SQL Tutorial

For installation best YouTube video <https://www.youtube.com/watch?v=WuBcTJnIuzo&ab_channel=ProgrammingKnowledge>

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](https://www.javatpoint.com/oracle-tutorial), [MySQL](https://www.javatpoint.com/mysql-tutorial), [MongoDB](https://www.javatpoint.com/mongodb-tutorial), [PostgreSQL](https://www.javatpoint.com/postgresql-tutorial), [SQL Server](https://www.javatpoint.com/sql-server-tutorial), [DB2](https://www.javatpoint.com/db2-tutorial), etc.

Also, they are using different dialects, such as −

* MS SQL Server using T-SQL,
* Oracle using PL/SQL,
* MS Access version of SQL is called JET SQL (native format) etc.

What is SQL?

SQL is a short-form of the structured query language, and it is pronounced as S-Q-L or sometimes as See-Quell.

This database language is mainly designed for maintaining the data in relational database management systems. It is a special tool used by data professionals for handling structured data (data which is stored in the form of tables). It is also designed for stream processing in RDSMS.

You can easily create and manipulate the database, access and modify the table rows and columns, etc. This query language became the standard of ANSI in the year of 1986 and ISO in the year of 1987.

If you want to get a job in the field of data science, then it is the most important query language to learn. Big enterprises like Facebook, Instagram, and LinkedIn, use SQL for storing the data in the back-end.

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.
* Allows users to access data in the relational database management systems.
* Allows users to describe the data.
* Allows users to define the data in a database and manipulate that data.
* Allows to embed within other languages using SQL modules, libraries & pre-compilers.
* Allows users to create and drop databases and tables.
* Allows users to create view, stored procedure, functions in a database.
* Allows users to set permissions on tables, procedures and views.

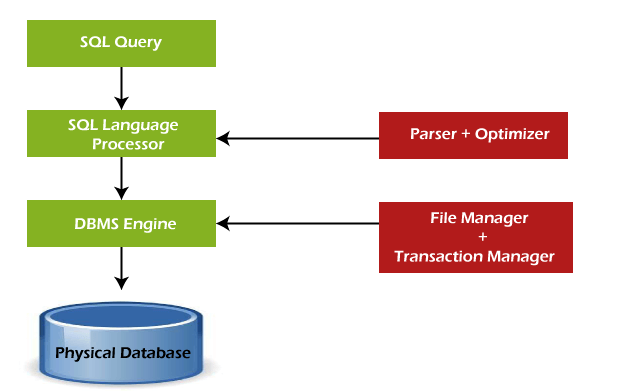
Process of SQL

When we are executing the command of SQL on any Relational database management system, then the system automatically finds the best routine to carry out our request, and the SQL engine determines how to interpret that particular command.

Structured Query Language contains the following four components in its process:

* Query Dispatcher
* Optimization Engines
* Classic Query Engine
* SQL Query Engine, etc.

A classic query engine allows data professionals and users to maintain non-SQL queries. The architecture of SQL is shown in the following diagram:



Some SQL Commands

The SQL commands help in creating and managing the database. The most common SQL commands which are highly used are mentioned below:

1. CREATE command
2. UPDATE command
3. DELETE command
4. SELECT command
5. DROP command
6. INSERT command

## **SQL Commands**

The standard SQL commands to interact with relational databases are CREATE, SELECT, INSERT, UPDATE, DELETE and DROP. These commands can be classified into the following groups based on their nature −

### **DDL - Data Definition Language**

|  |  |
| --- | --- |
| **Sr.No.** | **Command & Description** |
| 1 | **CREATE**  Creates a new table, a view of a table, or other object in the database. |
| 2 | **ALTER**  Modifies an existing database object, such as a table. |
| 3 | **DROP**  Deletes an entire table, a view of a table or other objects in the database. |

### **DML - Data Manipulation Language**

|  |  |
| --- | --- |
| **Sr.No.** | **Command & Description** |
| 1 | **SELECT**  Retrieves certain records from one or more tables. |
| 2 | **INSERT**  Creates a record. |
| 3 | **UPDATE**  Modifies records. |
| 4 | **DELETE**  Deletes records. |

### **DCL - Data Control Language**

|  |  |
| --- | --- |
| **Sr.No.** | **Command & Description** |
| 1 | **GRANT**  Gives a privilege to user. |
| 2 | **REVOKE**  Takes back privileges granted from user. |

SQL vs No-SQL



The following table describes the [differences between the SQL and NoSQL](https://www.javatpoint.com/sql-vs-nosql), which are necessary to understand:

|  |  |
| --- | --- |
| **SQL** | **No-SQL** |
| 1. SQL is a relational database management system. | 1. While No-SQL is a non-relational or distributed database management system. |
| 2. The query language used in this database system is a structured query language. | 2. The query language used in the No-SQL database systems is a non-declarative query language. |
| 3. The schema of SQL databases is predefined, fixed, and static. | 3. The schema of No-SQL databases is a dynamic schema for unstructured data. |
| 4. These databases are vertically scalable. | 4. These databases are horizontally scalable. |
| 5. The database type of SQL is in the form of tables, i.e., in the form of rows and columns. | 5. The database type of No-SQL is in the form of documents, key-value, and graphs. |
| 6. It follows the ACID model. | 6. It follows the BASE model. |
| 7. Complex queries are easily managed in the SQL database. | 7. NoSQL databases cannot handle complex queries. |
| 8. This database is not the best choice for storing hierarchical data. | 8. While No-SQL database is a perfect option for storing hierarchical data. |
| 9. All SQL databases require object-relational mapping. | 9. Many No-SQL databases do not require object-relational mapping. |
| 10. Gauges, CircleCI, Hootsuite, etc., are the top enterprises that are using this query language. | 10. Airbnb, Uber, and Kickstarter are the top enterprises that are using this query language. |
| 11. SQLite, Ms-SQL, Oracle, PostgreSQL, and MySQL are examples of SQL database systems. | 11. Redis, MongoDB, Hbase, BigTable, CouchDB, and Cassandra are examples of NoSQL database systems. |

If you want to know more about the vertical and horizontal scaling you can visit this link

[https://stackoverflow.com/questions/11707879/difference-between-scaling-horizontally-and-vertically-for-databases#](https://stackoverflow.com/questions/11707879/difference-between-scaling-horizontally-and-vertically-for-databases)

Advantages of SQL

SQL provides various advantages which make it more popular in the field of data science. It is a perfect query language which allows data professionals and users to communicate with the database. Following are the best advantages or benefits of Structured Query Language:

**1. No programming needed**

SQL does not require a large number of coding lines for managing the database systems. We can easily access and maintain the database by using simple SQL syntactical rules. These simple rules make the SQL user-friendly.

**2. High-Speed Query Processing**

A large amount of data is accessed quickly and efficiently from the database by using SQL queries. Insertion, deletion, and up-dation operations on data are also performed in less time.

**3. Standardized Language**

SQL follows the long-established standards of ISO and ANSI, which offer a uniform platform across the globe to all its users.

**4. Portability**

The structured query language can be easily used in desktop computers, laptops, tablets, and even smartphones. It can also be used with other applications according to the user's requirements.

**5. Interactive language**

We can easily learn and understand the SQL language. We can also use this language for communicating with the database because it is a simple query language. This language is also used for receiving the answers to complex queries in a few seconds.

**6. More than one Data View**

The SQL language also helps in making the multiple views of the database structure for the different database users.

Disadvantages of SQL

With the advantages of SQL, it also has some disadvantages, which are as follows:

**1. Cost**

The operation cost of some SQL versions is high. That's why some programmers cannot use the Structured Query Language.

**2. Interface is Complex**

Another big disadvantage is that the interface of Structured query language is difficult, which makes it difficult for SQL users to use and manage it.

**3. Partial Database control**

The business rules are hidden. So, the data professionals and users who are using this query language cannot have full database control.

SQL Syntax

When you want to do some operations on the data in the database, then you must have to write the query in the predefined syntax of SQL.

The syntax of the structured query language is a unique set of rules and guidelines, which is not case-sensitive. Its Syntax is defined and maintained by the ISO and ANSI standards.

Following are some most important points about the SQL syntax which are to remember:

* You can write the keywords of SQL in both uppercase and lowercase, but writing the SQL keywords in uppercase improves the readability of the SQL query.
* SQL statements or syntax are dependent on text lines. We can place a single SQL statement on one or multiple text lines.
* You can perform most of the action in a database with SQL statements.
* SQL syntax depends on relational algebra and tuple relational calculus.

SQL Statement

[SQL](https://www.javatpoint.com/sql-tutorial) statements tell the database what operation you want to perform on the structured data and what information you would like to access from the database.

The statements of SQL are very simple and easy to use and understand. They are like plain English but with a particular syntax.

**Simple Example of SQL statement:**

1. **SELECT** "column\_name" **FROM** "table\_name";

Each SQL statement begins with any of the SQL keywords and ends with the semicolon (;). The semicolon is used in the SQL for separating the multiple Sql statements which are going to execute in the same call. In this SQL tutorial, we will use the semicolon (;) at the end of each SQL query or statement.

Most Important SQL Commands and Statements

1. [Select Statement](https://www.javatpoint.com/sql-select)
2. [Update Statement](https://www.javatpoint.com/sql-update)
3. [Delete Statement](https://www.javatpoint.com/sql-delete)
4. [Create Table Statement](https://www.javatpoint.com/sql-create-table)
5. [Alter Table Statement](https://www.javatpoint.com/sql-alter-table)
6. [Drop Table Statement](https://www.javatpoint.com/sql-drop-table)
7. [Create Database Statement](https://www.javatpoint.com/sql-create-database)
8. [Drop Database Statement](https://www.javatpoint.com/sql-drop-database)
9. [Insert Into Statement](https://www.javatpoint.com/sql-insert)
10. [Truncate Table Statement](https://www.javatpoint.com/sql-truncate-table)
11. Describe Statement
12. [Distinct Clause](https://www.javatpoint.com/sql-select-distinct)
13. Commit Statement
14. Rollback Statement
15. Create Index Statement
16. Drop Index Statement
17. Use Statement

1. SELECT Statement

This SQL statement reads the data from the SQL database and shows it as the output to the database user.

**Syntax of SELECT Statement:**

1. **SELECT** column\_name1, column\_name2, .…, column\_nameN
2. [ **FROM** table\_name ]
3. [ **WHERE** condition ]
4. [ **ORDER** **BY** order\_column\_name1 [ **ASC** | **DESC** ], .... ];

**Example of SELECT Statement:**

1. **SELECT** Emp\_ID, First\_Name, Last\_Name, Salary, City
2. **FROM** Employee\_details
3. **WHERE** Salary = 100000
4. **ORDER** **BY** Last\_Name

This example shows the **Emp\_ID, First\_Name, Last\_Name, Salary, and City** of those employees from the **Employee\_details** table whose **Salary** is **100000**. The output shows all the specified details according to the ascending alphabetical order of **Last\_Name**.

2. UPDATE Statement

This SQL statement changes or modifies the stored data in the SQL database.

**Syntax of UPDATE Statement:**

1. **UPDATE** table\_name
2. **SET** column\_name1 = new\_value\_1, column\_name2 = new\_value\_2, ...., column\_nameN = new\_value\_N
3. [ **WHERE**  CONDITION ];

**Example of UPDATE Statement:**

1. **UPDATE** Employee\_details
2. **SET** Salary = 100000
3. **WHERE** Emp\_ID = 10;

This example changes the **Salary** of those employees of the **Employee\_detail**s table whose **Emp\_ID** is **10** in the table.

3. DELETE Statement

This SQL statement deletes the stored data from the SQL database.

**Syntax of DELETE Statement:**

1. **DELETE** **FROM** table\_name
2. [ **WHERE** CONDITION ];

**Example of DELETE Statement:**

1. **DELETE** **FROM** Employee\_details
2. **WHERE** First\_Name = 'Sumit';

This example deletes the record of those employees from the **Employee\_details** table whose **First\_Name** is **Sumit** in the table.

4. CREATE TABLE Statement

This SQL statement creates the new table in the SQL database.

**Syntax of CREATE TABLE Statement:**

1. **CREATE** **TABLE** table\_name
2. (
3. column\_name1 data\_type [column1 **constraint**(s)],
4. column\_name2 data\_type [column2 **constraint**(s)],
5. .....
6. .....,
7. column\_nameN data\_type [columnN **constraint**(s)],
8. **PRIMARY** **KEY**(one or more col)
9. );

**Example of CREATE TABLE Statement:**

1. **CREATE** **TABLE** Employee\_details(
2. Emp\_Id NUMBER(4) NOT NULL,
3. First\_name **VARCHAR**(30),
4. Last\_name **VARCHAR**(30),
5. Salary Money,
6. City **VARCHAR**(30),
7. **PRIMARY** **KEY** (Emp\_Id)
8. );

This example creates the table **Employee\_details** with five columns or fields in the SQL database. The fields in the table are **Emp\_Id, First\_Name, Last\_Name, Salary,** and **City**. The **Emp\_Id** column in the table acts as a **primary key**, which means that the Emp\_Id column cannot contain duplicate values and null values.

5. ALTER TABLE Statement

This SQL statement adds, deletes, and modifies the columns of the table in the SQL database.

**Syntax of ALTER TABLE Statement:**

1. **ALTER** **TABLE** table\_name **ADD** column\_name datatype[(**size**)];

The above SQL alter statement adds the column with its datatype in the existing database table.

1. **ALTER** **TABLE** table\_name **MODIFY** column\_name column\_datatype[(**size**)];

The above 'SQL alter statement' renames the old column name to the new column name of the existing database table.

1. **ALTER** **TABLE** table\_name **DROP** **COLUMN** column\_name;

The above SQL alter statement deletes the column of the existing database table.

**Example of ALTER TABLE Statement:**

1. **ALTER** **TABLE** Employee\_details
2. **ADD** Designation **VARCHAR**(18);

This example adds the new field whose name is **Designation** with size **18** in the **Employee\_details** table of the SQL database.

6. DROP TABLE Statement

This SQL statement deletes or removes the table and the structure, views, permissions, and triggers associated with that table.

**Syntax of DROP TABLE Statement:**

1. **DROP** **TABLE** [ IF EXISTS ]
2. table\_name1, table\_name2, ……, table\_nameN;

The above syntax of the drop statement deletes specified tables completely if they exist in the database.

**Example of DROP TABLE Statement:**

1. **DROP** **TABLE** Employee\_details;

This example drops the **Employee\_details** table if it exists in the SQL database. This removes the complete information if available in the table.

7. CREATE DATABASE Statement

This SQL statement creates the new database in the database management system.

**Syntax of CREATE DATABASE Statement:**

1. **CREATE** **DATABASE** database\_name;

**Example of CREATE DATABASE Statement:**

1. **CREATE** **DATABASE** Company;

The above example creates the company database in the system.

8. DROP DATABASE Statement

This SQL statement deletes the existing database with all the data tables and views from the database management system.

**Syntax of DROP DATABASE Statement:**

1. **DROP** **DATABASE** database\_name;

**Example of DROP DATABASE Statement:**

1. **DROP** **DATABASE** Company;

The above example deletes the company database from the system.

9. INSERT INTO Statement

This SQL statement inserts the data or records in the existing table of the SQL database. This statement can easily insert single and multiple records in a single query statement.

**Syntax of insert a single record:**

1. **INSERT** **INTO** table\_name
2. (
3. column\_name1,
4. column\_name2, .…,
5. column\_nameN
6. )
7. **VALUES**
8. (value\_1,
9. value\_2, ..…,
10. value\_N
11. );

**Example of insert a single record:**

1. **INSERT** **INTO** Employee\_details
2. (
3. Emp\_ID,
4. First\_name,
5. Last\_name,
6. Salary,
7. City
8. )
9. **VALUES**
10. (101,
11. Akhil,
12. Sharma,
13. 40000,
14. Bangalore
15. );

This example inserts **101** in the first column, **Akhil** in the second column, **Sharma** in the third column, **40000** in the fourth column, and **Bangalore** in the last column of the table **Employee\_details**.

**Syntax of inserting a multiple records in a single query:**

1. **INSERT** **INTO** table\_name
2. ( column\_name1, column\_name2, .…, column\_nameN)
3. **VALUES** (value\_1, value\_2, ..…, value\_N), (value\_1, value\_2, ..…, value\_N),….;

**Example of inserting multiple records in a single query:**

1. **INSERT** **INTO** Employee\_details
2. ( Emp\_ID, First\_name, Last\_name, Salary, City )
3. **VALUES** (101, Amit, Gupta, 50000, Mumbai), (101,  John, Aggarwal, 45000, Calcutta), (101, Sidhu, Arora, 55000, Mumbai);

This example inserts the records of three employees in the **Employee\_details** table in the single query statement.

10. TRUNCATE TABLE Statement

This SQL statement deletes all the stored records from the table of the SQL database.

**Syntax of TRUNCATE TABLE Statement:**

1. **TRUNCATE** **TABLE** table\_name;

**Example of TRUNCATE TABLE Statement:**

1. **TRUNCATE** **TABLE** Employee\_details;

This example deletes the record of all employees from the Employee\_details table of the database.

11. DESCRIBE Statement

This SQL statement tells something about the specified table or view in the query.

**Syntax of DESCRIBE Statement:**

1. DESCRIBE table\_name | view\_name;

**Example of DESCRIBE Statement:**

1. DESCRIBE Employee\_details;

This example explains the structure and other details about the **Employee\_details** table.

12. DISTINCT Clause

This SQL statement shows the distinct values from the specified columns of the database table. This statement is used with the **SELECT** keyword.

**Syntax of DISTINCT Clause:**

1. **SELECT** **DISTINCT** column\_name1, column\_name2, ...
2. **FROM** table\_name;

**Example of DISTINCT Clause:**

1. **SELECT** **DISTINCT** City, Salary
2. **FROM** Employee\_details;

This example shows the distinct values of the **City** and **Salary** column from the **Employee\_details** table.

13. COMMIT Statement

This SQL statement saves the changes permanently, which are done in the transaction of the SQL database.

**Syntax of COMMIT Statement:**

1. **COMMIT**

**Example of COMMIT Statement:**

1. **DELETE** **FROM** Employee\_details
2. **WHERE** salary = 30000;
3. **COMMIT**;

This example deletes the records of those employees whose **Salary** is **30000** and then saves the changes permanently in the database.

14. ROLLBACK Statement

This SQL statement undo the transactions and operations which are not yet saved to the SQL database.

**Syntax of ROLLBACK Statement:**

1. **ROLLBACK**

**Example of ROLLBACK Statement:**

1. **DELETE** **FROM** Employee\_details
2. **WHERE** City = Mumbai;
3. **ROLLBACK**;

This example deletes the records of those employees whose **City** is **Mumbai** and then undo the changes in the database.

15. CREATE INDEX Statement

This SQL statement creates the new index in the SQL database table.

**Syntax of CREATE INDEX Statement:**

1. **CREATE** **INDEX** index\_name
2. **ON** table\_name ( column\_name1, column\_name2, …, column\_nameN );

**Example of CREATE INDEX Statement:**

1. **CREATE** **INDEX** idx\_First\_Name
2. **ON** employee\_details (First\_Name);

This example creates an index **idx\_First\_Name** on the **First\_Name** column of the **Employee\_details** table.

16. DROP INDEX Statement

This SQL statement deletes the existing index of the SQL database table.

**Syntax of DROP INDEX Statement:**

1. **DROP** **INDEX** index\_name;

**Example of DROP INDEX Statement:**

1. **DROP** **INDEX** idx\_First\_Name;

This example deletes the index **idx\_First\_Name** from the SQL database.

17. USE Statement

This SQL statement selects the existing SQL database. Before performing the operations on the database table, you have to select the database from the multiple existing databases.

**Syntax of USE Statement:**

1. USE database\_name;

**Example of USE DATABASE Statement:**

1. USE Company;

This example uses the company database.

SQL Data Types

Data types are used to represent the nature of the data that can be stored in the database table. For example, in a particular column of a table, if we want to store a string type of data then we will have to declare a string data type of this column.

Data types mainly classified into three categories for every database.

* String Data types
* Numeric Data types
* Date and time Data types

You can see more about the data type [Here](https://www.javatpoint.com/sql-data-types).

SQL Operators

SQL statements generally contain some reserved words or characters that are used to perform operations such as comparison and arithmetical operations etc. These reserved words or characters are known as operators.

Generally, there are three types of operators in SQL:

1. SQL Arithmetic Operators
2. SQL Comparison Operators
3. SQL Logical Operators

SQL Arithmetic Operators:

Let's assume two variables "a" and "b". Here "a" is valued 50 and "b" valued 100.

**Example:**

|  |  |  |
| --- | --- | --- |
| **Operators** | **Descriptions** | **Examples** |
| + | It is used to add containing values of both operands | a+b will give 150 |
| - | It subtracts right hand operand from left hand operand | a-b will give -50 |
| \* | It multiply both operand's values | a\*b will give 5000 |
| / | It divides left hand operand by right hand operand | b/a will give 2 |
| % | It divides left hand operand by right hand operand and returns reminder | b%a will give 0 |

SQL Comparison Operators:

Let's take two variables "a" and "b" that are valued 50 and 100.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Examine both operands value that are equal or not,if yes condition become true. | (a=b) is not true |
| != | This is used to check the value of both operands equal or not,if not condition become true. | (a!=b) is true |
| < > | Examines the operand's value equal or not, if values are not equal condition is true | (a<>b) is true |
| > | Examine the left operand value is greater than right Operand, if yes condition becomes true | (a>b) is not true |
| < | Examines the left operand value is less than right Operand, if yes condition becomes true | (a<="" td=""> |
| >= | Examines that the value of left operand is greater than or equal to the value of right operand or not,if yes condition become true | (a>=b) is not true |
| <= | Examines that the value of left operand is less than or equal to the value of right operand or not, if yes condition becomes true | (a<=b) is true |
| !< | Examines that the left operand value is not less than the right operand value | (a!<="" td=""> |
| !> | Examines that the value of left operand is not greater than the value of right operand | (a!>b) is true |

SQL Logical Operators:

This is the list of logical operators used in SQL.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| ALL | this is used to compare a value to all values in another value set. |
| AND | this operator allows the existence of multiple conditions in an SQL statement. |
| ANY | this operator is used to compare the value in list according to the condition. |
| BETWEEN | this operator is used to search for values, that are within a set of values |
| IN | this operator is used to compare a value to that specified list value |
| NOT | the NOT operator reverse the meaning of any logical operator |
| OR | this operator is used to combine multiple conditions in SQL statements |
| EXISTS | the EXISTS operator is used to search for the presence of a row in a specified table |
| LIKE | this operator is used to compare a value to similar values using wildcard operator |

# SQL CREATE Database

The **SQL CREATE DATABASE** statement is used by a developer to create a database.

Let's see the syntax of SQL CREATE DATABASE:

1. **CREATE** **DATABASE** database\_name;

If you want to add tables in that database, you can use CREATE TABLE statement.

Create Database in MySQL

In MySQL, same command is used to create a database.

1. **CREATE** **DATABASE** database\_name;

## **Create Database in Oracle**

You don't need to create database in Oracle. In Oracle database, you can create tables directly.

SQL DROP Database

**SQL DROP statement** is used to delete or remove indexes from a table in the database.

If you want to delete or drop an existing database in a SQL schema, you can use SQL DROP DATABASE

Let's see the syntax of sql DROP DATABASE:

**DROP** **DATABASE** database\_name;

If you delete or drop the database, all the tables and views will also be deleted. So be careful while using this command.

# SQL RENAME Database

SQL RENAME DATABASE is used when you need to change the name of your database. Sometimes it is used because you think that the original name is not more relevant to the database or you want to give a temporary name to that database.

Let's see how to rename MySql and SQL Server databases.

#### **Rename MySQL database**

To rename the mysql database, you need to follow the following syntax:

1. RENAME **DATABASE** old\_db\_name **TO** new\_db\_name;

Rename SQL server database using T-SQL

This command is useful for SQL server 2005, 2008, 2008R2 and 2012.

1. **ALTER** **DATABASE** old\_name **MODIFY** **NAME** = new\_name

If you are using SQL server 2000, you can also use this command to rename the database. But, Microsoft phased out it.

1. **EXEC** sp\_renamedb 'old\_name' , 'new\_name'

# SQL SELECT Database

In MySQL database, you need to select a database first before executing any query on table, view etc. To do so, we use following query:

Use database\_name;

# SQL Table

Table is a collection of data, organized in terms of rows and columns. In DBMS term, table is known as relation and row as tuple.

#### **Note: A table has a specified number of columns, but can have any number of rows.**

Table is the simple form of data storage. A table is also considered as a convenient representation of relations.

Let's see an example of an employee table:

|  |  |  |
| --- | --- | --- |
| Employee | | |
| **EMP\_NAME** | **ADDRESS** | **SALARY** |
| Ankit | Lucknow | 15000 |
| Raman | Allahabad | 18000 |
| Mike | New York | 20000 |

In the above table, "Employee" is the table name, "EMP\_NAME", "ADDRESS" and "SALARY" are the column names. The combination of data of multiple columns forms a row e.g. "Ankit", "Lucknow" and 15000 are the data of one row.

## **SQL TABLE Variable**

The **SQL Table variable** is used to create, modify, rename, copy and delete tables. Table variable was introduced by Microsoft.

It was introduced with SQL server 2000 to be an alternative of temporary tables.

It is a variable where we temporary store records and results. This is same like temp table but in the case of temp table we need to explicitly drop it.

Table variables are used to store a set of records. So declaration syntax generally looks like CREATE TABLE syntax.

1. **create** **table** "tablename"
2. ("column1" "data type",
3. "column2" "data type",
4. ...
5. "columnN" "data type");

When a transaction rolled back the data associated with table variable is not rolled back.

A table variable generally uses lesser resources than a temporary variable.

Table variable cannot be used as an input or an output parameter.

# SQL CREATE TABLE

SQL CREATE TABLE statement is used to create table in a database.

If you want to create a table, you should name the table and define its column and each column's data type.

Let's see the simple syntax to create the table.

1. **create** **table** "tablename"
2. ("column1" "data type",
3. "column2" "data type",
4. "column3" "data type",
5. ...
6. "columnN" "data type");

The data type of the columns may vary from one database to another. For example, NUMBER is supported in Oracle database for integer value whereas INT is supported in MySQL.

Let us take an example to create a STUDENTS table with ID as primary key and NOT NULL are the constraint showing that these fields cannot be NULL while creating records in the table.

**CREATE** **TABLE** STUDENTS (

ID **INT**   NOT NULL,

**NAME** **VARCHAR** (20) NOT NULL,

AGE **INT**                         NOT NULL,

ADDRESS **CHAR** (25),

**PRIMARY** **KEY** (ID)

);

You can verify it, if you have created the table successfully by looking at the message displayed by the SQL Server, else you can use DESC command as follows:

DESC STUDENTS;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **FIELD** | **TYPE** | **NULL** | **KEY** | **DEFAULT** | **EXTRA** |
| ID | Int(11) | NO | PRI |  |  |
| NAME | Varchar(20) | NO |  |  |  |
| AGE | Int(11) | NO |  |  |  |
| ADDRESS | Varchar(25) | YES |  | NULL |  |

4 rows in set (0.00 sec)

Now you have the STUDENTS table available in your database and you can use to store required information related to students.

## **SQL CREATE TABLE Example in MySQL**

Let's see the command to create a table in MySQL database.

**CREATE** **TABLE** Employee

(

EmployeeID **int**,

FirstName **varchar**(255),

LastName **varchar**(255),

Email **varchar**(255),

AddressLine **varchar**(255),

City **varchar**(255)

1. );

## **SQL CREATE TABLE Example in Oracle**

Let's see the command to create a table in Oracle database.

**CREATE** **TABLE** Employee

(

EmployeeID number(10),

FirstName varchar2(255),

LastName varchar2(255),

Email varchar2(255),

AddressLine varchar2(255),

City varchar2(255)

);

## **SQL CREATE TABLE Example in Microsoft SQLServer**

Let's see the command to create a table in SQLServer database. It is same as MySQL and Oracle.

1. **CREATE** **TABLE** Employee
2. (
3. EmployeeID **int**,
4. FirstName **varchar**(255),
5. LastName **varchar**(255),
6. Email **varchar**(255),
7. AddressLine **varchar**(255),
8. City **varchar**(255)
9. );

## **Create a Table using another table**

We can create a copy of an existing table using the create table command. The new table gets the same column signature as the old table. We can select all columns or some specific columns.

If we create a new table using an old table, the new table will be filled with the existing value from the old table.

The basic syntax for creating a table with the other table is:

**CREATE** **TABLE** table\_name  **AS**

**SELECT** column1, column2,...

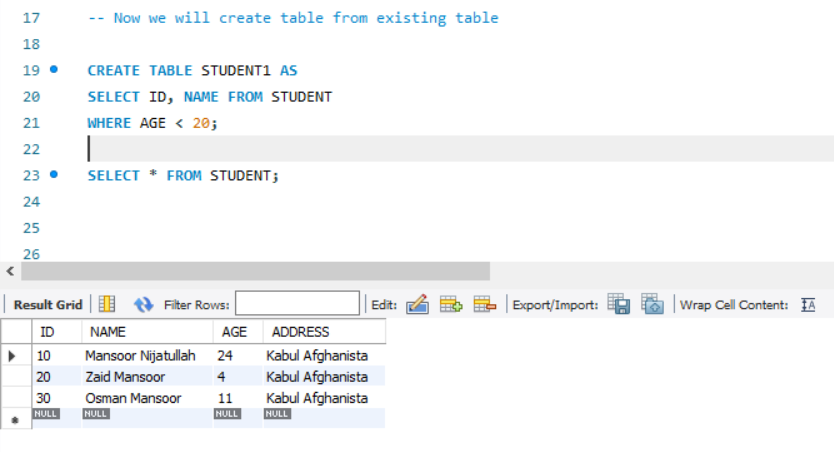
**FROM** old\_table\_name **WHERE** ..... ;

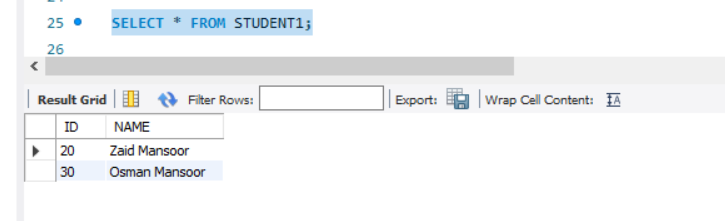
The following SQL creates a copy **of** the employee **table**.

**CREATE** **TABLE** EmployeeCopy **AS**

**SELECT** EmployeeID, FirstName, Email

**FROM** Employee;





## **SQL Primary Key with CREATE TABLE Statement**

The following query creates a PRIMARY KEY on the "D" column when the "Employee" table is created.

### **MySQL**

**CREATE** **TABLE** Employee(

EmployeeID NOT NULL,

FirstName **varchar**(255) NOT NULL,

LastName **varchar**(255),

City **varchar**(255),

**PRIMARY** **KEY** (EmployeeID)

);

### **SQL Server / Oracle / MS Access**

**CREATE** **TABLE** Employee(

EmployeeID NOT NULL **PRIMARY** **KEY**,

FirstName **varchar**(255) NOT NULL,

LastName **varchar**(255),

City **varchar**(255)

);

**Use the following query to define a PRIMARY KEY constraints on multiple columns, and to allow naming of a PRIMARY KEY constraints.**

### **For MySQL / SQL Server /Oracle / MS Access**

1. **CREATE** **TABLE** Employee(
2. EmployeeID NOT NULL,
3. FirstName **varchar**(255) NOT NULL,
4. LastName **varchar**(255),
5. City **varchar**(255),
6. **CONSTRAINT**     PK\_Employee **PRIMARY** **KEY** (EmployeeID, FirstName)
7. );

SQL DROP TABLE

A SQL DROP TABLE statement is used to delete a table definition and all data from a table.

