(SELECT column1 FROM tablename WHERE column1 ='somevalue')

If you observe above SQL **SOME** operator syntax, we will get values only when **column1** values matches with **column1** data returned by subquery otherwise it will not return any data.

We will check some operator with examples for that create “**EmployeeDetails**” table by using following script in your SQL database.

create table EmployeeDetails(empid int, empname varchar(50),designation varchar(50),salary int,Location varchar(50))

insert into EmployeeDetails

values(1,'suresh','software engineer',25000,'chennai'),

(2,'rohini','AEO',15000,'chennai'),

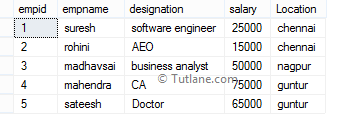
(3,'madhavsai','business analyst',50000,'nagpur'),

(4,'mahendra','CA',75000,'guntur'),

(5,'sateesh','Doctor',65000,'guntur')

select \* from EmployeeDetails

When we execute the above script, a new table called “**EmployeeDetails**” will be created and the result will be as shown below.



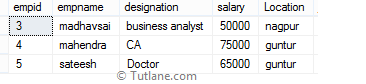
Now execute following examples to check how **SOME** operator will work in SQL server.

## SQL SOME Operator Example

The following sql statement will return employee details whose **salary** column values match with the data returned by the subquery.

SELECT \* FROM EmployeeDetails WHERE salary = SOME(SELECT salary FROM EmployeeDetails WHERE salary >25000)

When we execute above sql query, we will get the result will be like as shown below.

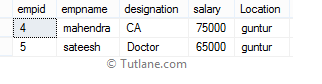


## SQL Some Operator Example2

The following sql statement will return employee details whose **salary** column value matches with at least one value with the data returned by subquery and that value must be preceded by [comparison operators](https://www.tutlane.com/tutorial/sql-server/sql-comparison-operators).

SELECT \* FROM EmployeeDetails WHERE salary > SOME(SELECT salary FROM EmployeeDetails WHERE salary >25000)

When we execute the above query, we will get the result as shown below.



In SQL, **ALL** operator is useful to return true when the given value matches with all the values in single column set of values. The ALL operator in SQL is like an **AND operator** and it will compare a value against all the values in column.

## Syntax of SQL ALL Operator

Following is the syntax of defining an ALL operator in the SQL server.

SELECT column1, column2 FROM tablename WHERE column1 = ALL(SELECT column1 FROM tablename2)

To check all operator in SQL with example execute following script in database to create a tables.

create table example1(id int, name varchar(50))

insert into example1 values(1,'suresh'),(2,'dasari'),(3,'rohini'),(4,'madhav'),(5,'honey')

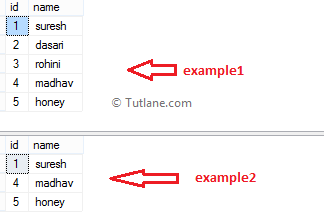
create table example2(id int, name varchar(50))

insert into example2 values(1,'suresh'),(4,'madhav'),(5,'honey')

select \* from example1

select \* from example2

Once we execute the above script new tables **example1** and **example2** will be created in the SQL database and result will be like as shown below.



Now our tables are ready and we can write queries like as shown following to check ALL Operator.

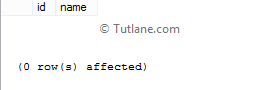
## SQL ALL Operator Example1

Following is the example of using an all operator in the SQL server.

SELECT id,name FROM example1

WHERE id =ALL(SELECT Id FROM example2)

When we execute the above SQL all operator query, we will get the result like as shown below.



If you observe the above result we got 0 rows because in the above query “**id**” in **example1** table will compare with all ids in **example2** table like as shown below.

   (example1:id1 = example2:id1)

   AND (example1:id1 = example2:id4)

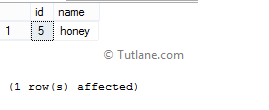
   AND (example1:id1 = example2:id5)

## SQL ALL Operator Example2

Following is the example of using an all operator in sql server.

SELECT id,name FROM example1 WHERE id >=ALL(SELECT Id FROM example2)

When we execute the above SQL all operator query, we will get the result like as shown below.



If you observe above result we got a records because in above query “**id**” column of **example1** table will compare with all **ids** in **example2** table like as shown below.

   (example1:id1 > example2:id1)

   AND (example1:id1 > example2:id4)

   AND (example1:id1 > example2:id5)

This is how we can use all operator in sql server statements based on our requirements.

n SQL, **Any** operator is used to return values that match with any value in a single column set of values. It’s like an [OR operator](https://www.tutlane.com/tutorial/sql-server/sql-or-operator) and it will compare a value against any value in the defined column.

To check any operator in SQL with examples execute the following script to create a new table in the SQL database.

create table example1(id int, name varchar(50))

insert into example1 values(1,'suresh'),(2,'dasari'),(3,'rohini'),(4,'madhav'),(5,'honey')

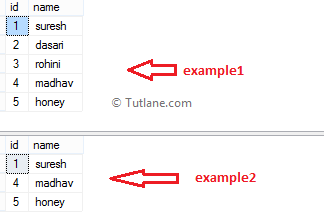
create table example2(id int, name varchar(50))

insert into example2 values(1,'suresh'),(4,'madhav'),(5,'honey')

select \* from example1

select \* from example2

Once we execute the above script, the new table’s **example1** and **example2** will be created and the result will be as shown below.



Now our tables are ready, to check how ANY operator will work in SQL with examples, then execute the following queries.

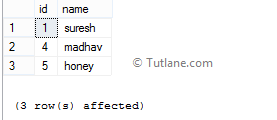
## SQL Any Operator Example

Following is the example of using any operator in SQL.

SELECT id,name FROM example1

WHERE id = ANY(SELECT Id FROM example2)

When we execute the above SQL query, we will get a SQL any operator result like as shown below.



In the above query, the “**id**” column in **example1** table will compare with all **id's** in **example2** table like as shown below.

   (example1:id1 = example2:id1)

   OR (example1:id1 = example2:id4)

   OR (example1:id1 = example2:id5)

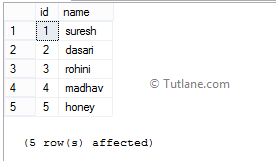
## SQL Any Operator Example2

Following is the another example of using any operator in sql.

SELECT id,name FROM example1

id >= ANY(SELECT Id FROM example2)

When we execute the above query, we will get the result as shown below.

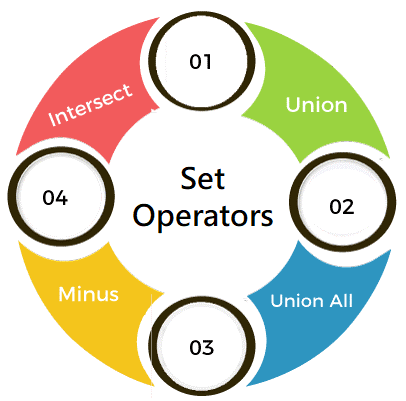


# **SET Operators in SQL**

SET operators are special type of operators which are used to combine the result of two queries.

Operators covered under SET operators are:

1. **UNION**
2. **UNION ALL**
3. **INTERSECT**
4. **MINUS**



There are certain rules which must be followed to perform operations using SET operators in SQL. Rules are as follows:

1. **The number and order of columns must be the same.**
2. **Data types must be compatible.**

Let us see each of the SET operators in more detail with the help of examples.

34M

649

Difference between JDK, JRE, and JVM

**Next**

**Stay**

All the examples will be written using the MySQL database.

Consider we have the following tables with the given data.

**Table 1: t\_employees**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Name** | **Department** | **Salary** | **Year\_of\_Experience** |
| 1 | Aakash Singh | Development | 72000 | 2 |
| 2 | Abhishek Pawar | Production | 45000 | 1 |
| 3 | Pranav Deshmukh | HR | 59900 | 3 |
| 4 | Shubham Mahale | Accounts | 57000 | 2 |
| 5 | Sunil Kulkarni | Development | 87000 | 3 |
| 6 | Bhushan Wagh | R&D | 75000 | 2 |
| 7 | Paras Jaiswal | Marketing | 32000 | 1 |

**Table 2: t2\_employees**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Name** | **Department** | **Salary** | **Year\_of\_Experience** |
| 1 | Prashant Wagh | R&D | 49000 | 1 |
| 2 | Abhishek Pawar | Production | 45000 | 1 |
| 3 | Gautam Jain | Development | 56000 | 4 |
| 4 | Shubham Mahale | Accounts | 57000 | 2 |
| 5 | Rahul Thakur | Production | 76000 | 4 |
| 6 | Bhushan Wagh | R&D | 75000 | 2 |
| 7 | Anand Singh | Marketing | 28000 | 1 |

**Table 3: t\_students**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Name** | **Hometown** | **Percentage** | **Favourite\_Subject** |
| 1 | Soniya Jain | Udaipur | 89 | Physics |
| 2 | Harshada Sharma | Kanpur | 92 | Chemistry |
| 3 | Anuja Rajput | Jaipur | 78 | History |
| 4 | Pranali Singh | Nashik | 88 | Geography |
| 5 | Renuka Deshmukh | Panipat | 90 | Biology |
| 6 | Swati Kumari | Faridabad | 93 | English |
| 7 | Prachi Jaiswal | Gurugram | 96 | Hindi |

**Table 4: t2\_students**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Name** | **Hometown** | **Percentage** | **Favourite\_Subject** |
| 1 | Soniya Jain | Udaipur | 89 | Physics |
| 2 | Ishwari Dixit | Delhi | 86 | Hindi |
| 3 | Anuja Rajput | Jaipur | 78 | History |
| 4 | Pakhi Arora | Surat | 70 | Sanskrit |
| 5 | Renuka Deshmukh | Panipat | 90 | Biology |
| 6 | Jayshree Patel | Pune | 91 | Maths |
| 7 | Prachi Jaiswal | Gurugram | 96 | Hindi |

## 1. UNION:

* UNION will be used to combine the result of two select statements.
* Duplicate rows will be eliminated from the results obtained after performing the UNION operation.

**Example 1:**

Write a query to perform union between the table t\_employees and the table t2\_employees.

**Query:**

1. mysql> **SELECT** \***FROM** t\_employees **UNION** **SELECT** \***FROM** t2\_employees;

Here, in a single query, we have written two SELECT queries. The first SELECT query will fetch the records from the t\_employees table and perform a UNION operation with the records fetched by the second SELECT query from the t2\_employees table.

You will get the following output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Name** | **Department** | **Salary** | **Year\_of\_Experience** |
| 1 | Aakash Singh | Development | 72000 | 2 |
| 2 | Abhishek Pawar | Production | 45000 | 1 |
| 3 | Pranav Deshmukh | HR | 59900 | 3 |
| 4 | Shubham Mahale | Accounts | 57000 | 2 |
| 5 | Sunil Kulkarni | Development | 87000 | 3 |
| 6 | Bhushan Wagh | R&D | 75000 | 2 |
| 7 | Paras Jaiswal | Marketing | 32000 | 1 |
| 1 | Prashant Wagh | R&D | 49000 | 1 |
| 3 | Gautam Jain | Development | 56000 | 4 |
| 5 | Rahul Thakur | Production | 76000 | 4 |
| 7 | Anand Singh | Marketing | 28000 | 1 |

Since we have performed union operation between both the tables, so only the records from the first and second table are displayed except for the duplicate records.

**Example 2:**

Write a query to perform union between the table t\_students and the table t2\_students.

**Query:**

1. mysql> **SELECT** \***FROM** t\_students **UNION** **SELECT** \***FROM** t2\_students;

Here, in a single query, we have written two SELECT queries. The first SELECT query will fetch the records from the t\_students table and perform a UNION operation with the records fetched by the second SELECT query from the t2\_students table.

You will get the following output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Name** | **Department** | **Salary** | **Year\_of\_Experience** |
| 1 | Soniya Jain | Udaipur | 89 | Physics |
| 2 | Harshada Sharma | Kanpur | 92 | Chemistry |
| 3 | Anuja Rajput | Jaipur | 78 | History |
| 4 | Pranali Singh | Nashik | 88 | Geography |
| 5 | Renuka Deshmukh | Panipat | 90 | Biology |
| 6 | Swati Kumari | Faridabad | 93 | English |
| 7 | Prachi Jaiswal | Gurugram | 96 | Hindi |
| 2 | Ishwari Dixit | Delhi | 86 | Hindi |
| 4 | Pakhi Arora | Surat | 70 | Sanskrit |
| 6 | Jayshree Patel | Pune | 91 | Maths |

Since we have performed union operation between both the tables, so only the records from the first and second table are displayed except for the duplicate records.

## 2. UNION ALL

* This operator combines all the records from both the queries.
* Duplicate rows will be not be eliminated from the results obtained after performing the UNION ALL operation.

**Example 1:**

Write a query to perform union all operation between the table t\_employees and the table t2\_employees.

**Query:**

1. mysql> **SELECT** \***FROM** t\_employees **UNION** ALL **SELECT** \***FROM** t2\_employees;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Name** | **Department** | **Salary** | **Year\_of\_Experience** |
| 1 | Aakash Singh | Development | 72000 | 2 |
| 2 | Abhishek Pawar | Production | 45000 | 1 |
| 3 | Pranav Deshmukh | HR | 59900 | 3 |
| 4 | Shubham Mahale | Accounts | 57000 | 2 |
| 5 | Sunil Kulkarni | Development | 87000 | 3 |
| 6 | Bhushan Wagh | R&D | 75000 | 2 |
| 7 | Paras Jaiswal | Marketing | 32000 | 1 |
| 1 | Prashant Wagh | R&D | 49000 | 1 |
| 2 | Abhishek Pawar | Production | 45000 | 1 |
| 3 | Gautam Jain | Development | 56000 | 4 |
| 4 | Shubham Mahale | Accounts | 57000 | 2 |
| 5 | Rahul Thakur | Production | 76000 | 4 |
| 6 | Bhushan Wagh | R&D | 75000 | 2 |
| 7 | Anand Singh | Marketing | 28000 | 1 |

Here, in a single query, we have written two SELECT queries. The first SELECT query will fetch the records from the t\_employees table and perform UNION ALL operation with the records fetched by the second SELECT query from the t2\_employees table.

You will get the following output:

Since we have performed union all operation between both the tables, so all the records from the first and second table are displayed, including the duplicate records.

**Example 2:**

Write a query to perform union all operation between the table t\_students and the table t2\_students.

**Query:**

1. mysql> **SELECT** \***FROM** t\_students **UNION** ALL **SELECT** \***FROM** t2\_students;

Here, in a single query, we have written two SELECT queries. The first SELECT query will fetch the records from the t\_students table and perform UNION ALL operation with the records fetched by the second SELECT query from the t2\_students table.

You will get the following output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Name** | **Hometown** | **Percentage** | **Favourite\_Subject** |
| 1 | Soniya Jain | Udaipur | 89 | Physics |
| 2 | Harshada Sharma | Kanpur | 92 | Chemistry |
| 3 | Anuja Rajput | Jaipur | 78 | History |
| 4 | Pranali Singh | Nashik | 88 | Geography |
| 5 | Renuka Deshmukh | Panipat | 90 | Biology |
| 6 | Swati Kumari | Faridabad | 93 | English |
| 7 | Prachi Jaiswal | Gurugram | 96 | Hindi |
| 1 | Soniya Jain | Udaipur | 89 | Physics |
| 2 | Ishwari Dixit | Delhi | 86 | Hindi |
| 3 | Anuja Rajput | Jaipur | 78 | History |
| 4 | Pakhi Arora | Surat | 70 | Sanskrit |
| 5 | Renuka Deshmukh | Panipat | 90 | Biology |
| 6 | Jayshree Patel | Pune | 91 | Maths |
| 7 | Prachi Jaiswal | Gurugram | 96 | Hindi |

Since we have performed union all operation between both the tables, so all the records from the first and second table are displayed, including the duplicate records.

## 3. INTERSECT:

* It is used to combine two SELECT statements, but it only returns the records which are common from both SELECT statements.

**Example 1:**

Write a query to perform intersect operation between the table t\_employees and the table t2\_employees.

**Query:**

1. mysql> **SELECT** \***FROM** t\_employees **INTERSECT** **SELECT** \***FROM** t2\_employees;

Here, in a single query, we have written two SELECT queries. The first SELECT query will fetch the records from the t\_employees table and perform INTERSECT operation with the records fetched by the second SELECT query from the t2\_employees table.

You will get the following output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Name** | **Hometown** | **Percentage** | **Favourite\_Subject** |
| 2 | Abhishek Pawar | Production | 45000 | 1 |
| 4 | Shubham Mahale | Accounts | 57000 | 2 |
| 6 | Bhushan Wagh | R&D | 75000 | 2 |

Since we have performed intersect operation between both the tables, so only the common records from both the tables are displayed.

**Example 2:**

Write a query to perform intersect operation between the table t\_students and the table t2\_students.

**Query:**

1. mysql> **SELECT** \***FROM** t\_students **INTERSECT** **SELECT** \***FROM** t2\_students;

Here, in a single query, we have written two SELECT queries. The first SELECT query will fetch the records from the t\_students table and perform a UNION operation with the records fetched by the second SELECT query from the t2\_students table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Name** | **Hometown** | **Percentage** | **Favourite\_Subject** |
| 1 | Soniya Jain | Udaipur | 89 | Physics |
| 3 | Anuja Rajput | Jaipur | 78 | History |
| 5 | Renuka Deshmukh | Panipat | 90 | Biology |
| 7 | Prachi Jaiswal | Gurugram | 96 | Hindi |

You will get the following output:

Since we have performed intersect operation between both the tables, so only the common records from both the tables are displayed.

1. **MINUS**

* It displays the rows which are present in the first query but absent in the second query with no duplicates.

**Example 1:**

Write a query to perform a minus operation between the table t\_employees and the table t2\_employees.

**Query:**

1. mysql> **SELECT** \***FROM** t\_employees MINUS **SELECT** \***FROM** t2\_employees;

Here, in a single query, we have written two SELECT queries. The first SELECT query will fetch the records from the t\_employees table and perform MINUS operation with the records fetched by the second SELECT query from the t2\_employees table.

You will get the following output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Name** | **Department** | **Salary** | **Year\_of\_Experience** |
| 1 | Aakash Singh | Development | 72000 | 2 |
| 3 | Pranav Deshmukh | HR | 59900 | 3 |
| 5 | Sunil Kulkarni | Development | 87000 | 3 |
| 7 | Paras Jaiswal | Marketing | 32000 | 1 |

Since we have performed Minus operation between both the tables, so only the unmatched records from both the tables are displayed.

**Example 2:**

Write a query to perform a minus operation between the table t\_students and the table t2\_students.

**Query:**

1. mysql> **SELECT** \***FROM** t\_students MINUS **SELECT** \***FROM** t2\_students;

Here, in a single query, we have written two SELECT queries. The first SELECT query will fetch the records from the t\_employees table and perform a UNION operation with the records fetched by the second SELECT query from the t2\_employees table.

You will get the following output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Name** | **Hometown** | **Percentage** | **Favourite\_Subject** |
| 2 | Harshada Sharma | Kanpur | 92 | Chemistry |
| 4 | Pranali Singh | Nashik | 88 | Geography |
| 6 | Swati Kumari | Faridabad | 93 | English |

Since we have performed a minus operation between both the tables, so only the Unmatched records from both the tables are displayed.

[What is an Aggregate Function in SQL?](https://www.simplilearn.com/tutorials/sql-tutorial/sql-aggregate-functions#what_is_an_aggregate_function_in_sql)

[COUNT() Function](https://www.simplilearn.com/tutorials/sql-tutorial/sql-aggregate-functions#count_function)

[SUM() Function](https://www.simplilearn.com/tutorials/sql-tutorial/sql-aggregate-functions#sum_function)

[AVG() Function](https://www.simplilearn.com/tutorials/sql-tutorial/sql-aggregate-functions#avg_function)

[MIN() Function](https://www.simplilearn.com/tutorials/sql-tutorial/sql-aggregate-functions#min_function)

## What is an Aggregate Function in SQL?

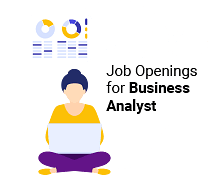
An aggregate function in SQL returns one value after calculating multiple values of a column. We often use aggregate functions with the GROUP BY and HAVING clauses of the SELECT statement.

Various types of SQL aggregate functions are:

* Count()
* Sum()
* Avg()
* Min()
* Max()

#### **Business Analyst Master's Program**

Gain expertise in Business analytics tools[EXPLORE PROGRAM](https://www.simplilearn.com/business-analyst-certification-training-course?source=GhPreviewCTABanner)



## COUNT() Function

The COUNT() function returns the number of rows in a database table.

Syntax:

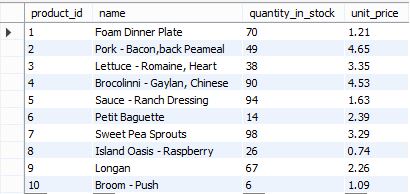
COUNT(\*)

or

COUNT( [ALL|DISTINCT] expression )

Example:

We will use the ‘products’ table from the sample database for our demonstration.



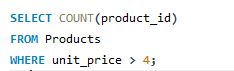
The following SQL statement fetches the number of products in the table.

count-ex-1

This will produce the following result.

output-count-ex-1

The below-given command will display those product ids where the unit price is greater than 4.

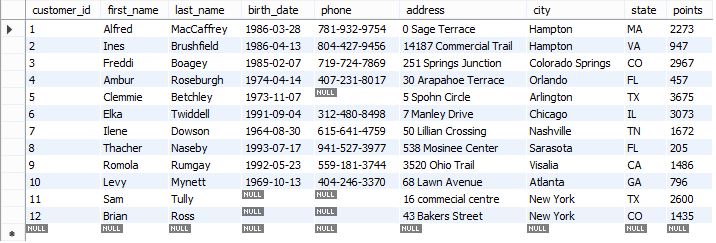


This will display the following result.

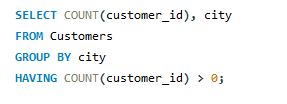
output-count-ex-2

Let's look at how we can use GROUP BY and HAVING functions with the COUNT function.

Consider the following dataset:



The SQL command given below will list the number of customers in each city.



This will produce the following results:



#### **FREE Course: Introduction to Data Analytics**

Mastery to Data Analytics Basics is a Click Away![START LEARNING](https://www.simplilearn.com/learn-data-analytics-for-beginners-skillup?utm_source=frs&utm_medium=skillup-course-banner&utm_campaign=frs-skillup-course-promotion)



## SUM() Function

The SUM() function returns the total sum of a numeric column.

Syntax:

SUM()

or

SUM( [ALL|DISTINCT] expression )

Example:

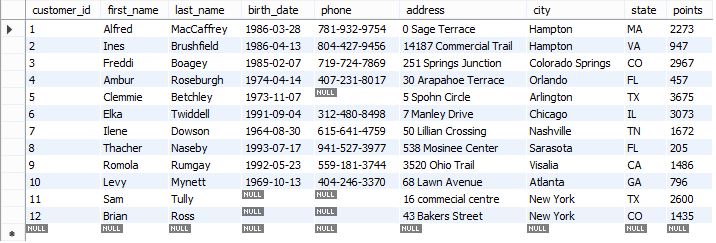
The following SQL statement finds the sum of the "unit price" fields in the "products" table:

sum-sql-aggregate-function

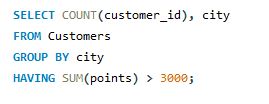
This will produce the following result.

Let’s look at how we can use GROUP BY and HAVING functions with the SUM function.

Consider the following dataset:



The SQL command below will list the number of customers in each city, having a sum of points greater than 3000.



This will produce the following result:



## AVG() Function

The AVG() function calculates the average of a set of values.

Syntax:

AVG()

or

AVG( [ALL|DISTINCT] expression )

Example:

The following SQL command calculates the average quantity in stock.

avg-aggregate-functions

This will produce the following result.

output-avg-1

## MIN() Function

The MIN() aggregate function returns the lowest value (minimum) in a set of non-NULL values.

Syntax:

MIN()

or

MIN( [ALL|DISTINCT] expression )

Example:

min-sql-aggregate-function

The above code will give us the minimum quantity in stock in the products table.

output-min

## MAX() Function

The MAX() aggregate function returns the highest value (maximum) in a set of non-NULL values.

Syntax:

AVG()

or

AVG( [ALL|DISTINCT] expression )

Example:

The code depicted below will give us the maximum quantity in stock in the products table.

max-aggregate-functions

This will produce the following result.

output-max-1

* [String functions](https://ramkedem.com/en/mysql-scalar-functions/#StringFunctions)– functions that perform operations on character values.
* [Numeric functions](https://ramkedem.com/en/mysql-scalar-functions/#NumericFunctions)– functions that perform operations on numeric values.
* [Date functions](https://ramkedem.com/en/mysql-scalar-functions/#DateFunctions)– functions that perform operations on date values.
* [Conversion functions](https://ramkedem.com/en/mysql-scalar-functions/#ConversionFunctions)– functions that convert column data types.
* [NULL-related Functions](https://ramkedem.com/en/mysql-scalar-functions/#NullRelatedFunctions)– functions for handling null values.

String functions:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2 | SELECT CONCAT('Hello' , 'World')  -- Result: 'HelloWorld' | | Returns text strings concatenated | **CONCAT** |
| |  |  | | --- | --- | | 1  2 | SELECT INSTR('hello' , 'e')  -- Result: 2 | | Returns the location of a substring in a string. | **INSTR** |
| |  |  | | --- | --- | | 1  2 | SELECT LENGTH('hello')  -- Result: 5 | | Returns the number of characters of the specified string expression. | **LENGTH** |
| |  |  | | --- | --- | | 1  2 | SELECT RTRIM(' hello    ')  -- Result: ' hello' | | Returns a character string after truncating all trailing blanks. | **RTRIM** |
| |  |  | | --- | --- | | 1  2 | SELECT LTRIM('  hello    ')  -- Result: 'hello    ' | | Returns a character expression after it removes leading blanks. | **LTRIM** |
| |  |  | | --- | --- | | 1  2 | SELECT REPLACE('hello' , 'e' , '$')  -- Result: 'h$llo' | | Replaces all occurrences of a specified string value with another string value. | **REPLACE** |
| |  |  | | --- | --- | | 1  2 | SELECT REVERSE('hello')  -- Result: 'olleh' | | Returns the reverse order of a string value. | **REVERSE** |
| |  |  | | --- | --- | | 1  2 | SELECT SUBSTR('hello' , 2,3)  -- Result: 'ell' | | Returns part of a text. | **SUBSTR** |
| |  |  | | --- | --- | | 1  2 | SELECT LOWER('HELLO')  -- Result: 'hello' | | Returns a character expression after converting uppercase character data to lowercase. | **LOWER** |
| |  |  | | --- | --- | | 1  2 | SELECT UPPER('hello')  -- Result: 'HELLO' | | Returns a character expression with lowercase character data converted to uppercase. | **UPPER** |
| |  |  | | --- | --- | | 1  2 | SELECT SELECT CONCAT('hello', space(3), 'world')  -- Result: 'hello   world' | | Returns a string consisting of *N* spaces | **SPACE** |

Date Functions:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6 | SELECT DATE\_ADD('2008-01-02', INTERVAL 1 DAY);  -- Result : 2008-01-03  SELECT DATE\_ADD('2008-01-02', INTERVAL 1 WEEK);  -- Result : 2008-01-09  SELECT DATE\_ADD('2008-01-02', INTERVAL 1 MONTH);  -- Result : 2008-02-02 | | Returns a specified date with additional time values. | **DATE\_ADD** |
| |  |  | | --- | --- | | 1  2 | SELECT DAYOFMONTH('2015-08-30')  -- Result : 30 | | Returns an integer representing the day (day of the month) of the specified date. | **DAYOFMONTH** |
| |  |  | | --- | --- | | 1  2 | SELECT LAST\_DAY('2015-08-02')  -- Result : 2015-08-31 | | Returns a date representing the last day of the month for specified date. | **LAST\_DAY** |
| |  |  | | --- | --- | | 1  2 | SELECT DATEDIFF('2010-04-01', '2010-03-01')  -- Result : 31 | | Returns the difference between two days, expressed as a value in days. | **DATEDIFF** |
| |  |  | | --- | --- | | 1  2 | SELECT PERIOD\_DIFF(201005, 201003)  -- Result : 2 | | returns the number of months between two periods. | **PREIOD\_DIFF** |
| |  |  | | --- | --- | | 1  2 | SELECT SYSDATE  -- Result: (current date) | | Returns the current database system date. This value is derived from the operating system of the computer on which the instance of MySQL is running. | **SYSDATE()** |