This is very important to know that once a table is deleted all the information available in the table is lost forever, so we have to be very careful when using this command.

Let's see the syntax to drop the table from the database.

1. **DROP** **TABLE** "table\_name";

Let us take an example:

First we verify STUDENTS table and then we would delete it from the database.

1. SQL> **DESC** STUDENTS;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **FIELD** | **TYPE** | **NULL** | **KEY** | **DEFAULT** | **EXTRA** |
| ID | Int(11) | NO | PRI |  |  |
| NAME | Varchar(20) | NO |  |  |  |
| AGE | Int(11) | NO |  |  |  |
| ADDRESS | Varchar(25) | YES |  | NULL |  |

1. 4 rows in set (0.00 sec)  This shows that STUDENTS table is available in the database, so we can drop it as follows:
2. SQL>**DROP** **TABLE** STUDENTS;

Now, use the following command to check whether table exists or not.

1. SQL> **DESC** STUDENTS;

As you can see, table is dropped so it doesn't display it.

SQL DROP TABLE Example in MySQL

Let's see the command to drop a table from the MySQL database.

1. **DROP** **TABLE** table\_name;

SQL DROP TABLE Example in Oracle

Let's see the command to drop a table from Oracle database. It is same as MySQL.

1. **DROP** **TABLE** table\_name;

SQL DROP TABLE Example in Microsoft SQLServer

Let's see the command to drop a table from SQLServer database. It is same as MySQL.

1. **DROP** **TABLE** table\_name;

# SQL DELETE TABLE

The DELETE statement is used to delete rows from a table. If you want to remove a specific row from a table you should use WHERE condition.

1. **DELETE** **FROM** table\_name [**WHERE** condition];

But if you do not specify the WHERE condition it will remove all the rows from the table.

1. **DELETE** **FROM** table\_name;

There are some more terms similar to DELETE statement like as DROP statement and TRUNCATE statement but they are not exactly same there are some differences between them.

## **Difference between DELETE and TRUNCATE statements**

There is a slight difference b/w delete and truncate statement. The **DELETE statement** only deletes the rows from the table based on the condition defined by WHERE clause or delete all the rows from the table when condition is not specified.

But it does not free the space containing by the table.

The **TRUNCATE statement:** it is used to delete all the rows from the table **and free the containing space.**

Let's see an "employee" table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Emp\_id** | **Name** | **Address** | **Salary** |
| 1 | Aryan | Allahabad | 22000 |
| 2 | Shurabhi | Varanasi | 13000 |
| 3 | Pappu | Delhi | 24000 |

Execute the following query to truncate the table:

1. **TRUNCATE** **TABLE** employee;

## **Difference b/w DROP and TRUNCATE statements**

When you use the drop statement it deletes the table's row together with the table's definition so all the relationships of that table with other tables will no longer be valid.

**When you drop a table:**

* Table structure will be dropped
* Relationship will be dropped
* Integrity constraints will be dropped
* Access privileges will also be dropped

On the other hand when we **TRUNCATE** a table, the table structure remains the same, so you will not face any of the above problems.

# SQL RENAME TABLE

**SQL RENAME TABLE** syntax is used to change the name of a table. Sometimes, we choose non-meaningful name for the table. So it is required to be changed.

Let's see the syntax to rename a table from the database.

1. **ALTER** **TABLE** table\_name
2. RENAME **TO** new\_table\_name;

Optionally, you can write following command to rename the table.

1. RENAME old\_table \_name **To** new\_table\_name;

Let us take an example of a table named "STUDENTS", now due to some reason we want to change it into table name "ARTISTS".

Table1: students

|  |  |  |
| --- | --- | --- |
| **Name** | **Age** | **City** |
| Amrita gill | 25 | Amritsar |
| Amrender sirohi | 22 | Ghaziabad |
| Divya khosla | 20 | Delhi |

You should use any one of the following syntax to RENAME the table name:

1. **ALTER** **TABLE** STUDENTS
2. RENAME **TO** ARTISTS;

Or

1. RENAME STUDENTS **TO** ARTISTS;
2. **After** that the **table** "students" will be changed **into** **table** **name** "artists"

# SQL TRUNCATE TABLE

A truncate SQL statement is used to remove all rows (complete data) from a table. It is similar to the DELETE statement with no WHERE clause.

#### **TRUNCATE TABLE Vs DELETE TABLE**

Truncate table is faster and uses lesser resources than DELETE TABLE command.

#### **TRUNCATE TABLE Vs DROP TABLE**

Drop table command can also be used to delete complete table but it deletes table structure too. TRUNCATE TABLE doesn't delete the structure of the table.

Let's see the syntax to truncate the table from the database.

1. **TRUNCATE** **TABLE** table\_name;

For example, you can write following command to truncate the data of employee table

1. **TRUNCATE** **TABLE** Employee;

**Note:** The rollback process is not possible after truncate table statement. Once you truncate a table you cannot use a flashback table statement to retrieve the content of the table.

# SQL COPY TABLE

If you want to copy a SQL table into another table in the same SQL server database, it is possible by using the select statement.

The syntax of copying table from one to another is given below:

1. **SELECT** \* **INTO** <destination\_table> **FROM** <source\_table>

For example, you can write following command to copy the records of hr\_employee table into employee table.

1. **SELECT** \* **INTO** admin\_employee **FROM** hr\_employee;

#### **Note: SELECT INTO is totally different from INSERT INTO statement.**

# SQL TEMP TABLE

The concept of temporary table is introduced by SQL server. It helps developers in many ways:

**Temporary tables** can be created at run-time and can do all kinds of operations that a normal table can do. These temporary tables are created inside tempdb database.

There are two types of temp tables based on the behavior and scope.

1. Local Temp Variable
2. Global Temp Variable

## **Local Temp Variable**

Local temp tables are only available at current connection time. It is automatically deleted when user disconnects from instances. It is started with hash (#) sign.

1. **CREATE** **TABLE** #**local** **temp** **table** (
2. User id **int**,
3. Username **varchar** (50),
4. User address **varchar** (150)
5. )

## **Global Temp Variable**

Global temp tables name starts with double hash (##). Once this table is created, it is like a permanent table. It is always ready for all users and not deleted until the total connection is withdrawn.

1. **CREATE** **TABLE** ##new **global** **temp** **table** (
2. User id **int**,
3. User **name** **varchar** (50),
4. User address **varchar** (150)
5. )

SQL ALTER TABLE

The ALTER TABLE statement is used to add, modify or delete columns in an existing table. It is also used to rename a table.

You can also use SQL ALTER TABLE command to add and drop various constraints on an existing table.

SQL ALTER TABLE Add Column

If you want to add columns in SQL table, the SQL alter table syntax is given below:

1. **ALTER** **TABLE** table\_name **ADD** column\_name **column**-definition;

If you want to add multiple columns in table, the SQL table will be

1. **ALTER** **TABLE** table\_name
2. **ADD** (column\_1 **column**-definition,
3. column\_2 **column**-definition,
4. .....
5. column\_n **column**-definition);

SQL ALTER TABLE Modify Column

If you want to modify an existing column in SQL table, syntax is given below:

1. **ALTER** **TABLE** table\_name **MODIFY** column\_name column\_type;

If you want to modify multiple columns in table, the SQL table will be

1. **ALTER** **TABLE** table\_name
2. **MODIFY** (column\_1 column\_type,
3. column\_2 column\_type,
4. .....
5. column\_n column\_type);

SQL ALTER TABLE DROP Column

The syntax of alter table drop column is given below:

1. **ALTER** **TABLE** table\_name **DROP** **COLUMN** column\_name;

SQL ALTER TABLE RENAME Column

The syntax of alter table rename column is given below:

1. **ALTER** **TABLE** table\_name
2. RENAME **COLUMN** old\_name **to** new\_name;

# SQL SELECT

The most commonly used SQL command is **SELECT statement**. It is used to query the database and retrieve selected data that follow the conditions we want.

In simple words, we can say that the select statement used to query or retrieve data from a table in the database.

Let's see the syntax of select statement.

1. **SELECT** expressions
2. **FROM** tables
3. **WHERE** conditions;

Here expression is the column that we want to retrieve.

Tables indicate the tables, we want to retrieve records from.

## **Optional clauses in SELECT statement**

There are some optional clauses in SELECT statement:

**[WHERE Clause]** : It specifies which rows to retrieve.

**[GROUP BY Clause]** : Groups rows that share a property so that the aggregate function can be applied to each group.

**[HAVING Clause]** : It selects among the groups defined by the GROUP BY clause.

**[ORDER BY Clause]** : It specifies an order in which to return the rows.

For example, let a database table: student\_details;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **First\_name** | **Last\_name** | **Age** | **Subject** | **Hobby** |
| 1 | Amar | Sharma | 20 | Maths | Cricket |
| 2 | Akbar | Khan | 22 | Biology | Football |
| 3 | Anthony | Milton | 25 | Commerce | Gambling |

From the above example, select the first name of all the students. To do so, query should be like this:

1. **SELECT** first\_name **FROM** student\_details;

Note: the SQL commands are not case sensitive. We can also write the above SELECT statement as:

1. **select** first\_name **from** student\_details;

Now, you will get following data:

|  |
| --- |
| Amar |
| Akbar |
| Anthony |

We can also retrieve data from more than one column. For example, to select first name and last name of all the students, you need to write

1. **SELECT** first\_name, last\_name **FROM** student\_details;

Now, you will get following data:

|  |  |
| --- | --- |
| Amar | Sharma |
| Akbar | Khan |
| Anthony | Milton |

We can also use clauses like WHERE, GROUP BY, HAVING, ORDER BY with SELECT statement.

Here a point is notable that only SELECT and FROM statements are necessary in SQL SELECT statements. Other clauses like WHERE, GROUP BY, ORDER BY, HAVING may be optional.

# SQL SELECT UNIQUE

Actually, there is no difference between DISTINCT and UNIQUE.

**SELECT UNIQUE** is an old syntax which was used in oracle description but later ANSI standard defines DISTINCT as the official keyword.

After that oracle also added DISTINCT but did not withdraw the service of UNIQUE keyword for the sake of backward compatibility.

In simple words, we can say that SELECT UNIQUE statement is used to retrieve a unique or distinct element from the table.

Let's see the syntax of select unique statement.

1. **SELECT** **UNIQUE** column\_name
2. **FROM** table\_name;

SQL SELECT DISTINCT statement can also be used for the same cause.

# SQL SELECT DISTINCT

The **SQL DISTINCT command** is used with SELECT key word to retrieve only distinct or unique data.

In a table, there may be a chance to exist a duplicate value and sometimes we want to retrieve only unique values. In such scenarios, SQL SELECT DISTINCT statement is used.

#### **Note: SQL SELECT UNIQUE and SQL SELECT DISTINCT statements are same.**

Let's see the syntax of select distinct statement.

1. **SELECT** **DISTINCT** column\_name ,column\_name
2. **FROM**  table\_name;

Let's try to understand it by the table given below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_Name** | **Gender** | **Mobile\_Number** | **HOME\_TOWN** |
| Rahul Ojha | Male | 7503896532 | Lucknow |
| Disha Rai | Female | 9270568893 | Varanasi |
| Sonoo Jaiswal | Male | 9990449935 | Lucknow |

Here is a table of students from where we want to retrieve distinct information For example: distinct home-town.

1. **SELECT** **DISTINCT** home\_town
2. **FROM** students

Now, it will return two rows.

|  |
| --- |
| **HOME\_TOWN** |
| Lucknow |
| Varanasi |

# SQL SELECT COUNT

The **SQL COUNT()** function is used to return the number of rows in a query.

The COUNT() function is used with SQL SELECT statement and it is very useful to count the number of rows in a table having enormous data.

**For example:** If you have a record of the voters in selected area and want to count the number of voters then it is very difficult to do it manually but you can do it easily by using the SQL SELECT COUNT query.

Let's see the syntax of SQL COUNT statement.

**SELECT** COUNT (expression)

**FROM** tables

**WHERE** conditions;

Let's see the examples of sql select count function.

#### **SQL SELECT COUNT(column\_name)**

1. **SELECT** COUNT(**name**) **FROM** employee\_table;

**It will return the total number of names of employee\_table. But null fields will not be counted.**

#### **SQL SELECT COUNT(\*)**

1. **SELECT** COUNT(\*) **FROM** employee\_table;

The "select count(\*) from table" is used to return the number of records in table.

#### **SQL SELECT COUNT(DISTINCT column\_name)**

1. **SELECT** COUNT(**DISTINCT** **name**) **FROM** employee\_table;

It will return the total distinct names of employee\_table.

# SQL SELECT TOP

The SQL SELECT TOP Statement is used to select top data from a table. The top clause specifies that how many rows are returned.

Let's see an example. If a table has a large number of data, select top statement determines that how many rows will be retrieved from the given table.

There is an example of employee table:

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_ID** | **NAME** | **SIR\_NAME** | **USER\_NAME** |
| 1 | RAHUL | OJHA | ra@jha |
| 2 | ANU | SHARMA | anusha1 |
| 3 | RAVI | SINGHAL | ravin |

Let's see the syntax for the select top statement.

1. **SELECT** COUNT (expression)

Let's see the example of sql select top statement.

1. **SELECT** **TOP** 2 \* **FROM** employee

It will return the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_ID** | **NAME** | **SIR\_NAME** | **USER\_NAME** |
| 1 | RAHUL | OJHA | ra@jha |
| 2 | ANU | SHARMA | anusha1 |

# SQL SELECT FIRST

The SQL first() function is used to return the first value of the selected column.

Let's see the syntax of sql select first() function:

1. **SELECT** **FIRST**(column\_name) **FROM** table\_name;

Here a point is notable that first function is only supported by MS Access.

If you want to retrieve the first value of the "customer\_name" column from the "customers" table, you need to write following query:

1. **SELECT** **FIRST**(customer\_name) **AS** first\_customer **FROM** customers;

Let us take the example of CUSTOMERS to examine SQL SELECT FIRST command:

Table CUSTOMERS

|  |  |  |  |
| --- | --- | --- | --- |
| **CUSTOMER\_NAME** | **AGE** | **ADDRESS** | **EXPENDITURE** |
| KAMAL SHARMA | 26 | GHAZIABAD | 6000 |
| ROBERT PETT | 23 | NEWYORK | 26000 |
| SHIKHA SRIVASTAV | 22 | DELHI | 9000 |

If you want to retrieve the first value of the "customer\_name" column from the "customers" table, you need to write following query:

Let's see the syntax of sql select first() function:

1. **SELECT** **FIRST** (CUSTOMER\_NAME) **AS** first\_customer **FROM** CUSTOMERS;
2. **After** that query, you will find the result:
3. KAMAL SHARMA

#### **Note: The SELECT FIRST statement is only supported by MS Access. This statement doesn't work with other databases like Oracle, MySQL etc.**

# SQL SELECT LAST

The last() function is used to return the last value of the specified column.

Syntax for SQL SELECT LAST() FUNCTION:

1. **SELECT** **LAST** (column\_name) **FROM** table\_name;

You should note that the last() function is only supported in MS Access. But there are ways to get the last record in MySql, SQL Server, Oracle etc. databases.

**My SQL syntax:**

1. **SELECT** column\_name **FROM** table\_name
2. **ORDER** **BY** column\_name **DESC**
3. LIMIT 1;

**SQL Server syntax:**

1. **SELECT** **TOP** 1 column\_name **FROM** table\_name
2. **ORDER** **BY** column\_name **DESC**;

**Oracle syntax:**

1. **SELECT** column\_name **FROM** table\_name
2. **ORDER** **BY** column\_name **DESC**
3. **WHERE** ROWNUM <=1;

Let us take the example of CUSTOMERS to examine SQL SELECT LAST command:

Table CUSTOMERS

|  |  |  |  |
| --- | --- | --- | --- |
| **CUSTOMER\_NAME** | **AGE** | **ADDRESS** | **EXPENDITURE** |
| KAMAL SHARMA | 26 | GHAZIABAD | 6000 |
| ROBERT PETT | 23 | NEWYORK | 26000 |
| SHIKHA SRIVASTAV | 22 | DELHI | 9000 |

If you want to retrieve the last value of the "customer\_name" column from the "customers" table, you need to write following query:

1. **SELECT** **LAST** (CUSTOMER\_NAME) **AS** LAST\_CUSTOMER **FROM** CUSTOMERS;
2. **After** that query, you will find the result:
3. SHIKHA SRIVASTAV

# SQL SELECT RANDOM

The SQL SELECT RANDOM() function returns the random row. It can be used in online exam to display the random questions.

There are a lot of ways to select a random record or row from a database table. Each database server needs different SQL syntax.

If you want to select a random row with **MY SQL**:

1. **SELECT** **column** **FROM** **table**
2. **ORDER** **BY** RAND ( )
3. LIMIT 1

If you want to select a random row with **Microsoft SQL server**:

1. **SELECT** **TOP** 1 **column** **FROM** **table**
2. **ORDER** **BY** NEW ID()

If you want to select a random record with **ORACLE**:

1. **SELECT** **column** **FROM**
2. (**SELECT** **column** **FROM** **table**
3. **ORDER** **BY** dbms\_random.value)
4. **WHERE** rownum =1

If you want to select a random row with **PostgreSQL**:

1. **SELECT** **column** **FROM** **table**
2. **ORDER** **BY** RAND()
3. LIMIT 1

# SQL SELECT AS

**SQL AS** is used to assign temporarily a new name to a table column.

It makes easy presentation of query results and allows the developer to label results more accurately without permanently renaming table columns.

Let's see the example of select as:

**SELECT** day\_of\_order **AS** "Date"

Customer **As** "Client",

Product,

Quantity,

**FROM** orders;

Let us take a table named orders, it contains:

|  |  |  |  |
| --- | --- | --- | --- |
| **Day\_of\_order** | **Customer** | **Product** | **Quantity** |
| 11-09-2001 | Ajeet | Mobile | 2 |
| 13-12-2001 | Mayank | Laptop | 20 |
| 26-12-2004 | Balaswamy | Water cannon | 35 |

After applying this SQL AS example syntax

**SELECT** day\_of\_order **AS** "Date"

Customer **As** "Client",

Product,

Quantity,

**FROM** orders;

Result will be shown as this table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Client** | **Product** | **Quantity** |
| 11-09-2001 | Ajeet | Mobile | 2 |
| 13-12-2001 | Mayank | Laptop | 20 |
| 26-12-2004 | Balaswamy | Water cannon | 35 |

#### **Note: SQL AS is same as SQL ALIAS.**

# SQL SELECT IN

SQL IN is an operator used in a SQL query to help reduce the need to use multiple SQL "OR" conditions.

It is used in SELECT, INSERT, UPDATE or DELETE statement.

#### **Advantage of SQL SELECT IN**

It minimizes the use of SQL OR operator.

Let's see the syntax for SQL IN:

1. Expression IN (value 1, value 2 ... value n);

Take an example with character values.

1. **SELECT** \*
2. **FROM** students
3. **WHERE** students\_name IN ( Amit , Raghav, Rajeev)

Let's take another example with numeric values.

**SELECT** \*

**FROM** marks

**WHERE** roll\_no IN (001, 023, 024);

# SQL SELECT from Multiple Tables

This statement is used to retrieve fields from multiple tables. To do so, we need to use join query to get data from multiple tables.

Let's see the example for the select from multiple tables:

**SELECT** orders.order\_id, suppliers.**name**

**FROM** suppliers

**INNER** JOIN orders

**ON** suppliers.supplier\_id = orders.supplier\_id

**ORDER** **BY** order\_id;

Let us take three tables, two tables of customers named customer1 and customer2 and the third table is product table.

**Customer1 table**

|  |  |
| --- | --- |
| **Cus\_id** | **Name1** |
| 1 | Jack |
| 2 | Jill |

**Customer2 table**

|  |  |
| --- | --- |
| **Cus\_id** | **Name2** |
| 1 | Sandy |
| 2 | Venus |

**Product table**

|  |  |  |
| --- | --- | --- |
| **P\_id** | **Cus\_id** | **P\_name** |
| 1 | 1 | Laptop |
| 2 | 2 | Phone |
| 3 | P1 | Pen |
| 4 | P2 | Notebook |

**Example syntax to select from multiple tables:**

**SELECT** p. p\_id, p.cus\_id, p.p\_name, c1.name1, c2.name2

**FROM** product **AS** p

LEFT JOIN customer1 **AS** c1

**ON** p.cus\_id=c1.cus\_id

LEFT JOIN customer2 **AS** c2

**ON** p.cus\_id = c2.cus\_id

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_id** | **Cus\_id** | **P\_name** | **P\_name** | **P\_name** |
| 1 | 1 | Laptop | Jack | NULL |
| 2 | 2 | Phone | Jill | NULL |
| 3 | P1 | Pen | NULL | Sandy |
| 4 | P2 | Notebook | NULL | Venus |

# SQL SELECT DATE

SQL SELECT DATE is used to retrieve a date from a database. If you want to find a particular date from a database, you can use this statement.

**For example:** let's see the query to get all the records after '2013-12-12'.

1. **SELECT** \* **FROM**
2. **table**-**name** **WHERE** your **date**-**column** >= '2013-12-12'

Let's see the another query to get all the records after '2013-12-12' and before '2013-12-13' date.

1. **SELECT**\* **FROM**
2. **table**-**name** **where** your **date**-**column** < '2013-12-13' and your **date**-**column** >= '2013-12-12'

If you want to compare the dates within the query, you should use BETWEEN operator to compare the dates.

1. **SELECT** \* **FROM**
2. table\_name **WHERE** yourdate BETWEEN '2012-12-12' and '2013-12-12'

Or if you are looking for one date in particular you can use. You should change the date parameter into the acceptable form.

1. **SELECT**\* **FROM**
2. table\_name **WHERE** cast (datediff (day, 0, yourdate) **as** datetime) = '2012-12-12'

# SQL SELECT SUM

It is also known as SQL SUM() function. It is used in a SQL query to return summed value of an expression.

Let's see the Syntax for the select sum function:

1. **SELECT** SUM (expression)
2. **FROM** tables
3. **WHERE** conditions;

expression may be numeric field or formula.

This would produce the following result.

|  |  |  |
| --- | --- | --- |
| **ID** | **EMPLOYEE\_NAME** | **SALARY** |
| 1 | JACK REACHER | 32000 |
| 2 | PADMA MAHESHWARI | 22000 |
| 3 | JOE PETRA | 41000 |
| 4 | AMBUJ AGRAWAL | 21000 |

After using this SQL SELECT SUM example, it will produce the result containing the sum of the salary greater than 20000.

Total salary: 116,000

#### **SQL SUM EXAMPLE with single field:**

If you want to know how the combined total salary of all employee whose salary is above 20000 per month.

1. **SELECT** SUM (salary) **AS** "Total Salary"
2. **FROM** employees
3. **WHERE** salary > 20000;

In this example, you will find the expression as "Total Salary" when the result set is returned.

#### **SQL SUM EXAMPLE with SQL DISTINCT:**

You can also use SQL DISTINCT clause with SQL SUM function.

1. **SELECT** SUM (**DISTINCT** salary) **AS** "Total Salary"
2. **FROM** employees
3. **WHERE** salary > 20000;

#### **SQL SUM EXAMPLE with SQL GROUP BY:**

Sometimes there is a need to use the SQL GROUP BY statement with the SQL SUM function.

For example, we could also use the SQL SUM function to return the name of department and the total sales related to department.

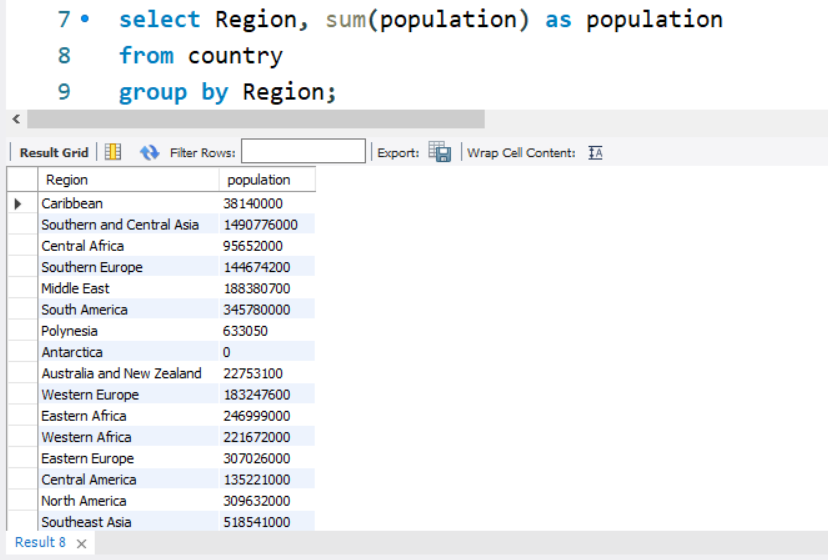
1. **SELECT** department, SUM (sales) **AS** "Total Sales"
2. **FROM** order\_details
3. **GROUP** **BY** department;

Let us take a table named order\_details

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **DEPARTMENT** | **DATE** | **DAILY SALES** |
| 1 | Mechanical | 2012-08-13 | 360 |
| 2 | Electrical | 2012-08-13 | 100 |
| 2 | Electrical | 2012-08-14 | 110 |
| 3 | Electronics | 2012-08-13 | 150 |
| 3 | Electronics | 2012-08-14 | 170 |

After using the SQL GROUP BY statement with SUM, you will find the following result.

|  |  |
| --- | --- |
| **DEPARTMENT** | **SUM(DAILY SALES)** |
| Mechanical | 360 |
| Electrical | 210 |
| electronics | 320 |



# SQL SELECT NULL

First of all we should know that what null value is? Null values are used to represent missing unknown data.

There can be two conditions:

1. Where SQL is NULL
2. Where SQL is NOT NULL

If in a table, a column is optional, it is very easy to insert data in column or update an existing record without adding a value in this column. This means that field has null value.

#### **Note: we should not compare null value with 0. They are not equivalent.**

## **Where SQL is NULL:**

How to select records with null values only? (in the marks column)

There is an example of student table:

|  |  |  |
| --- | --- | --- |
| **SIR\_NAME** | **NAME** | **MARKS** |
| TYAGI | SEEMA |  |
| SINGH | RAMAN | 5.5 |
| SHARMA | AMAR |  |
| JAISWAL | VICKY | 6.2 |

Let's see the query to get all the records where marks is NULL:

1. **SELECT** SIR\_NAME, **NAME**, MARKS **FROM** STUDENTS
2. **WHERE** MARKS **IS** NULL

It will return the following records:

|  |  |  |
| --- | --- | --- |
| **SIR\_NAME** | **NAME** | **MARKS** |
| SHARMA | AMAR |  |
| TYAGI | SEEMA |  |

## **Where SQL is NOT NULL:**

How to select records with no null values(in marks column)? Let's see the query to get all the records where marks is NOT NULL

1. **SELECT** SIR\_NAME, FIRSTNAME, MARKS **FROM** STUDENTS
2. **WHERE** MARKS **IS** NOT NULL

|  |  |  |
| --- | --- | --- |
| **SIR\_NAME** | **NAME** | **MARKS** |
| SINGH | RAMAN | 5.5 |
| JAISWAL | VICKY | 6.2 |

# SQL WHERE

A **WHERE clause** in SQL is a data manipulation language statement.

WHERE clauses are not mandatory clauses of SQL DML statements. But it can be used to limit the number of rows affected by a SQL DML statement or returned by a query.

Actually. it filters the records. It returns only those queries which fulfill the specific conditions.

WHERE clause is used in SELECT, UPDATE, DELETE statement etc.

Let's see the syntax for sql where:

1. **SELECT** column1, **column** 2, ... **column** n
2. **FROM**    table\_name
3. **WHERE** [conditions]

WHERE clause uses some conditional selection

|  |  |
| --- | --- |
| = | equal |
| > | greater than |
| < | less than |
| >= | greater than or equal |
| <= | less than or equal |
| < > | not equal to |

# SQL AND

The SQL AND condition is used in SQL query to create two or more conditions to be met.

It is used in SQL SELECT, INSERT, UPDATE and DELETE statements.

Let's see the syntax for SQL AND:

1. **SELECT** columns
2. **FROM** tables
3. **WHERE** condition 1
4. AND condition 2;

The SQL AND condition requires that both conditions should be met.

The SQL AND condition also can be used to join multiple tables in a SQL statement.

## **SQL "AND" example with "INSERT" statement**

This is how an SQL "AND" condition can be used in the SQL INSERT statement.

For example:

1. **INSERT** **INTO** suppliers
2. (supplier\_id, supplier\_name)
3. **SELECT** account\_no, **name**
4. **FROM** customers
5. **WHERE** customer\_name ='IBM'
6. AND employees =1000;

## **SQL "AND" example with "UPDATE" statement**

This is how the "AND" condition can be used in the SQL UPDATE statement.

For example:

1. **UPDATE** suppliers
2. **SET** supplier\_name = 'HP'
3. **WHERE** supplier\_name = 'IBM'
4. AND offices = 8;

## **SQL "AND" example with "DELETE" statement**

This is how an SQL "AND" condition can be used in the SQL DELETE statement.

For example:

1. **DELETE** **FROM** suppliers
2. **WHERE** supplier\_name = 'IBM'
3. AND product = 'PC computers';

# SQL OR

The **SQL OR condition** is used in a SQL query to create a SQL statement where records are returned when any one of the condition met. It can be used in a SELECT statement, INSERT statement, UPDATE statement or DELETE statement.

Let's see the syntax for OR condition:

1. **SELECT** columns
2. **FROM** tables
3. **WHERE** condition 1
4. OR condition 2;

## **SQL "OR" example with SQL SELECT**

1. **SELECT** \*
2. **FROM** suppliers
3. **WHERE** city = 'New York'
4. OR available\_products >= 250;

## **SQL "OR" example with SQL INSERT**

You can see in below example that how an SQL "OR" condition is used with SQL insert statement.

For example:

1. **INSERT** **INTO** suppliers(supplier\_id, supplier\_name)
2. **SELECT** account\_no, **name**
3. **FROM** customers
4. **WHERE** city = 'New Delhi'
5. OR city = 'Ghaziabad';

## **SQL "OR" example with SQL UPDATE**

For example:

1. **UPDATE** suppliers
2. **SET** supplier\_name = 'HP'
3. **WHERE** supplier\_name = 'IBM'
4. OR available\_product >36;

## **SQL "OR" example with SQL DELETE**

For example:

1. **DELETE** **FROM** suppliers
2. **WHERE** supplier\_name = 'IBM'
3. OR employee <=100;

# SQL WITH CLAUSE

The SQL WITH clause is used to provide a sub-query block which can be referenced in several places within the main SQL query. It was introduced by oracle in oracle 9i release2 database.

There is an example of employee table:

**Syntax for the SQL WITH clause -**

This syntax is for SQL WITH clause using a single sub-query alias.

1. **WITH** <alias\_name> **AS** (sql\_sub-query\_statement)
2. **SELECT** column\_list **FROM** <alias\_name> [**table** **name**]
3. [**WHERE** <join\_condition>]

When you use multiple sub-query aliases, the syntax will be as follows.