## Numeric Functions

|  |  |  |
| --- | --- | --- |
| **Syntax** | **Description** | **Function** |
| |  |  | | --- | --- | | 1  2 | SELECT FLOOR(59.9)  -- Result: 59 | | Returns an integer that is less than or equal to the specified numeric expression. | **FLOOR** |
| |  |  | | --- | --- | | 1  2 | SELECT CEIL(59.1)  -- Result: 60 | | Returns an integer that is greater than, or equal to, the specified numeric expression. | **CEIL** |
| |  |  | | --- | --- | | 1  2  3  4  5 | SELECT ROUND(59.9)  -- Result: 60    SELECT ROUND(59.1)  -- Result: 59 | | Returns a numeric value, rounded to the specified length or precision. | **ROUND** |

Conversions functions:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 1  2 | SELECT DATE\_FORMAT(SYSDATE(), '%Y-%m-%d')  -- 2015-09-16 | | Converts a a date into a string | **DATE\_FORMAT** |
| |  |  | | --- | --- | | 1  2 | SELECT FORMAT(1003423 , 3)  -- Result : 1,003,423.000 | | Converts a number into a string | **FORMAT** |
| |  |  | | --- | --- | | 1  2 | SELECT CONVERT('11', UNSIGNED INTEGER)  -- Result 11 | | Used to convert one datatype into another, may be used to convert a string into a number | **CONVERT** |

Null releated functions:

|  |  |  |
| --- | --- | --- |
| **Syntax** | **Description** | **Function** |
| |  |  | | --- | --- | | 1  2 | SELECT IFNULL(NULL, 'Hello')  -- Hello | | Accepts two arguments and returns the first if its not NULL | **IFNULL** |

SQL Expressions:

An expression is a combination of one or more values, operators and SQL functions that evaluate to a value. These SQL EXPRESSIONs are like formulae and they are written in query language. You can also use them to query the database for a specific set of data.

### **Syntax**

Consider the basic syntax of the SELECT statement as follows −

SELECT column1, column2, columnN

FROM table\_name

WHERE [CONDITION|EXPRESSION];

There are different types of SQL expressions, which are mentioned below −

* Boolean
* Numeric
* Date

Let us now discuss each of these in detail.

## Boolean Expressions

SQL Boolean Expressions fetch the data based on matching a single value. Following is the syntax −

SELECT column1, column2, columnN

FROM table\_name

WHERE SINGLE VALUE MATCHING EXPRESSION;

Consider the CUSTOMERS table having the following records −

SQL> SELECT \* FROM CUSTOMERS;

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

| 7 | Muffy | 24 | Indore | 10000.00 |

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

7 rows in set (0.00 sec)

The following table is a simple example showing the usage of various SQL Boolean Expressions −

SQL> SELECT \* FROM CUSTOMERS WHERE SALARY = 10000;

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 7 | Muffy | 24 | Indore | 10000.00 |

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

1 row in set (0.00 sec)

## Numeric Expression

These expressions are used to perform any mathematical operation in any query. Following is the syntax −

SELECT numerical\_expression as OPERATION\_NAME

[FROM table\_name

WHERE CONDITION] ;

Here, the numerical\_expression is used for a mathematical expression or any formula. Following is a simple example showing the usage of SQL Numeric Expressions −

SQL> SELECT (15 + 6) AS ADDITION

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

| ADDITION |

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

| 21 |

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

1 row in set (0.00 sec)

There are several built-in functions like avg(), sum(), count(), etc., to perform what is known as the aggregate data calculations against a table or a specific table column.

SQL> SELECT COUNT(\*) AS "RECORDS" FROM CUSTOMERS;

+---------+

| RECORDS |

+---------+

| 7 |

+---------+

1 row in set (0.00 sec)

## Date Expressions

Date Expressions return current system date and time values −

SQL> SELECT CURRENT\_TIMESTAMP;

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

| Current\_Timestamp |

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

| 2009-11-12 06:40:23 |

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

1 row in set (0.00 sec)

Another date expression is as shown below −

SQL> SELECT GETDATE();;

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

| GETDATE |

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

| 2009-10-22 12:07:18.140 |

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

1 row in set (0.00 sec)

SQL Clauses:

* WHERE Clause
* ORDER BY clause
* HAVING Clause
* TOP Clause
* GROUP BY Clause

#### 1. SQL WHERE Clause

In MySQL, we use the SQL SELECT statement to select data from a table in the database. Here, the WHERE clause allows filtering certain records that exactly match a specified condition. Thus, it helps us to fetch only the necessary data from the database that satisfies the given expressional conditions. The WHERE clause is used with SELECT statement as well as with UPDATE, DELETE type statements and aggregate functions to restrict the no. of records to be retrieved by the table. We can also use logical or comparison operators such as LIKE,<,>,=, etc. with WHERE clause to fulfill certain conditions.

**Query:**

SELECT Column1,….ColumnN From Table\_name WHERE [condition];

For example, we are considering a table named Books as the demo:

The query to get records from this table:

**Query:**

SELECT BookName, Price, Lang From Books WHERE CatID >1;

**Output:**

In the above example, we have fetched the rows from a table using WHERE clause where CatID is greater than 1.

**Query:**

SELECT Price, NumPage From Books WHERE BookName=’Networking’;

**Output:**

#### 2. SQL ORDER BY Clause

The ORDER BY clause is used in SQL for sorting records. It is used to arrange the result set either in ascending or descending order. When we query using SELECT statement the result is not in an ordered form. Hence, the result rows can be sorted when we combine the SELECT statement with the ORDER BY clause.

**Query:**

SELECT column1, …,columnN FROM TableName ORDER BY column1,...,column  ASC|DESC;

We add ASC for ascending and DSC for descending with the column name in the query to display the result rows in an ordered form.

**Query:**

SELECT BookName, Price From Books ORDER BY Price ASC;

The result table is retrieved with columns that are sorted in ascending order and the below table is returned in descending.

SELECT BookName, NumPage From Books ORDER BY NumPage DESC;

#### 3. SQL GROUP BY Clause

The GROUP BY clause is used to group rows that have the same values in the result set. Like if we find the names of books from the table grouped by CatID.

**Query:**

SELECT Column FROM Table WHERE condition GROUP BY Column [ORDER BY Column];

This clause is generally used with aggregate functions that allow grouping the query result rows by multiple columns. The aggregate functions are COUNT, MAX, MIN, SUM, AVG, etc.

We have the following example:

SELECT COUNT(BookName), CatID From Books GROUP BY CatID;

The SQL GROUP BY clause returns the aggregated value applying the functions on the columns of the table. The above screenshot shows that the result is returned grouped by CatID where no. of BookName present in those CatID is fetched.

#### 4. SQL HAVING Clause

Actually, this clause is introduced to apply functions in the query with the WHERE clause. In SQL, the HAVING clause was added because the WHERE clause could not be applied with aggregate functions.

**Query:**

SELECT Column FROM Table WHERE condition GROUP BY Column HAVING condition [ORDER BY Column];

We can also use the HAVING clause with logical operators such as OR and AND.

Let us consider the SQL statement below to learn the clause:

SELECT COUNT (CatID), Lang From Books GROUP BY Lang HAVING COUNT(CATID) <3;

Here the result table is returned where the columns are grouped by Lang and no. of rows is restricted by the HAVING clause by providing a condition that CatID should be less than 3.

#### 5. SQL TOP Clause

The TOP clause is used to determine the number of record rows to be shown in the result. This TOP clause is used with SELECT statement specially implemented on large tables with many records. But the clause is not supported in many database systems, like MySQL supports the LIMIT clause to select limited no. of rows and in Oracle ROWNUM is used.

**For SQL Server / MS Access Query:**

SELECT TOP no|percentage ColumName(s) FROM TableName WHERE condition;

**MySQL Query:**

SELECT ColumnName(s) FROM TableName WHERE condition LIMIT no;

**Oracle Query:**

SELECT ColumnName(s) FROM TableName WHERE ROWNUM <= no;

For example, we can explain this clause by these SQL statements where we can return the rows using TOP Clause with SELECT and WHERE for different database platforms:

SELECT TOP 3 \* FROM Books;  
SELECT \* FROM Books LIMIT 3;  
SELECT \* FROM Books WHERE ROWNUM <= 3;

**Output:**

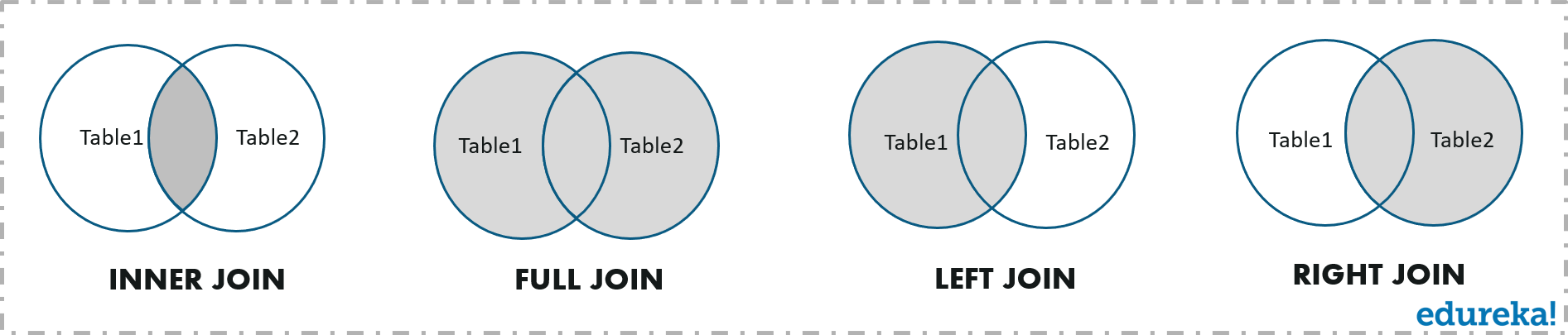
Joins in SQL:

## ****How many types of Joins are there in SQL?****

There are mainly four types of joins that you need to understand. They are:

* [INNER JOIN](https://www.edureka.co/blog/sql-joins-types#INNER%20JOIN)
* [FULL JOIN](https://www.edureka.co/blog/sql-joins-types#FULL%20JOIN)
* [LEFT JOIN](https://www.edureka.co/blog/sql-joins-types#LEFT%20JOIN)
* [RIGHT JOIN](https://www.edureka.co/blog/sql-joins-types#RIGHT%20JOIN)

You can refer to the below image.



## ****How do I know which join to use in SQL?****

Let us look into each one of them. For your better understanding of this concept, I will be considering the following three tables to show you how to perform the Join operations on such tables.

### **Employee Table:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **EmpID** | **EmpFname** | **EmpLname** | **Age** | **EmailID** | **PhoneNo** | **Address** |
| 1 | Vardhan | Kumar | 22 | vardy@abc.com | 9876543210 | Delhi |
| 2 | Himani | Sharma | 32 | himani@abc.com | 9977554422 | Mumbai |
| 3 | Aayushi | Shreshth | 24 | aayushi@abc.com | 9977555121 | Kolkata |
| 4 | Hemanth | Sharma | 25 | hemanth@abc.com | 9876545666 | Bengaluru |
| 5 | Swatee | Kapoor | 26 | swatee@abc.com | 9544567777 | Hyderabad |

### **Project Table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ProjectID** | **EmpID** | **ClientID** | **ProjectName** | **ProjectStartDate** |
| 111 | 1 | 3 | Project1 | 2019-04-21 |
| 222 | 2 | 1 | Project2 | 2019-02-12 |
| 333 | 3 | 5 | Project3 | 2019-01-10 |
| 444 | 3 | 2 | Project4 | 2019-04-16 |
| 555 | 5 | 4 | Project5 | 2019-05-23 |
| 666 | 9 | 1 | Project6 | 2019-01-12 |
| 777 | 7 | 2 | Project7 | 2019-07-25 |
| 888 | 8 | 3 | Project8 | 2019-08-20 |

**Client Table:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **ClientID** | **ClientFname** | **ClientLname** | **Age** | **ClientEmailID** | **PhoneNo** | **Address** | **EmpID** |
| 1 | Susan | Smith | 30 | susan@adn.com | 9765411231 | Kolkata | 3 |
| 2 | Mois | Ali | 27 | mois@jsq.com | 9876543561 | Kolkata | 3 |
| 3 | Soma | Paul | 22 | soma@wja.com | 9966332211 | Delhi | 1 |
| 4 | Zainab | Daginawala | 40 | zainab@qkq.com | 9955884422 | Hyderabad | 5 |
| 5 | Bhaskar | Reddy | 32 | bhaskar@xyz.com | 9636963269 | Mumbai | 2 |

### **INNER JOIN**

This type of join returns those records which have matching values in both tables. So, if you perform an INNER join operation between the Employee table and the Projects table, all the tuples which have matching values in both the tables will be given as output.

#### **Syntax:**

SELECT Table1.Column1,Table1.Column2,Table2.Column1,....

FROM Table1

INNER JOIN Table2

ON Table1.MatchingColumnName = Table2.MatchingColumnName;

NOTE: You can either use the keyword INNER JOIN or JOIN to perform this operation.