1. **WITH** <alias\_name\_A>  **AS** (sql\_sub-query\_statement)
2. <alias\_name\_B> **AS** (sql\_sub-query\_statement\_from\_alias\_name\_A
3. Or sql\_sub-query\_statement)
4. **SELECT** <column\_list>
5. **FROM** <alias\_name\_A >,< alias\_name\_B >, [tablenames]
6. [**WHERE** < join\_condition>]

# SQL ORDER BY Clause

The SQL ORDER BY clause is used for sorting data in ascending and descending order based on one or more columns.

Some databases sort query results in ascending order by default.

SQL ORDER BY syntax:

**SELECT** expressions

**FROM** tables

**WHERE** conditions

**ORDER** **BY** expression [**ASC** | **DESC**];

Let us take a CUSTOMERS table having the following records:

|  |
| --- |
|  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Himani gupta | 21 | Modinagar | 22000 |
| 2 | Shiva tiwari | 22 | Bhopal | 21000 |
| 3 | Ajeet bhargav | 45 | Meerut | 65000 |
| 4 | Ritesh yadav | 36 | Azamgarh | 26000 |
| 5 | Balwant singh | 45 | Varanasi | 36000 |
| 6 | Mahesh sharma | 26 | Mathura | 22000 |

This is an example that would sort the result in ascending order by NAME and SALARY.

1. **SELECT** \* **FROM** CUSTOMERS
2. **ORDER** **BY** **NAME**, SALARY;

This would produce the following result.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 3 | Ajeet bhargav | 45 | Meerut | 65000 |
| 5 | Balwant singh | 45 | Varanasi | 36000 |
| 1 | Himani gupta | 21 | Modinagar | 22000 |
| 6 | Mahesh sharma | 26 | Mathura | 22000 |
| 4 | Ritesh yadav | 36 | Azamgarh | 26000 |
| 2 | Shiva tiwari | 22 | Bhopal | 21000 |

This is an example to sort the result in descending order by NAME.

1. **SELECT** \* **FROM** CUSTOMERS
2. **ORDER** **BY** **NAME** **DESC**;

This would produce the following result.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 2 | Shiva tiwari | 22 | Bhopal | 21000 |
| 4 | Ritesh yadav | 36 | Azamgarh | 26000 |
| 6 | Mahesh sharma | 26 | Mathura | 22000 |
| 1 | Himani gupta | 21 | Modinagar | 22000 |
| 5 | Balwant singh | 45 | Varanasi | 36000 |
| 3 | Ajeet bhargav | 45 | Meerut | 65000 |

# SQL ORDER BY CLAUSE WITH ASCENDING ORDER

This statement is used to sort data in ascending order. If you miss the ASC attribute, SQL ORDER BY query takes ascending order by default.

**Let's take an example of supplier**

**SELECT** supplier\_city

**FROM** suppliers

**WHERE** supplier\_name = 'IBM'

**ORDER** **BY** supplier\_city;

Let us take a CUSTOMERS table having the following records:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Himani gupta | 21 | Modinagar | 22000 |
| 2 | Shiva tiwari | 22 | Bhopal | 21000 |
| 3 | Ajeet bhargav | 45 | Meerut | 65000 |
| 4 | Ritesh yadav | 36 | Azamgarh | 26000 |
| 5 | Balwant singh | 45 | Varanasi | 36000 |
| 6 | Mahesh sharma | 26 | Mathura | 22000 |

This is an example to sort the result in ascending order by NAME and SALARY.

1. **SELECT** \* **FROM** CUSTOMERS
2. **ORDER** **BY** **NAME**, SALARY;

This would produce the following result.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 3 | Ajeet bhargav | 45 | Meerut | 65000 |
| 5 | Balwant singh | 45 | Varanasi | 36000 |
| 1 | Himani gupta | 21 | Modinagar | 22000 |
| 6 | Mahesh sharma | 26 | Mathura | 22000 |
| 4 | Ritesh yadav | 36 | Azamgarh | 26000 |
| 2 | Shiva tiwari | 22 | Bhopal | 21000 |

# SQL ORDER BY CLAUSE WITH DESCENDING ORDER:

This statement is used to sort data in descending order. You should use the DESC attribute in your ORDER BY clause as follows.

**SELECT** supplier\_city

**FROM** suppliers

**WHERE** supplier\_name = 'IBM'

**ORDER** **BY** supplier\_city **DESC**;

Let's see an example of an employee table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Himani gupta | 21 | Modinagar | 22000 |
| 2 | Shiva tiwari | 22 | Bhopal | 21000 |
| 3 | Ajeet bhargav | 45 | Meerut | 65000 |
| 4 | Ritesh yadav | 36 | Azamgarh | 26000 |
| 5 | Balwant singh | 45 | Varanasi | 36000 |
| 6 | Mahesh sharma | 26 | Mathura | 22000 |

This is an example to sort the result in descending order by NAME.

1. **SELECT** \* **FROM** CUSTOMERS
2. **ORDER** **BY** **NAME** **DESC**;

This would produce the following result.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 2 | Shiva tiwari | 22 | Bhopal | 21000 |
| 4 | Ritesh yadav | 36 | Azamgarh | 26000 |
| 6 | Mahesh sharma | 26 | Mathura | 22000 |
| 1 | Himani gupta | 21 | Modinagar | 22000 |
| 5 | Balwant singh | 45 | Varanasi | 36000 |
| 3 | Ajeet bhargav | 45 | Meerut | 65000 |

# SQL ORDER BY RANDOM

If you want the resulting record to be ordered randomly, you should use the following codes according to several databases.

Here a question occurs that what is the need of fetching a random record or a row from a database?

Sometimes you may want to display random information like articles, links, pages etc. to your user.

If you want to fetch random rows from any of the databases you have to use some queries which are altered according to the databases.

## **Select a random row with MySQL:**

If you want to return a random row with MY SQL, Use the following code:

1. **SELECT** **column** **FROM** **table**
2. **ORDER** **BY** RAND ()
3. LIMIT 1

## **Select a random row with Postgre SQL:**

1. **SELECT** **column** **FROM** **table**
2. **ORDER** **BY** RANDOM ()
3. LIMIT 1

## **Select a random row with SQL Server:**

1. **SELECT** **TOP** 1 **column** **FROM** **table**
2. **ORDER** **BY** NEWID ()

## **Select a random row with oracle:**

1. **SELECT** **column** **FROM**
2. (**SELECT** **column** **FROM** **table**
3. **ORDER** **BY** dbms\_random.value)
4. **WHERE** rownum = 1

## **Select a random row with IBM DB2:**

1. **SELECT** **column** RAND () **as** IDX
2. **FROM** **table**
3. **ORDER** **BY**  IDX **FETCH** **FIRST** 1 **ROWS** **ONLY**

# SQL ORDER BY LIMIT

We can retrieve limited rows from the database. I can be used in pagination where are forced to show only limited records like 10, 50, 100 etc.

#### **LIMIT CLAUSE FOR ORACLE SQL:**

If you want to use LIMIT clause with SQL, you have to use ROWNUM queries because it is used after result are selected.

You should use the following code:

1. **SELECT** **name**, age
2. **FROM**
3. (**SELECT** **name**, age, ROWNUM r
4. **FROM**
5. (**SELECT** **name**, age, **FROM** employee\_data
6. **ORDER** **BY** age **DESC**
7. )
8. **WHERE** ROWNUM <=40
9. )
10. **WHERE** r >= 21;

This query will give you 21th to 40th rows.

# SQL SORTING ON MULTIPLE COLUMNS

Let's take an example of customer table which has many columns, the following SQL statement selects all customers from the table named "customer", stored by the "country" and "Customer-Name" columns:

1. **SELECT** \* **FROM** customers
2. **ORDER** **BY** country, Customer-**Name**;

# SQL INSERT STATEMENT

SQL INSERT statement is a SQL query. It is used to insert a single or a multiple records in a table.

There are two ways to insert data in a table:

1. By SQL insert into statement
   1. By specifying column names
   2. Without specifying column names
2. By SQL insert into select statement

## **1) Inserting data directly into a table**

You can insert a row in the table by using SQL INSERT INTO command.

There are two ways to insert values in a table.

**In the first method there is no need to specify the column name where the data will be inserted, you need only their values.**

1. **INSERT** **INTO** table\_name
2. **VALUES** (value1, value2, value3....);

**The second method specifies both the column name and values which you want to insert.**

1. **INSERT** **INTO** table\_name (column1, column2, column3....)
2. **VALUES** (value1, value2, value3.....);

Let's take an example of table which has five records within it.

**INSERT** **INTO** STUDENTS (ROLL\_NO, **NAME**, AGE, CITY)

**VALUES** (1, ABHIRAM, 22, ALLAHABAD);

**INSERT** **INTO** STUDENTS (ROLL\_NO, **NAME**, AGE, CITY)

**VALUES** (2, ALKA, 20, GHAZIABAD);

**INSERT** **INTO** STUDENTS (ROLL\_NO, **NAME**, AGE, CITY)

**VALUES** (3, DISHA, 21, VARANASI);

**INSERT** **INTO** STUDENTS (ROLL\_NO, **NAME**, AGE, CITY)

**VALUES** (4, ESHA, 21, DELHI);

**INSERT** **INTO** STUDENTS (ROLL\_NO, **NAME**, AGE, CITY)

**VALUES** (5, MANMEET, 23, JALANDHAR);

It will show the following table as the final result.

|  |  |  |  |
| --- | --- | --- | --- |
| **ROLL\_NO** | **NAME** | **AGE** | **CITY** |
| 1 | ABHIRAM | 22 | ALLAHABAD |
| 2 | ALKA | 20 | GHAZIABAD |
| 3 | DISHA | 21 | VARANASI |
| 4 | ESHA | 21 | DELHI |
| 5 | MANMEET | 23 | JALANDHAR |

You can create a record in CUSTOMERS table by using this syntax also.

1. **INSERT** **INTO** CUSTOMERS
2. **VALUES** (6, PRATIK, 24, KANPUR);

The following table will be as follow:

|  |  |  |  |
| --- | --- | --- | --- |
| **ROLL\_NO** | **NAME** | **AGE** | **CITY** |
| 1 | ABHIRAM | 22 | ALLAHABAD |
| 2 | ALKA | 20 | GHAZIABAD |
| 3 | DISHA | 21 | VARANASI |
| 4 | ESHA | 21 | DELHI |
| 5 | MANMEET | 23 | JALANDHAR |
| 6 | PRATIK | 24 | KANPUR |

## **2) Inserting data through SELECT Statement**

**SQL INSERT INTO SELECT Syntax**

1. **INSERT** **INTO** table\_name
2. [(column1, column2, .... **column**)]
3. **SELECT** column1, column2, .... **Column** N
4. **FROM** table\_name [**WHERE** condition];

Note: when you add a new row, you should make sure that data type of the value and the column should be matched.

If any integrity constraints are defined for the table, you must follow them.

# SQL INSERT MULTIPLE ROWS

Many times, developers ask that is it possible to insert multiple rows into a single table in a single statement. Currently developers have to write multiple insert statement when they insert values in a table. It is not only boring, also time consuming. To get rid from this you should try this syntax. Actually, there are three different methods to insert multiple values into a single table.

1. Traditional method (simple insert)
2. SQL insert select
3. SQL server 2008+ Row Construction

Insert multiple values in SQL server

1. **CREATE** **TABLE** student (ID **INT** VALUE **VARCHAR** (100));

1. SQL INSERT: (TRADITIONAL INSERT)

1. **INSERT** **INTO** student (ID, **NAME**)
2. **VALUES** (1, 'ARMAAN');
3. **INSERT** **INTO** student (ID, **NAME**)
4. **VALUES** (2, 'BILLY');
5. **INSERT** **INTO** student (ID, **NAME**)
6. **VALUES** (3, 'CHARLIE');

TO CLEAN-UP:

1. **TRUNCATE** **TABLE** student;

2. INSERT SELECT: (SELECT UNION INSERT)

1. **INSERT** **INTO** student (ID, **NAME**)
2. **SELECT** 1, 'ARMAAN'
3. **UNION** ALL
4. **SELECT** 2, 'BILLY'
5. **UNION** ALL
6. **SELECT** 3, 'CHARLIE';
7. **TO** CLEAN-UP:
8. **TRUNCATE** **TABLE** student;

3.SQL Server 2008+ Row Construction

1. **INSERT** **INTO** student (ID, **NAME**)
2. **VALUES** (1, 'ARMAAN'), (2, 'BILLY'), (3, 'CHARLIE');
3. **TO** CLEAN-UP:
4. **DROP** **TABLE** student;

**Note:**Row Constructor is a new feature for SQL Server 2008. It is not supported by SQL Server 2005.

# SQL UPDATE

The SQL commands (*UPDATE* and *DELETE*) are used to modify the data that is already in the database. The SQL DELETE command uses a WHERE clause.

**SQL UPDATE** statement is used to change the data of the records held by tables. Which rows is to be update, it is decided by a condition. To specify condition, we use WHERE clause.

The UPDATE statement can be written in following form:

1. **UPDATE table\_name SET [column\_name1= value1,... column\_nameN = valueN] [WHERE condition]**

Let's see the Syntax:

1. **UPDATE** table\_name
2. **SET** column\_name = expression
3. **WHERE** conditions

Let's take an example: here we are going to update an entry in the source table.

SQL statement:

1. **UPDATE** students
2. **SET** User\_Name = 'beinghuman'
3. **WHERE** Student\_Id = '3'

**Source Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_Id** | **FirstName** | **LastName** | **User\_Name** |
| 1 | Ada | Sharma | sharmili |
| 2 | Rahul | Maurya | sofamous |
| 3 | James | Walker | jonny |

See the result after updating value:

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_Id** | **FirstName** | **LastName** | **User\_Name** |
| 1 | Ada | Sharma | sharmili |
| 2 | Rahul | Maurya | sofamous |
| 3 | James | Walker | **beinghuman** |

## **Updating Multiple Fields:**

If you are going to update multiple fields, you should separate each field assignment with a comma.

SQL UPDATE statement for multiple fields:

1. **UPDATE** students
2. **SET** User\_Name = 'beserious', First\_Name = 'Johnny'
3. **WHERE** Student\_Id = '3'

Result of the table is given below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_Id** | **FirstName** | **LastName** | **User\_Name** |
| 1 | Ada | Sharma | sharmili |
| 2 | Rahul | Maurya | sofamous |
| 3 | **Johnny** | Walker | **beserious** |

MYSQL SYNTAX FOR UPDATING TABLE:

1. **UPDATE** table\_name
2. **SET** field1 = new-value1, field2 = new-value2,
3. [**WHERE** CLAUSE]

**SQL UPDATE SELECT:**

**SQL UPDATE WITH SELECT QUERY:**

We can use SELECT statement to update records through UPDATE statement.

**SYNTAX:**

**UPDATE** tableDestination

**SET** tableDestination.col = value

**WHERE** EXISTS (

**SELECT** col2.value

**FROM**  tblSource

**WHERE** tblSource.join\_col = tblDestination. Join\_col

AND  tblSource.**Constraint** = value)

You can also try this one -

**UPDATE**

**Table**

**SET**

**Table**.column1 = othertable.**column** 1,

**Table**.column2 = othertable.**column** 2

**FROM**

**Table**

**INNER** JOIN

Other\_table

**ON**

**Table**.id = other\_table.id

**My SQL SYNTAX:**

If you want to UPDATE with SELECT in My SQL, you can use this syntax:

Let's take an example having two tables. Here,

First table contains -

Cat\_id, cat\_name,

And the second table contains -

Rel\_cat\_id, rel\_cat\_name

**SQL UPDATE COLUMN:**

We can update a single or multiple columns in SQL with SQL UPDATE query.

**SQL UPDATE EXAMPLE WITH UPDATING SINGLE COLUMN:**

1. **UPDATE** students
2. **SET** student\_id = 001
3. **WHERE** student\_name = 'AJEET';

This SQL UPDATE example would update the student\_id to '001' in the student table where student\_name is 'AJEET'.

**SQL UPDATE EXAMPLE WITH UPDATING MULTIPLE COLUMNS:**

To update more than one column with a single update statement:

1. **UPDATE** students
2. **SET** student\_name = 'AJEET',
3. Religion = 'HINDU'
4. **WHERE** student\_name = 'RAJU';

This SQL UPDATE statement will change the student name to 'AJEET' and religion to 'HINDU' where the student name is 'RAJU'.

# SQL UPDATE with JOIN

**SQL UPDATE JOIN** means we will update one table using another table and join condition.

Let us take an example of a customer table. I have updated customer table that contains latest customer details from another source system. I want to update the customer table with latest data. In such case, I will perform join between target table and source table using join on customer ID.

Let's see the *syntax* of SQL UPDATE query with JOIN statement.

**UPDATE** customer\_table

**INNER** JOIN

Customer\_table

**ON** customer\_table.rel\_cust\_name = customer\_table.cust\_id

**SET** customer\_table.rel\_cust\_name = customer\_table.cust\_name

### **How to use multiple tables in SQL UPDATE statement with JOIN**

Let's take two tables, table 1 and table 2.

**Create table1**

**CREATE** **TABLE** table1 (column1 **INT**, column2 **INT**, column3 **VARCHAR** (100))

**INSERT** **INTO** table1 (col1, col2, col3)

**SELECT** 1, 11, 'FIRST'

**UNION** ALL

**SELECT** 11,12, 'SECOND'

**UNION** ALL

**SELECT** 21, 13, 'THIRD'

**UNION** ALL

**SELECT** 31, 14, 'FOURTH'

**Create table2**

**CREATE** **TABLE** table2 (column1 **INT**, column2 **INT**, column3 **VARCHAR** (100))

**INSERT** **INTO** table2 (col1, col2, col3)

**SELECT** 1, 21, 'TWO-ONE'

**UNION** ALL

**SELECT** 11, 22, 'TWO-TWO'

**UNION** ALL

**SELECT** 21, 23, 'TWO-THREE'

**UNION** ALL

**SELECT** 31, 24, 'TWO-FOUR'

Now check the content in the table.

1. **SELECT** \* **FROM** table\_1
2. **SELECT** \* **FROM** table\_2

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Col 1** | **Col 2** | **Col 3** |
| 1 | 1 | 11 | First |
| 2 | 11 | 12 | Second |
| 3 | 21 | **13** | **Third** |
| 4 | 31 | **14** | **Fourth** |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Col 1** | **Col 2** | **Col 3** |
| 1 | 1 | 21 | Two-One |
| 2 | 11 | 22 | Two-Two |
| 3 | 21 | **23** | **Two-Three** |
| 4 | 31 | **24** | **Two-Four** |

Our requirement is that we have table 2 which has two rows where Col 1 is 21 and 31. We want to update the value from table 2 to table 1 for the rows where Col 1 is 21 and 31.

We want to also update the values of Col 2 and Col 3 only.

The most easiest and common way is to use join clause in the update statement and use multiple tables in the update statement.

**UPDATE** **table** 1

**SET** Col 2 = t2.Col2,

Col 3 = t2.Col3

**FROM** table1 t1

**INNER** JOIN **table** 2 t2 **ON** t1.Col1 = t2.col1

**WHERE** t1.Col1 IN (21,31)

Check the content of the table

SELECT FROM table 1

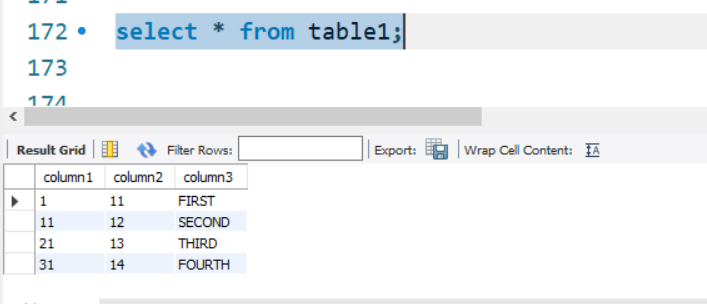
SELECT FROM table 2

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Col 1** | **Col 2** | **Col 3** |
| 1 | 1 | 11 | First |
| 2 | 11 | 12 | Second |
| 3 | 21 | **23** | **Two-Three** |
| 4 | 31 | **24** | **Two-Four** |

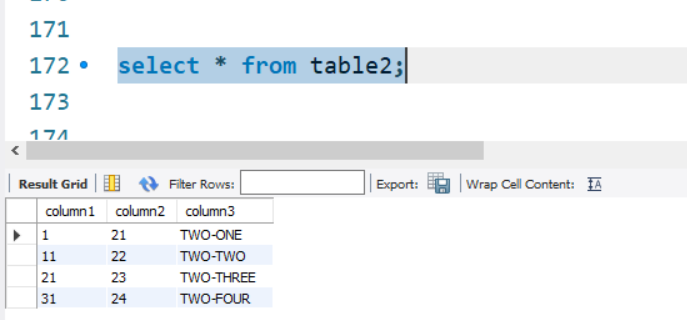
|  |  |  |  |
| --- | --- | --- | --- |
|  | **Col 1** | **Col 2** | **Col 3** |
| 1 | 1 | 21 | First |
| 2 | 11 | 22 | Second |
| 3 | 21 | **23** | **Two-Three** |
| 4 | 31 | **24** | **Two-Four** |

Here we can see that using join clause in update statement. We have merged two tables by the use of join clause.

This is table1



This is table 2



Now let’s perform the operations on the tables.

SQL UPDATE DATE

How to update a date and time field in SQL?

If you want to update a date & time field in SQL, you should use the following query.

let's see the syntax of sql update date.

**UPDATE** **table**

**SET** Column\_Name = 'YYYY-MM-DD HH:MM:SS'

**WHERE** Id = value

Let us check this by an example:

Firstly we take a table in which we want to update date and time fields.

If you want to change the first row which id is 1 then you should write the following syntax:

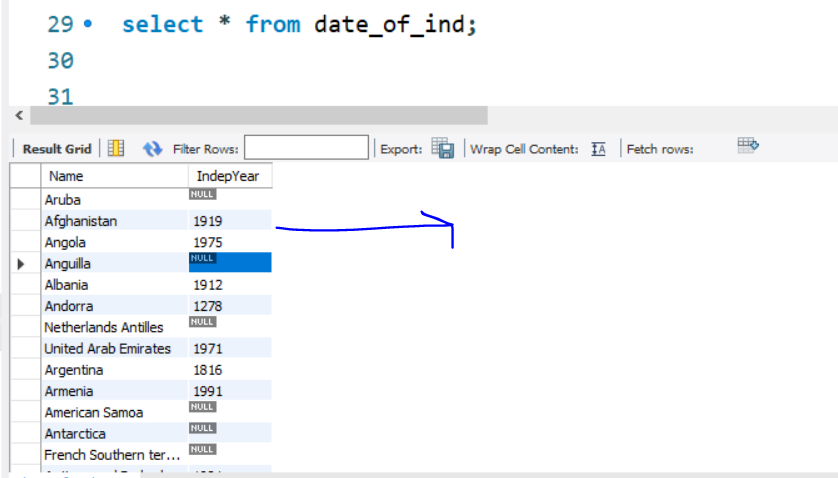
**UPDATE** **table**

**SET** EndDate = '2014-03-16 00:00:00.000'

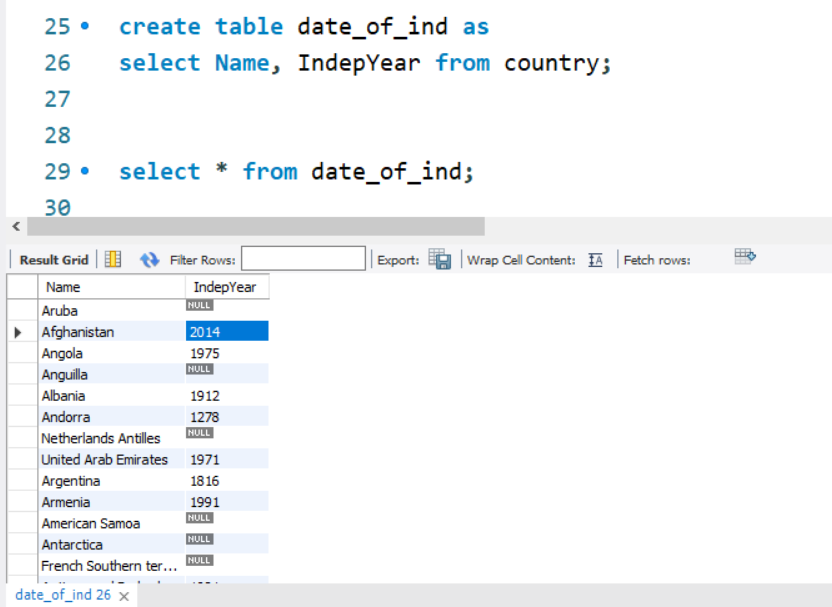
**WHERE** Id = 1

Note: you should always remember that SQL must attach default 00:00:00.000 automatically.

This query will change the date and time field of the first row in that above assumed table.



Let’s change the year for Afghanistan of independence



SQL DELETE

The **SQL DELETE statement** is used to delete rows from a table. Generally DELETE statement removes one or more records from a table.

SQL DELETE Syntax

Let's see the Syntax for the SQL DELETE statement:

1. **DELETE** **FROM** table\_name [**WHERE** condition];

Here table\_name is the table which has to be deleted. The *WHERE clause* in SQL DELETE statement is optional here.

SQL DELETE Example

Let us take a table, named "EMPLOYEE" table.

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **EMP\_NAME** | **CITY** | **SALARY** |
| 101 | Adarsh Singh | Obra | **20000** |
| 102 | Sanjay Singh | Meerut | **21000** |
| 103 | Priyanka Sharma | Raipur | **25000** |
| 104 | Esha Singhal | Delhi | **26000** |

Example of delete with WHERE clause is given below:

1. **DELETE** **FROM** EMPLOYEE **WHERE** ID=101;

Resulting table after the query:

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **EMP\_NAME** | **CITY** | **SALARY** |
| 102 | Sanjay Singh | Meerut | **21000** |
| 103 | Priyanka Sharma | Raipur | **25000** |
| 104 | Esha Singhal | Delhi | **26000** |

Another example of delete statement is given below

1. **DELETE** **FROM** EMPLOYEE;

Resulting table after the query:

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **EMP\_NAME** | **CITY** | **SALARY** |

It will delete all the records of EMPLOYEE table.

It will delete the all the records of EMPLOYEE table where ID is 101.

The WHERE clause in the SQL DELETE statement is optional and it identifies the rows in the column that gets deleted.

WHERE clause is used to prevent the deletion of all the rows in the table, If you don't use the WHERE clause you might loss all the rows.

Invalid DELETE Statement for ORACLE database

You cannot use \* (asterisk) symbol to delete all the records.

1. **DELETE** \* **FROM** EMPLOYEE;

# SQL DELETE TABLE

The DELETE statement is used to delete rows from a table. If you want to remove a specific row from a table you should use WHERE condition.

1. **DELETE** **FROM** table\_name [**WHERE** condition];

But if you do not specify the WHERE condition it will remove all the rows from the table.

1. **DELETE** **FROM** table\_name;

**There are some more terms similar to DELETE statement like as DROP statement and TRUNCATE statement but they are not exactly same there are some differences between them.**

## **Difference between DELETE and TRUNCATE statements**

There is a slight difference b/w delete and truncate statement. The **DELETE statement** only deletes the rows from the table based on the condition defined by WHERE clause or delete all the rows from the table when condition is not specified.

**But it does not free the space containing by the table.**

The **TRUNCATE statement:** it is used to delete all the rows from the table **and free the containing space.**

Let's see an "employee" table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Emp\_id** | **Name** | **Address** | **Salary** |
| 1 | Aryan | Allahabad | 22000 |
| 2 | Shurabhi | Varanasi | 13000 |
| 3 | Pappu | Delhi | 24000 |

Execute the following query to truncate the table:

1. **TRUNCATE** **TABLE** employee;

## **Difference b/w DROP and TRUNCATE statements**

When you use the drop statement it deletes the table's row together with the table's definition so all the relationships of that table with other tables will no longer be valid.

**When you drop a table:**

* Table structure will be dropped
* Relationship will be dropped
* Integrity constraints will be dropped
* Access privileges will also be dropped

On the other hand when we **TRUNCATE** a table, the table structure remains the same, so you will not face any of the above problems.

# SQL RENAME TABLE

**SQL RENAME TABLE** syntax is used to change the name of a table. Sometimes, we choose non-meaningful name for the table. So it is required to be changed.

Let's see the syntax to rename a table from the database.

1. **ALTER** **TABLE** table\_name
2. RENAME **TO** new\_table\_name;

Optionally, you can write following command to rename the table.

1. RENAME old\_table \_name **To** new\_table\_name;

Let us take an example of a table named "STUDENTS", now due to some reason we want to change it into table name "ARTISTS".

Table1: students

|  |  |  |
| --- | --- | --- |
| **Name** | **Age** | **City** |
| Amrita gill | 25 | Amritsar |
| Amrender sirohi | 22 | Ghaziabad |
| Divya khosla | 20 | Delhi |

You should use any one of the following syntax to RENAME the table name:

1. **ALTER** **TABLE** STUDENTS
2. RENAME **TO** ARTISTS;

Or

1. RENAME STUDENTS **TO** ARTISTS;
2. **After** that the **table** "students" will be changed **into** **table** **name** "artists"

# SQL TRUNCATE TABLE

A truncate SQL statement is used to remove all rows (complete data) from a table. It is similar to the DELETE statement with no WHERE clause.

#### **TRUNCATE TABLE Vs DELETE TABLE**

Truncate table is faster and uses lesser resources than DELETE TABLE command.

#### **TRUNCATE TABLE Vs DROP TABLE**

Drop table command can also be used to delete complete table but it deletes table structure too. TRUNCATE TABLE doesn't delete the structure of the table.

Let's see the syntax to truncate the table from the database.

1. **TRUNCATE** **TABLE** table\_name;

For example, you can write following command to truncate the data of employee table

1. **TRUNCATE** **TABLE** Employee;

**Note:** The rollback process is not possible after truncate table statement. Once you truncate a table you cannot use a flashback table statement to retrieve the content of the table.

# SQL COPY TABLE

If you want to copy a SQL table into another table in the same SQL server database, it is possible by using the select statement.

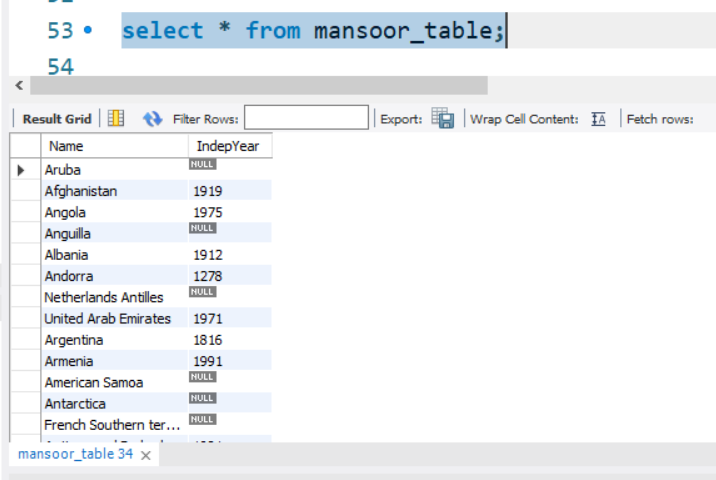
The syntax of copying table from one to another is given below:

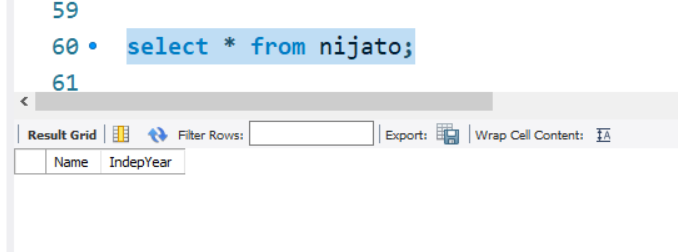
1. **SELECT** \* **INTO** <destination\_table> **FROM** <source\_table>

For example, you can write following command to copy the records of hr\_employee table into employee table.