#### **Example:**

|  |  |
| --- | --- |
| 1  2  3 | SELECT Employee.EmpID, Employee.EmpFname, Employee.EmpLname, Projects.ProjectID, Projects.ProjectName  FROM Employee  INNER JOIN Projects ON Employee.EmpID=Projects.EmpID; |

#### **Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **EmpID** | **EmpFname** | **EmpLname** | **ProjectID** | **ProjectName** |
| 1 | Vardhan | Kumar | 111 | Project1 |
| 2 | Himani | Sharma | 222 | Project2 |
| 3 | Aayushi | Shreshth | 333 | Project3 |
| 3 | Aayushi | Shreshth | 444 | Project4 |
| 5 | Swatee | Kapoor | 555 | Project5 |

### **FULL JOIN**

Full Join or the Full Outer Join returns all those records which either have a match in the left(Table1) or the right(Table2) table.

[[](https://www.edureka.co/mysql-dba)](https://www.edureka.co/mysql-dba" \t "_blank)

### [MySQL DBA Certification Training](https://www.edureka.co/mysql-dba" \t "_blank)

[Explore Curriculum](https://www.edureka.co/mysql-dba" \t "_blank)

#### **Syntax:**

SELECT Table1.Column1,Table1.Column2,Table2.Column1,....

FROM Table1

FULL JOIN Table2

ON Table1.MatchingColumnName = Table2.MatchingColumnName;

#### **Example:**

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT Employee.EmpFname, Employee.EmpLname, Projects.ProjectID  FROM Employee  FULL JOIN Projects  ON Employee.EmpID = Projects.EmpID; |

#### **Output:**

|  |  |  |
| --- | --- | --- |
| **EmpFname** | **EmpLname** | **ProjectID** |
| Vardhan | Kumar | 111 |
| Himani | Sharma | 222 |
| Aayushi | Shreshth | 333 |
| Aayushi | Shreshth | 444 |
| Hemanth | Sharma | NULL |
| Swatee | Kapoor | 555 |
| NULL | NULL | 666 |
| NULL | NULL | 777 |
| NULL | NULL | 888 |

### **LEFT JOIN**

The LEFT JOIN or the LEFT OUTER JOIN  returns all the records from the left table and also those records which satisfy a condition from the right table. Also, for the records having no matching values in the right table, the output or the result-set will contain the NULL values.

#### **Syntax:**

SELECT Table1.Column1,Table1.Column2,Table2.Column1,....

FROM Table1

LEFT JOIN Table2

ON Table1.MatchingColumnName = Table2.MatchingColumnName;

#### **Example:**

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT Employee.EmpFname, Employee.EmpLname, Projects.ProjectID, Projects.ProjectName  FROM Employee  LEFT JOIN  ON Employee.EmpID = Projects.EmpID ; |

#### **Output:**

|  |  |  |  |
| --- | --- | --- | --- |
| **EmpFname** | **EmpLname** | **ProjectID** | **ProjectName** |
| Vardhan | Kumar | 111 | Project1 |
| Himani | Sharma | 222 | Project2 |
| Aayushi | Shreshth | 333 | Project3 |
| Aayushi | Shreshth | 444 | Project4 |
| Swatee | Kapoor | 555 | Project5 |
| Hemanth | Sharma | NULL | NULL |

### **RIGHT JOIN**

The RIGHT JOIN or the RIGHT OUTER JOIN  returns all the records from the right table and also those records which satisfy a condition from the left table. Also, for the records having no matching values in the left table, the output or the result-set will contain the NULL values.

#### **Syntax:**

SELECT Table1.Column1,Table1.Column2,Table2.Column1,....

FROM Table1

RIGHT JOIN Table2

ON Table1.MatchingColumnName = Table2.MatchingColumnName;

#### **Example:**

|  |  |
| --- | --- |
| 1  2  3  4 | SELECT Employee.EmpFname, Employee.EmpLname, Projects.ProjectID, Projects.ProjectName  FROM Employee  RIGHT JOIN  ON Employee.EmpID = Projects.EmpID; |

#### **Output:**

|  |  |  |  |
| --- | --- | --- | --- |
| **EmpFname** | **EmpLname** | **ProjectID** | **ProjectName** |
| Vardhan | Kumar | 111 | Project1 |
| Himani | Sharma | 222 | Project2 |
| Aayushi | Shreshth | 333 | Project3 |
| Aayushi | Shreshth | 444 | Project4 |
| Swatee | Kapoor | 555 | Project5 |
| NULL | NULL | 666 | Project6 |
| NULL | NULL | 777 | Project7 |
| NULL | NULL | 888 | Project8 |

# **MySQL CROSS JOIN**

MySQL CROSS JOIN is used to combine all possibilities of the two or more tables and returns the result that contains every row from all contributing tables. The CROSS JOIN is also known as CARTESIAN JOIN, which provides the Cartesian product of all associated tables. The Cartesian product can be explained as all rows present in the first table multiplied by all rows present in the second table. It is similar to the Inner Join, where the join condition is not available with this clause.

We can understand it with the following visual representation where CROSS JOIN returns all the records from table1 and table2, and each row is the combination of rows of both tables.

Examples:

For example, with two sets A {x,y,z} and B {1,2,3}, the Cartesian product of A x B is the set of all ordered pairs (x,1), (x,2), (x,3), (y,1) (y,2), (y,3), (z,1), (z,2), (z,3).

The following illustrates syntax of the CROSS JOIN clause:

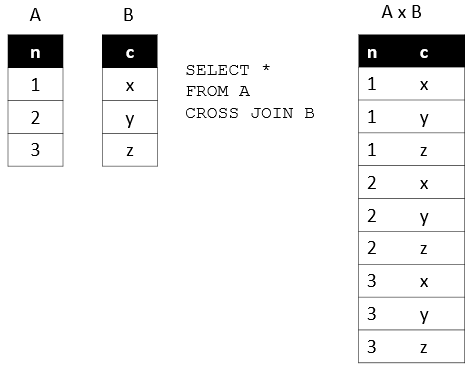
SELECT column\_list

FROM A

CROSS JOIN B;

Code language: SQL (Structured Query Language) (sql)

The following picture illustrates the result of the cross join between the table A and table B. In this illustration, the table A has three rows 1, 2 and 3 and the table B also has three rows x, y and z. As the result, the Cartesian product has nine rows:



**Code:**

CREATE TABLE products (ProductID INT PRIMARY KEY AUTO\_INCREMENT, Product\_NameVARCHAR(255) NOT NULL, Cost INT NOT NULL);

**Step 4:**  Also, let us input some records into the table products using the query below

**Code:**

INSERT INTO products(ProductID, Product\_Name, Cost) VALUES  
('1','Parle G','100')  
('2','Maggie','112')  
('3','GoodDay Buiscuit','150');

**Step 5:** To display the contents of the table as follows

**Code:**

SELECT \* FROM Products;

**Output:**

**Step 6:**Also, we need to create another table as Suppliers having fields as follows

**Code:**

CREATE TABLE Suppliers(Supplier\_ID INT PRIMARY KEY AUTO\_INCREMENT, CategoryVARCHAR(255) NOT NULL, Unit VARCHAR(255) NOT NULL);

**Step 7:**Adding some entries into the table Suppliers using the query statement below

**Code:**

INSERT INTO suppliers(Supplier\_ID, Category, Unit, CostEach) VALUES  
('10','Snacks','10 pcs'),  
('11','Drinks','25 bottles'),  
('12','Kitchen Needs','200 packs');

**Step 8:**View the records in the table as

**Code:**

SELECT \* FROM Suppliers;

**Output:**

**Step 9:**Now, let us implement the CROSS JOIN on theses tables by the help of succeeding query command and view the result set

**Code:**

SELECT ProductID, Supplier\_ID, Product\_Name, Unit, Cost FROM Products CROSS JOIN Suppliers;

**Output:**

### **MySQL CROSS JOIN Syntax**

The CROSS JOIN keyword is always used with the SELECT statement and must be written after the FROM clause. The following syntax fetches all records from both joining tables:

1. **SELECT** **column**-lists
2. **FROM** table1
3. CROSS JOIN table2;

In the above syntax, the column-lists is the name of the column or field that you want to return and table1 and table2 is the table name from which you fetch the records.

14.8M

233

Triggers in SQL (Hindi)

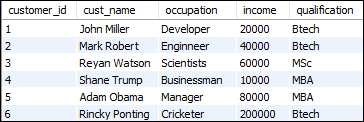
### **MySQL CROSS JOIN Example**

Let us take some examples to understand the working of Left Join or Left Outer Join clause:

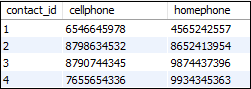
### **CROSS JOIN clause for joining two tables**

Here, we are going to create two tables **"customers"** and **"contacts"** that contains the following data:

**Table: customers**



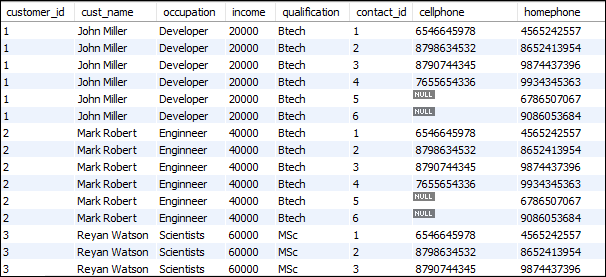
**Table: contacts**



To fetch all records from both tables, execute the following query:

1. **SELECT** \*
2. **FROM** customers
3. CROSS JOIN contacts;

After successful execution of the query, it will give the following output:



When the CROSS JOIN statement executed, you will observe that it displays 42 rows. It means seven rows from customers table multiplies by the six rows from the contacts table.

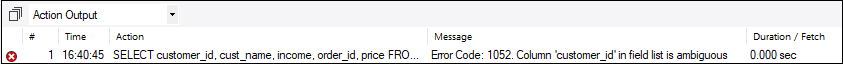
#### **NOTE: To avoid the result of repeated columns twice, it is recommended to use individual column names instead of SELECT \* statement.**

### **Ambiguous Columns problem in MySQL CROSS JOIN**

Sometimes, we need to fetch the selected column records from multiple tables. These tables can contain some column names similar. In that case, MySQL CROSS JOIN statement throws an error: the column name is ambiguous. It means the name of the column is present in both tables, and MySQL gets confused about which column you want to display. The following examples explain it more clearly:

1. **SELECT** customer\_id, cust\_name, income, order\_id, price
2. **FROM** customer
3. CROSS JOIN orders;

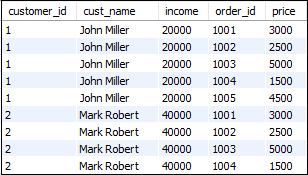
The above CROSS JOIN throws an error as given in the image below:



This problem can be resolved by using the table name before the column name. The above query can be re-written as:

1. **SELECT** customer.customer\_id, customer.cust\_name, customer.income, orders.order\_id, orders.price
2. **FROM** customer
3. CROSS JOIN orders;

After executing the above query, we will get the following output:

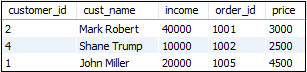


### **LEFT JOIN with WHERE Clause**

The WHERE clause is used to return the **filter** result from the table. The following example illustrates this with the CROSS JOIN clause:

1. **SELECT** customers.customer\_id, customers.cust\_name, customers.income, orders.order\_id, orders.price
2. **FROM** customers
3. CROSS JOIN orders
4. USING(customer\_id) **WHERE** price>1500 AND price<5000;

This statement gives the below result:



SUB QUERIES:

**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 |  |

* To delete students from Student2 table whose rollno is same as that in Student1 table and having location as chennai
* DELETE FROM Student2
* WHERE ROLL\_NO IN ( SELECT ROLL\_NO
* FROM Student1
* WHERE LOCATION = ’chennai’);

Output:

1 row delete successfully.

**Display Student2 table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| NAME | ROLL\_NO | LOCATION | PHONE\_NUMBER |  |
| Sai | 112 | mumbai | 6565656565 |  |
| Sri | 113 | coimbatore | 7878787878 |  |

* To update name of the students to geeks in Student2 table whose location is same as Raju,Ravi in Student1 table
* UPDATE Student2
* SET NAME=’geeks’
* WHERE LOCATION IN ( SELECT LOCATION
* FROM Student1
* WHERE NAME IN (‘Raju’,’Ravi’));

Output:

1 row updated successfully.

**Display Student2 table:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| NAME | ROLL\_NO | LOCATION | PHONE\_NUMBER |  |
| Sai | 112 | mumbai | 6565656565 |  |
| geeks | 113 | coimbatore | 7878787878 |  |

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.

* The subquery can be nested inside a SELECT, INSERT, UPDATE, or DELETE statement or inside another subquery.
* A subquery is usually added within the WHERE Clause of another SQL SELECT statement.
* You can use the comparison operators, such as >, <, or =. The comparison operator can also be a multiple-row operator, such as IN, ANY, or ALL.
* A subquery is also called an inner query or inner select, while the statement containing a subquery is also called an outer query or outer select.
* The inner query executes first before its parent query so that the results of an inner query can be passed to the outer query.

You can use a subquery in a SELECT, INSERT, DELETE, or UPDATE statement to perform the following tasks:

* Compare an expression to the result of the query.
* Determine if an expression is included in the results of the query.
* Check whether the query selects any rows.

Syntax:

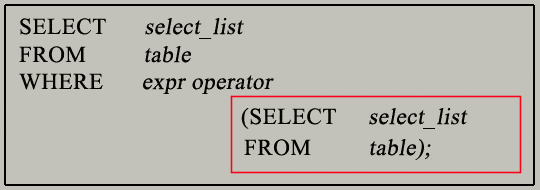
SELECT column\_name

FROM table\_name

WHERE column\_name expression operator

( SELECT COLUMN\_NAME from TABLE\_NAME WHERE ... );

**Syntax :**



* The subquery (inner query) executes once before the main query (outer query) executes.
* The main query (outer query) use the subquery result.

# SQL Subqueries

Last update on October 14 2021 13:24:36 (UTC/GMT +8 hours)

## What is subquery in SQL?

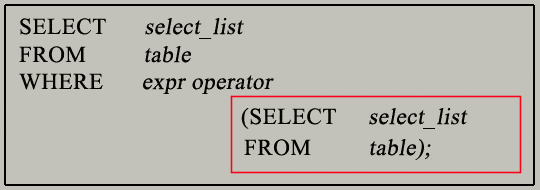
A subquery is a SQL query nested inside a larger query.