1. **SELECT** \* **INTO** admin\_employee **FROM** hr\_employee;

#### **Note: SELECT INTO is totally different from INSERT INTO statement.**





# SQL TEMP TABLE

The concept of temporary table is introduced by SQL server. It helps developers in many ways:

**Temporary tables** can be created at run-time and can do all kinds of operations that a normal table can do. These temporary tables are created inside tempdb database.

**There are two types of temp tables based on the behavior and scope.**

1. **Local Temp Variable**
2. **Global Temp Variable**

## **Local Temp Variable**

Local temp tables are only available at current connection time. It is automatically deleted when user disconnects from instances. It is started with hash (#) sign.

**CREATE** **TABLE** #**local** **temp** **table** (

User id **int**,

Username **varchar** (50),

User address **varchar** (150)

)

## **Global Temp Variable**

Global temp tables name starts with double hash (##). Once this table is created, it is like a permanent table. It is always ready for all users and not deleted until the total connection is withdrawn.

**CREATE** **TABLE** ##new **global** **temp** **table** (

User id **int**,

User **name** **varchar** (50),

User address **varchar** (150)

)

# SQL ALTER TABLE

The ALTER TABLE statement is used to add, modify or delete columns in an existing table. It is also used to rename a table.

You can also use SQL ALTER TABLE command to add and drop various constraints on an existing table.

### **SQL ALTER TABLE Add Column**

If you want to add columns in SQL table, the SQL alter table syntax is given below:

1. **ALTER** **TABLE** table\_name **ADD** column\_name **column**-definition;

If you want to add multiple columns in table, the SQL table will be

1. **ALTER** **TABLE** table\_name
2. **ADD** (column\_1 **column**-definition,
3. column\_2 **column**-definition,
4. .....
5. column\_n **column**-definition);

### **SQL ALTER TABLE Modify Column**

If you want to modify an existing column in SQL table, syntax is given below:

1. **ALTER** **TABLE** table\_name **MODIFY** column\_name column\_type;

If you want to modify multiple columns in table, the SQL table will be

1. **ALTER** **TABLE** table\_name
2. **MODIFY** (column\_1 column\_type,
3. column\_2 column\_type,
4. .....
5. column\_n column\_type);

### **SQL ALTER TABLE DROP Column**

The syntax of alter table drop column is given below:

1. **ALTER** **TABLE** table\_name **DROP** **COLUMN** column\_name;

### **SQL ALTER TABLE RENAME Column**

The syntax of alter table rename column is given below:

1. **ALTER** **TABLE** table\_name
2. RENAME **COLUMN** old\_name **to** new\_name;

# SQL SELECT

**The most commonly used SQL command is SELECT statement. It is used to query the database and retrieve selected data that follow the conditions we want.**

In simple words, we can say that the select statement used to query or retrieve data from a table in the database.

Let's see the syntax of select statement.

1. **SELECT** expressions
2. **FROM** tables
3. **WHERE** conditions;

Here expression is the column that we want to retrieve.

Tables indicate the tables, we want to retrieve records from.

## **Optional clauses in SELECT statement**

There are some optional clauses in SELECT statement:

**[WHERE Clause]** : It specifies which rows to retrieve.

**[GROUP BY Clause]** : Groups rows that share a property so that the aggregate function can be applied to each group.

**[HAVING Clause]** : It selects among the groups defined by the GROUP BY clause.

**[ORDER BY Clause]** : It specifies an order in which to return the rows.

For example, let a database table: student\_details;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **First\_name** | **Last\_name** | **Age** | **Subject** | **Hobby** |
| 1 | Amar | Sharma | 20 | Maths | Cricket |
| 2 | Akbar | Khan | 22 | Biology | Football |
| 3 | Anthony | Milton | 25 | Commerce | Gambling |

From the above example, select the first name of all the students. To do so, query should be like this:

1. **SELECT** first\_name **FROM** student\_details;

Note: the SQL commands are not case sensitive. We can also write the above SELECT statement as:

1. **select** first\_name **from** student\_details;

Now, you will get following data:

|  |
| --- |
| Amar |
| Akbar |
| Anthony |

We can also retrieve data from more than one column. For example, to select first name and last name of all the students, you need to write

1. **SELECT** first\_name, last\_name **FROM** student\_details;

Now, you will get following data:

|  |  |
| --- | --- |
| Amar | Sharma |
| Akbar | Khan |
| Anthony | Milton |

We can also use clauses like WHERE, GROUP BY, HAVING, ORDER BY with SELECT statement.

Here a point is notable that only SELECT and FROM statements are necessary in SQL SELECT statements. Other clauses like WHERE, GROUP BY, ORDER BY, HAVING may be optional.

# SQL SELECT UNIQUE

Actually, there is no difference between DISTINCT and UNIQUE.

**SELECT UNIQUE** is an old syntax which was used in oracle description but later ANSI standard defines DISTINCT as the official keyword.

After that oracle also added DISTINCT but did not withdraw the service of UNIQUE keyword for the sake of backward compatibility.

In simple words, we can say that SELECT UNIQUE statement is used to retrieve a unique or distinct element from the table.

Let's see the syntax of select unique statement.

1. **SELECT** **UNIQUE** column\_name
2. **FROM** table\_name;

SQL SELECT DISTINCT statement can also be used for the same cause.

# SQL SELECT DISTINCT

The **SQL DISTINCT command** is used with SELECT key word to retrieve only distinct or unique data.

In a table, there may be a chance to exist a duplicate value and sometimes we want to retrieve only unique values. In such scenarios, SQL SELECT DISTINCT statement is used.

#### **Note: SQL SELECT UNIQUE and SQL SELECT DISTINCT statements are same.**

Let's see the syntax of select distinct statement.

1. **SELECT** **DISTINCT** column\_name ,column\_name
2. **FROM**  table\_name;

Let's try to understand it by the table given below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_Name** | **Gender** | **Mobile\_Number** | **HOME\_TOWN** |
| Rahul Ojha | Male | 7503896532 | Lucknow |
| Disha Rai | Female | 9270568893 | Varanasi |
| Sonoo Jaiswal | Male | 9990449935 | Lucknow |

Here is a table of students from where we want to retrieve distinct information For example: distinct home-town.

1. **SELECT** **DISTINCT** home\_town
2. **FROM** students

Now, it will return two rows.

|  |
| --- |
| **HOME\_TOWN** |
| Lucknow |
| Varanasi |

# SQL SELECT COUNT

The **SQL COUNT()** function is used to return the number of rows in a query.

The COUNT() function is used with SQL SELECT statement and it is very useful to count the number of rows in a table having enormous data.

**For example:** If you have a record of the voters in selected area and want to count the number of voters then it is very difficult to do it manually but you can do it easily by using the SQL SELECT COUNT query.

Let's see the syntax of SQL COUNT statement.

**SELECT** COUNT (expression)

**FROM** tables

**WHERE** conditions;

Let's see the examples of sql select count function.

#### **SQL SELECT COUNT(column\_name)**

1. **SELECT** COUNT(**name**) **FROM** employee\_table;

It will return the total number of names of employee\_table. But null fields will not be counted.

#### **SQL SELECT COUNT(\*)**

1. **SELECT** COUNT(\*) **FROM** employee\_table;

The "select count(\*) from table" is used to return the number of records in table.

#### **SQL SELECT COUNT(DISTINCT column\_name)**

1. **SELECT** COUNT(**DISTINCT** **name**) **FROM** employee\_table;

It will return the total distinct names of employee\_table.

# SQL SELECT AS

**SQL AS** is used to assign temporarily a new name to a table column.

It makes easy presentation of query results and allows the developer to label results more accurately without permanently renaming table columns.

Let's see the example of select as:

**SELECT** day\_of\_order **AS** "Date"

Customer **As** "Client",

Product,

Quantity,

**FROM** orders;

Let us take a table named orders, it contains:

|  |  |  |  |
| --- | --- | --- | --- |
| **Day\_of\_order** | **Customer** | **Product** | **Quantity** |
| 11-09-2001 | Ajeet | Mobile | 2 |
| 13-12-2001 | Mayank | Laptop | 20 |
| 26-12-2004 | Balaswamy | Water cannon | 35 |

After applying this SQL AS example syntax

1. **SELECT** day\_of\_order **AS** "Date"
2. Customer **As** "Client",
3. Product,
4. Quantity,
5. **FROM** orders;

Result will be shown as this table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Client** | **Product** | **Quantity** |
| 11-09-2001 | Ajeet | Mobile | 2 |
| 13-12-2001 | Mayank | Laptop | 20 |
| 26-12-2004 | Balaswamy | Water cannon | 35 |

#### **Note: SQL AS is same as SQL ALIAS.**

# SQL DELETE DUPLICATE ROWS

If you have got a situation that you have multiple duplicate records in a table, so at the time of fetching records from the table you should be more careful. You make sure that you are fetching unique records instead of fetching duplicate records.

To overcome with this problem we use DISTINCT keyword.

It is used along with SELECT statement to eliminate all duplicate records and fetching only unique records.

**SYNTAX:**

The basic syntax to eliminate duplicate records from a table is:

**SELECT** **DISTINCT** column1, column2,....columnN

**FROM** **table** \_name

**WHERE** [conditions]

**EXAMPLE:**

Let us take an example of STUDENT table.

|  |  |  |  |
| --- | --- | --- | --- |
| **ROLL\_NO** | **NAME** | **PERCENTAGE** | **ADDRESS** |
| 1 | AJEET MAURYA | 72.8 | ALLAHABAD |
| 2 | CHANDAN SHARMA | 63.5 | MATHURA |
| 3 | DIVYA AGRAWAL | 72.3 | VARANASI |
| 4 | RAJAT KUMAR | 72.3 | DELHI |
| 5 | RAVI TYAGI | 75.5 | HAPUR |
| 6 | SONU JAISWAL | 71.2 | GHAZIABAD |

Firstly we should check the SELECT query and see how it returns the duplicate percentage records.

**SELECT** PERCENTAGE **FROM** STUDENTS

**ORDER** **BY** PERCENTAGE;

|  |
| --- |
| **PERCENTAGE** |
| 63.5 |
| 71.2 |
| 72.3 |
| 72.3 |
| 72.8 |
| 75.5 |

Now let us use SELECT query with DISTINCT keyword and see the result. This will eliminate the duplicate entry.

**SELECT** **DISTINCT** PERCENTAGE **FROM** STUDENTS

**ORDER** **BY** PERCENTAGE;

|  |
| --- |
| **PERCENTAGE** |
| 63.5 |
| 71.2 |
| 72.3 |
| 72.8 |
| 75.5 |

# SQL DELETE DATABASE

You can easily remove or delete indexes, tables and databases with the DROP statement.

**The DROP index statement is:**

Used to delete index in the table

**DROP INDEX SYNTAX for MS Access:**

1. **DROP** **INDEX** index\_name **ON** table\_name

**DROP INDEX SYNTAX for MS SQL Server:**

1. **DROP** **INDEX** table\_name.index\_name

**DROP INDEX syntax for DB2/Oracle:**

1. **DROP** **INDEX** index\_name

**DROP INDEX syntax for MySQL:**

1. **ALTER** **TABLE** table\_name **DROP** **INDEX** index\_name

**DROP DATABASE Statement:**

The drop database statement is used to delete a database.

1. **DROP** **DATABASE** database\_name

**Note:**

We should always note that in RDBMS, database name should be unique.

We should always remember that DROP database command may be the cause of loss of all information so we should always be careful while doing this operation.

# SQL DELETE VIEW

**Before knowing about what is SQL delete view, it is important to know -**

**What is SQL view?**

A view is a result set of a stored query on the data.

The SQL view is a table which does not physically exist. It is only a virtual table.

SQL VIEW can be created by a SQL query by joining one or more table.

**Syntax for SQL create view -**

**CREATE** **VIEW** view\_name **AS**

**SELECT** columns

**FROM** tables

**WHERE** conditions;

If you want to delete a SQL view, It is done by SQL DROP command you should use the following syntax:

**SQL DROP VIEW syntax:**

1. **DROP** **VIEW** view\_name

**Why use the SQL DROP VIEW statement?**

When a view no longer useful you may drop the view permanently. Also if a view needs change within it, it would be dropped and then created again with changes in appropriate places.

# SQL DELETE JOIN

This is very commonly asked question that how to delete or update rows using join clause

It is not a very easy process, sometimes, we need to update or delete records on the basis of complex WHERE clauses.

There are three tables which we use to operate on SQL syntax for DELETE JOIN.

These tables are table1, table2 and target table.

**SQL Syntax for delete JOIN**

**DELETE** [target **table**]

**FROM**    [table1]

**INNER** JOIN [table2]

**ON** [table1.[joining **column**] = [table2].[joining **column**]

**WHERE**   [condition]

**Syntax for update**

1. **UPDATE** [target **table**]
2. **SET** [target **column**] = [new value]
3. **FROM**    [table1]
4. **INNER** JOIN [table2]
5. **ON** [table1.[joining **column**] = [table2].[joining **column**]
6. **WHERE**   [condition]

# SQL JOIN

As the name shows, JOIN means *to combine something*. In case of SQL, JOIN means **"to combine two or more tables"**.

The SQL JOIN clause takes records from two or more tables in a database and combines it together.

**ANSI standard SQL** defines five types of JOIN:

1. inner join,
2. left outer join,
3. right outer join,
4. full outer join, and
5. cross join.

In the process of joining, rows of both tables are combined in a single table.

#### **Why SQL JOIN is used?**

If you want to access more than one table through a select statement.

If you want to combine two or more table then SQL JOIN statement is used .it combines rows of that tables in one table and one can retrieve the information by a SELECT statement.

**The joining of two or more tables is based on common field between them.**

**SQL INNER JOIN also known as simple join is the most common type of join.**

## **How to use SQL join or SQL Inner Join?**

Let an example to deploy SQL JOIN process:

1.Staff table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Staff\_NAME** | **Staff\_AGE** | **STAFF\_ADDRESS** | **Monthley\_Package** |
| 1 | ARYAN | 22 | MUMBAI | 18000 |
| 2 | SUSHIL | 32 | DELHI | 20000 |
| 3 | MONTY | 25 | MOHALI | 22000 |
| 4 | AMIT | 20 | ALLAHABAD | 12000 |

2.Payment table

|  |  |  |  |
| --- | --- | --- | --- |
| **Payment\_ID** | **DATE** | **Staff\_ID** | **AMOUNT** |
| 101 | 30/12/2009 | 1 | 3000.00 |
| 102 | 22/02/2010 | 3 | 2500.00 |
| 103 | 23/02/2010 | 4 | 3500.00 |

So, if you follow this JOIN statement to join these two tables?

**SELECT** Staff\_ID, Staff\_NAME, Staff\_AGE, AMOUNT

**FROM** STAFF s, PAYMENT p

**WHERE** s.ID =p.STAFF\_ID;

This will produce the result like this:

|  |  |  |  |
| --- | --- | --- | --- |
| **STAFF\_ID** | **NAME** | **Staff\_AGE** | **AMOUNT** |
| 3 | MONTY | 25 | 2500 |
| 1 | ARYAN | 22 | 3000 |
| 4 | AMIT | 25 | 3500 |
| 1 | ARYAN | 22 | 3000 |

# SQL OUTER JOIN

In the SQL outer JOIN all the content of the both tables are integrated together either they are matched or not.

If you take an example of employee table

**Outer join of two types:**

**1.Left outer join** (also known as left join): this join returns all the rows from left table combine with the matching rows of the right table. If you get no matching in the right table it returns NULL values.

**2.Right outer join** (also known as right join): this join returns all the rows from right table are combined with the matching rows of left table .If you get no column matching in the left table .it returns null value.

This diagram shows the different type of joins:

# SQL LEFT JOIN

The SQL left join returns all the values from the left table and it also includes matching values from right table, if there are no matching join value it returns NULL.

**BASIC SYNTAX FOR LEFT JOIN:**

**SELECT** table1.column1, table2.column2....

**FROM** table1

LEFTJOIN table2

**ON** table1.column\_field = table2.column\_field;

let us take two tables in this example to elaborate all the things:

**CUSTOMER TABLE:**

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **SALARY** |
| 1 | ARYAN | 51 | 56000 |
| 2 | AROHI | 21 | 25000 |
| 3 | VINEET | 24 | 31000 |
| 4 | AJEET | 23 | 32000 |
| 5 | RAVI | 23 | 42000 |

This is second table

**ORDER TABLE:**

|  |  |  |  |
| --- | --- | --- | --- |
| **O\_ID** | **DATE** | **CUSTOMER\_ID** | **AMOUNT** |
| 001 | 20-01-2012 | 2 | 3000 |
| 002 | 12-02-2012 | 2 | 2000 |
| 003 | 22-03-2012 | 3 | 4000 |
| 004 | 11-04-2012 | 4 | 5000 |

**join these two tables with LEFT JOIN:**

**SELECT** ID, **NAME**, AMOUNT,**DATE**

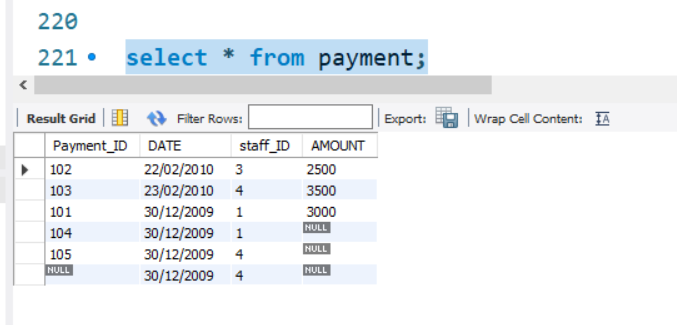
**FROM** CUSTOMER

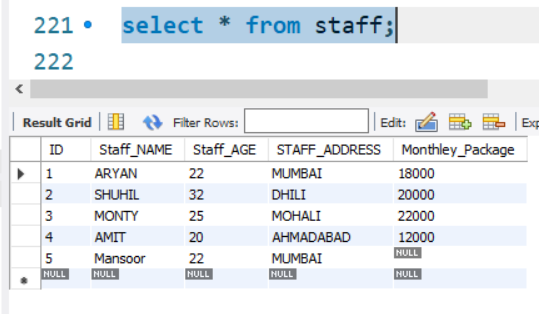
LEFT JOIN **ORDER**

**ON** CUSTOMER.ID = **ORDER**.CUSTOMER\_ID;

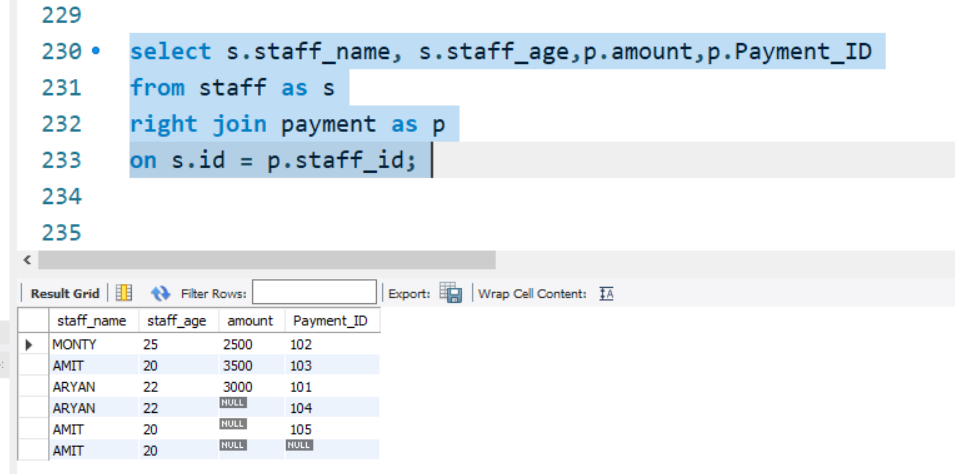
This will produce the following result:

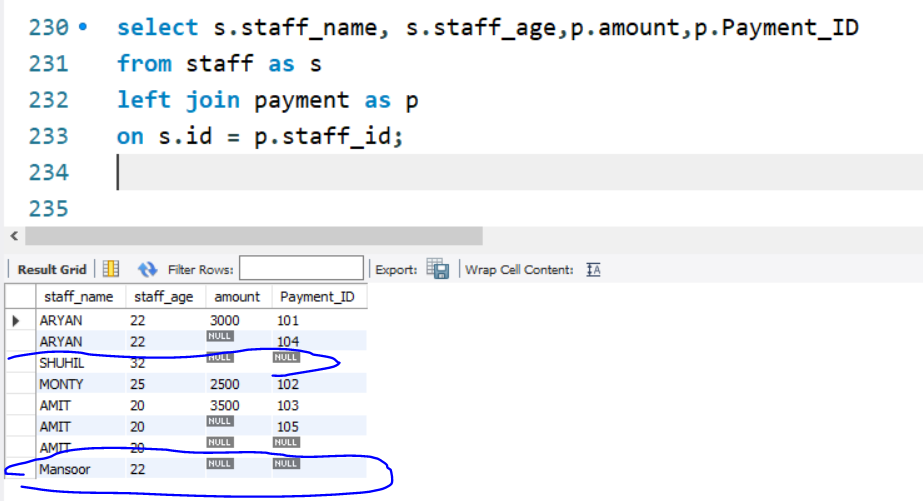
|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **AMOUNT** | **DATE** |
| 1 | ARYAN | NULL | NULL |
| 2 | AROHI | 3000 | 20-01-2012 |
| 2 | AROHI | 2000 | 12-02-2012 |
| 3 | VINEET | 4000 | 22-03-2012 |
| 4 | AJEET | 5000 | 11-04-2012 |
| 5 | RAVI | NULL | NULL |





This is right join it will return all the right table data and all the match data from the left.





# SQL RIGHT JOIN

The SQL right join returns all the values from the rows of right table. It also includes the matched values from left table but if there is no matching in both tables, it returns NULL.

Basic syntax for right join:

**SELECT** table1.column1, table2.column2.....

**FROM** table1

RIGHT JOIN table2

**ON** table1.column\_field = table2.column\_field;

let us take an example with 2 tables table1 is CUSTOMERS table and table2 is ORDERS table.

**CUSTOMER TABLE:**

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **SALARY** |
| 1 | ARYAN | 51 | 56000 |
| 2 | AROHI | 21 | 25000 |
| 3 | VINEET | 24 | 31000 |
| 4 | AJEET | 23 | 32000 |
| 5 | RAVI | 23 | 42000 |

and this is the second table:

**ORDER TABLE:**

|  |  |  |  |
| --- | --- | --- | --- |
| **DATE** | **O\_ID** | **CUSTOMER\_ID** | **AMOUNT** |
| 20-01-2012 | 001 | 2 | 3000 |
| 12-02-2012 | 002 | 2 | 2000 |
| 22-03-2012 | 003 | 3 | 4000 |
| 11-04-2012 | 004 | 4 | 5000 |

Here we will join these two tables with SQL RIGHT JOIN:

**SELECT** ID,**NAME**,AMOUNT,**DATE**

**FROM** CUSTOMER

RIGHT JOIN **ORDER**

**ON** CUSTOMER.ID = **ORDER**.CUSTOMER\_ID;

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **AMOUNT** | **DATE** |
| 2 | AROHI | 3000 | 20-01-2012 |
| 2 | AROHI | 2000 | 12-02-2012 |
| 3 | VINEET | 4000 | 22-03-2012 |
| 4 | AJEET | 5000 | 11-04-2012 |

# SQL FULL JOIN

The SQL full join is the result of combination of both left and right outer join and the join tables have all the records from both tables. It puts NULL on the place of matches not found.

**SQL full outer join and SQL join are same. generally it is known as SQL FULL JOIN.**

## **SQL full outer join:**

What is SQL full outer join?

SQL full outer join is used to combine the result of both left and right outer join and returns all rows (don't care its matched or unmatched) from the both participating tables.

**Syntax for full outer join:**

**SELECT** \*

**FROM** table1

**FULL** OUTER JOIN table2

**ON** table1.column\_name = table2.column\_name;

**Note:**here table1 and table2 are the name of the tables participating in joining and column\_name is the column of the participating tables.

Let us take two tables to demonstrate full outer join:

**table\_A**

|  |  |
| --- | --- |
| **A** | **M** |
| 1 | m |
| 2 | n |
| 4 | o |

**table\_B**

|  |  |
| --- | --- |
| **A** | **N** |
| 2 | p |
| 3 | q |
| 5 | r |

**Resulting table**

|  |  |  |  |
| --- | --- | --- | --- |
| **A** | **M** | **A** | **N** |
| 2 | n | 2 | p |
| 1 | m | - | - |
| 4 | o | - | - |
| - | - | 3 | q |
| - | - | 5 | r |

Because this is a full outer join so all rows (both matching and non-matching) from both tables are included in the output. Here only one row of output displays values in all columns because there is only one match between table\_A and table\_B.

Pictorial representation of full outer join:

# SQL Cross Join

When each row of first table is combined with each row from the second table, known as Cartesian join or cross join. In general words we can say that SQL CROSS JOIN returns the Cartesian product of the sets of rows from the joined table.

**We can specify a CROSS JOIN in two ways:**

1. Using the JOIN syntax.
2. the table in the FROM clause without using a WHERE clause.

**SYNTAX of SQL Cross Join**

1. **SELECT** \* **FROM** [TABLE1] CROSS JOIN [TABLE2]
2. OR
3. **SELECT** \* **FROM** [ TABLE1] , [TABLE2]

Let us take an example of two tables,

**Table1 - MatchScore**

|  |  |  |
| --- | --- | --- |
| **Player** | **Department\_id** | **Goals** |
| Franklin | 1 | 2 |
| Alan | 1 | 3 |
| Priyanka | 2 | 2 |
| Rajesh | 3 | 5 |

**Table2 - Departments**

|  |  |
| --- | --- |
| **Department\_id** | **Department\_name** |
| 1 | IT |
| 2 | HR |
| 3 | Marketing |

**SQL Statement:**

1. **SELECT** \* **FROM** MatchScore CROSS JOIN Departments

**After executing this query , you will find the following result:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Player** | **Department\_id** | **Goals** | **Depatment\_id** | **Department\_name** |
| Franklin | 1 | 2 | 1 | IT |
| Alan | 1 | 3 | 1 | IT |
| Priyanka | 2 | 2 | 1 | IT |
| Rajesh | 3 | 5 | 1 | IT |
| Franklin | 1 | 2 | 2 | HR |
| Alan | 1 | 3 | 2 | HR |
| Priyanka | 2 | 2 | 2 | HR |
| Rajesh | 3 | 5 | 2 | HR |
| Franklin | 1 | 2 | 3 | Marketing |
| Alan | 1 | 3 | 3 | Marketing |
| Priyanka | 2 | 2 | 3 | Marketing |
| Rajesh | 3 | 5 | 3 | Marketing |

# SQL PRIMARY KEY

A column or columns is called **primary key (PK)** that *uniquely identifies each row in the table*.

If you want to create a primary key, you should define a PRIMARY KEY constraint when you create or modify a table.

When multiple columns are used as a primary key, it is known as **composite primary key**.

In designing the composite primary key, you should use as few columns as possible. It is good for storage and performance both, the more columns you use for primary key the more storage space you require.

Inn terms of performance, less data means the database can process faster.

#### **Points to remember for primary key:**

* Primary key enforces the entity integrity of the table.
* Primary key always has unique data.
* A primary key length cannot be exceeded than 900 bytes.
* A primary key cannot have null value.
* There can be no duplicate value for a primary key.
* A table can contain only one primary key constraint.

When we specify a primary key constraint for a table, database engine automatically creates a unique index for the primary key column.

#### **Main advantage of primary key:**

The main advantage of this uniqueness is that we get **fast access**.

In oracle, it is not allowed for a primary key to contain more than 32 columns.

## **SQL primary key for one column:**

The following SQL command creates a PRIMARY KEY on the "S\_Id" column when the "students" table is created.

**MySQL:**

**CREATE** **TABLE** students

(

S\_Id **int** NOT NULL,

LastName **varchar** (255) NOT NULL,

FirstName **varchar** (255),

Address **varchar** (255),

City **varchar** (255),

**PRIMARY** **KEY** (S\_Id)

)

**SQL Server, Oracle, MS Access:**

1. **CREATE** **TABLE** students
2. (
3. S\_Id **int** NOT NULL **PRIMARY** **KEY**,
4. LastName **varchar** (255) NOT NULL,
5. FirstName **varchar** (255),
6. Address **varchar** (255),
7. City **varchar** (255),
8. )

## **SQL primary key for multiple columns:**

**MySQL, SQL Server, Oracle, MS Access:**

**CREATE** **TABLE** students

(

S\_Id **int** NOT NULL,

LastName **varchar** (255) NOT NULL,

FirstName **varchar** (255),

Address **varchar** (255),

City **varchar** (255),

**CONSTRAINT** pk\_StudentID **PRIMARY** **KEY** (S\_Id, LastName)

)

**Note:** you should note that in the above example there is only one PRIMARY KEY (pk\_StudentID). However it is made up of two columns (S\_Id and LastName).

## **SQL primary key on ALTER TABLE**

When table is already created and you want to create a PRIMARY KEY constraint on the "S\_Id" column you should use the following SQL:

**Primary key on one column:**

1. **ALTER** **TABLE** students
2. **ADD** **PRIMARY** **KEY** (S\_Id)

**Primary key on multiple column:**

1. **ALTER** **TABLE** students
2. **ADD** **CONSTRAINT** pk\_StudentID **PRIMARY** **KEY** (S\_Id,LastName)

When you use ALTER TABLE statement to add a primary key, the primary key columns must not contain NULL values (when the table was first created).

## **How to DROP a PRIMARY KEY constraint?**

If you want to DROP (remove) a primary key constraint, you should use following syntax:

**MySQL:**

1. **ALTER** **TABLE** students
2. **DROP** **PRIMARY** **KEY**

**SQL Server / Oracle / MS Access:**

1. **ALTER** **TABLE** students
2. **DROP** **CONSTRAINT** pk\_StudentID

# SQL FOREIGN KEY

In the relational databases, a foreign key is a field or a column that is used to establish a link between two tables.

In simple words you can say that, a foreign key in one table used to point primary key in another table.

Let us take an example to explain it:

Here are two tables first one is students table and second is orders table.

Here orders are given by students.

**First table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **S\_Id** | **LastName** | **FirstName** | **CITY** |
| 1 | MAURYA | AJEET | ALLAHABAD |
| 2 | JAISWAL | RATAN | GHAZIABAD |
| 3 | ARORA | SAUMYA | MODINAGAR |

**Second table:**

|  |  |  |
| --- | --- | --- |
| **O\_Id** | **OrderNo** | **S\_Id** |
| 1 | 99586465 | 2 |
| 2 | 78466588 | 2 |
| 3 | 22354846 | 3 |
| 4 | 57698656 | 1 |

Here you see that "S\_Id" column in the "Orders" table points to the "S\_Id" column in "Students" table.

* The "S\_Id" column in the "Students" table is the PRIMARY KEY in the "Students" table.
* The "S\_Id" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.

The foreign key constraint is generally prevents action that destroy links between tables.

It also prevents invalid data to enter in foreign key column.

## **SQL FOREIGN KEY constraint ON CREATE TABLE:**

(Defining a foreign key constraint on single column)

To create a foreign key on the "S\_Id" column when the "Orders" table is created:

**MySQL:**

**CREATE** **TABLE** orders

(

O\_Id **int** NOT NULL,

Order\_No  **int** NOT NULL,

S\_Id **int**,

PRIMAY **KEY** (O\_Id),

**FOREIGN** **KEY** (S\_Id) **REFERENCES** Persons (S\_Id)

)

**SQL Server /Oracle / MS Access:**

1. **CREATE** **TABLE** Orders
2. (
3. O\_Id **int** NOT NULL PRIMAY **KEY**,
4. Order\_No **int** NOT NULL,
5. S\_Id **int** **FOREIGN** **KEY** **REFERENCES** persons (S\_Id)
6. )

## **SQL FOREIGN KEY constraint for ALTER TABLE:**

If the Order table is already created and you want to create a FOREIGN KEY constraint on the "S\_Id" column, you should write the following syntax:

**Defining a foreign key constraint on single column:**

**MySQL / SQL Server / Oracle / MS Access:**

**ALTER** **TABLE** Orders

**ADD** **CONSTRAINT** fk\_PerOrders

**FOREIGN** **KEY**(S\_Id)

**REFERENCES** Students (S\_Id)

## **DROP SYNTAX for FOREIGN KEY COSTRAINT:**

If you want to drop a FOREIGN KEY constraint, use the following syntax:

**MySQL:**

1. **ALTER** **TABLE** Orders
2. DROP **FOREIGN** **KEY** fk\_PerOrders

**SQL Server / Oracle / MS Access:**

1. **ALTER** **TABLE** Orders
2. **DROP** **CONSTRAINT** fk\_PerOrders

## **Difference between primary key and foreign key in SQL:**

These are some important difference between primary key and foreign key in SQL-

Primary key cannot be null on the other hand foreign key can be null.

Primary key is always unique while foreign key can be duplicated.

Primary key uniquely identifies a record in a table while foreign key is a field in a table that is primary key in another table.

There is only one primary key in the table on the other hand we can have more than one foreign key in the table.

By default primary key adds a clustered index on the other hand foreign key does not automatically create an index, clustered or non-clustered. You must manually create an index for foreign key.

# SQL Composite Key

A composite key is a combination of two or more columns in a table that can be used to uniquely identify each row in the table when the columns are combined uniqueness is guaranteed, but when it taken individually it does not guarantee uniqueness.

Sometimes more than one attributes are needed to uniquely identify an entity. A primary key that is made by the combination of more than one attribute is known as a composite key.

In other words we can say that:

Composite key is a key which is the combination of more than one field or column of a given table. It may be a candidate key or primary key.

Columns that make up the composite key can be of different data types.

**SQL Syntax to specify composite key:**

**CREATE** **TABLE** TABLE\_NAME

(COLUMN\_1, DATA\_TYPE\_1,

COLUMN\_2, DATA\_TYPE\_2,

???

**PRIMARY** **KEY** (COLUMN\_1, COLUMN\_2, ...));

In all cases composite key created consist of COLUMN1 and COLUMN2.

MySQL:

1. **CREATE** **TABLE** SAMPLE\_TABLE
2. (COL1 **integer**,
3. COL2 **varchar**(30),
4. COL3 **varchar**(50),
5. **PRIMARY** **KEY** (COL1, COL2));

MySQL:

1. **CREATE** **TABLE** SAMPLE\_TABLE
2. (COL1 **integer**,
3. COL2 **varchar**(30),
4. COL3 **varchar**(50),
5. **PRIMARY** **KEY** (COL1, COL2));

Oracle:

1. **CREATE** **TABLE** SAMPLE\_TABLE
2. **CREATE** **TABLE** SAMPLE\_TABLE
3. (COL1 **integer**,
4. COL2 **varchar**(30),
5. COL3 **varchar**(50),
6. **PRIMARY** **KEY** (COL1, COL2));

**SQL Server:**

Let's see the Syntax for the select top statement:

1. **CREATE** **TABLE** SAMPLE\_TABLE
2. (COL1 **integer**,
3. COL2 nvarchar(30),
4. COL3 nvarchar(50),
5. **PRIMARY** **KEY** (COL1, COL2));

# Unique Key in SQL

A unique key is a set of one or more than one fields/columns of a table that uniquely identify a record in a database table.

You can say that it is little like primary key but it can accept only one null value and it cannot have duplicate values.

The unique key and primary key both provide a guarantee for uniqueness for a column or a set of columns.

There is an automatically defined unique key constraint within a primary key constraint.

There may be many unique key constraints for one table, but only one PRIMARY KEY constraint for one table.

**SQL UNIQUE KEY constraint on CREATE TABLE:**

If you want to create a UNIQUE constraint on the "S\_Id" column when the "students" table is created, use the following SQL syntax:

**SQL Server / Oracle / MS Access:**

**(Defining a unique key constraint on single column):**

1. **CREATE** **TABLE** students
2. (
3. S\_Id **int** NOT NULL **UNIQUE**,
4. LastName **varchar** (255) NOT NULL,
5. FirstName **varchar** (255),
6. City **varchar** (255)
7. )

**MySQL:**

1. **CREATE** **TABLE** students
2. **CREATE** **TABLE** students
3. (
4. S\_Id **int** NOT NULL,
5. LastName **varchar** (255) NOT NULL,
6. FirstName **varchar** (255),
7. City **varchar** (255),
8. **UNIQUE** (S\_Id)
9. )

**(Defining a unique key constraint on multiple columns):**

**MySQL / SQL Server / Oracle / MS Access:**

1. **CREATE** **TABLE** students
2. (
3. S\_Id **int** NOT NULL,
4. LastName **varchar** (255) NOT NULL,
5. FirstName **varchar** (255),
6. City **varchar** (255),
7. **CONSTRAINT** uc\_studentId **UNIQUE** (S\_Id, LastName)
8. )

**SQL UNIQUE KEY constraint on ALTER TABLE:**

If you want to create a unique constraint on "S\_Id" column when the table is already created, you should use the following SQL syntax:

**(Defining a unique key constraint on single column):**

**MySQL / SQL Server / Oracle / MS Access:**

1. **ALTER** **TABLE** students
2. **ADD** **UNIQUE** (S\_Id)

**(Defining a unique key constraint on multiple columns):**

**MySQL / SQL Server / Oracle / MS Access:**

1. **ALTER** **TABLE** students
2. **ADD** **CONSTRAINT** uc\_StudentId **UNIQUE**  (S\_Id, LastName)

**DROP SYNTAX FOR A FOREIGN KEY constraint:**

If you want to drop a UNIQUE constraint, use the following SQL syntax:

**MySQL:**

1. **ALTER** **TABLE** students
2. **DROP** **INDEX** uc\_studentID

**SQL Server / Oracle / MS Access:**

1. **ALTER** **TABLE** students
2. **DROP** **CONSTRAINT** uc\_studentID

# Alternate Key in SQL

Alternate key is a secondary key it can be simple to understand by an example:

Let's take an example of student it can contain NAME, ROLL NO., ID and CLASS.

Here ROLL NO. is primary key and rest of all columns like NAME, ID and CLASS are alternate keys.

If a table has more than one candidate key, one of them will become the primary key and rest of all are called alternate keys.

In simple words, you can say that any of the candidate key which is not part of primary key is called an alternate key. So when we talk about alternate key, the column may not be primary key but still it is a unique key in the column.

#### **An alternate key is just a candidate key that has not been selected as the primary key.**

# SQL vs NoSQL

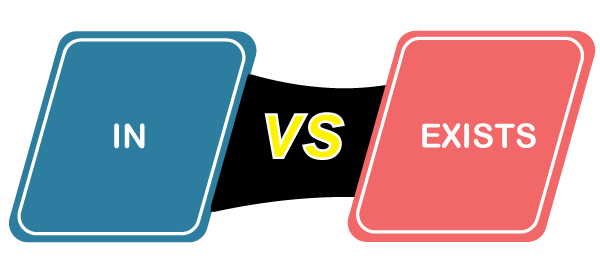
There are a lot of databases used today in the industry. Some are SQL databases, some are NoSQL databases. The conventional database is SQL database system that uses tabular relational model to represent data and their relationship. The NoSQL database is the newer one database that provides a mechanism for storage and retrieval of data other than tabular relations model used in relational databases.

Following is a list of differences between SQL and NoSQL database:

|  |  |  |
| --- | --- | --- |
| **Index** | **SQL** | **NoSQL** |
| 1) | Databases are categorized as Relational Database Management System (RDBMS). | NoSQL databases are categorized as Non-relational or distributed database system. |
| 2) | SQL databases have fixed or static or predefined schema. | NoSQL databases have dynamic schema. |
| 3) | SQL databases display data in form of tables so it is known as table-based database. | NoSQL databases display data as collection of key-value pair, documents, graph databases or wide-column stores. |
| 4) | SQL databases are vertically scalable. | NoSQL databases are horizontally scalable. |
| 5) | SQL databases use a powerful language "Structured Query Language" to define and manipulate the data. | In NoSQL databases, collection of documents are used to query the data. It is also called unstructured query language. It varies from database to database. |
| 6) | SQL databases are best suited for complex queries. | NoSQL databases are not so good for complex queries because these are not as powerful as SQL queries. |
| 7) | SQL databases are not best suited for hierarchical data storage. | NoSQL databases are best suited for hierarchical data storage. |
| 8) | MySQL, Oracle, Sqlite, PostgreSQL and MS-SQL etc. are the example of SQL database. | MongoDB, BigTable, Redis, RavenDB, Cassandra, Hbase, Neo4j, CouchDB etc. are the example of nosql database |

# IN vs. EXISTS

This article explains the complete overview of IN and EXISTS clause. It is one of the most common questions asked by developers who write SQL queries to filter for specific values. **The main difference between them is that IN selects a list of matching values, whereas EXISTS returns the Boolean value TRUE or FALSE**. Before making the comparison, we will first know these [SQL](https://www.javatpoint.com/sql-tutorial) clauses.



## **IN Operator**

The IN operator is used to **retrieves results when the specified value matches any value in a set of values or is returned by a subquery**. This operator allows us to specify multiple values along with the [WHERE clause](https://www.javatpoint.com/sql-where). It reduces the use of **multiple OR conditions** in [SELECT](https://www.javatpoint.com/sql-select), [INSERT](https://www.javatpoint.com/sql-insert), [UPDATE](https://www.javatpoint.com/sql-update), and [DELETE](https://www.javatpoint.com/sql-delete) queries; that's why it is also called the shorthand for multiple OR conditions.

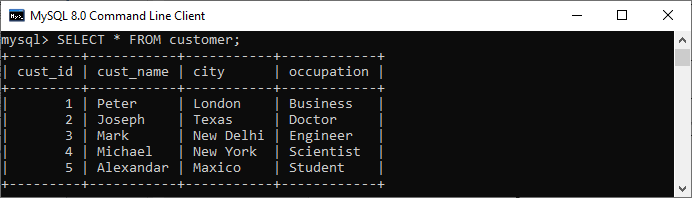
In this operator, the inner query executes first, and the result obtained is used by the outer query to display the output. It should be remembered that the inner query is executed only once. The **IN operator** has the following syntax:

**SELECT** column\_name(s)

**FROM** table\_name

**WHERE** column\_name IN (value1, value2, - - - - );

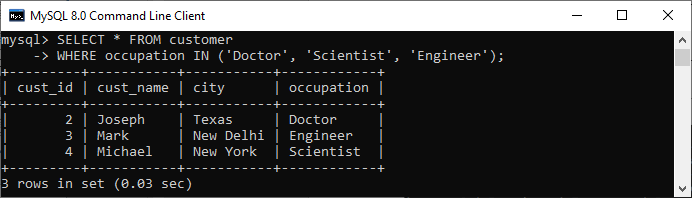
Let us take an example to understand this operator. Suppose we have a table named **customer** that contains the following data:



If we want **to get all customer details whose occupation is either doctor, engineer, or scientist**, then we can use the statement as follows:

1. mysql> **SELECT** \* **FROM** customer
2. **WHERE** occupation IN ('Doctor', 'Scientist', 'Engineer');

Here is the output:



## **EXISTS Operator**

EXISTS is a Boolean operator which **checks the subquery result and returns an either TRUE or FALSE value**. It is used in combination with subquery and checks whether a row is returned through this subquery or not. This operator returns **TRUE** if the subquery returns single or multiple records. Otherwise, it gives a **FALSE** result when no records are returned.

When the EXISTS operator detects the first true event, it automatically terminates for further processing. This feature enhances the query's efficiency. We can use the EXISTS operator with SELECT, UPDATE, DELETE, and INSERT statements. The following is the **syntax of EXISTS operator**:

**SELECT** col\_names

**FROM** tab\_name

**WHERE** [NOT] EXISTS (

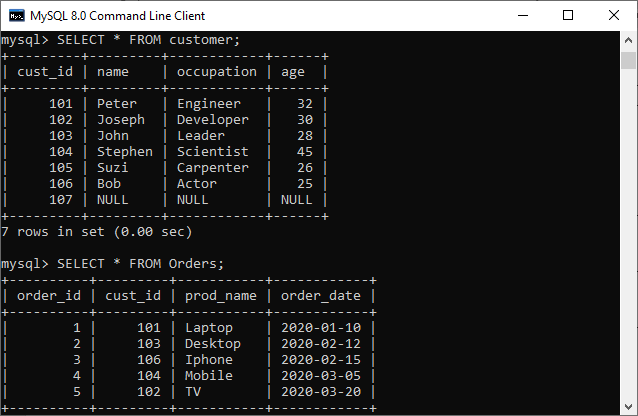
**SELECT** col\_names

**FROM** tab\_name

**WHERE** condition

);

Let us take an example to understand this operator. Suppose we have a table named **customer** and **order** containing the following data:



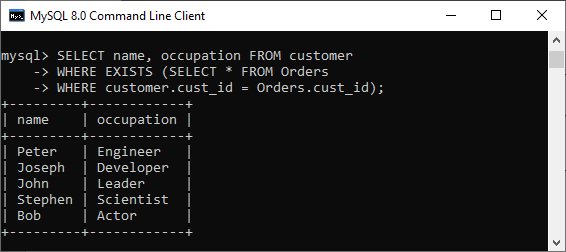
If we want **to get all customer names and occupation who has placed at least one order**, then we can use the statement as follows:

**SELECT** **name**, occupation **FROM** customer

**WHERE** EXISTS (**SELECT** \* **FROM** Orders

**WHERE** customer.cust\_id = Orders.cust\_id);

Here is the output:



## **Key differences between IN and EXISTS Operator**

The following points explain the main differences between IN and EXISTS clause:

* The IN clause scan all records fetched from the given subquery column, whereas EXISTS clause evaluates true or false, and the SQL engine quits the scanning process as soon as it found a match.
* When the subquery results are large, EXISTS operator provides better performance. In contrast, when the sub-query results are small, the IN operator is faster than EXISTS.
* IN operator always picks the matching values list, whereas EXISTS returns the Boolean values TRUE or FALSE.
* EXISTS operator can only be used with subqueries, whereas we can use the IN operator on subqueries and values both.
* The EXISTS clause can compare everything with NULLs, while the IN clause can't compare anything with NULL.
* IN operator performs a direct match between the columns specified before the IN keyword and a subquery result. Conversely, EXISTS operator does not check for a match because it only verifies data existence in a subquery.

## **IN vs. EXISTS Comparison Chart**

The following comparison chart explains their main differences in a quick manner:

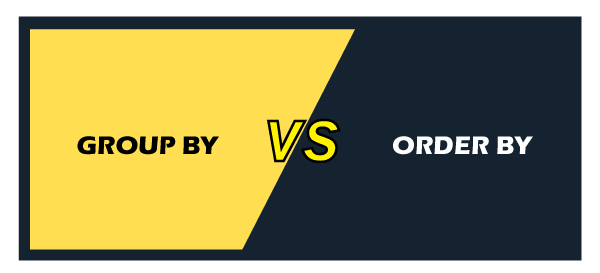
|  |  |  |
| --- | --- | --- |
| **SN** | **IN Operator** | **EXISTS Operator** |
| 1. | It is used to minimize the multiple OR conditions. | It is used to check the existence of data in a subquery. In other words, it determines whether the value will be returned or not. |
| 2. | It compares the values between subquery (child query) and parent query. | It does not compare the values between subquery and parent query. |
| 3. | It scans all values inside the IN block. | It stops for further execution once the single positive condition is met. |
| 4. | It can return TRUE, FALSE, or NULL. Hence, we can use it to compare NULL values. | It returns either TRUE or FALSE. Hence, we cannot use it to compare NULL values. |
| 5. | We can use it on subqueries as well as with values. | We can use it only on subqueries. |
| 6. | It executes faster when the subquery result is less. | It executes faster when the subquery result is large. It is more efficient than IN because it processes Boolean values rather than values itself. |
| 7. | Syntax to use IN clause:  SELECT col\_names  FROM tab\_name  WHERE col\_name IN (subquery); | Syntax to use EXISTS clause:  SELECT col\_names  FROM tab\_name  WHERE [NOT] EXISTS (subquery); |

## **Conclusion**

In this article, we have made a comparison between IN and EXISTS operators. Here, we conclude that both clauses work for the same purpose, but their internal working is different. In other words, they **differ in their logical working**. We can select any of them according to our requirement, but if we have a table that contains several records (large data), it is better to use EXISTS rather than IN operator.

# GROUP BY vs. ORDER BY

This article explains the complete overview of the GROUP BY and ORDER BY clause. They are mainly used for organizing data obtained by SQL queries. The difference between these clauses is one of the most common places to get stuck when learning [SQL](https://www.javatpoint.com/sql-tutorial). The main difference between them is that **the GROUP BY clause is applicable when we want to use aggregate functions to more than one set of rows. The ORDER BY clause is applicable when we want to get the data obtained by a query in the sorting order**. Before making the comparison, we will first know these SQL clauses.



## **ORDER BY Clause**

The [ORDER BY clause](https://www.javatpoint.com/sql-order-by) is used in SQL queries to sort the data returned by a query in ascending or descending order. If we omit the sorting order, it sorts the summarized result in the ascending order by default. The ORDER BY clause, like the GROUP BY clause, could be used in conjunction with the SELECT statement. **ASC** denotes ascending order, while **DESC** denotes descending order.

**The following is the syntax to use the ORDER BY clause in a SQL statement:**

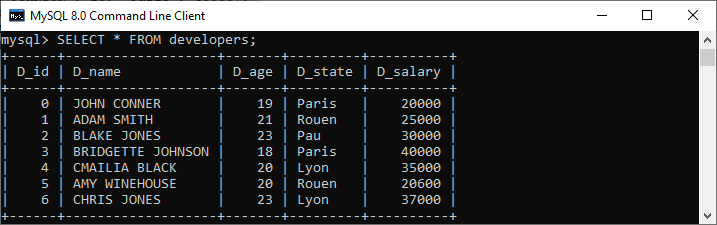
**SELECT** expressions

**FROM** tables

[**WHERE** conditions]

**ORDER** **BY** expression [ **ASC** | **DESC** ];

Let us understand how the ORDER BY clause works with the help of the following example. Suppose we have a table **developer** that contains the following data:



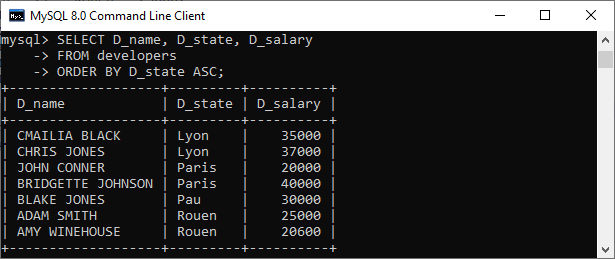
We can see that these results are not displayed in an organized way. Suppose we want to organize results in ascending or descending order based on the **state column**. In that case, we would need the ORDER BY command to get the desired result. We can do this by executing the command as follows:

**SELECT** D\_name, D\_state, D\_salary

**FROM** developers

**ORDER** **BY** D\_state **ASC**;

Here is the output where we will get the desired results:



## **GROUP BY Clause**

The [GROUP BY clause](https://www.javatpoint.com/sql-group-by) is used in SQL queries to organize data that have the same attribute values. Usually, we use it with the [SELECT statement](https://www.javatpoint.com/sql-select). It is always to remember that we have to place the GROUP BY clause after the [WHERE clause](https://www.javatpoint.com/sql-where). Additionally, it is paced before the ORDER BY clause.

We can often use this clause in collaboration with aggregate functions like SUM, AVG, MIN, MAX, and COUNT to produce summary reports from the database. It's important to remember that the attribute in this clause must appear in the SELECT clause, not under an aggregate function. If we do so, the query would be incorrect. As a result, the GROUP BY clause is always used in conjunction with the SELECT clause. The query for the GROUP BY clause is grouped query, and it returns a single row for each grouped object.

**The following is the syntax to use GROUP BY clause in a SQL statement:**

**SELECT** column\_name, **function**(column\_name)

**FROM** table\_name

**WHERE** condition

**GROUP** **BY** column\_name;

Let us understand how the GROUP BY clause works with the help of an example. Here we will demonstrate it with the same table.

Suppose we want to know **developer's average salary in a particular state** and organize results in descending order based on the state column. In that case, we would need both the GROUP BY and ORDER BY command to get the desired result. We can do this by executing the command as follows:

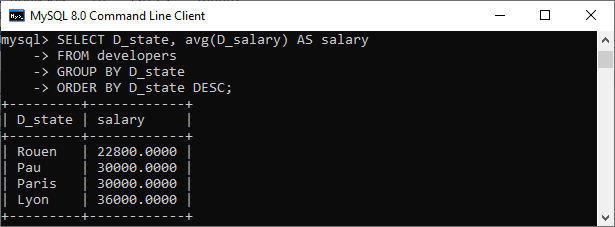
**SELECT** D\_state, avg(D\_salary) **AS** salary

**FROM** developers

**GROUP** **BY** D\_state

**ORDER** **BY** D\_state **DESC**;

This query initially formed an intermediate result that has grouped the state. Next, the **AVG** function is performed on each group of states, then sort the result in descending order, and finally, we will get the desired results as shown below:



## **Key Differences between GROUP BY and ORDER BY**

The following are the key distinctions between the Group By and Order By clause:

* The Group By clause is used to group data based on the same value in a specific column. The ORDER BY clause, on the other hand, sorts the result and shows it in ascending or descending order.
* It is mandatory to use the aggregate function to use the Group By. On the other hand, it's not mandatory to use the aggregate function to use the Order By.
* The attribute cannot be under GROUP BY statement under aggregate function, whereas the attribute can be under ORDER BY statement under aggregate function.
* **Group By clause controls the presentation of tuples that means grouping is done based on the similarity among the row's attribute values.** In contrast, the ORDER BY clause controls the presentation of columns that means the ordering or sorting is done based on the column's attribute values either in ascending or descending order.
* GROUP BY is always placed after the WHERE clause but before the ORDER BY statement. On the other hand, ORDER BY is always used after the GROUP BY statement.

## **GROUP BY vs. ORDER BY Comparison Chart**

The following comparison chart explains their main differences in a quick manner:

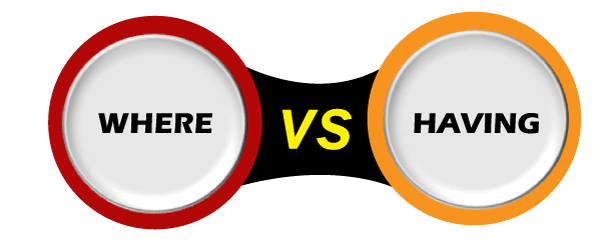
|  |  |  |
| --- | --- | --- |
| **SN** | **GROUP BY** | **ORDER BY** |
| 1. | It is used to group the rows that have the same values. | It sorts the result set either in ascending or descending order. |
| 2. | It may be allowed in CREATE VIEW statement. | It is not allowed in CREATE VIEW statement |
| 3. | It controls the presentation of rows. | It controls the presentation of columns. |
| 4. | The attribute cannot be under aggregate function under GROUP BY statement. | The attribute can be under aggregate function under ORDER BY statement. |
| 5. | It is always used before the ORDER BY clause in the SELECT statement. | It is always used after the GROUP BY clause in the SELECT statement. |
| 6. | It is mandatory to use aggregate functions in the GROUP BY. | It's not mandatory to use aggregate functions in the ORDER BY. |
| 7. | Here, the grouping is done based on the similarity among the row's attribute values. | Here, the result-set is sorted based on the column's attribute values, either ascending or descending order. |

## **Conclusion**

The GROUP BY and ORDER BY clauses are compared in this article. Both clauses are extremely useful SQL database features. When we want to form a group of rows, we use the GROUP BY clause. If we want to organize data in ascending or descending order based on a particular column, we use the ORDER BY clause. They do not have any relationship because both are used for two different purposes. However, we can combine them to serve some special purpose or can use them individually depending on the circumstances. We can use these clauses only with the SELECT statement.

# Difference between WHERE and HAVING

The WHERE and HAVING clauses are discussed in depth in this article. They're also used to filter records in SQL queries. The difference between the WHERE and HAVING clause is the most common question posed during an interview time. **The main difference between them is that the WHERE clause is used to specify a condition for filtering records before any groupings are made, while the HAVING clause is used to specify a condition for filtering values from a group.** Before making the comparison, we will first know these [SQL](https://www.javatpoint.com/sql-tutorial) clauses.



## **WHERE Clause**

The WHERE clause in MySQL is used with [SELECT](https://www.javatpoint.com/sql-select), [INSERT](https://www.javatpoint.com/sql-insert), [UPDATE](https://www.javatpoint.com/sql-update), and [DELETE](https://www.javatpoint.com/sql-delete) queries to filter data from the table or relation. It describes a specific condition when retrieving records from a single table or multiple tables using the [JOIN clause](https://www.javatpoint.com/sql-join). If the specified condition is satisfied, it returns the particular value from the table. The [WHERE clause](https://www.javatpoint.com/sql-where) places conditions on the selected columns.

The WHERE clause in MySQL can also **implement the logical connectives** [AND](https://www.javatpoint.com/sql-and), [OR](https://www.javatpoint.com/sql-or), and NOT. They are known as the Boolean condition that must be **true** to retrieve the data. The logical connectives expressions use the comparison operators as their operands like <, <=, >, >=, =, and <>. The comparison operators are usually used to compare strings and arithmetic expressions.

**The following syntax illustrates the use of the WHERE clause:**

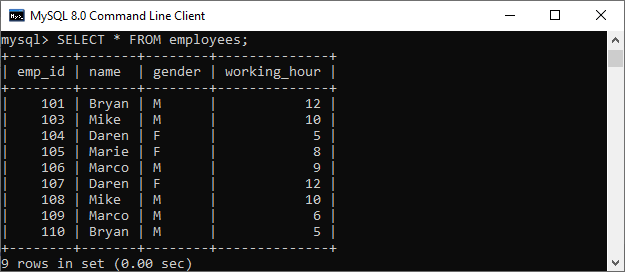
**SELECT** column\_lists,

**FROM** table\_name

**WHERE** conditions

**GROUP** **BY** column\_lists;

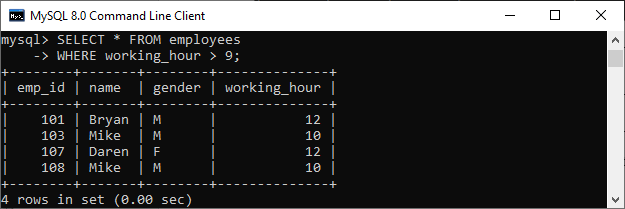
Let us take an example to understand this clause. Suppose we have a table named **employees** that contain the following data:



If we want **to get the employee detail whose working hours are greater than 9**, then we can use the statement as follows:

1. mysql> **SELECT** \* **FROM** employees
2. **WHERE** working\_hour > 9;

We will get the below output where we can see employee detail whose working hours are greater than 9:



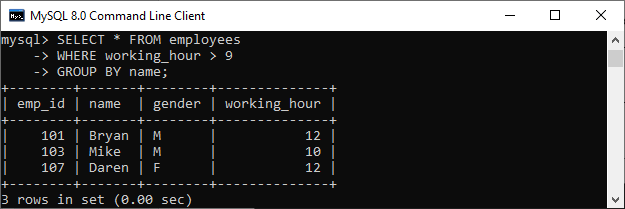
If we use the above query with the [**GROUP BY** clause](https://www.javatpoint.com/sql-group-by), we will get the different result:

**SELECT** \* **FROM** employees

**WHERE** working\_hour > 9

**GROUP** **BY** **name**;

Here is the output:



## **HAVING Clause**

HAVING clause in MySQL **used in conjunction with GROUP BY** clause enables us to specify conditions that filter which group results appear in the result. It returns only those values from the groups in the final result that fulfills certain conditions. We can also use the WHERE and HAVING clause together during selection. In this case, WHERE clause first filters the individual rows, then rows are grouped, performs aggregate calculations, and at last HAVING clause filter the groups.

This clause places conditions on groups created by the GROUP BY clause. It behaves like the WHERE clause when the SQL statement does not use the GROUP BY keyword. We can use the aggregate (group) functions such as [SUM](https://www.javatpoint.com/sql-select-sum), MIN, MAX, AVG, and [COUNT](https://www.javatpoint.com/sql-select-count) only with two clauses: SELECT and HAVING.

**The following syntax illustrates the use of the HAVING clause:**

**SELECT** column\_lists,

aggregate\_function (expression)

**FROM** table\_name

**WHERE** conditions

**GROUP** **BY** column\_lists

**HAVING** condition;

Let us take an example to understand this clause. Here we are considering the same table **employees** for demonstration.

If we want **to get the total working hours for each employee whose working time is greater than 6 hour**, then we can use the statement as follows:

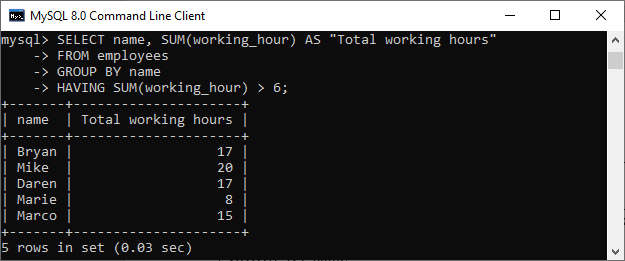
**SELECT** **name**, SUM(working\_hour) **AS** "Total working hours"

**FROM** employees

**GROUP** **BY** **name**

**HAVING** SUM(working\_hour) > 6;

We will get the below output where we can see each employee total working hours:



## **Key Differences between WHERE and HAVING Clause**

The following points explain the main differences between database and schema:

* WHERE clause filters individual rows, whereas the HAVING clause filters groups instead of one row at a time.
* We cannot use the WHERE clause with aggregate functions because it works for filtering individual rows. In contrast, HAVING can works with aggregate functions because it is used to filter groups.
* Row operations are handled by the WHERE clause, while the HAVING clause handles column operations to summarized rows or groups.
* WHERE comes before GROUP BY, which means WHERE clause filter rows before performing aggregate calculations. HAVING comes after GROUP BY, which means the HAVING clause filters rows after performing aggregate calculations. Consequently, HAVING is slower than WHERE in terms of efficiency and should be avoided wherever possible.
* We can combine the WHERE and HAVING clause together in a SELECT query. In this case, the WHERE clause is used first to filter individual rows. The rows are then grouped, perform aggregate calculations, and finally, the HAVING clause is used to filter the groups.
* The WHERE clause retrieves the desired data based on the specified condition. On the other hand, the HAVING clause first fetches whole data, and then separation is done based on the specified condition.
* Without a SELECT statement, we cannot use the HAVING clause. Conversely, we can use a WHERE with SELECT, UPDATE, and DELETE statements.
* WHERE clause is a pre-filter, whereas HAVING clause is a post-filter.