* A subquery may occur in :
  + - A SELECT clause
  + - A FROM clause
  + - A WHERE clause
* The subquery can be nested inside a SELECT, INSERT, UPDATE, or DELETE statement or inside another subquery.
* A subquery is usually added within the WHERE Clause of another SQL SELECT statement.
* You can use the comparison operators, such as >, <, or =. The comparison operator can also be a multiple-row operator, such as IN, ANY, or ALL.
* A subquery is also called an inner query or inner select, while the statement containing a subquery is also called an outer query or outer select.
* The inner query executes first before its parent query so that the results of an inner query can be passed to the outer query.

You can use a subquery in a SELECT, INSERT, DELETE, or UPDATE statement to perform the following tasks:

* Compare an expression to the result of the query.
* Determine if an expression is included in the results of the query.
* Check whether the query selects any rows.

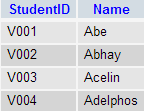
**Syntax :**

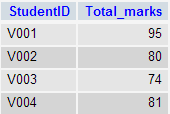


* The subquery (inner query) executes once before the main query (outer query) executes.
* The main query (outer query) use the subquery result.

## SQL Subqueries Example :

We have the following two tables 'student' and 'marks' with common field 'StudentID'.



Now we want to write a query to identify all students who get better marks than that of the student who's StudentID is 'V002', but we do not know the marks of 'V002'.  
- To solve the problem, we require two queries. One query returns the marks (stored in Total\_marks field) of 'V002' and a second query identifies the students who get better marks than the result of the first query.

**First query:**

SELECT \*

FROM `marks`

WHERE studentid = 'V002';

Copy

**Query result:**

student query

The result of the query is 80.  
- Using the result of this query, here we have written another query to identify the students who get better marks than 80. Here is the query :

**Second query:**

SELECT a.studentid, a.name, b.total\_marks

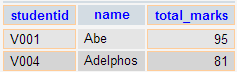
FROM student a, marks b

WHERE a.studentid = b.studentid

AND b.total\_marks >80;

Copy

**Query result:**



Above two queries identified students who get the better number than the student who's StudentID is 'V002' (Abhay).

You can combine the above two queries by placing one query inside the other. The subquery (also called the 'inner query') is the query inside the parentheses. See the following code and query result :

**SQL Code:**

SELECT a.studentid, a.name, b.total\_marks

FROM student a, marks b

WHERE a.studentid = b.studentid AND b.total\_marks >

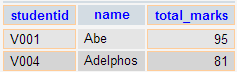
(SELECT total\_marks

FROM marks

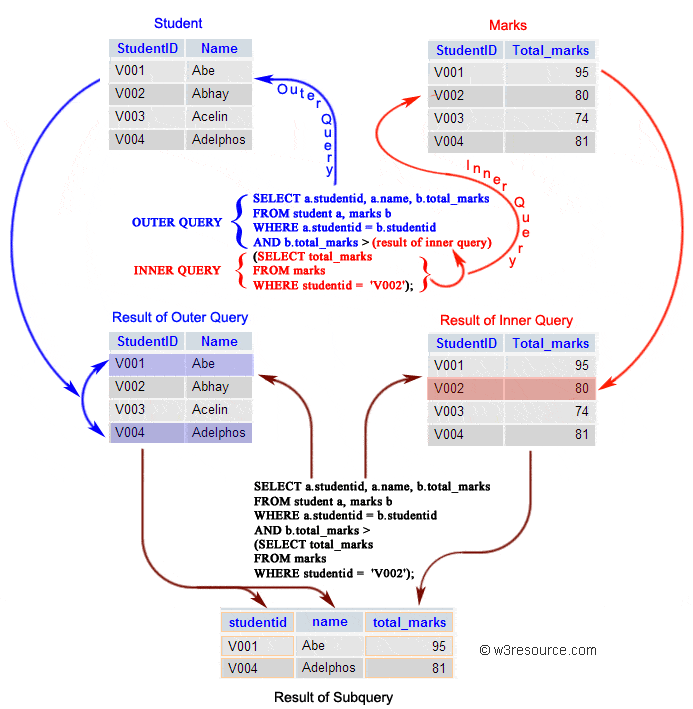
WHERE studentid = 'V002');

Copy

**Query result:**



**Pictorial Presentation of SQL Subquery:**

[](https://www.w3resource.com/sql/subqueries/sql-subqueries.gif)

## Subqueries with INSERT statement

INSERT statement can be used with subqueries. Here are the syntax and an example of subqueries using INSERT statement.

**Syntax:**

INSERT INTO table\_name [ (column1 [, column2 ]) ]

SELECT [ \*|column1 [, column2 ]

FROM table1 [, table2 ]

[ WHERE VALUE OPERATOR ];

If we want to insert those orders from 'orders' table which have the advance\_amount 2000 or 5000 into 'neworder' table the following SQL can be used:

Sample table: orders

ORD\_NUM ORD\_AMOUNT ADVANCE\_AMOUNT ORD\_DATE CUST\_CODE AGENT\_CODE ORD\_DESCRIPTION

---------- ---------- -------------- --------- --------------- --------------- -----------------

200114 3500 2000 15-AUG-08 C00002 A008

200122 2500 400 16-SEP-08 C00003 A004

200118 500 100 20-JUL-08 C00023 A006

200119 4000 700 16-SEP-08 C00007 A010

200121 1500 600 23-SEP-08 C00008 A004

200130 2500 400 30-JUL-08 C00025 A011

200134 4200 1800 25-SEP-08 C00004 A005

200108 4000 600 15-FEB-08 C00008 A004

200103 1500 700 15-MAY-08 C00021 A005

200105 2500 500 18-JUL-08 C00025 A011

200109 3500 800 30-JUL-08 C00011 A010

200101 3000 1000 15-JUL-08 C00001 A008

200111 1000 300 10-JUL-08 C00020 A008

200104 1500 500 13-MAR-08 C00006 A004

200106 2500 700 20-APR-08 C00005 A002

200125 2000 600 10-OCT-08 C00018 A005

200117 800 200 20-OCT-08 C00014 A001

200123 500 100 16-SEP-08 C00022 A002

200120 500 100 20-JUL-08 C00009 A002

200116 500 100 13-JUL-08 C00010 A009

200124 500 100 20-JUN-08 C00017 A007

200126 500 100 24-JUN-08 C00022 A002

200129 2500 500 20-JUL-08 C00024 A006

200127 2500 400 20-JUL-08 C00015 A003

200128 3500 1500 20-JUL-08 C00009 A002

200135 2000 800 16-SEP-08 C00007 A010

200131 900 150 26-AUG-08 C00012 A012

200133 1200 400 29-JUN-08 C00009 A002

200100 1000 600 08-JAN-08 C00015 A003

200110 3000 500 15-APR-08 C00019 A010

200107 4500 900 30-AUG-08 C00007 A010

200112 2000 400 30-MAY-08 C00016 A007

200113 4000 600 10-JUN-08 C00022 A002

200102 2000 300 25-MAY-08 C00012 A012

**SQL Code:**

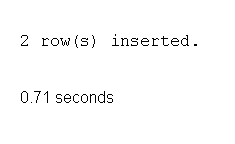
INSERT INTO neworder

SELECT \* FROM orders

WHERE advance\_amount in(2000);

Copy

Output:



To see more details of subqueries using INSERT statement [click here](https://www.w3resource.com/sql/insert-statement/insert-using-subqueries.php).

## Subqueries with UPDATE statement

In a UPDATE statement, you can set new column value equal to the result returned by a single row subquery. Here are the syntax and an example of subqueries using UPDATE statement.

**Syntax:**

UPDATE table SET column\_name = new\_value

[ WHERE OPERATOR [ VALUE ]

(SELECT COLUMN\_NAME

FROM TABLE\_NAME)

[ WHERE) ]

If we want to update that ord\_date in 'neworder' table with '15-JAN-10' which have the difference of ord\_amount and advance\_amount is less than the minimum ord\_amount of 'orders' table the following SQL can be used:

Sample table: neworder

ORD\_NUM ORD\_AMOUNT ADVANCE\_AMOUNT ORD\_DATE CUST\_CODE AGENT\_CODE ORD\_DESCRIPTION

---------- ---------- -------------- --------- --------------- --------------- -----------------

200114 3500 2000 15-AUG-08 C00002 A008

200122 2500 400 16-SEP-08 C00003 A004

200118 500 100 20-JUL-08 C00023 A006

200119 4000 700 16-SEP-08 C00007 A010

200121 1500 600 23-SEP-08 C00008 A004

200130 2500 400 30-JUL-08 C00025 A011

200134 4200 1800 25-SEP-08 C00004 A005

200108 4000 600 15-FEB-08 C00008 A004

200103 1500 700 15-MAY-08 C00021 A005

200105 2500 500 18-JUL-08 C00025 A011

200109 3500 800 30-JUL-08 C00011 A010

200101 3000 1000 15-JUL-08 C00001 A008

200111 1000 300 10-JUL-08 C00020 A008

200104 1500 500 13-MAR-08 C00006 A004

200106 2500 700 20-APR-08 C00005 A002

200125 2000 600 10-OCT-08 C00018 A005

200117 800 200 20-OCT-08 C00014 A001

200123 500 100 16-SEP-08 C00022 A002

200120 500 100 20-JUL-08 C00009 A002

200116 500 100 13-JUL-08 C00010 A009

200124 500 100 20-JUN-08 C00017 A007

200126 500 100 24-JUN-08 C00022 A002

200129 2500 500 20-JUL-08 C00024 A006

200127 2500 400 20-JUL-08 C00015 A003

200128 3500 1500 20-JUL-08 C00009 A002

200135 2000 800 16-SEP-08 C00007 A010

200131 900 150 26-AUG-08 C00012 A012

200133 1200 400 29-JUN-08 C00009 A002

200100 1000 600 08-JAN-08 C00015 A003

200110 3000 500 15-APR-08 C00019 A010

200107 4500 900 30-AUG-08 C00007 A010

200112 2000 400 30-MAY-08 C00016 A007

200113 4000 600 10-JUN-08 C00022 A002

200102 2000 300 25-MAY-08 C00012 A012

**SQL Code:**

UPDATE neworder

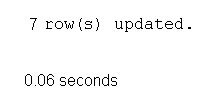
SET ord\_date='15-JAN-10'

WHERE ord\_amount-advance\_amount<

(SELECT MIN(ord\_amount) FROM orders);

Inner query result will be 500.

Output:



To see more details of subqueries using UPDATE statement [click here](https://www.w3resource.com/sql/update-statement/update-using-subqueries.php).

## Subqueries with DELETE statement

DELETE statement can be used with subqueries. Here are the syntax and an example of subqueries using DELETE statement.

**Syntax:**

DELETE FROM TABLE\_NAME

[ WHERE OPERATOR [ VALUE ]

(SELECT COLUMN\_NAME

FROM TABLE\_NAME)

[ WHERE) ]

If we want to delete those orders from 'neworder' table which advance\_amount are less than the maximum advance\_amount of 'orders' table, the following SQL can be used:

Sample table: neworder

**SQL Code:**

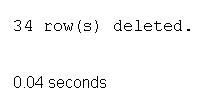
DELETE FROM neworder

WHERE advance\_amount<

(SELECT MAX(advance\_amount) FROM orders);

Copy

Output:

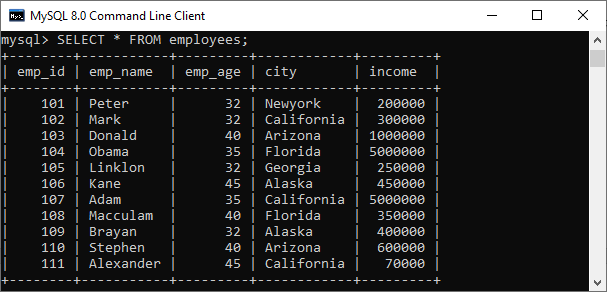


1. **SELECT** column\_list (s) **FROM**  table\_name
2. **WHERE**  column\_name OPERATOR
3. (**SELECT** column\_list (s)  **FROM** table\_name [**WHERE**])

## MySQL Subquery Example

Let us understand it with the help of an example. Suppose we have a table named **"employees"** that contains the following data:

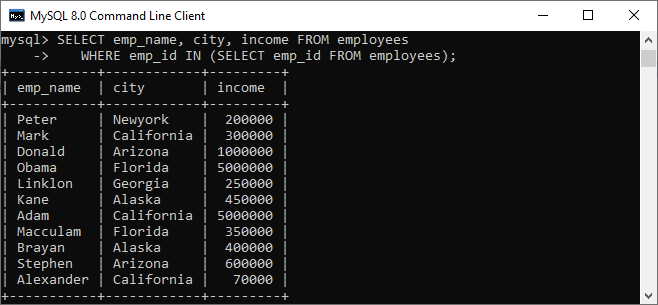
**Table: employees**



Following is a simple SQL statement that returns the **employee detail whose id matches in a subquery**:

1. **SELECT** emp\_name, city, income **FROM** employees
2. **WHERE** emp\_id IN (**SELECT** emp\_id **FROM** employees);

This query will return the following output:



### **MySQL Subquery with Comparison Operator**

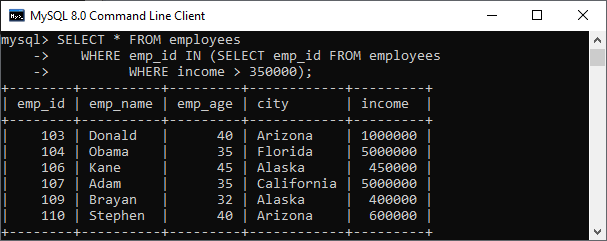
A comparison operator is an operator used to compare values and returns the result, either true or false. The following comparison operators are used in MySQL <, >, =, <>, <=>, etc. We can use the subquery before or after the comparison operators that return a single value. The returned value can be the arithmetic expression or a column function. After that, SQL compares the subquery results with the value on the other side of the comparison operator. The below example explains it more clearly:

Following is a simple [SQL](https://www.javatpoint.com/sql-tutorial) statement that returns the **employee detail whose income is more than 350000** with the help of subquery:

1. **SELECT** \* **FROM** employees
2. **WHERE** emp\_id IN (**SELECT** emp\_id **FROM** employees
3. **WHERE** income > 350000);

This query first executes the subquery that returns the **employee id whose income > 350000**. Second, the main query will return the employees all details whose employee id are in the result set returned by the subquery.