## **WHERE vs. HAVING Comparison Chart**

The following comparison chart explains their main differences in a quick manner:

|  |  |  |
| --- | --- | --- |
| **Comparison Basis** | **WHERE Clause** | **HAVING Clause** |
| **Definition** | It is used to perform filtration on individual rows. | It is used to perform filtration on groups. |
| **Basic** | It is implemented in row operations. | It is implemented in column operations. |
| **Data fetching** | The WHERE clause fetches the specific data from particular rows based on the specified condition | The HAVING clause first fetches the complete data. It then separates them according to the given condition. |
| **Aggregate Functions** | The WHERE clause does not allow to work with aggregate functions. | The HAVING clause can work with aggregate functions. |
| **Act as** | The WHERE clause acts as a pre-filter. | The HAVING clause acts as a post-filter. |
| **Used with** | We can use the WHERE clause with the SELECT, UPDATE, and DELETE statements. | The HAVING clause can only use with the SELECT statement. |
| **GROUP BY** | The GROUP BY clause comes after the WHERE clause. | The GROUP BY clause comes before the HAVING clause. |

## **Conclusion**

In this article, we have made a comparison between the WHERE and HAVING clause. Here, we conclude that both clauses work in the same way in filtering the data, **except some additional feature makes the HAVING clause more popular.** We can efficiently work with aggregate functions in the HAVING clause while WHERE does not allow for aggregate functions.

# Where condition in SQL

The MySQL WHERE statement is used to define a circumstance when the data is collected from a separate list or joined to several tables. If the specified circumstance is met, then only the unique value of the table is returned. To extract the information and retrieve only the correct records, you can use the WHERE statement.

Not only, the [WHERE](https://www.javatpoint.com/mysql-where) condition used in the Selection statement, but it is also used in the [UPDATE](https://www.javatpoint.com/sql-update) assertion, [DELETE](https://www.javatpoint.com/sql-delete) assertion, etc.

### **Syntax**

Now, we have discussed the syntax of the [SELECT statement](https://www.javatpoint.com/sql-select) associated with the [WHERE](https://www.javatpoint.com/sql-where) condition. It is shown below-

1. **SELECT** column1, column2, ..... columnN
2. **FROM** table\_name
3. **WHERE** {condition}

#### **Note: The WHERE condition is used not only in the Selection assertion but also in the UPDATE assertion, the Remove assertion, etc.!**

You may use contrast or logical operators, such as >, <, =, LIKE, NOT, etc. To define a state. The below-given example will explicitly state this notation.

**For Example-**

Here, we consider the table of CUSTOMERS with the below-given records.

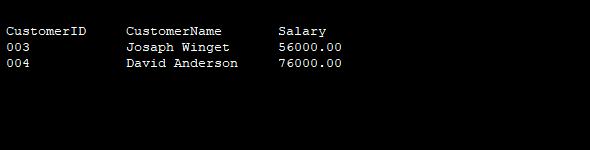
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CustomerID** | **CustomerName** | **Address** | **Age** | **Salary** |
| 001 | Hardy Thompson | Maxico | 25 | 10000.00 |
| 002 | Sturt Broad Los Angeles | 28 | 25000.00 |  |
| 003 | Joseph Winget | California | 30 | 56000.00 |
| 004 | David Anderson | Norway | 24 | 76000.00 |
| 005 | Alexa | Denmark | 35 | 23000.00 |

Now, we will run the below-given query on [SQL](https://www.javatpoint.com/sql-tutorial) to fetch the CustomerID, CustomerName and Salary from the above table CUSTOMERS, where the salary is greater than 25000.

1. **Select** CustomerID, CustomerName, Salary
2. **from** CUSTOMERS
3. **Where** Salary > 25000;

**Output**

After the successful execution of the above SQL query, we got the following output.

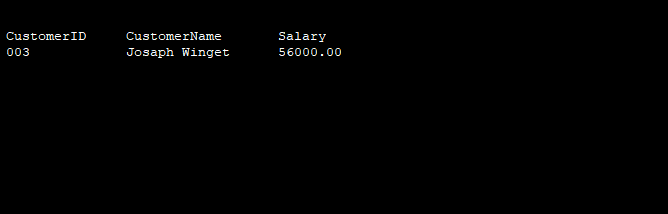


Here, we have another instance where we would get the CustomerID, CustomerName and Salary from the database table named CUSTOMERS for a specific customer named **Joseph Winget.**

1. **Select** CustomerID, CustomerName, Salary
2. **from** CUSTOMERS
3. **Where** CustomerName = 'Joseph Winget';

**Output**

After the successful execution of the above SQL query, we got the above-given outcome.



#### **Note- Here, it should be noted that all strings should be specified within single quotes (''). So, as in the instance discussed earlier, should provide numerical data without any quotation.**

# SQL Injection

The SQL Injection is a code penetration technique that might cause loss to our database. It is one of the most practiced web hacking techniques to place malicious code in SQL statements, via webpage input. SQL injection can be used to manipulate the application's web server by malicious users.

SQL injection generally occurs when we ask a user to input their username/userID. Instead of a name or ID, the user gives us an SQL statement that we will unknowingly run on our database. For Example - we create a SELECT statement by adding a variable "demoUserID" to select a string. The variable will be fetched from user input (getRequestString).

1. demoUserI = getrequestString("UserId");
2. demoSQL = "SELECT \* FROM users WHERE UserId =" +demoUserId;

## **Types of SQL injection attacks**

SQL injections can do more harm other than passing the login algorithms. Some of the SQL injection attacks include:

* Updating, deleting, and inserting the data: An attack can modify the cookies to poison a web application's database query.
* It is executing commands on the server that can download and install malicious programs such as Trojans.
* We are exporting valuable data such as credit card details, email, and passwords to the attacker's remote server.
* Getting user login details: It is the simplest form of SQL injection. Web application typically accepts user input through a form, and the front end passes the user input to the back end database for processing.

## **Example of SQL Injection**

We have an application based on employee records. Any employee can view only their own records by entering a unique and private employee ID. We have a field like an Employee ID. And the employee enters the following in the input field:

236893238 or 1=1

It will translate to:

1. **SELECT** \* **from** EMPLOYEE **where** EMPLOYEE\_ID == 236893238 or 1=1

The SQL code above is valid and will return EMPLOYEE\_ID row from the EMPLOYEE table. The 1=1 will return all records for which this holds true. All the employee data is compromised; now, the malicious user can also similarly delete the employee records.

Example:

1. **SELECT** \* **from** Employee **where** (Username == "" or 1=1) AND (**Password**="" or 1=1).

Now the malicious user can use the '=' operator sensibly to retrieve private and secure user information. So instead of the query mentioned above, the following query, when exhausted, retrieve protected data, not intended to be shown to users.

1. **SELECT** \* **from** EMPLOYEE **where** (Employee\_name =" " or 1=1) AND (**Password**=" " or 1=1)

## **SQL injection based on Batched SQL statements**

Several databases support batched SQL statements. It is a group of two or more SQL statements separated by semicolons.

The SQL statement given below will return all rows from the Employee table, then delete the Employee\_Add table.

1. **SELECT** \* **From** Employee; **DROP** **Table** Employee\_Add

## **How to detect SQL Injection attacks**

Creating a SQL Injection attack is not difficult, but even the best and good-intentioned developers make mistakes. The detection of SQL Injection is, therefore, an essential component of creating the risk of an SQL injection attack. Web Application Firewall can detect and block basic SQL injection attacks, but we should depend on it as the sole preventive measure.

Intrusion Detection System (IDS) is both network-based and host-based. It can be tuned to detect SQL injection attacks. Network-based IDSec can monitor all connections to our database server, and flags suspicious activities. The host-based IDS can monitor web server logs and alert when something strange happens.

## **Impact of SQL Injection**

The intruder can retrieve all the user-data present in the database, such as user details, credit card information, and social security numbers, and can also gain access to protected areas like the administrator portal. It is also possible to delete the user data from the tables. These days all the online shopping applications, bank transactions use back-end database servers. If the intruder can exploit SQL injection, the entire server is compromised.

## **How to prevent SQL Injection attack**

* We should use user authentication to validate input from the user by pre-defining length, input type, and the input field.
* Restricting the access privileges of users and defining the amount of data any outsider can access from the database. Generally, the user cannot be granted permission to access everything in the database.
* We should not use system administrator accounts.

# SQL CAST Function

The SQL CAST function is mainly used to convert the expression from one data type to another data type. **If the SQL Server CAST function is unable to convert a declaration to the desired data type, this function returns an error. We use the CAST function to convert numeric data into character or string data.**

### **Syntax:**

1. CAST (expression **AS** [data type])

Here, the [data type] is a type of valid data types in RDBMS.

The syntax is:

CAST (EXPRESSION **AS** Data\_ Type[(Length)]

\_ \_ CAST in the SQL example

**SELECT** CAST (123 **AS** **VARCHAR** (20)) [result\_name]

**FROM** [Source]

* **Expression**: It is a valid expression where we want to convert a data type into the SQL.
* **Data\_type**: It is a Data Type to which we want to convert the expression.
* **Length**: It is the optional parameter of an integer type. We can use the setting to define the length of any targeted data type.

By default, it is 30.

### **Examples:**

**Here, we will use the below tables.**

**Table Manager \_ Score**

|  |  |
| --- | --- |
| **Column Name** | **Date Type** |
| **Manager Id** | Integer |
| **First \_ Name** | char( 20) |
| **Score** | float |

The table contains the following rows:

**Table Manager\_Score**

|  |  |  |
| --- | --- | --- |
| **ManagerID** | **First\_Name** | **Score** |
| 1 | Jame | 92.2 |
| 2 | Boby | 87.5 |
| 3 | Marry | 70 |
| 4 | Sanju | 120.2 |

### **Example 1:**

**SELECT** First\_Name, CAST (Score **AS** **Integer**)

Int\_Score **FROM** Student\_Score;

### **Result:**

|  |  |
| --- | --- |
| **First\_Name** | **Int\_Score** |
| Jame | 92 |
| Boby | 87 |
| Marry | 70 |
| sanju | 120 |

In **Example 1**, we are using the CAST function to convert the SCORE column from type FLOAT to INTEGER. When we do it, various RDBMS have many rules to handle the numbers to the point of decimal.

According to the above example, the numbers after the decimal point are truncated.

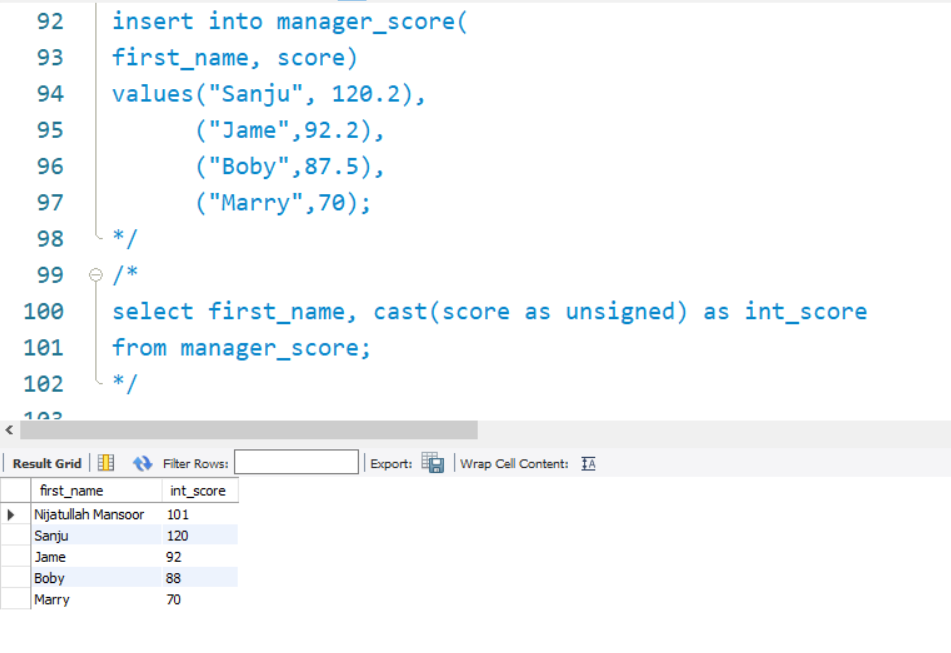
### **Example 2:**

1. **SELECT** First\_Name, CAST (Score **AS** **char** (3))
2. Char\_Score **FROM** Student\_Score;

**Result:**

|  |  |
| --- | --- |
| **First\_\_Name** | **Char\_\_Score** |
| Jame | 85. |
| Boby | 92. |
| Marry | 90 |
| sanju | 110 |

In Example 2, we use the CAST function to convert the **SCORE** column from type FLOAT to CHAR (3). When we do it, we only hold the first 3 character. If there are more than three characters, everything after the first three characters is discarded.



# SQL Comments

SQL Comments are used to explain the sections of the SQL statements, and used to prevent the statements of SQL. In many programming languages, comments matter a lot.

A Microsoft Access database does not support the comments. So, **Mozilla Firefox** and **Microsoft Edge** use the Microsoft Access database in the examples.

There are three types of comments, which are given below:

1. **Single line comments.**
2. **Multi-line comments**
3. **Inline comments**

## **Single Line Comment**

Comments starting and ending with a single line are said as individual line comments. The line which starts with '–' is a single line comment, and that particular line is not executed.

The text between **--** and end of the line is ignored and cannot be executed.

**Syntax:**

* -- single-line comment
* -- another comment
* SELECT \* FROM Customers;

The following example uses a single-line comment:

### **Example 1**

1. --Select all:
2. **SELECT** \* **FROM** Employees;

The given example uses a single-line comment to ignore the end of the line:

### **Example 2**

1. **SELECT** \* **FROM** Customers -- WHERE City='London';

The following example uses the single-line comment to ignore the statements:

### **Example 3**

1. --SELECT \* FROM Employees;
2. **SELECT** \* **FROM** Products;\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## **Multi-line Comments**

Comments that start in one line and end in different front are said as multi-line comments. The text between **/\*** and **\*/** is ignored in the code part.

The line starting with '/\*' is considered as a starting point of comment and terminated when '\*/' lies at the end.

**Syntax**:

1. /\* multi-line comment
2. another comment \*/
3. **SELECT** \* **FROM** Customers;

### **Example 1**

1. /\***Select** all the columns
2. **of** all the records
3. in the Customers **table**:\*/
4. **SELECT** \* **FROM** Employees;

The below example uses a multi-line comment to ignore more statements:

### **Example 2**

1. /\***SELECT** \* **FROM** Customers;
2. **SELECT** \* **FROM** Products;
3. **SELECT** \* **FROM** Orders;
4. **SELECT** \* **FROM** Categories;\*/
5. **SELECT** \* **FROM** Suppliers;

To ignore some part of a statement, use the **/\*....... \*/** comment.

The following example uses a comment to ignore some part of any code:

### **Example of SQL Multi-line Comment:**

1. /\***SELECT** \* **FROM** Customers;
2. **SELECT** \* **FROM** Products;
3. **SELECT** \* **FROM** Orders;
4. **SELECT** \* **FROM** Categories;\*/
5. **SELECT** \* **FROM** Suppliers;

### **Example**

1. **SELECT** CustomerName, /\*City,\*/ Country **FROM** Customers;

The following example uses a comment to not to be the part of a statement:

### **Example**

1. **SELECT** \* **FROM** Customers **WHERE** (CustomerName LIKE 'L%.'
2. OR CustomerName LIKE 'R%' /\*OR CustomerName LIKE 'S%'
3. OR CustomerName LIKE 'T%'\*/ OR CustomerName LIKE 'W%')
4. AND Country='America.'
5. **ORDER** **BY** CustomerName;

## **Inline comments:**

Inline comments are an extension of multi-line comments, and comments can be stated between the statements and are enclosed in between '/\*' and '\*/.'

**Syntax**:

SELECT \* FROM /\*Employees; \*/

### **Examples:**

1. Multi line comment ->
2. /\* **SELECT** \* **FROM** Teachers;
3. **SELECT** \* **FROM** Teacher\_DETAILS;
4. **SELECT** \* **FROM** Orders; \*/
5. **SELECT** \* **FROM** Course;

In line comment ->

1. **SELECT** \* **FROM** Students;
2. **SELECT** \* **FROM** /\* Employee\_DETAILS;
3. **SELECT** \* **FROM** Orders;
4. **SELECT** \* **FROM** \*/ Topics;

## **SQL Comment Indicators**

SQL Comment Indicator is indicated according to the given examples

It includes the double hyphen ( — ), braces ( { } ), and C-style ( /\* . . . \*/ ) comment delimiters. It also includes the comments after the statement.

1. **SELECT** \* **FROM** customer; -- Selects all rows and columns
2. **SELECT** \* **FROM** employee; {Selects all **rows** and columns}
3. **SELECT** \* **FROM** employee; /\*Selects all columns and **rows**\*/copy **to** the clipboard

In the below examples, we place the comments on a single line code -

1. **SELECT** \* **FROM** customer;
2. -- Selects all the rows and columns
3. **SELECT** \* **FROM** employee;
4. {Selects all columns and **rows**}
5. **SELECT** \* **FROM** customer;
6. /\*Selects all columns and **rows**\*/

### **Examples of multi-line statements -**

1. **SELECT** \* **FROM** customer;
2. -- Selects all columns and rows
3. -- from the customer table
4. **SELECT** \* **FROM** customer;
5. {Selects all columns and **rows**
6. **from** the customer **table**}
7. **SELECT** \* **FROM** customer;
8. /\*Selects all columns and **rows**
9. **from** the customer **table**\*/copy **to** clipboard
10. **SELECT** \* -- Selects all columns and rows
11. **FROM** customer; -- from the customer table
12. **SELECT** \* {Selects all columns and **rows**}
13. **FROM** customer; {**from** the customer **table**}
14. **SELECT** \* /\*Selects all columns and **rows**\*/
15. **FROM** customer; /\***from** the customer **table**\*/copy **to** clipboard

# SQL CONCAT Function

The CONCAT function in SQL is a String function, which is used to merge two or more strings. The Concat service converts the Null values to an Empty string when we display the result. This function is used to concatenate two strings to make a single string. The**operator**is used to link **character strings**and **column string**.

We can use a **literal in**CONCAT Function. A literal is a **number, character**, or **date** that includes the SELECT statement.

### **Syntax of CONCAT function:**

1. **SELECT** CONCAT (String 1, String 2, String3.., String N)
2. **FROM** [Source]

### **Example-**

1. SQL>   **SELECT** CONCAT ('FIRST', 'SECOND');

|  |  |
| --- | --- |
| **CONCAT(' FIRST','SECOND')** | **FIRST SECOND** |

To understand the CONCAT function in detail, consider an employee\_tbl table, which has the following records -

1. SQL> **SELECT** \* **FROM** employee\_ tbl ;

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **NAME** | **WORK\_DATE** | **DAILY\_TYPING\_PAGES** |
| 1 | Michal | 2009-02-15 | 270 |
| 2 | Zeena | 2003-03-24 | 250 |
| 2 | kachner | 2007-08-19 | 277 |
| 2 | warner | 2007-04-25 | 264 |
| 3 | Joy | 2007-05-17 | 250 |
| 4 | atire | 2006-06-23 | 270 |
| 5 | delph | 2004-05-28 | 230 |

So if we want to concatenate all the names, employee IDs, and work\_ date of above table, then we can do it using the following command -

1. **SELECT** CONCAT (id , **name** , work\_date )
2. **FROM** employee\_ tbl;
3. CONCAT(id, **name**, work\_date)

|  |
| --- |
| **1Michal2009-02-15** |
| **2Zeena2003-03-24** |
| **2kachner2007-08-19** |
| **2warner2007-04-25** |
| **3joy2007-05-17** |
| **4atire2006-06-23** |
| **5delph2004-05-28** |

### **Example 2:**

1. **SELECT** id, first\_name, last\_name, first\_name || last\_name,
2. salary, first\_name || salary **FROM** myTable
3. **Output** (Third and Fifth Columns show  **values** concatenated **by** operator ||)

**Output:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **id** | **last\_name** | **first\_name** | **first\_name||last\_name** | **salary** | **first\_name||salary** |
| 1 | bean | Mr. | Mr.bean | 10000 | Mr.10000 |
| 2 | William | Sunita | Sunita William | 50000 | Sunita50000 |
| 3 | tpoint | Java | Javatpoint | 20000 | Java20000 |
| 4 | &example | tutorial | tutorial&example | 90000 | Tutorial90000 |

#### **Note: In above example, we have used "||", which is known as the Concatenation operator, and it is used to link two or more columns in select query. This operator is independent of the data type of column. Here, we have linkined 2 columns i.e, first\_name+last\_name as well as first\_name+salary.**

We can use **string literals** in CONCAT operator.

**Example 1**: Using the character literal

**Syntax**

1. **SELECT** id, first\_name, last\_name, salary,
2. first\_name||' has salary '||salary **as** "new"  **FROM** myTable

**Output:** (Concatenating three values and giving a new 'name')

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **first\_name** | **last\_name** | **salary** | **new** |
| 1 | Javatpoint | tpoint | 20000 | Java has salary 20000 |
| 2 | tutorial | &example | 30000 | the tutorial has salary 30000 |
| 3 | Shane | Watson | 40000 | Shane has salary 40000 |
| 4 | Jennifer | louse | 60000 | Jennifer has salary 60000 |

#### **Note: We have used salary as a character literal in the select statement. We can use the date literal and number literal according to our requirement in the table.**

**Example 2:** Using character as well as the number literal

**Syntax:**

1. **SELECT** id, first\_name, last\_name, salary, first\_name||100||'
2. has id '||id **AS** "new" **FROM** myTable
3. **Output** (Making  the **output** readable **by** concatenating a string
4. **with** **values**)

**Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **first\_name** | **last\_name** | **salary** | **new** |
| 1 | Javatpoint | tpoint | 20000 | Java100 has id 1 |
| 2 | tutorial | &example | 30000 | Tutorial100 has id 2 |
| 3 | Shane | Watson | 40000 | Shane100 has id 3 |
| 4 | Jennifer | louse | 60000 | Jennifer100 has id 4 |

In the above example, we have used **the salary** as a character literal as well as **100** as number authentic in our select statement.

CTE (Common Table Expression)SQL

The **Common Table Expressions** (CTE) are imported into the SQL to simplify many classes of the Structured Query Language (SQL) for a derived table, which is unsuitable. **It** was introduced in **2005 SQL SERVER** version.

The common table expressions (**CTE**) are a result set, which we reference with the **SELECT**, **INSERT, UPDATE**, or **DELETE** statement. In [SQL](https://www.javatpoint.com/sql-tutorial) **2008**, we add a **CTE** for the unique MERGE statement.

How to use CTEs in T-SQL?

The use of **Common Text Expression** is to add the clause "**WITH**" before the **SELECT, INSERT, UPDATE, DELETE or MERGE** statement. The WITH clause contain **one or more CTEs,** which are separated by commas.

1. [**WITH** [, ....]]
2. ::::=
3. cte\_name [(column\_name[, ...])]\_\_Write the **name** **of** **column** here
4. **AS** (cte\_query)

We generate CTEs when we refer to any table. The CTE result set is not accessible to any statements when we run the particular statement.

Creating a Recursive table expression

The recursive CTE is used when we are working with hierarchical data. An example of hierarchical data in the table is the list of students in the group. For each student, the counter generates a ReferenceID and a NAME. The ReferenceID references itself like an employee ID in a recursive table. We use the CTE to display the position of employee's database.

If the CTE is created wrong, it enters into the infinite loop.

To prevent the endless loop, **MAXRECURSION** will added in the **OPTION** clause of the **INSERT, DELETE, UPDATE, SELECT or MERGE** statement.

Use below code to create a table:

1. **CREATE** **TABLE** Employees
2. {
3. EmployeeID **int** NOT NULL **PRIMARY** **KEY**,
4. FirstName **varchar**(50) NOT NULL,
5. LastNamevarchar(50) NOT NULL,
6. Manager ID **int** NULL
7. }
8. **INSERT** **INTO** Employees **VALUES**(1, 'Ken', 'Thompson', NULL)
9. **INSERT** **INTO** Employees **VALUES**(2, 'Kent', 'Thompson', 1)
10. **INSERT** **INTO** Employees **VALUES**(3, 'Williams', 'Thompson', 1)
11. **INSERT** **INTO** Employees **VALUES**(4, 'Charles', 'Thompson', 2)
12. **INSERT** **INTO** Employees **VALUES**(5, 'Michal', 'Thompson', 3)
13. **INSERT** **INTO** Employees **VALUES**(6, 'Gill', 'Thompson', 3)
14. **INSERT** **INTO** Employees **VALUES**(7, 'Danyl', 'Thompson', 3)
15. **INSERT** **INTO** Employees **VALUES**(8, 'Monty', 'Thompson', 5)
16. **INSERT** **INTO** Employees **VALUES**(9, 'Rob', 'Thompson', 6)
17. **INSERT** **INTO** Employees **VALUES**(10, 'Robert', 'Thompson',6)

After the Employee table is created, a SELECT statement, which is preceded by a WITH clause that includes a CTE named cteReports is created:

1. **WITH**
2. cteReports (EmpID, FirstName, LastName, MgrID, EmpLevel
3. **AS**
4. (
5. **Select** EmployeeID, FirstName, LastName, ManagerID, 1
6. **FROM** Employees
7. //**WHERE** ManagerID **IS** NULL
8. **UNION** ALL
9. **SELECT** e.StudentID, e.FirstName, e.LastName, e.ManagerID,
10. r. StuLevel + 1
11. **FROM** Students p
12. **INNER** JOIN cteReports s
13. **ON** e.ManagerID = r.StuID
14. )
15. **SELECT**
16. **First** **Name**+ ' ' + **LAST** **NAME** **AS** FullName,StuLevel,
17. (**SELECT** FirstName + '.....' +LastName **FROM** Students
18. **WHERE** StudentID = cteReports.MgrID) **AS** ManagerFROM cteReports
19. **ORDER** **BY** StuLevel, MgrID

**It** is an essential tool to generate the inconsistent result set, and retrieved in the **SELECT, UPDATE, INSERT, MERGE,** or **DELETE** the statement.

# How to use distinct in SQL?

SQL **DISTINCT** clause is used to remove the duplicates columns from the result set.

The distinct keyword is used with select keyword in conjunction. It is helpful when we avoid duplicate values present in the specific **columns/tables**. The **unique values** are fetched when we use the distinct keyword.

* SELECT DISTINCT returns only distinct (**different**) values.
* DISTINCT eliminates duplicate records from the table.
* DISTINCT can be used with aggregates: **COUNT, AVG, MAX**, etc.
* DISTINCT operates on a single column.
* Multiple columns are not supported for DISTINCT.

### **Syntax:**

**SELECT** **DISTINCT** expressions

**FROM** tables

[**WHERE** conditions];

### **Parameters:**

**Expressions:** The columns or calculations that we want to retrieve are called expression.

**Tables:** The tables that we want to retrieve the records. There is only one table in the FROM clause.

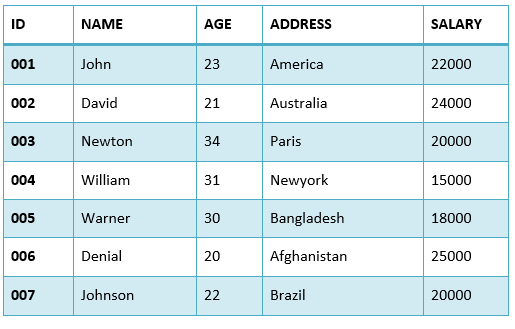
**WHERE conditions:** The conditions may meet for the records which are selected and it is optional.

**Note:**

* When one expression is provided in the **DISTINCT** clause then the query will return the unique values of the expressions.
* The query will retrieve the unique combinations for the listed expressions if more than one expression is provided in the **DISTINCT** clause here.
* In SQL, the **DISTINCT** clause cannot ignore the NULL values. So when we use the DISTINCT clause in the SQL statement, our result set will include NULL as a distinct value.

### **Example:**

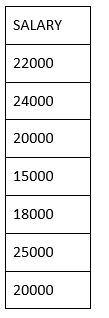
Consider the following **EMPLOYEES** table.



First, let us see the following SELECT query returns the duplicate salary records.

1. **SELECT** SALARY **FROM** EMPLOYEES
2. **ORDER** **BY** SALARY;

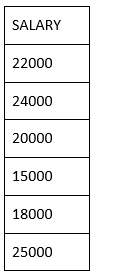
When we execute the above SQL query, it fetches all the records including the duplicate records. In the above table, salary of Newton and Johnson is same 20000.



Now, let us use the **DISTINCT** keyword with the above SELECT query.

1. **SELECT** **DISTINCT** SALARY **FROM** EMPLOYEES
2. **ORDER** **BY** SALARY;

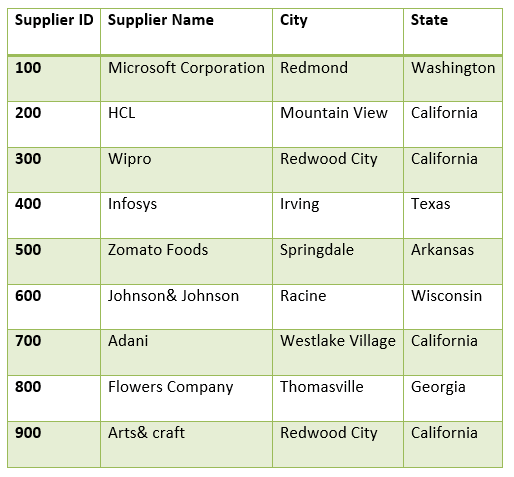
The above SQL query removes the duplicate records and shows the following result.



### **Example: Finding Unique Values in the Column**

Look at the [DISTINCT clause](https://www.javatpoint.com/sql-select-distinct) to find the unique values within one column in the table.

We have a table called suppliers with the following data:



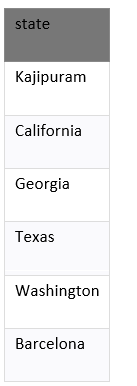
From the above table, we are going to find the unique states.

**SELECT** **DISTINCT** state

**FROM** suppliers

**ORDER** **BY** state;

These are **six the** records.



The example returns the unique state from ***suppliers table and removes*** the duplicate records from the result set.

### **Example: Finding Unique Values in Multiple Column**

The **SQL DISTINCT** clause is used to remove the duplicate records from many fields in the SELECT statement.

Enter the **SQL statement**:

1. **SELECT** **DISTINCT** city, state
2. **FROM** suppliers
3. **ORDER** **BY** city, state;

**Output:**

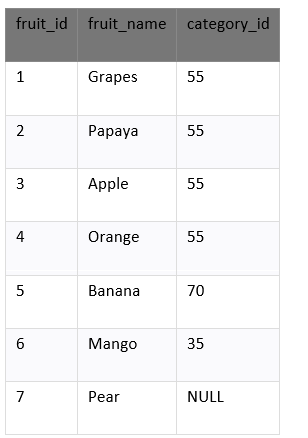
These are 8 records:



The example returns each unique ***city and state*** combination. We see the **Redwood City** and **California**, appears in the result set.

### **Example: DISTINCT Clause handles NULL Values**

The DISTINCT clause considers **NULL** to the unique value in **SQL**. We have a table called ***products*** which contains the below data.