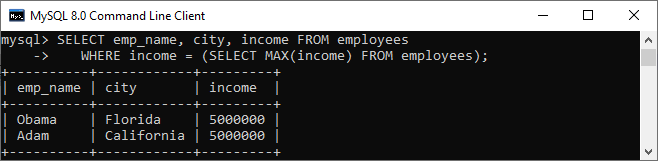
After executing the statement, we will get the below output, where we can see the employee detail whose income>350000.



Let us see an example of another comparison operator, such as equality (=) to find employee details with **maximum income** using a subquery.

1. **SELECT** emp\_name, city, income **FROM** employees
2. **WHERE** income = (**SELECT** **MAX**(income) **FROM** employees);

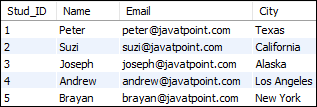
It will give the output where we can see two employees detail who have maximum income.



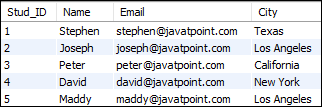
### **MySQL Subquery with IN or NOT-IN Operator**

If the subquery produces more than one value, we need to use the IN or NOT IN operator with the [WHERE clause](https://www.javatpoint.com/mysql-where). Suppose we have a table named **"Student"** and **"Student2"** that contains the following data:

**Table: Student**



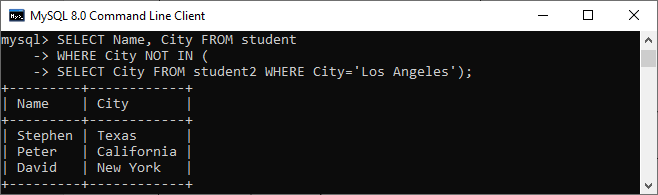
**Table: Student2**



The following subquery with NOT IN operator returns the **student detail who does not belong to Los Angeles City** from both tables as follows:

1. **SELECT** **Name**, City **FROM** student
2. **WHERE** City NOT IN (
3. **SELECT** City **FROM** student2 **WHERE** City='Los Angeles');

After execution, we can see that the result contains the student details not belonging to Los Angeles City.



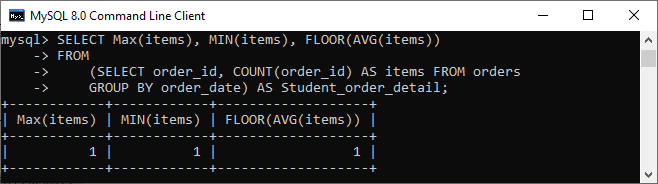
### **MySQL Subquery in the FROM Clause**

If we use a subquery in the FROM clause, MySQL will return the output from a subquery is used as a temporary table. We called this table as a derived table, inline views, or materialized subquery.

The following subquery returns the maximum, minimum, and average number of items in the order table:

1. **SELECT** **Max**(items), **MIN**(items), FLOOR(AVG(items))
2. **FROM**
3. (**SELECT** order\_id, COUNT(order\_id) **AS** items **FROM** orders
4. **GROUP** **BY** order\_date) **AS** Student\_order\_detail;

It will give the output as follows:

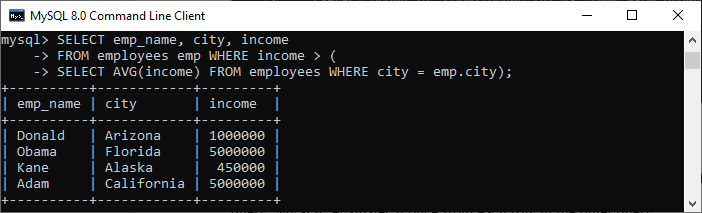


### **MySQL Correlated Subqueries**

A correlated subquery in MySQL is a subquery that depends on the outer query. It uses the data from the outer query or contains a reference to a parent query that also appears in the outer query. MySQL evaluates it once from each row in the outer query.

1. **SELECT** emp\_name, city, income
2. **FROM** employees emp **WHERE** income > (
3. **SELECT** AVG(income) **FROM** employees **WHERE** city = emp.city);

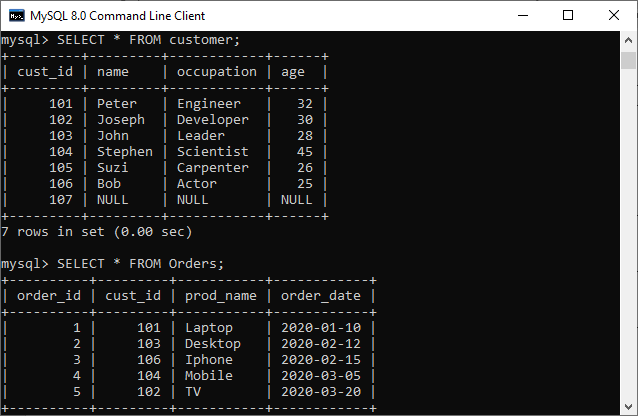
In the above query, we select an **employee name and city** whose income is higher than the average income of all employees in each city.



The subquery executes for every city of the specified table because it will change for every row. Therefore, the average income will also be changed. Then, the main query filters employee detail whose income is higher than the average income from the subquery.

### **MySQL Subqueries with EXISTS or NOT EXISTS**

The [EXISTS operator](https://www.javatpoint.com/mysql-exists) is a Boolean operator that returns either true or false result. It is used with a subquery and checks the existence of data in a subquery. If a subquery returns any record at all, this operator returns true. Otherwise, it will return false. The NOT EXISTS operator used for negation that gives true value when the subquery does not return any row. Otherwise, it returns false. Both EXISTS and NOT EXISTS used with correlated subqueries. The following example illustrates it more clearly. Suppose we have a table **customer and order** that contains the data as follows:



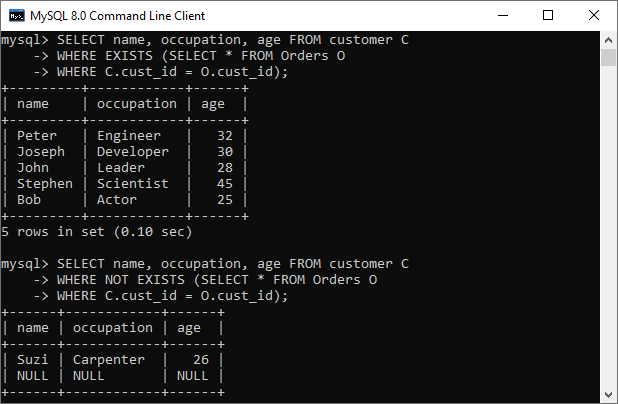
The below SQL statements uses EXISTS operator to find the name, occupation, and age of the customer who has placed at least one order.

1. **SELECT** **name**, occupation, age **FROM** customer C
2. **WHERE** EXISTS (**SELECT** \* **FROM** Orders O
3. **WHERE** C.cust\_id = O.cust\_id);

This statement uses NOT EXISTS operator that returns the customer details who have not placed an order.

1. **SELECT** **name**, occupation, age **FROM** customer C
2. **WHERE** NOT EXISTS (**SELECT** \* **FROM** Orders O
3. **WHERE** C.cust\_id = O.cust\_id);

We can see the below output to understand the above queries result.



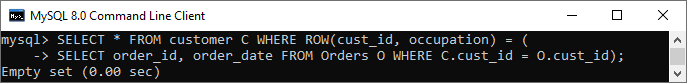
To read more information about the EXISTS operator, [click here](https://www.javatpoint.com/mysql-exists).

### **MySQL ROW Subqueries**

It is a subquery that returns a single row where we can get more than one column values. We can use the following operators for comparing row subqueries =, >, <, >=, <=, <>, !=, <=>. Let us see the following example:

1. **SELECT** \* **FROM** customer C **WHERE** ROW(cust\_id, occupation) = (
2. **SELECT** order\_id, order\_date **FROM** Orders O **WHERE** C.cust\_id = O.cust\_id);

If given row has cust\_id, occupation values equal to the order\_id, order\_date values of any rows in the first table, the WHERE expression is TRUE, and each query returns those first table rows. Otherwise, the expression is FALSE, and the query produces an empty set, which can be shown in the below image:



### **MySQL Subqueries with ALL, ANY, and SOME**

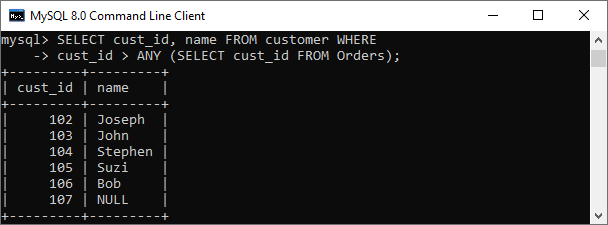
We can use a subquery which is followed by the keyword ALL, ANY, or SOME after a comparison operator. The following are the syntax to use subqueries with ALL, ANY, or SOME:

1. operand comparison\_operator ANY (subquery)
2. operand comparison\_operator ALL (subquery)
3. operand comparison\_operator SOME (subquery)

The ALL keyword compares values with the value returned by a subquery. Therefore, it returns TRUE if the comparison is TRUE for ALL of the values returned by a subquery. The ANY keyword returns TRUE if the comparison is TRUE for ANY of the values returned by a subquery. The ANY and SOME keywords are the same because they are the alias of each other. The following example explains it more clearly:

1. **SELECT** cust\_id, **name** **FROM** customer **WHERE**
2. cust\_id > ANY (**SELECT** cust\_id **FROM** Orders);

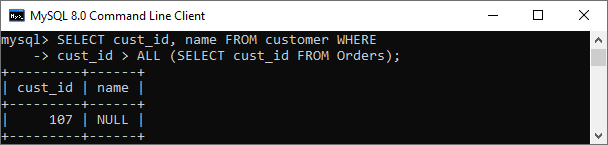
We will get the output as follows:



If we use ALL in place of ANY, it will return TRUE when the comparison is TRUE for ALL values in the column returned by a subquery. For example:

1. **SELECT** cust\_id, **name** **FROM** customer **WHERE**
2. cust\_id > ALL (**SELECT** cust\_id **FROM** Orders);

We can see the output as below:



## Difference between Correlated and non corelated Subquery in SQL

here are the *main difference between correlated and non-correlated subqueries in SQL*

### 1. Working

A non-correlated subquery is executed only once and its result can be swapped back for a query, on the other hand, **a correlated subquery is executed multiple times**, precisely once for each row returned by the outer query.  
  
For example, the following query is an example of a non-correlated subquery:

SELECT MAX(Salary) FROM Employee

WHERE Salary NOT IN ( SELECT MAX(Salary) FROM Employee)

Here the subquery is SELECT MAX(Salary) from Employee, you can execute and substitute the result of that query e.g. if subquery return 10000 then the outer query is reduced to

SELECT MAX(Salary) from Employee where Salary NOT IN (10000).

This is not possible with a correlated subquery, which needs to be executed multiple times as shown below:

SELECT e.Name, e.Salary FROM Employee e

WHERE 2 = (

SELECT COUNT(Salary) FROM Employee p WHERE p.salary >= e.salary)

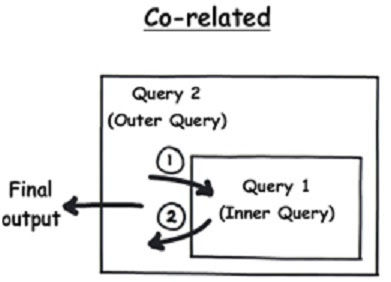
In this example, the subquery is SELECT COUNT(Salary) FROM Employee p WHERE p.salary >= e.salary, you cannot swap its value for the outer query because it needs to be executed for each employee.  
  
Let's say the first row of employees has a salary of 5000, in this case, e.salary will be 500 and subquery will be

SELECT COUNT(Salary) FROM Employee p WHERE p.salary >= 5000

and subquery will find how many salaries are higher than 5000 if count return 2 then it's the [second-highest salary](http://javarevisited.blogspot.sg/2012/12/how-to-find-second-highest-or-maximum-salary-sql.html#axzz4jneWOUNe). This logic needs to be executed for each row the outer query will process.

### 2. Dependency

A *correlated subquery depends upon the outer query* and *cannot execute in isolation*, but a regular or non-correlated subquery doesn't depend on the outer query and can execute in isolation.  
  
From the above example, you can see that a correlated subquery like SELECT COUNT(Salary) FROM Employee p WHERE p.salary >= e.salary depends upon outer query because it needs the value of e.salary, which comes from the table listed on the outer query.  
  
On the other hand, regular subquery, SELECT MAX(Salary) FROM Employee doesn't depends upon the outer query and can be executed in isolation or independently of the outer query. You can further join these [free SQL and database courses](https://www.java67.com/2018/02/5-free-database-and-sql-query-courses-programmers.html) to learn more about Correlated and non-correlated subqueries and how to use them. 

[](https://medium.com/javarevisited/top-5-sql-and-database-courses-to-learn-online-48424533ac61)

### 3.Speed and Performance

A **correlated subquery is much slower** than a non-correlated subquery because in the former, the inner query executes for each row of the outer query. This means if your table has n rows then whole processing will take the **n \* n = n^2** time, as compared to 2n times taken by a non-correlated subquery.  
  
This happens because to execute a non-correlated subquery you need to examine just n rows of the table and similar to execute the outer query you need to examine n rows, so in total n + n = 2n rows.  
  
This is the reason you should be very careful using a correlated subquery with large tables e.g. tables with millions of rows because that can take a long time and could potentially block other jobs and queries from accessing the table.  
  
In many cases, you can replace correlated subquery with [inner join](http://javarevisited.blogspot.sg/2012/11/how-to-join-three-tables-in-sql-query-mysql-sqlserver.html#axzz4jneWOUNe) which would result in better performance. For example, to find all employees whose salary is greater than the average salary of the department you can write the following correlated subquery:

SELECT e.id, e.name

FROM Employee e

WHERE salary > (

SELECT AVG(salary)

FROM Employee p

WHERE p.department = e.department)

Now, you can convert this [correlated subquery](http://javarevisited.blogspot.sg/2012/07/subquery-example-in-sql-correlated-vs.html) to a [JOIN based query](http://javarevisited.blogspot.com/2013/05/difference-between-left-and-right-outer-join-sql-mysql.html) for better performance as shown below:

SELECT e.id, e.name

FROM Employee INNER JOIN

(SELECT department, AVG(salary) AS department\_average

FROM Employee

GROUP BY department) AS t ON e.department = t.department

WHERE e.salary > t.department\_average;

A **database object** is any defined object in a database that is used to store or reference data.Anything which we make from **create command**is known as Database Object.It can be used to hold and manipulate the data.Some of the examples of database objects are : view, sequence, indexes, etc.

* **Table –** Basic unit of storage; composed rows and columns
* **View –** Logically represents subsets of data from one or more tables
* **Sequence –** Generates primary key values
* **Index –** Improves the performance of some queries
* **Synonym –** Alternative name for an object:

Different database Objects :

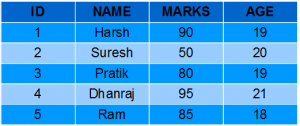
Views in SQL are kind of virtual tables. A view also has rows and columns as they are in a real table in the database. We can create a view by selecting fields from one or more tables present in the database. A View can either have all the rows of a table or specific rows based on certain condition.

In this article we will learn about creating , deleting and updating Views.  
**Sample Tables**:

StudentDetails

[](https://media.geeksforgeeks.org/wp-content/uploads/Screenshot-57.png)

StudentMarks

[](https://media.geeksforgeeks.org/wp-content/uploads/Screenshot-58.png)

**CREATING VIEWS**

We can create View using **CREATE VIEW** statement. A View can be created from a single table or multiple tables.

**Syntax**:

CREATE VIEW view\_name AS

SELECT column1, column2.....

FROM table\_name

WHERE condition;

**view\_name**: Name for the View

**table\_name**: Name of the table

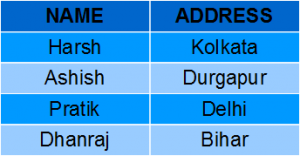
**condition**: Condition to select rows

**Examples**:

* **Creating View from a single table:**
  + In this example we will create a View named DetailsView from the table StudentDetails.  
    Query:
  + CREATE VIEW DetailsView AS
  + SELECT NAME, ADDRESS
  + FROM StudentDetails
  + WHERE S\_ID < 5;

To see the data in the View, we can query the view in the same manner as we query a table.

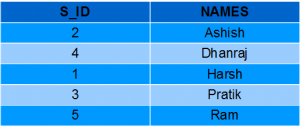
SELECT \* FROM DetailsView;

Output:  
[](https://media.geeksforgeeks.org/wp-content/uploads/Screenshot-571.png)

* + In this example, we will create a view named StudentNames from the table StudentDetails.  
    Query:
  + CREATE VIEW StudentNames AS
  + SELECT S\_ID, NAME
  + FROM StudentDetails
  + ORDER BY NAME;

If we now query the view as,

SELECT \* FROM StudentNames;

Output:  
[](https://media.geeksforgeeks.org/wp-content/uploads/Screenshot-64.png)

* **Creating View from multiple tables**: In this example we will create a View named MarksView from two tables StudentDetails and StudentMarks. To create a View from multiple tables we can simply include multiple tables in the SELECT statement. Query:
* CREATE VIEW MarksView AS
* SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS
* FROM StudentDetails, StudentMarks
* WHERE StudentDetails.NAME = StudentMarks.NAME;

To display data of View MarksView:

SELECT \* FROM MarksView;

Output:  
[](https://media.geeksforgeeks.org/wp-content/uploads/Screenshot-591.png)

**DELETING VIEWS**

We have learned about creating a View, but what if a created View is not needed any more? Obviously we will want to delete it. SQL allows us to delete an existing View. We can delete or drop a View using the DROP statement.

**Syntax**:

DROP VIEW view\_name;

**view\_name**: Name of the View which we want to delete.

For example, if we want to delete the View **MarksView**, we can do this as:

DROP VIEW MarksView;

**UPDATING VIEWS**

There are certain conditions needed to be satisfied to update a view. If any one of these conditions is **not** met, then we will not be allowed to update the view.

1. The SELECT statement which is used to create the view should not include GROUP BY clause or ORDER BY clause.
2. The SELECT statement should not have the DISTINCT keyword.
3. The View should have all NOT NULL values.
4. The view should not be created using nested queries or complex queries.
5. The view should be created from a single table. If the view is created using multiple tables then we will not be allowed to update the view.

* We can use the **CREATE OR REPLACE VIEW** statement to add or remove fields from a view.  
  **Syntax**:
* CREATE OR REPLACE VIEW view\_name AS
* SELECT column1,coulmn2,..
* FROM table\_name
* WHERE condition;

For example, if we want to update the view **MarksView** and add the field AGE to this View from **StudentMarks**Table, we can do this as:

CREATE OR REPLACE VIEW MarksView AS

SELECT StudentDetails.NAME, StudentDetails.ADDRESS, StudentMarks.MARKS, StudentMarks.AGE

FROM StudentDetails, StudentMarks

WHERE StudentDetails.NAME = StudentMarks.NAME;

If we fetch all the data from MarksView now as:

SELECT \* FROM MarksView;

Output:  
[](https://media.geeksforgeeks.org/wp-content/uploads/Screenshot-60.png)

* **Inserting a row in a view**:  
  We can insert a row in a View in a same way as we do in a table. We can use the INSERT INTO statement of SQL to insert a row in a View.**Syntax**:
* INSERT INTO view\_name(column1, column2 , column3,..)
* VALUES(value1, value2, value3..);
* **view\_name**: Name of the View

**Example**:  
In the below example we will insert a new row in the View DetailsView which we have created above in the example of “creating views from a single table”.

INSERT INTO DetailsView(NAME, ADDRESS)

VALUES("Suresh","Gurgaon");

If we fetch all the data from DetailsView now as,

SELECT \* FROM DetailsView;

Output:  
[](https://media.geeksforgeeks.org/wp-content/uploads/Screenshot-62.png)

* **Deleting a row from a View**:  
  Deleting rows from a view is also as simple as deleting rows from a table. We can use the DELETE statement of SQL to delete rows from a view. Also deleting a row from a view first delete the row from the actual table and the change is then reflected in the view.**Syntax**:
* DELETE FROM view\_name
* WHERE condition;
* **view\_name**:Name of view from where we want to delete rows
* **condition**: Condition to select rows

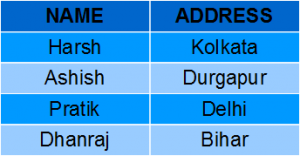
**Example**:  
In this example we will delete the last row from the view DetailsView which we just added in the above example of inserting rows.

DELETE FROM DetailsView

WHERE NAME="Suresh";

If we fetch all the data from DetailsView now as,

SELECT \* FROM DetailsView;

Output:  
[](https://media.geeksforgeeks.org/wp-content/uploads/Screenshot-571.png)

**WITH CHECK OPTION**

The WITH CHECK OPTION clause in SQL is a very useful clause for views. It is applicable to a updatable view. If the view is not updatable, then there is no meaning of including this clause in the CREATE VIEW statement.

* The WITH CHECK OPTION clause is used to prevent the insertion of rows in the view where the condition in the WHERE clause in CREATE VIEW statement is not satisfied.
* If we have used the WITH CHECK OPTION clause in the CREATE VIEW statement, and if the UPDATE or INSERT clause does not satisfy the conditions then they will return an error.

**Example**:  
In the below example we are creating a View SampleView from StudentDetails Table with WITH CHECK OPTION clause.

CREATE VIEW SampleView AS

SELECT S\_ID, NAME

FROM StudentDetails

WHERE NAME IS NOT NULL

WITH CHECK OPTION;

In this View if we now try to insert a new row with null value in the NAME column then it will give an error because the view is created with the condition for NAME column as NOT NULL.  
For example,though the View is updatable but then also the below query for this View is not valid:

INSERT INTO SampleView(S\_ID)

VALUES(6);

### Syntax to Create View

A View can be created using CREATE VIEW statement as below:

CREATE VIEW VIEW\_NAME AS  
SELECT column1, column2, column3.......  
FROM table\_name WHERE [condition];

#### Creating a view in SQL with Examples:

Let us consider the below table EMPLOYEE:

We can create a view from the above table as below.

CREATE VIEW EMLOYEE\_VW AS  
SELECT NAME, AGE, SALARY  
FROM EMPLOYEE;

In the above view EMPLOYEE\_VW, the below columns will be created.

SELECT \* FROM EMPLOYEE\_VW;

We can create views from more than one table.

Let us consider another table DEPARTMENT as below.

We can create a view from the above EMPLOYEE and DEPARTMENT tables as below:

CREATE VIEW DEPT\_VIEW AS  
SELECT EMPLOYEE.ID, EMPLOYEE.NAME, DEPARTMENT.DEPTNAME  
FROM EMPLOYEE, DEPARTMENT  
WHERE EMPLOYEE.ID = DEPARTMENT.ID;

The view DEPT\_VIEW will have the below result.

SELECT \* FROM DEPT\_VIEW;

### Different view Operations in SQL Views

Below are various view operations are as follows:

#### Update in View

Though a view can be updated, we need to keep a few conditions in the notice. Such as, while updating a view the select statement should not contain a DISTINCT keyword, set functions, order by clause, Group By or Having, sub-queries, etc. Also, the FROM clause should not contain multiple tables. In addition to the above, the view should have NOT NULL values if it needs to be updated. So when we want to update the view EMPLOYEE\_VW keeping the above points in focus, the table EMPLOYEE will be updated.

CREATE OR REPLACE VIEW statement is used to add or remove fields from a view.

**SYNTAX for Update**

CREATE OR REPLACE VIEW view\_name AS  
SELECT column1, column2....  
FROM table\_name  
WHERE [condition];

Let us update the view DEPT\_VIEW as below to add the location column.

 Popular Course in this category

**[View Course](https://www.educba.com/software-development/courses/sql-training-certification/?btnz=edu-blg-inline-banner1" \t "_blank)**

CREATE OR REPLACE VIEW DEPT\_VIEW AS  
SELECT EMPLOYEE.ID, EMPLOYEE.NAME, DEPARTMENT.DEPTNAME, DEPARTMENT.LOCATION  
FROM EMPLOYEE, DEPARTMENT  
WHERE EMPLOYEE.ID = DEPARTMENT.ID;

SELECT \* FROM EMPLOYEE;

From the above query, we will get the below result.

#### INSERTING into VIEW

Syntax to insert into a view

INSERT INTO view\_name(column1, column 2, column3,....) VALUES(value1, value2, value3,...);

A row can be inserted to view EMPLOYEE\_VW by using insert into a statement as below:

INSERT INTO EMPLOYEE\_VW (NAME, AGE, SALARY) VALUES(‘RAM’, ‘24’, ‘27000.00’);

After insertion, we can see the result by the below select query.

SELECT \* FROM EMPLOYEE\_VW;

#### DELETING FROM A VIEW

Deleting a row from a view deletes the row from the table on which the view was created.

The syntax for deleting from a view

DELETE FROM view\_name WHERE [condition];

We can delete the row from view as below:

DELETE FROM EMPLOYEE\_VW WHERE NAME = ‘MALAY’;

After deletion, the result can be displayed by the below query.

SELECT \* FROM EMPLOYEE\_VW;

#### DROPPING A VIEW

The views can be dropped by using the below syntax:

DROP VIEW view\_name;

If we want to delete the view EMPLOYEE\_VW, it can be deleted as below:

DROP VIEW EMPLOYEE\_VW;

### Advantages and Disadvantages of  SQL Views

Below are some pros and cons as follows:

#### Advantages

Below are the advantages of using views:

* If we need to maintain any sensitive information by providing limited access to the users, views are used for that purpose. Views are used to only display the required data to the users by keeping sensitive data safe.
* As a database view is associated with many tables upon which the view is created, it simplifies the complexity of the query.
* The view is used to hide the complexity of the underlying tables used in a database from the end-users.
* Views are useful in case of re-designing the database so as not to affect any other applications using the same database.
* The data of the computed columns can be calculated very easily when we query the data from the view, as views enable computed columns.

Syntax to Create View

A View can be created using CREATE VIEW statement as below:

CREATE VIEW VIEW\_NAME AS

SELECT column1, column2, column3.......

FROM table\_name WHERE [condition];

# SQL indexes

An index is a schema object. It is used by the server to speed up the retrieval of rows by using a pointer.. An index helps to speed up select queries and where clauses, but it slows down data input, with the update and the insert statements. Indexes can be created or dropped with no effect on the data.  we will see how to create, delete, and uses the INDEX in the database.

For example, if you want to reference all pages in a book that discusses a certain topic, you first refer to the index, which lists all the topics alphabetically and is then referred to one or more specific page numbers.

### **Creating an Index:**

**Syntax:**

CREATE INDEX index

ON TABLE column;

where the **index** is the name given to that index and **TABLE** is the name of the table on which that index is created and **column** is the name of that column for which it is applied.

### **For multiple columns:**

**Syntax:**

CREATE INDEX index

ON TABLE (column1, column2,.....);

### **Unique Indexes:**

Unique indexes are used for the maintenance of the integrity of the data present in the table as well as for the fast performance, it does not allow multiple values to enter into the table.   
 **Syntax:**

CREATE UNIQUE INDEX index

ON TABLE column;

### **When should indexes be created:**

* A column contains a wide range of values.
* A column does not contain a large number of null values.
* One or more columns are frequently used together in a where clause or a join condition.

**When should indexes be avoided:**

* The table is small
* The columns are not often used as a condition in the query
* The column is updated frequently

**Removing an Index:**

To remove an index from the data dictionary by using the **DROP INDEX** command.

**Syntax:**

DROP INDEX index;

To drop an index, you must be the owner of the index or have the **DROP ANY INDEX** privilege. 