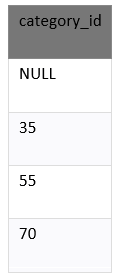
Select the unique values from the field fruit\_id which contains the null value. Enter the below [SQL](https://www.javatpoint.com/sql-tutorial) syntax:

**SELECT** **DISTINCT** fruit\_id

**FROM** fruits

**ORDER** **BY** category\_id;

There are four records selected. These are the results which we see below:



In the above example, the query returns the unique values that are in the ***category\_id*** column. We see by the first row in the result set, **NULL** is an exceptional value which is returned by the **DISTINCT** clause.

# Joining Three or More Tables in SQL

***Joining multiple tables*** in SQL is some tricky task. It can be more difficult if you need to ***join*** more than two tables in single SQL query, we will analyze how to retrieve data from multiple tables using **INNER JOINs**. In this section, we have used two approaches to **join three or more tables in SQL.**

### **Example:**

**We are creating three tables, as follows:**

1. student
2. marks
3. details

### **Table 1: student**

**create** **table** student(s\_id **int** **primary** **key**, s\_name **varchar**(17));

**insert** **into** student **values**(1, 'Jack');

**insert** **into** student **values**(2, 'Rithvik');

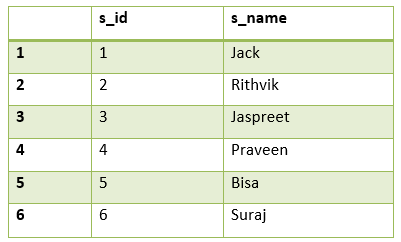
**insert** **into** student **values**(3, 'Jaspreet');

**insert** **into** student **values**(4, 'Praveen');

**insert** **into** student **values**(5, 'Bisa');

**insert** **into** student **values**(6, 'Suraj');

**STUDENT TABLE**



In the above table **s\_id** is the primary key.

### **Table 2: marks**

**create** **table** marks(school\_id **int** **primary** **key**, s\_id **int**, score **int**, status **varchar**(20));

**insert** **into** marks **values**(1004, 1, 23, 'fail');

**insert** **into** marks **values**(1008, 6, 95, 'pass');

**insert** **into** marks **values**(1012, 2, 97, 'pass');

**insert** **into** marks **values**(1016, 7, 67, 'pass');

**insert** **into** marks **values**(1020, 3, 100, 'pass');

**insert** **into** marks **values**(1025, 8, 73, 'pass');

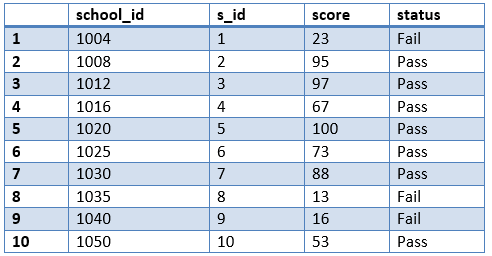
**insert** **into** marks **values**(1030, 4, 88, 'pass');

**insert** **into** marks **values**(1035, 9,  13, 'fail');

**insert** **into** marks **values**(1040, 5,  16, 'fail');

**insert** **into** marks **values**(1050, 10, 53, 'pass');

**MARKS TABLE**



In the above table, **school\_id is primary key and s\_id is the foreign key.**

### **Table 3: details**

**create** **table** details(address\_city **varchar**(20), email\_ID **varchar**(20),    school\_id **int**, accomplishments **varchar**(50));

**insert** **into** details **values**('Bangalore',  'jsingh@jtp.com',

                                1020, 'ACM ICPC selected');

**insert** **into** details **values**('Hyderabad', 'praveen@jtp.com',

                                1030, 'Geek of the month');

**insert** **into** details **values**('Delhi',     'rithvik@jtp.com',

                                    1012, 'IOI finalist');

**insert** **into** details **values**('Chennai',   'om@jtp.com',

                                 1111, 'Geek of the year');

**insert** **into** details **values**('Banglore', ' suraj@jtp.com',

                                 1008, 'IMO finalist');

**insert** **into** details **values**('Mumbai',    'sasukeh@jtp.com',

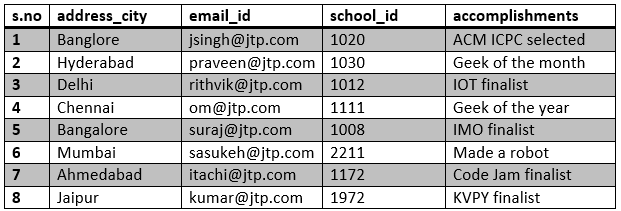
                                  2211, 'Made a robot');

**insert** **into** details **values**('Ahmedabad', 'itachi@jtp.com',

                               1172, 'Code Jam finalist');

**insert** **into** details **values**('Jaipur',    'kumar@jtp.com',

                                   1972, 'KVPY finalist');



In the above table, **school\_id** is the **foreign key**.

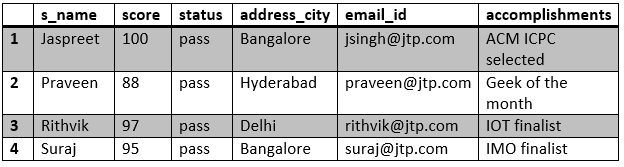
There are two approaches to join three or more tables in [SQL](https://www.javatpoint.com/sql-tutorial):

### **1. Using JOINS in SQL:**

The same logic is applied here which is used to join **two tables** i.e., the **minimum** number of join statements to join **n** tables are **(n-1)**.

1. **select** s\_name, score, status, address\_city, email\_id,
2. accomplishments **from** student s **inner** join mark m **on**
3. s.s\_id = m.s\_id **inner** join details d **on**
4. d.school\_id = m.school\_id;

**Output:**



### **2. Using the Parent-child Relationship:**

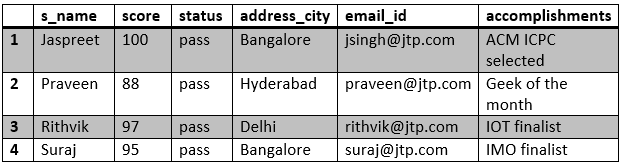
In the parent-child relationship, we use [**where** clause](https://www.javatpoint.com/sql-where) to join two or more tables. Create column **X** as a primary key in one table and a foreign key in another table

Look at the tables which are created:  
**s\_id** is the **primary key** in the student table and **foreign key** in the marks table. **(student (parent) - marks(child))**.  
**school\_id** is the **primary key** in the marks table and **foreign key** in the student table. **(marks(parent) - details(child))**.

**Query:**

1. **select** s\_name, score, status, address\_city,
2. email\_id, accomplishments **from** student s,
3. marks m, details d **where** s.s\_id = m.s\_id and
4. m.school\_id = d.school\_id;

**Output:**



# How to create functions in SQL?

**SQL** has many **built-in functions** for performing the calculation of data. **SQL** provides **built-in** functions to perform the **operations**. Some useful functions of SQL are performing the **mathematical calculations, string concatenation** and **sub-string** etc.

**SQL functions are divided into two parts:**

1. **Aggregate Functions**
2. **Scalar Functions**

## **SQL Aggregate Functions**

SQL **Aggregate** functions return a single value which is calculated from the values.

* **AVG():** It returns the average value of the column.
* **COUNT():** It returns the number of rows in the table.
* **FIRST():** It returns the first value of the column.
* **LAST():** It returns the last value
* **MAX():** It returns the largest value of the column.
* **MIN():** It returns the smallest value of the column.
* **SUM():** It returns the sum of rows of the table.

## **SQL Scalar functions**

SQL Scalar functions returns the single value according to the input value.

### **Scalar functions:**

* **UCASE():** It converts the database field to uppercase.
* **LCASE():** It converts a field to lowercase.
* **MID():** It extracts characters from the text field.
* **LEN():** It returns the length of a text field.
* **ROUND():** It rounds a numeric field to the number of decimals.
* **NOW():** It returns the current date and time.
* **FORMAT():** It formats how a field is to be displayed.

### **Aggregate Functions**

The aggregate functions return a **single value** after performing calculations on the group of values. Some of **Aggregate functions** are explained below.

### **AVG Function**

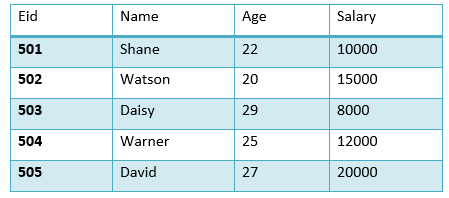
**AVG ()** returns the average value of the database after calculating the values in **numeric** column.

**Syntax :**

1. SELECT AVG(column\_name) FROM table\_name

**Using AVG() function**

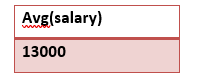
Consider the following **Emp** table:



The following [SQL](https://www.javatpoint.com/sql-tutorial) calculates the average salary of the employees.

1. SELECT avg(salary) from Emp;

**Result:**



## **COUNT() Function**

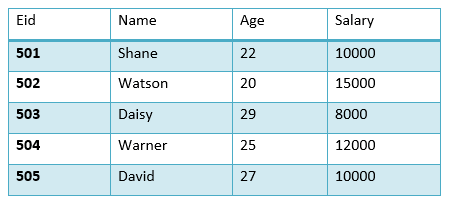
Count returns the number of rows which are present in the database, and either it is based on the condition or without condition.

Its basic **syntax** is,

1. SELECT COUNT(column\_name) FROM table-name

**Using COUNT() function**

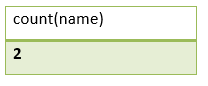
Consider the following **Emp** table:



SQL query to count the number of rows that satisfies the condition.

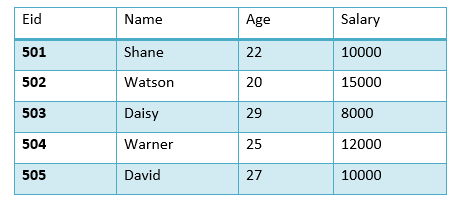
1. SELECT COUNT(name) FROM Emp WHERE salary = 10000;

**Output:**



**Example of COUNT (distinct)**

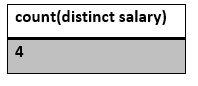
Consider the following **Emp** table:



The SQL query is:

1. SELECT COUNT(DISTINCT salary) FROM emp;

**Output:**



### **FIRST() Function**

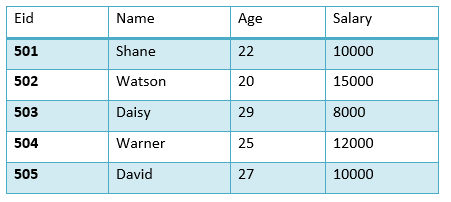
The function returns the first value of the specified column.

**Syntax:**

1. SELECT FIRST(column\_name) FROM table-name;

**Using FIRST() function**

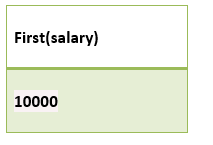
Consider the following **Emp** table:



The SQL query will be:

1. SELECT FIRST(salary) FROM EMP;

**Output:**



### **LAST() Function**

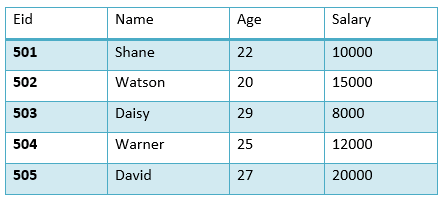
The LAST function returns the return last value of the selected column.

Syntax of the LAST function is:

1. SELECT LAST(column\_name) FROM table-name;

**Using LAST() function**

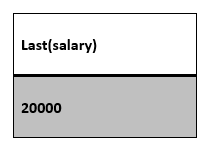
Consider the following **Emp** table:



SQL Query is:

1. SELECT LAST(salary) FROM emp;

**Output:**



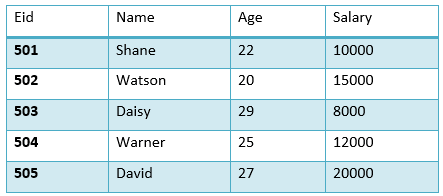
### **MAX() Function**

MAX() function returns the maximum value from the selected column of the table.

**Syntax:**

1. SELECT MAX(column\_name) from table-name;

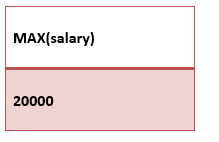
Consider the following **Emp** table:



The following SQL query fetch the maximum salary.

1. SELECT MAX(salary) FROM emp;

**Output:**



## **MIN() Function**

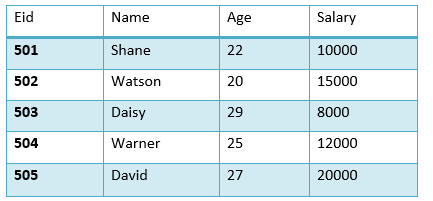
MIN function returns the minimum value of selected column.

**The syntax** of MIN function:

1. SELECT MIN(column\_name) from table-name;

**Using MIN () function**

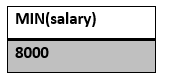
Consider the below **Emp** table:



SQL query to find the minimum salary:

1. SELECT MIN(salary) FROM emp;

**Output:**



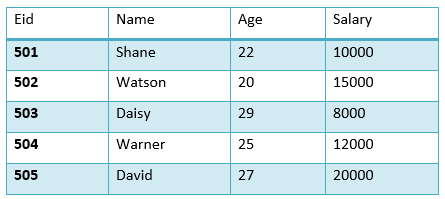
## **SUM() Function**

**SUM ()** function returns the total of the specified columns.

The syntax for **SUM:**

1. SELECT SUM (column\_name) from table-name;

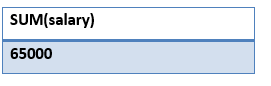
See the following **Emp** table



Sum of salaries are:

1. SELECT SUM(salary) FROM emp;

**Output:**



## **Scalar Functions**

Scalar functions return a single value from an input value. Some of the Scalar functions are given below:

### **UCASE () Function**

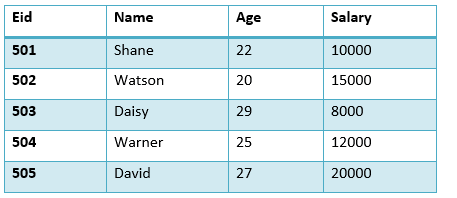
**UCASE ()** converts the value of the string column into the **Uppercase (Capital)** characters.

**Syntax**

1. SELECT UCASE(column\_name) from table-name;

**Using UCASE() function**

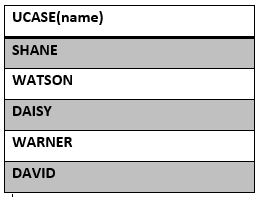
Consider the below **Emp** table:



SQL query of UCASE:

1. SELECT UCASE(name) FROM emp;

Result:



## **LCASE() Function**

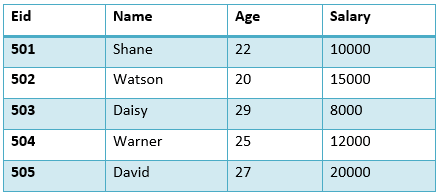
LCASE() function is used to convert the value of string columns to Lowercase.

**The syntax** for **LCASE:**

1. SELECT LCASE(column\_name) FROM table-name;

### **Using LCASE() function**

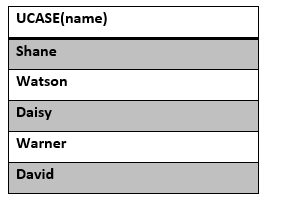
Consider the following **Emp** table



SQL query for converting the string value to **Lowercase:**

1. SELECT LCASE(name) FROM emp;

**Output:**



## **MID() Function**

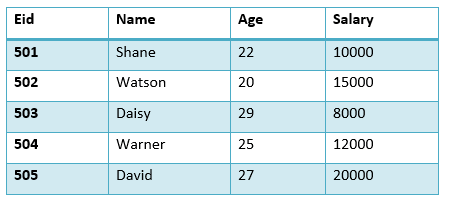
MID() function is used to extract substrings from column values in the table.

The syntax for the MID function is:

1. SELECT MID(column\_name, start, length) from table-name;

**Using MID() function**

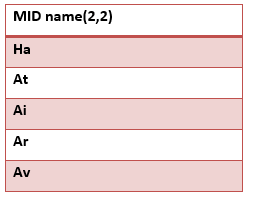
Consider the following **Emp** table:



The following SQL query returns the substring start from the second character.

1. SELECT MID(name,2,2) FROM emp;

**Output:**



## **ROUND() Function**

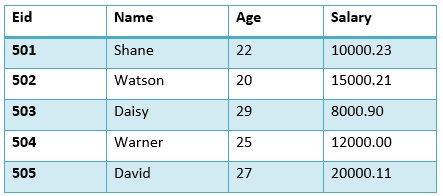
The ROUND() function is used to round a numeric field to a number of the nearest integer. It is used for decimal point.

**Syntax:**

1. SELECT ROUND(column\_name, decimals) from table-name;

**Using ROUND() function**

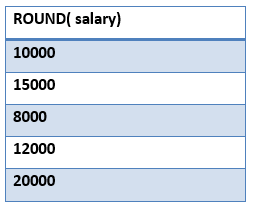
Consider the following **Emp** table:



The following SQL query rounds the amount of salary column.

1. SELECT ROUND(salary) from emp;

**Output:**



# How to run SQL Script?

### **What is SQL Script?**

The SQL script is a set of commands that saved as a file in **SQL Scripts** and it contain one or more SQL statements. We use **SQL Scripts** to **create, run, edit, view** or **delete** the script files.

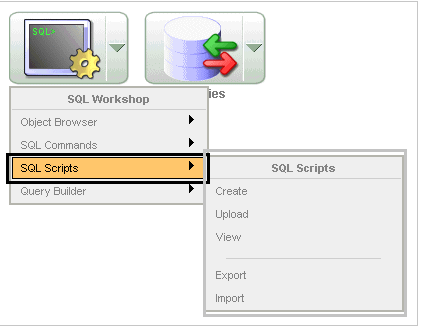
We must remember the following points while using the SQL Scripts:

* In SQL Scripts, SQL **\* Plus** commands are ignored at the run time.
* There is no any interaction between **SQL Scripts** and **SQL commands in SQL scripts**.
* We **cut** and **paste** SQL commands from **SQL script** editor.
* SQL script does not support the bind variables.

## **Accessing the SQL Scripts**

1. First, login to the Workstation.

2. To view the [SQL](https://www.javatpoint.com/sql-tutorial) Scripts page, do the following:

* Click on the **SQL Workshop** icon and **drill-out** the **SQL** page.
* Click on the down arrow to the right of the **SQL Workshop** icon to see the **drop-down** menu after choosing the **SQL Scripts** option.  
  

#### **Note: The document used the navigation path (drill-down approach) when we are explaining the navigation for the consistency.**

### **About the SQL Scripts Page**

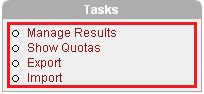
It displays the **SQL Scripts** that are created by the user. We control the appearance of the page by selecting it from the view list.

The SQL Scripts has **three** controls:

* **Script.** It finds the script by entering the partial name or script name in the script field. It displays the rows by selecting from the display list.
* **Owner.** Find the owner of the script you want to see by going to the owner's name and clicking on the owner's name.
* **Opinion.** Opinion changes the appearance of **SQL Scripts** page by selecting the view list.
  + Symbol displays every script as an icon that is identified by script name. Click on the Show Results checkbox to show additional results for icons that are determined by the script name.
  + Description: The description displays each script in one line. Every line has a checkbox to enable the selection process of script. It allows the script to load **script name, script editor**, and **owner of the script**. When the script updated in bytes, the text is linked in the run results, and an icon to **enable** the script to run.

**About the Tasks List**

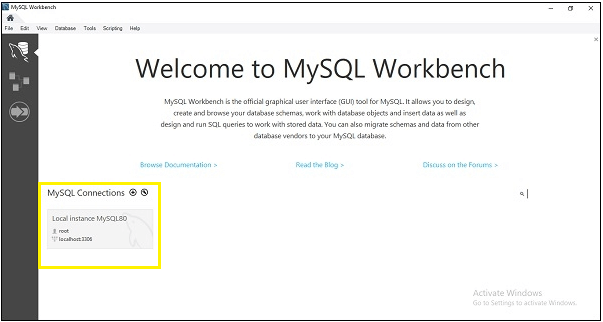
The tasks list is displayed on the right side of the page.



The Task List has these points:

* **Task List manages** the **results** and **enables the view, search**, and **display** the **results**.
* **Show Quota:** The script quota shows the maximum size of result.
  + Export enables the SQL repository to import the SQL Scripts in the workspace. The scripts are encoded in a single script and is written in local file system. By default export script is **"workspace\_name\_script / sql"**.
  + Import enables us to import scripts exported by many workspaces.
  + The export script accessible on local file system to import.

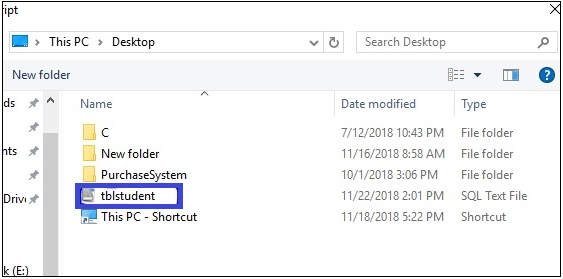
Use [MySQL **workbench**](https://www.javatpoint.com/mysql-workbench) to run the **SQL script**.



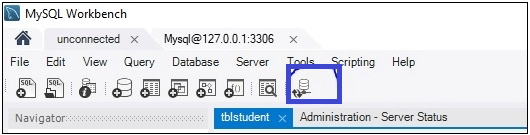
To open the SQL Script, go to **File -> Open**

Alternatively, use the shortcut key **Ctrl+Shift+O** to open the SQL Script.

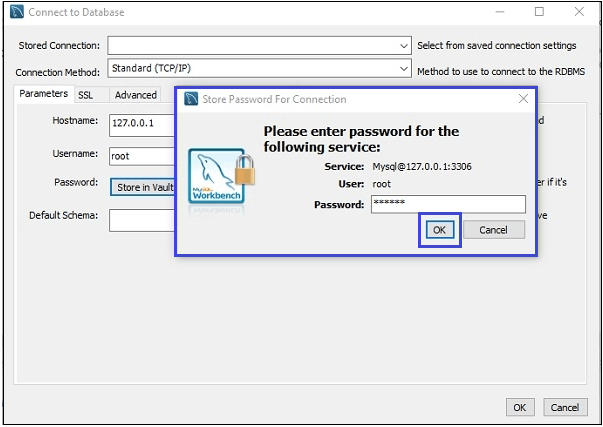
Import enables the import scripts exported by the workspace or different workspaces. Import only imports the scripts encoded in export scripts created by using export. The export script will accessible on the local file system for importing.



After browsing **.sql files**, we select the option **Reconnect to database**, as we have highlighted in the following image.

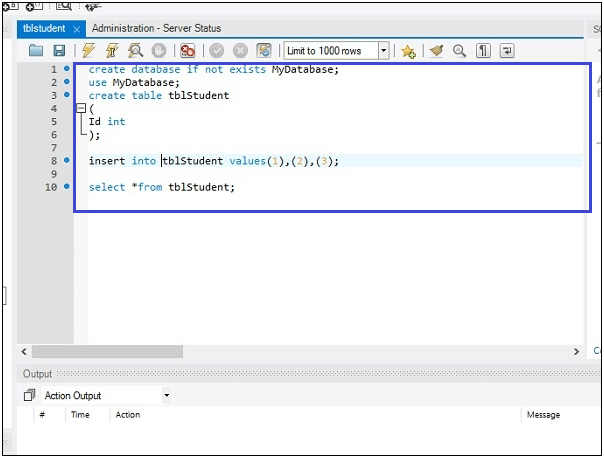


It asks for a password to connect with [**MySQL**](https://www.javatpoint.com/mysql-tutorial).



#### **Note: Press the OK button twice to connect with MySQL**

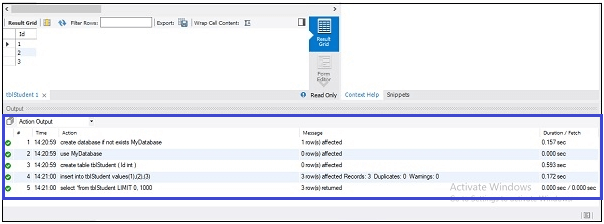
After the above process, the SQL files **tblstudent** appears on the screen that we have uploaded before.



After that, we execute the script by clicking on the symbol highlighted in the following image.

How to run SQL Script

After that, we get the following output:



## **Deleting Script in the Script Editor**

To delete the script, follow the steps given below:

1. First, click on **SQL Workshop** and then **SQL Scripts** on the Workspace **home page**. The **SQL Scripts** page appears in the screen.
2. Open the **script** which we want to delete in the **Script Editor**.
3. Click on the **Delete** button to remove the script from the **Script Repository**. We are prompted to confirm the action before the script is deleted.

The message **Script(s) deleted** shows above the updated list of **scripts**.

### **Copying a SQL Script**

To save the script, we copy the script in the script repository. Follow the steps given below, to copy a SQL Script.

1. Click on the SQL Workshop and open **SQL Script** on the workspace homepage.
2. Load the script to be copied to the editor.
3. Enter the name of a copied script into script field.
4. Click on Save button to save the copy of the script in the script repository field.

# How to Delete Duplicate Rows in SQL?

In this section, we learn different ways to delete duplicate rows in **MySQL and Oracle**. If the [SQL](https://www.javatpoint.com/sql-tutorial) table contains duplicate rows, then we have to remove the duplicate rows.

## **Preparing sample data**

The script creates the table named **contacts**.

DROP TABLE IF EXISTS contacts;

CREATE TABLE contacts (

id INT PRIMARY KEY AUTO\_INCREMENT,

first\_name VARCHAR(30) NOT NULL,

last\_name VARCHAR(25) NOT NULL,

    email VARCHAR(210) NOT NULL,

    age VARCHAR(22) NOT NULL

);

In the above table, we have inserted the following data.

INSERT INTO contacts (first\_name,last\_name,email,age)

VALUES ('Kavin','Peterson','kavin.peterson@verizon.net','21'),

       ('Nick','Jonas','nick.jonas@me.com','18'),

       ('Peter','Heaven','peter.heaven@google.com','23'),

       ('Michal','Jackson','michal.jackson@aol.com','22'),

       ('Sean','Bean','sean.bean@yahoo.com','23'),

       ('Tom ','Baker','tom.baker@aol.com','20'),

       ('Ben','Barnes','ben.barnes@comcast.net','17'),

       ('Mischa ','Barton','mischa.barton@att.net','18'),

       ('Sean','Bean','sean.bean@yahoo.com','16'),

       ('Eliza','Bennett','eliza.bennett@yahoo.com','25'),

       ('Michal','Krane','michal.Krane@me.com','25'),

       ('Peter','Heaven','peter.heaven@google.com','20'),

       ('Brian','Blessed','brian.blessed@yahoo.com','20');

       ('Kavin','Peterson','kavin.peterson@verizon.net','30'),

We execute the script to recreate test data after executing a [**DELETE** statement](https://www.javatpoint.com/sql-delete).

The query returns data from the contacts table:

1. SELECT \* FROM contacts
2. ORDER BY email;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| id | first\_name | last\_name | Email | age |
| 7 | Ben | Barnes | ben.barnes@comcast.net | 21 |
| 13 | Brian | Blessed | brian.blessed@yahoo.com | 18 |
| 10 | Eliza | Bennett | eliza.bennett@yahoo.cm | 23 |
| 1 | **Kavin** | **Peterson** | **kavin.peterson@verizon.net** | **22** |
| 14 | **Kavin** | **Peterson** | **kavin.peterson@verizon.net** | **23** |
| 8 | Mischa | Barton | mischa.barton@att.net | 20 |
| 11 | Michal | Krane | michal.Krane@me.com | 17 |
| 4 | Michal | Jackson | Michal.jackson@aol.com | 18 |
| 2 | Nick | Jonas | nick.jonas@me.com | 16 |
| 3 | **Peter** | **Heaven** | **Peter.heaven@google.com** | **25** |
| 12 | **Peter** | **Heaven** | **Peter.heaven@google.com** | **25** |
| 5 | **Sean** | **Bean** | **Sean.bean@yahoo.com** | **20** |
| 9 | **Sean** | **Bean** | **Sean.bean@yahoo.com** | **20** |
| 6 | Tom | Baker | tom.baker@aol.com | 30 |

The following SQL query returns the duplicate emails from the contact table:

1. SELECT
2. email, COUNT(email)
3. FROM
4. contacts
5. GROUP BY
6. email
7. HAVING
8. COUNT (email) > 1;

|  |  |
| --- | --- |
| email | COUNT(email) |
| kavin.peterson@verizon.net | 2 |
| Peter.heaven@google.com | 2 |
| Sean.bean@yahoo.com | 2 |

We have three rows with **duplicate** emails.

## **(A) Delete duplicate rows with the DELETE JOIN statement**

DELETE t1 FROM contacts t1

INNERJOIN contacts t2

WHERE

    t1.id < t2.id AND

    t1.email = t2.email;

**Output:**

1. Query OK, three rows affected (0.10 sec)

Three rows had been deleted. We execute the query, given below to finds the **duplicate emails** from the table.

1. SELECT
2. email,
3. COUNT (email)
4. FROM
5. contacts
6. GROUP BY
7. email
8. HAVING
9. COUNT (email) > 1;

The query returns the empty set. To verify the data from the contacts table, execute the following SQL query:

1. SELECT \* FROM contacts;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| id | first\_name | last\_name | Email | age |
| 7 | Ben | Barnes | ben.barnes@comcast.net | 21 |
| 13 | Brian | Blessed | brian.blessed@yahoo.com | 18 |
| 10 | Eliza | Bennett | eliza.bennett@yahoo.cm | 23 |
| 1 | Kavin | Peterson | kavin.peterson@verizon.net | 22 |
| 8 | Mischa | Barton | mischa.barton@att.net | 20 |
| 11 | Micha | Krane | michal.Krane@me.com | 17 |
| 4 | Michal | Jackson | Michal.jackson@aol.com | 18 |
| 2 | Nick | Jonas | nick.jonas@me.com | 16 |
| 3 | Peter | Heaven | Peter.heaven@google.com | 25 |
| 5 | Sean | Bean | Sean.bean@yahoo.com | 20 |
| 6 | Tom | Baker | tom.baker@aol.com | 30 |

The rows **id's 9, 12, and 14** have been deleted. We use the below statement to delete the duplicate rows:

Execute the script for **creating** the contact.

1. DELETE c1 FROM contacts c1
2. INNERJ OIN contacts c2
3. WHERE
4. c1.id > c2.id AND
5. c1.email = c2.email;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| id | first\_name | last\_name | email | age |
| 1 | Ben | Barnes | ben.barnes@comcast.net | 21 |
| 2 | **Kavin** | **Peterson** | **kavin.peterson@verizon.net** | **22** |
| 3 | Brian | Blessed | brian.blessed@yahoo.com | 18 |
| 4 | Nick | Jonas | nick.jonas@me.com | 16 |
| 5 | Michal | Krane | michal.Krane@me.com | 17 |
| 6 | Eliza | Bennett | eliza.bennett@yahoo.cm | 23 |
| 7 | Michal | Jackson | Michal.jackson@aol.com | 18 |
| 8 | **Sean** | **Bean** | **Sean.bean@yahoo.com** | **20** |
| 9 | Mischa | Barton | mischa.barton@att.net | 20 |
| 10 | **Peter** | **Heaven** | **Peter.heaven@google.com** | **25** |
| 11 | Tom | Baker | tom.baker@aol.com | 30 |

## **(B) Delete duplicate rows using an intermediate table**

To delete a duplicate row by using the intermediate table, follow the steps given below:

**Step 1**. Create a new table **structure**, same as the real table:

1. CREATE TABLE source\_copy LIKE source;

**Step 2**. Insert the distinct rows from the original schedule of the database:

1. INSERT INTO source\_copy
2. SELECT \* FROM source
3. GROUP BY col;

**Step 3**. Drop the original table and rename the immediate table to the original one.

1. DROP TABLE source;
2. ALTER TABLE source\_copy RENAME TO source;

For example, the following statements delete the **rows** with **duplicate** emails from the contacts table:

1. -- step 1
2. CREATE TABLE contacts\_temp
3. LIKE contacts;
5. -- step 2
6. INSERT INTO contacts\_temp
7. SELECT \* FROM contacts
8. GROUP BY email;
10. -- step 3
11. DROP TABLE contacts;
13. ALTER TABLE contacts\_temp
14. RENAME TO contacts;

## **(C) Delete duplicate rows using the ROW\_NUMBER() Function**

#### **Note: The ROW\_NUMBER() function has been supported since MySQL version 8.02, so we should check our MySQL version before using the function.**

The following statement uses the **ROW\_NUMBER ()** to assign a sequential integer to every row. If the email is duplicate, the row will higher than one.