### **Altering an Index:**

To modify an existing table’s index by rebuilding, or reorganizing the index.

ALTER INDEX IndexName

ON TableName REBUILD;

### **Confirming Indexes :**

You can check the different indexes present in a particular table given by the user or the server itself and their uniqueness.

**Syntax:**

select \* from USER\_INDEXES;

It will show you all the indexes present in the server, in which you can locate your own tables too.

### **Renaming an index :**

You can use the system stored procedure sp\_rename to rename any index in the database.

**Syntax:**

EXEC sp\_rename

index\_name,

new\_index\_name,

N'INDEX';

**Indexing** is a procedure that returns your requested data faster from the defined table. Without indexing, the SQL server has to scan the whole table for your data. By indexing, SQL server do the exact same thing when you searched for a content in a book by checking the index page. In the same way table’s index allows us to locate the exact data without scanning the whole table. There are two types of indexing in SQL.

1. Clustered index
2. Non-clustered index

**1. Clustered –**  
Clustered index is the type of indexing that established a physical sorting order of rows.Suppose you have a table *Student\_info* which contains *ROLL\_NO* as a primary key than Clustered index which is self created on that primary key will sort the *Student\_info* table as per *ROLL\_NO*. Clustered index is like Dictionary, in the dictionary sorting order is alphabetical there is no separate index page.

**Examples:**

**Input:**

CREATE TABLE Student\_info

(

ROLL\_NO int(10) primary key,

NAME varchar(20),

DEPARTMENT varchar(20),

);

insert into Student\_info values(1410110405, 'H Agarwal', 'CSE')

insert into Student\_info values(1410110404, 'S Samadder', 'CSE')

insert into Student\_info values(1410110403, 'MD Irfan', 'CSE')

SELECT \* FROM Student\_info

**Output:**

|  |  |  |
| --- | --- | --- |
| ROLL\_NO | NAME | DEPARTMENT |
| 1410110403 | MD Irfan | CSE |
| 1410110404 | S Samadder | CSE |
| 1410110405 | H Agarwal | CSE |

If we want to create Clustered index on other column then first we have to remove the primary key after that we can remove the previous index.

Note that defining a column as a primary key makes that column the Clustered Index of that table. To make any other column, clustered index first we have to remove the previous one as follows below procedure.

**Syntax:**

//Drop index

drop index table\_name.index\_name

//Create Clustered index index

create Clustered index IX\_table\_name\_column\_name

on table\_name (column\_name ASC)

**Note:** We can create only one clustered index in a table.

**2. Non-clustered:**  
The Non-Clustered index is an index structure separate from the data stored in a table that reorders one or more selected columns. The non-clustered index is created to improve the performance of frequently used queries not covered by clustered index. It’s like a textbook, the index page is created separately at the beginning of that book.

**Examples:**

**Input:**

CREATE TABLE Student\_info

(

ROLL\_NO int(10),

NAME varchar(20),

DEPARTMENT varchar(20),

);

insert into Student\_info values(1410110405, 'H Agarwal', 'CSE')

insert into Student\_info values(1410110404, 'S Samadder', 'CSE')

insert into Student\_info values(1410110403, 'MD Irfan', 'CSE')

SELECT \* FROM Student\_info

**Output:**

|  |  |  |
| --- | --- | --- |
| ROLL\_NO | NAME | DEPARTMENT |
| 1410110405 | H Agarwal | CSE |
| 1410110404 | S Samadder | CSE |
| 1410110403 | MD Irfan | CSE |

**Note:** We can create one or more Non\_Clustered index in a table.

**Syntax:**

//Create Non-Clustered index

create NonClustered index IX\_table\_name\_column\_name

on table\_name (column\_name ASC)

Table: Student\_info

|  |
| --- |
|  |
| ROLL\_NO | NAME | DEPARTMENT |
| 1410110405 | H Agarwal | CSE |
| 1410110404 | S Samadder | CSE |
| 1410110403 | MD Irfan | CSE |

**Input:**  
create NonClustered index IX\_Student\_info\_NAME on Student\_info (NAME ASC)  
**Output:**

Index

|  |
| --- |
|  |
| NAME | ROW\_ADDRESS |
| H Agarwal | 1 |
| MD Irfan | 3 |
| S Samadder | 2 |

**Clustered vs Non-Clustered index:**

* In a table there can be only one clustered index or one or more than one non\_clustered index.
* In Clustered index there is no separate index storage but in Non\_Clustered index there is separate index storage for the index.
* Clustered index is slower than Non\_Clustered index.

SEQUENCES:

MySQL does **not provide any built-in function to create a sequence** for a table's rows or columns. But we can generate it via [SQL](https://www.javatpoint.com/sql-tutorial)

### **Create Sequence Using AUTO\_INCREMENT**

The simplest way for creating a sequence in [MySQL](https://www.javatpoint.com/mysql-tutorial)

is by defining the column as **AUTO\_INCREMENT** during table creation, which should be a **primary key** column.

The following are the rules which should be considered when we use the AUTO\_INCREMENT attribute for the column:

* We can create only one AUTO\_INCREMENT column in each table, and the data type of this column is an integer.
* The AUTO\_INCREMENT column should also have either PRIMARY or UNIQUE KEY indexing.
* The AUTO\_INCREMENT column must contain **NOT NULL** However, MySQL automatically adds the NOT NULL constraint to the column implicitly when we set the column as an AUTO\_INCREMENT attribute.

**Example:**

Let us understand it with the help of the following example. First, we need to create a new table and make sure that there is one column with the AUTO\_INCREMENT attribute and that too, as PRIMARY KEY.

Execute the below query to create a table:

1. mysql> **CREATE** **TABLE** Insects (
2. Id **INT** UNSIGNED NOT NULL AUTO\_INCREMENT,
3. **PRIMARY** **KEY** (id),
4. **Name** **VARCHAR**(30) NOT NULL,
5. Type **VARCHAR**(30) NOT NULL,
6. Origin **VARCHAR**(30) NOT NULL
7. );

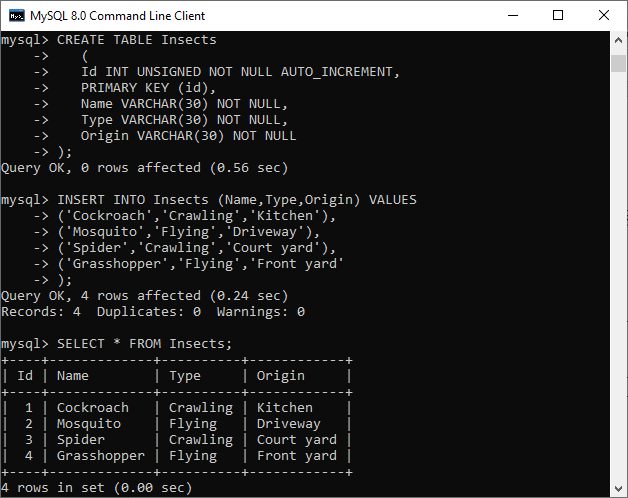
Next, we will insert a few rows into this table where no need to provide the id for each row because it is auto-incremented by MySQL.

1. mysql> **INSERT** **INTO** Insects (**Name**, Type, Origin) **VALUES**
2. ('Cockroach', 'Crawling', 'Kitchen'),
3. ('Mosquito', 'Flying', 'Driveway'),
4. ('Spider' ,'Crawling', 'Court yard'),
5. ('Grasshopper', 'Flying', 'Front yard');

Now execute the **SELECT statement** to verify the records:

1. mysql> **SELECT** \* **FROM** Insects;

We can see the results in the below image.



In the above image, we have defined the Id column with PRIMARY KEY and AUTO\_INCREMENT option that automatically incremented this column and always stored unique values in it.

When we execute the **[INSERT query](https://www.javatpoint.com/mysql-insert)**

, we do not provide values for the Id column, but MySQL automatically generates a sequence for it.

### **How MySQL Sequence Works?**

The AUTO\_INCREMENT column in MySQL contains the following attributes:

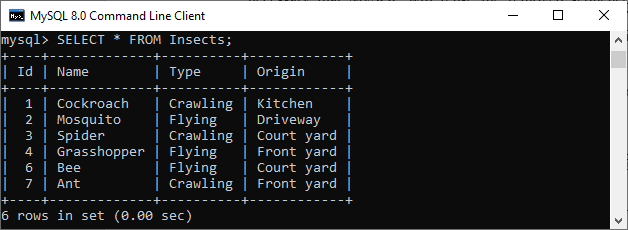
* The AUTO\_INCREMENT column's starting value is 1. This column is always incremented by 1 when we omit its value in the INSERT statement or insert a **NULL**
* We can use the **LAST\_INSERT\_ID()** function to get the last generated sequence number. However, we can also use the last insert ID for the subsequent statements that should be unique across sessions.
* If we will insert a new row into a table along with specifying a value for the sequence column, then MySQL first checks it whether the specified value has already existed or not. If it does not exist, it will insert the sequence number in the column; otherwise, issue an error. Again, if we insert a value greater than the next sequence number, MySQL will use it as the starting sequence number. Now, MySQL will generate the next sequencing value from the current sequence number. It is to note that it will create gaps in our sequence.
* If we update the AUTO\_INCREMENT column's value that already exists by using the **UPDATE** statement, MySQL will issue a duplicate-key error if the column stores only distinct value. If we update an AUTO\_INCREMENT column with a value greater than the existing values, MySQL inserts the next value of the last sequence number for the next row. For example, the AUTO\_INCREMENT column's last sequence value is 3, and we want to update it with 10, then the sequence number for the next row should be 4.
* If we want to delete the last inserted row using the **DELETE** statement, it is not necessary that MySQL will reuse the removed sequence number again because it depends on the table's storage engine. For example, if we use the **MyISAM** table and remove the last insert Id that is 5, MySQL still inserts the next sequence number as 6 for the new row.

Let us look at some more examples for a better understanding of the use of the MySQL sequence.

Insert two new records into the table.

1. mysql> **INSERT** **INTO** Insects (Id, **Name**, Type, Origin) **VALUES**
2. (6, 'Bee', 'Flying', 'Court yard'),
3. (7, 'Ant', 'Crawling', 'Front yard');

And execute the SELECT statement to verify the output:



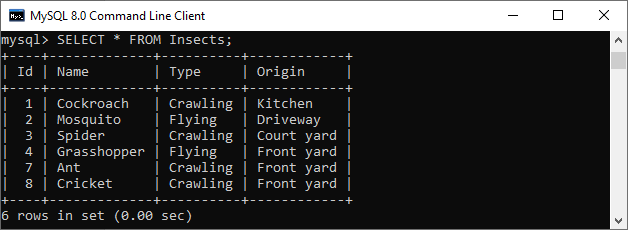
Next, we will **delete the insect whose id is 6** using the below query:

1. mysql> **DELETE** **FROM** Insects **WHERE** Id = 6;

Again, we will insert a new row into the table with the below statement:

1. mysql> **INSERT** **INTO** Insects (**Name**, Type, Origin) **VALUES**
2. ('Cricket', 'Crawling', 'Front yard');

We will execute the SELECT statement again to see the output:



In the above image, we can see that MySQL does not reuse the deleted sequence number. It is because the storage engine of the Insects table is **InnoDB**. Therefore, the insert query will add the new sequence in the Insects table as 8.

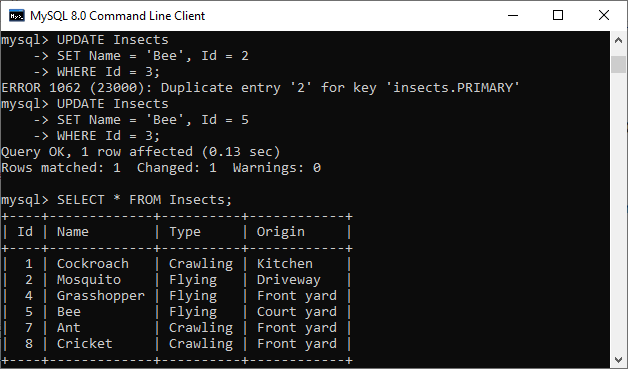
Now, we will update an existing insect whose **Id is 3 to the Id = 2:**

1. mysql> **UPDATE** Insects **SET** **Name** = 'Bee', Id = 2 **WHERE** Id = 3;

MySQL issued an error: **Duplicate entry '2' for key 'insects.PRIMARY' column**. Let's fix it

1. mysql> **UPDATE** Insects **SET** **Name** = 'Bee', Id = 5 **WHERE** Id = 3;

See the below image.



# SQL | SYNONYM

A **SYNONYM** provides another name for database object, referred to as original object, that may exist on a local or another server. A synonym belongs to schema, name of synonym should be unique. A synonym cannot be original object for an additional synonym and synonym cannot refer to user-defined function.

The query below results in an entry for each synonym in database. This query provides details about synonym metadata such as the name of synonym and name of the base object.

select \*

from sys.synonyms ;

**Note :** Synonyms are database dependent and cannot be accessed by other databases.

**Syntax –**

CREATE SYNONYM synonymname

FOR servername.databasename.schemaname.objectname;

GO

**Example –**  
Let us assume Geektabtable of GFGdatabase, Geeekshschema on server named Server1. To reference this table from another server, Server2, an application would have to use four-part named Server1.GFG.Geeeksh.Geektab. Also, if the location of table were to change, for example, to another server, application would have to be modified to reflect that change.

To address both these issues, one can create synonym, Geektable, on Server2for Geektabtable on Server1. Now, the application only has to use single-part name, Geektable, to point Geektab table. Also, if location of the Geektab table changes, you will have to modify synonym, Geektable, to point to new location of Geektab table.

Now, let us create synonym for Geektab table of GFG database, Geeeksh schema on server named Server1.

CREATE SYNONYM Geektable

FOR Server1.GFG.Geeeksh.Geektab;

GO

Find the output in Server2 by using synonym.

SELECT ID, Name

FROM Geektable;

**Output –**

| ID | Name |
| --- | --- |
| 1 | Nisha |
| 2 | Mira |
| 3 | Punit |
| 4 | Ram |

(4 row(s) affected)