1. SELECT id, email, ROW\_NUMBER()
2. OVER (PARTITION BY email
3. ORDER BY email
4. ) AS row\_num
5. FROM contacts;

The following SQL query returns **id list** of the duplicate rows:

1. SELECT id
2. FROM (SELECT id,
3. ROW\_NUMBER() OVER (
4. PARTITION BY email ORDER BY email) AS row\_num
5. FROM
6. contacts
7. ) t
8. WHERE
9. row\_num> 1;

**Output:**

|  |
| --- |
| id |
| 9 |
| 12 |
| 14 |

## **Delete Duplicate Records in Oracle**

When we found the duplicate records in the table, we had to delete the unwanted copies to keep our data clean and unique. If a table has duplicate rows, we can delete it by using the **DELETE** statement.

In the case, we have a column, which is not the part of **group** used to **evaluate** the **duplicate** records in the table.

Consider the table given below:

|  |  |  |
| --- | --- | --- |
| VEGETABLE\_ID | VEGETABLE\_NAME | COLOR |
| 01 | Potato | Brown |
| 02 | Potato | Brown |
| 03 | Onion | Red |
| 04 | Onion | Red |
| 05 | Onion | Red |
| 06 | Pumpkin | Green |
| 07 | Pumpkin | Yellow |

1. -- create the vegetable table
2. CREATE TABLE vegetables (
3. VEGETABLE\_ID NUMBER generated BY DEFAULT AS ID ENTITY,
4. VEGETABLE\_NAME VARCHAR2(100),
5. color VARCHAR2(20),
6. PRIMARY KEY (VEGETABLE\_ID)
7. );
8. -- insert sample rows

INSERT INTO vegetables (VEGETABLE\_NAME,color) VALUES('Potato','Brown');

INSERT INTO vegetables (VEGETABLE\_NAME,color) VALUES('Potato','Brown');

INSERT INTO vegetables (VEGETABLE\_NAME,color) VALUES('Onion','Red');

INSERT INTO vegetables (VEGETABLE\_NAME,color) VALUES('Onion','Red');

INSERT INTO vegetables (VEGETABLE\_NAME,color) VALUES('Onion','Red');

INSERT INTO vegetables (VEGETABLE\_NAME,color) VALUES('Pumpkin','Green');

INSERT INTO vegetables (VEGETABLE\_NAME,color) VALUES('Pumpkin','Yellow');

1. -- query data from the vegetable table
2. SELECT \* FROM vegetables;

Suppose, we want to keep the row with the highest **VEGETABLE\_ID** and delete all other copies.

1. SELECT
2. MAX (VEGETABLE\_ID)
3. FROM
4. vegetables
5. GROUP BY
6. VEGETABLE\_NAME,
7. color
8. ORDER BY
9. MAX(VEGETABLE\_ID);

|  |
| --- |
| MAX(VEGETABLE\_ID) |
| 2 |
| 5 |
| 6 |
| 7 |

We use the **DELETE** statement to delete the rows whose values in the **VEGETABLE\_ID COLUMN** are not the **highest**.

1. DELETE FROM
2. vegetables
3. WHERE
4. VEGETABLE\_IDNOTIN
5. (
6. SELECT
7. MAX(VEGETABLE\_ID)
8. FROM
9. vegetables
10. GROUP BY
11. VEGETABLE\_NAME,
12. color
13. );

Three rows have been deleted.

1. SELECT \*FROM vegetables;

|  |  |  |
| --- | --- | --- |
| VEGETABLE\_ID | VEGETABLE\_NAME | COLOR |
| **02** | Potato | Brown |
| **05** | Onion | Red |
| **06** | Pumpkin | Green |
| **07** |  | Yellow |

If we want to keep the row with the lowest id, use the **MIN()** function instead of the **MAX()** function.

1. DELETE FROM
2. vegetables
3. WHERE
4. VEGETABLE\_IDNOTIN
5. (
6. SELECT
7. MIN(VEGETABLE\_ID)
8. FROM
9. vegetables
10. GROUP BY
11. VEGETABLE\_NAME,
12. color
13. );

The above method works if we have a column that is not part of the group for evaluating duplicate. If all values in the columns have copies, then we cannot use the **VEGETABLE\_ID** column.

Let's drop and create the **vegetable** table with a new structure.

1. DROP TABLE vegetables;
2. CREATE TABLE vegetables (
3. VEGETABLE\_ID NUMBER,
4. VEGETABLE\_NAME VARCHAR2(100),
5. Color VARCHAR2(20)
6. );

INSERT INTO vegetables (VEGETABLE\_ID,VEGETABLE\_NAME,color) VALUES(1,'Potato','Brown');

INSERT INTO vegetables (VEGETABLE\_ID,VEGETABLE\_NAME,color) VALUES(1, 'Potato','Brown');

INSERT INTO vegetables (VEGETABLE\_ID,VEGETABLE\_NAME,color)VALUES(2,'Onion','Red');

INSERT INTO vegetables (VEGETABLE\_ID,VEGETABLE\_NAME,color)VALUES(2,'Onion','Red');

INSERT INTO vegetables (VEGETABLE\_ID,VEGETABLE\_NAME,color) VALUES(2,'Onion','Red');

INSERT INTO vegetables (VEGETABLE\_ID,VEGETABLE\_NAME,color) VALUES(3,'Pumpkin','Green');

INSERT INTO vegetables (VEGETABLE\_ID,VEGETABLE\_NAME,color) VALUES('4,Pumpkin','Yellow');

2. SELECT \* FROM vegetables;

|  |  |  |
| --- | --- | --- |
| VEGETABLE\_ID | VEGETABLE\_NAME | COLOR |
| 01 | Potato | Brown |
| 01 | Potato | Brown |
| 02 | Onion | Red |
| 02 | Onion | Red |
| 02 | Onion | Red |
| 03 | Pumpkin | Green |
| 04 | Pumpkin | Yellow |

In the vegetable table, the values in all columns **VEGETABLE\_ID, VEGETABLE\_NAME**, and color have been copied.

We can use the **rowid**, a locator that specifies where Oracle stores the row. Because the **rowid** is unique so that we can use it to remove the duplicates rows.

1. DELETE
2. FROM
3. Vegetables
4. WHERE
5. rowed NOT IN
6. (
7. SELECT
8. MIN(rowid)
9. FROM
10. vegetables
11. GROUP BY
12. VEGETABLE\_ID,
13. VEGETABLE\_NAME,
14. color
15. );

The query verifies the deletion operation:

1. SELECT \* FROM vegetables;

|  |  |  |
| --- | --- | --- |
| VEGETABLE\_ID | VEGETABLE\_NAME | COLOR |
| 01 | Potato | Brown |
| 02 | Onion | Red |
| 03 | Pumpkin | Green |
| 04 | Pumpkin | Yellow |

# Nth Highest salary

Finding the Nth highest salary( 2nd, 3rd, or nth highest) in a table is the most important and common question asked in various interviews.

Here we will show you the best and easiest way to write [SQL](https://www.javatpoint.com/sql-tutorial) queries to find nth highest salary in a table.

To show this, we are using Table **Emp** having employee details like EID, ENAME, and SALARY. Data present in the Emp Table is shown below:

**Table Name: Emp**

|  |  |  |
| --- | --- | --- |
| **EID** | **ENAME** | **SALARY** |
| 1 | Amit | 20000 |
| 2 | Bhaskar | 30000 |
| 3 | Chandan | 25000 |
| 4 | Durgesh | 28000 |
| 5 | Parul | 30000 |
| 6 | Garima | 25000 |
| 7 | Akshita | 28000 |
| 8 | Sonu | 40000 |
| 9 | Ravi | 37000 |
| 10 | Rajesh | 320000 |

### **The SQL query to calculate second highest salary in database table name as Emp**

**Query: 1**

1. SQL> **select** **min**(salary) **from**
2. (**select** **distinct** salary **from** emp **order** **by** salary **desc**)
3. **where** rownum < 3;
5. In **order** **to** calculate the **second** highest salary use rownum < 3
6. In **order** **to** calculate the third highest salary use rownum < 4

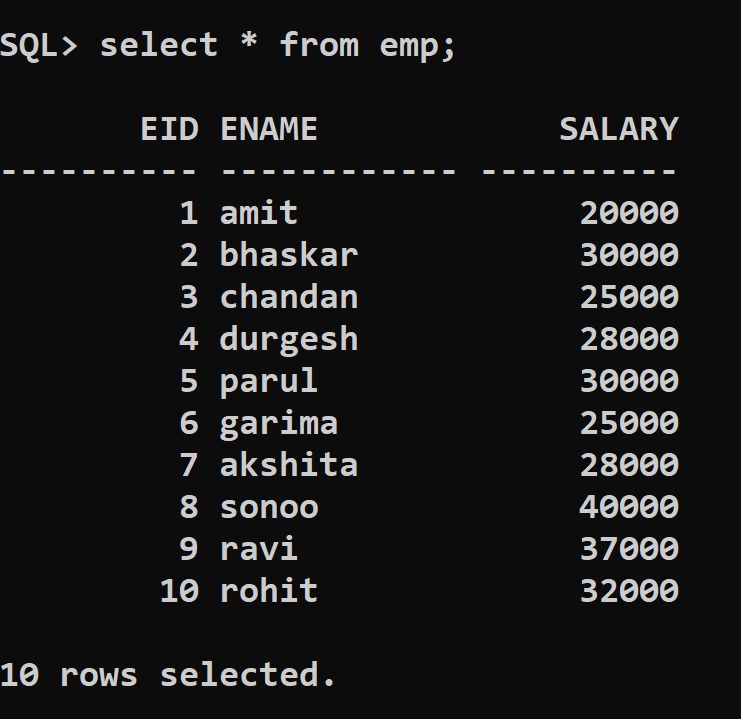
**Output:**

MIN(SALARY)

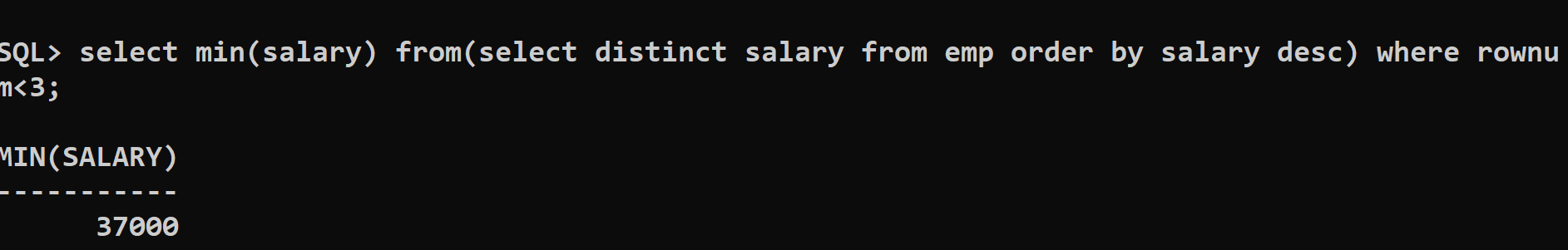
-----------

37000

### **The Structure and data in Emp Table**



### **The Output Screen**



### **Let us understand how this query is working:**

1. **As** this query **is** nested query lets understand each part step **by** step:
2. Step 1: **First** this part **of** the query will get executed **then** the outer part **of** the query will act **on** the result produced **by** this query:            **select** **distinct** salary **from** emp **order** **by** salary **desc**
3. **As** you can see that few employees are getting the same salary (**for** example Bhaskar, Parul, and Chandan, Garima are getting the same salary, therefore we have used **distinct** keyword, **order** **by** salary **desc** will arrange salary in descending **order**.
4. The **output** **of** **select** **distinct** salary **from** emp **order** **by** salary **desc**
5. SALARY
6. ----------
7. 40000
8. 37000
9. 32000
10. 30000
11. 28000
12. 25000
13. 20000
15. Step 2: SQL> **select** **min**(salary) **from**
16. (**select** **distinct** salary **from** emp **order** **by** salary **desc**)
17. **where** rownum < 3;
18. In step two we are applying the outer part **of** the nested query **into** the results we obtained **from** the internal query.
20. **Select** **min**(salary) **from**: will **select** **min** salary **as** 20000 which **is** not the **second**-highest salary, because **of** which we have used rownum < 3, the rownum < 3 will **only** give the number **of** **rows** **from** the **top** which **is** less than 3 i.e. 2.
21. The **output** **of** rownum< 3 will be:
22. SALARY
23. ----------
24. 40000
25. 37000
26. Step 3: Now **select** **min**(salary).
27. The **output** will be:
28. SALARY
29. ----------
30. 37000
31. 37000 **is** the **second**-highest salary.
33. Simillarly **to** find:
34. **To** find 3rd highest salary **set** rownum < 4
35. **To** find 4th highest salary **set** rownum < 5
36. And so **on**...

### **The SQL query to calculate second highest salary in database table name as Emp**

**Query: 2**

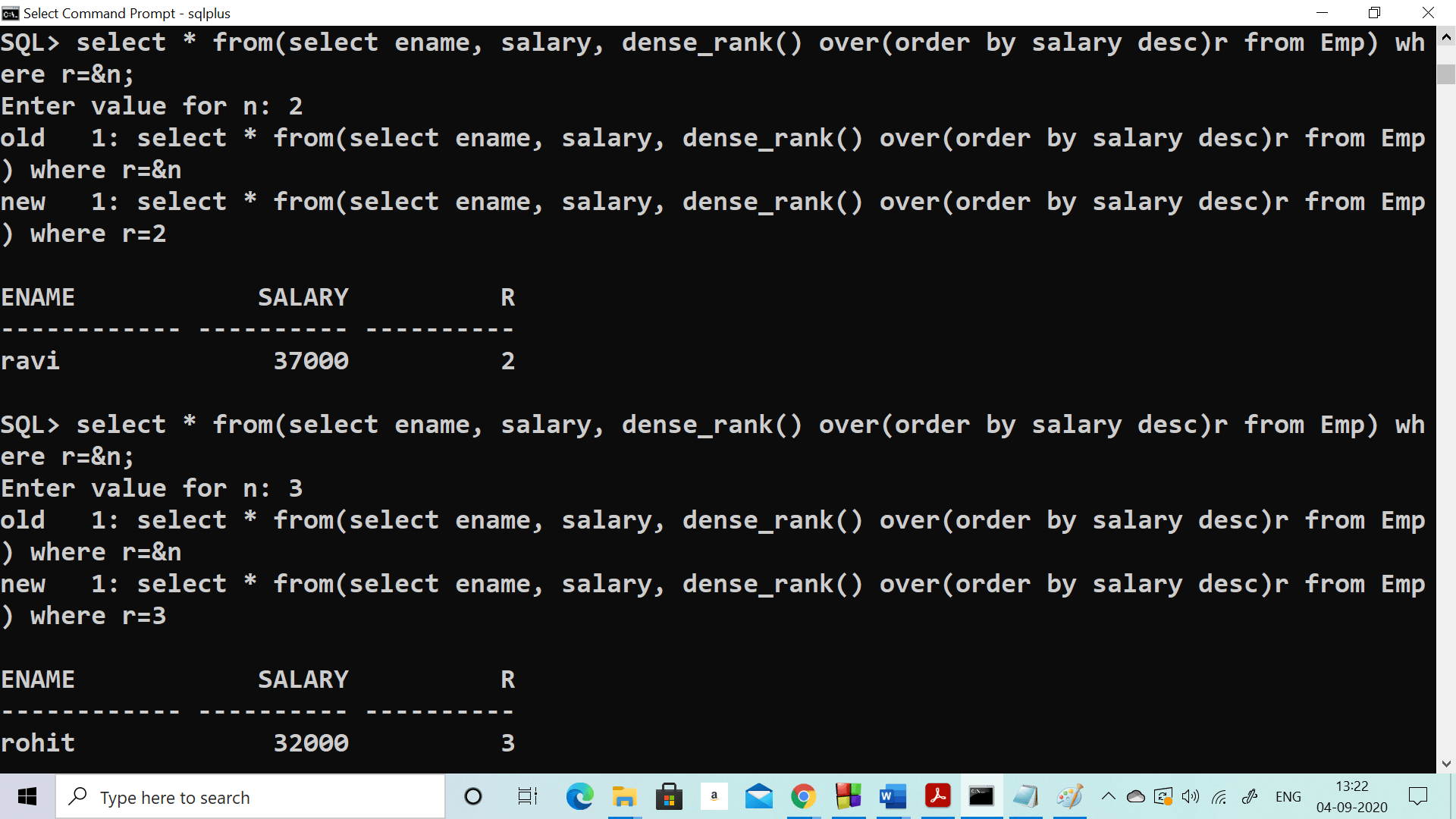
1. **select** \* **from**(
2. **select** ename, salary, dense\_rank()
3. over(**order** **by** salary **desc**)rank **from** Emp)
4. **where** rank = & num;
6. In **order** **to** calculate the **second** highest salary use num = 2
7. In **order** **to** calculate the third highest salary use num = 3
8. and so **on**...

**Output:**

ENAME SALARY Rank

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

ravi 37000 2



### **Let us understand how this query is working:**

1. **As** this query **is** nested query lets understand each part step **by** step:
2. Step 1: **First** this part **of** the query will get executed **then** the outer part **of** the query will act **on** the result produced **by** this query :
3. **select** ename, salary, dense\_rank() over(**order** **by** salary **desc**)rank **from** Emp
4. dense\_rank() calculates the rank **of** each row in an ordered **group** **of** **rows** and **returns** the rank **as** a number. The ranks start **from** **integer** 1 and so **on** in a consecutive manner.
6. If we talk about the above SQL query, based **on** the salary **of** the emp **table** the rank **is** returned. In case two or more than two **rows** have an equal salary, it assigns an equal rank **to** all the **rows**.
8. **As** you can see that few employees are getting the same salary(**for** example Bhaskar, Parul, and Chandan, Garima are getting the same salary, therefore we have used dense\_rank(), **order** **by** salary **desc** will arrange salary in descending **order**.



13. **Output** **of** : **select** ename, salary, dense\_rank() over(**order** **by** salary **desc**)rank **from** Emp
14. ENAME            SALARY       RANK
15. ------------ ---------- ----------
16. sonoo             40000          1
17. ravi              37000          2
18. rohit             32000          3
19. bhaskar           30000          4
20. parul             30000          4
21. akshita           28000          5
22. durgesh           28000          5
23. garima            25000          6
24. chandan           25000          6
25. amit              20000          7
27. you can see **from** the **output** that Bhaskar, Parul are getting rank 4 **as** they both are getting equal salary) and Akshita, Durgesh are getting rank 4(**as** they both are getting equal salary)  similarly Garima and Chandan.
29. Step 2: SQL> **select** \* **from**(
30. **select** ename, salary, dense\_rank()
31. over(**order** **by** salary **desc**)rank **from** Emp)
32. **where** r = &n;
33. In step 2 we are applying the outer part **of** the nested query **into** the result we obtained **from** the internal query.
35. **Select** \* **from**: will **select** all the **rows** which are not the **second**-highest salary, because **of** which we have used r = &n, the r = &n will **only** give the matching **rows** according **to** the **values** entered **by** the user **for** n. if n = 2 resultant will be
36. **Output** **for** n = 2 will be:
37. Enter the value **for** n: 2
38. old   1: **select** \* **from**(**select** ename, salary, dense\_rank() over(**order** **by** salary **desc**)r **from** Emp) **where** r=&n
39. new   1: **select** \* **from**(**select** ename, salary, dense\_rank() over(**order** **by** salary **desc**)r **from** Emp) **where** r=2
41. ENAME            SALARY          R
42. ------------ ---------- ----------
43. ravi              37000          2
45. **To** Find fourth highest salary:
46. Enter value **for** n: 4
47. old   1: **select** \* **from**(**select** ename, salary, dense\_rank() over(**order** **by** salary **desc**)r **from** Emp) **where** r=&n
48. new   1: **select** \* **from**(**select** ename, salary, dense\_rank() over(**order** **by** salary **desc**)r **from** Emp) **where** r=4
50. ENAME            SALARY          R
51. ------------ ---------- ----------
52. bhaskar           30000          4
53. parul             30000          4
55. Similarly, **to** find:
56. **To** find 5th highest salary **set** n = 5
57. **To** find 6th highest salary **set** n = 6
58. And so **on**...

## **The SQL query to calculate second highest salary in database table name as Emp**

Let's say the job is to calculate the Nth highest salary of employee from the above table. The procedure is as follows:

1. First task is to Identify the employee having TOP n non similar(distinct) salary.
2. Calculate the minimum salary among all the salaries resulted from above query, by doing this we get nth highest salary.
3. From the result of above query, identify the details of the employee whose salary is the minimum salary.

**Query No: 3**

1. **select** \* **from** emp **where** salary = ( **select** **min**( salary ) **from** emp
2. **where**  salary IN (**select** **distinct** **TOP** N
3. salary **from** emp **order** **by** salary **desc** )
4. )

The above SQL query will find out the details of the emp with the nth highest salary.

Let's see the working of the above SQL query in detail:

* Consider n = 5.

The processing done by server is that, it starts with most inner query, the query: "**select distinct TOP 5 salary from emp order by salary desc**" will generate following result:

1. •    40000
2. •    37000
3. •    32000
4. •    30000
5. •    28000

* The next outer query is: "select min(salary) from emp where salary IN (the result of a previous SQL query )". This will produce the following result:

1. •    28000

From above result it is verified that the required **Fifth** highest salary is 28000.

* Lastly, the query which is outer most, is: "select \* from emp where salary = result of previous SQL query ". The result of this query will be the details of employees having **Fifth** highest salary.

1. •   \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
2. •   ename             salary
3. •   \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_
4. •    akshita   |     28000
5. •              |

## **Working of query**

As these queries are nested queries so this query involves the use of an inner query. There are two versions of Inner queries. **Correlated** and **Uncorrelated** queries. Uncorrelated queries are those where the inner query can run independently of the outer query, and the correlated query is those where the inner query runs in conjunction with the outer query. The query we took to calculate nth highest salary is an example of a correlated query.

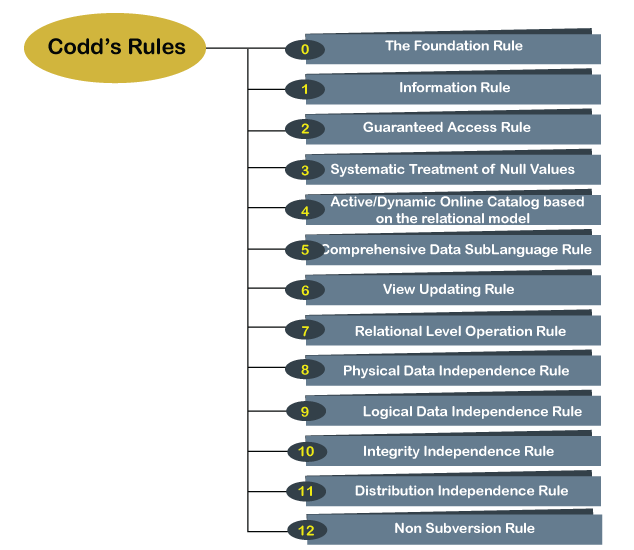
## **Performance analysis of SQL query**

From the above, we have learned that the inner query executes every time, single row of the outer query is processed, this ultimately brings a lot of performance overhead, especially when the number of rows is very large.

To avoid this, it is recommended to use Data Base specific keywords to get the result faster.

# 12 Codd's Rules

Every database has tables, and constraints cannot be referred to as a rational database system. And if any database has only relational data model, it cannot be a [**Relational Database System (RDBMS)**](https://www.javatpoint.com/what-is-rdbms). So, some rules define a database to be the correct RDBMS. These rules were developed by **Dr. Edgar F. Codd (E.F. Codd)** in **1985**, who has vast research knowledge on the Relational Model of database Systems. Codd presents his 13 rules for a database to test the concept of [DBMS](https://www.javatpoint.com/dbms-tutorial) against his relational model, and if a database follows the rule, it is called a **true relational database (RDBMS)**. These 13 rules are popular in RDBMS, known as **Codd's 12 rules**.



### **Rule 0: The Foundation Rule**

The database must be in relational form. So that the system can handle the database through its relational capabilities.

### **Rule 1: Information Rule**

A database contains various information, and this information must be stored in each cell of a table in the form of rows and columns.

### **Rule 2: Guaranteed Access Rule**

Every single or precise data (atomic value) may be accessed logically from a relational database using the combination of primary key value, table name, and column name.

### **Rule 3: Systematic Treatment of Null Values**

This rule defines the systematic treatment of Null values in database records. The null value has various meanings in the database, like missing the data, no value in a cell, inappropriate information, unknown data and the primary key should not be null.

### **Rule 4: Active/Dynamic Online Catalog based on the relational model**

It represents the entire logical structure of the descriptive database that must be stored online and is known as a database dictionary. It authorizes users to access the database and implement a similar query language to access the database.

### **Rule 5: Comprehensive Data SubLanguage Rule**

The relational database supports various languages, and if we want to access the database, the language must be the explicit, linear or well-defined syntax, character strings and supports the comprehensive: data definition, view definition, data manipulation, integrity constraints, and limit transaction management operations. If the database allows access to the data without any language, it is considered a violation of the database.

### **Rule 6: View Updating Rule**

All views table can be theoretically updated and must be practically updated by the database systems.

### **Rule 7: Relational Level Operation (High-Level Insert, Update and delete) Rule**

A database system should follow high-level relational operations such as insert, update, and delete in each level or a single row. It also supports union, intersection and minus operation in the database system.

### **Rule 8: Physical Data Independence Rule**

All stored data in a database or an application must be physically independent to access the database. Each data should not depend on other data or an application. If data is updated or the physical structure of the database is changed, it will not show any effect on external applications that are accessing the data from the database.

### **Rule 9: Logical Data Independence Rule**

It is similar to physical data independence. It means, if any changes occurred to the logical level (table structures), it should not affect the user's view (application). For example, suppose a table either split into two tables, or two table joins to create a single table, these changes should not be impacted on the user view application.

### **Rule 10: Integrity Independence Rule**

A database must maintain integrity independence when inserting data into table's cells using the SQL query language. All entered values should not be changed or rely on any external factor or application to maintain integrity. It is also helpful in making the database-independent for each front-end application.

### **Rule 11: Distribution Independence Rule**

The distribution independence rule represents a database that must work properly, even if it is stored in different locations and used by different end-users. Suppose a user accesses the database through an application; in that case, they should not be aware that another user uses particular data, and the data they always get is only located on one site. The end users can access the database, and these access data should be independent for every user to perform the SQL queries.

### **Rule 12: Non Subversion Rule**

The non-submersion rule defines RDBMS as a [SQL](https://www.javatpoint.com/sql-tutorial) language to store and manipulate the data in the database. If a system has a low-level or separate language other than SQL to access the database system, it should not subvert or bypass integrity to transform data.

# SQL EXCEPT

Usually, we use a JOIN clause to get the combined result from more than one table. Sometimes, we need a result set that contains records from one table but not available in the other table. In that case, SQL provides an EXCEPT clause/operator.

The EXCEPT clause in [SQL](https://www.javatpoint.com/sql-tutorial) is widely used to filter records from more than one table. This statement first combines the two [SELECT statements](https://www.javatpoint.com/sql-select) and returns records from the first SELECT query, which aren't present in the second SELECT query's result. In other words, it retrieves all rows from the first SELECT query while deleting redundant rows from the second.

This statement behaves the same as the minus operator does in mathematics. This article will illustrate how to use the SQL EXCEPT clause with the help of basic examples.

### **Rules for SQL EXCEPT**

We should consider the following rules before using the EXCEPT statement in SQL:

* In all SELECT statements, the number of columns and orders in the tables must be the same.
* The corresponding column's data types should be either the same or compatible.
* The fields in the respective columns of two SELECT statements cannot be the same.

### **SQL EXCEPT Syntax**

The following syntax illustrates the use of EXCEPT clause:

1. **SELECT** column\_lists **from** table\_name1
2. **EXCEPT**
3. **SELECT** column\_lists **from** table\_name2;

#### **NOTE: It is to note that MySQL does not support EXCEPT clause. So here we are going to use the PostgreSQL database to explain SQL EXCEPT examples.**

The below image explains the working of EXCEPT operation in the two tables T1 and T2:

#### **SQL EXCEPT**

**Illustration:**

#### **Table T1 includes data 1, 2, and 3.**

#### **Table T2 includes data 2, 3, and 4.**

When we execute the EXCEPT query on these tables, we will get 1, which is unique data from the T1, and it will not found in the T2.

### **SQL EXCEPT Example**

Let us first create two tables using the following scripts:

**Table: Customer**

1. **CREATE** **TABLE** **public**.customer
2. (
3. id **integer** NOT NULL,
4. **name** text **COLLATE** pg\_catalog."default" NOT NULL,
5. age **integer** NOT NULL,
6. salary **real** NOT NULL
7. )

**Table: Orders**

1. **CREATE** **TABLE** **public**."orders"
2. (
3. order\_id **integer** NOT NULL,
4. **date** **date** NOT NULL,
5. cust\_id **integer** NOT NULL,
6. amount **real** NOT NULL,
7. **CONSTRAINT** "order\_pkey" **PRIMARY** **KEY** ("order\_id")
8. )

Next, we will insert some records into both tables as follows:

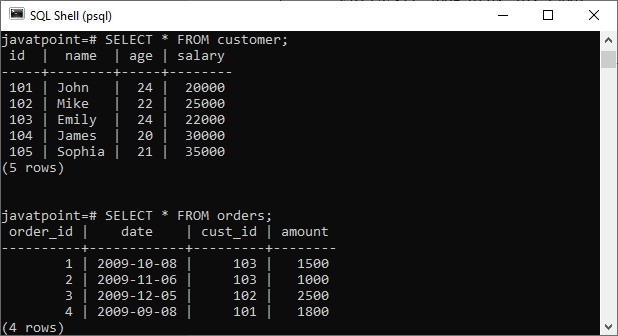
**Table: customer**

1. **INSERT** **INTO** **public**.customer(
2. id, **name**, age, salary)
3. **VALUES** (101, 'John', 24, 20000)
4. (102, 'Mike', 22, 25000),
5. (103, 'Emily', 24, 22000),
6. (104, 'James', 20, 30000),
7. (105, 'Sophia', 21, 35000);

**Table: orders**

1. **INSERT** **INTO** **public**.orders(
2. order\_id, **date**, cust\_id, amount)
3. **VALUES** (1, '2009-10-08', 103, 1500),
4. (2, '2009-11-06', 103, 1000),
5. (3, '2009-12-05', 102, 2500),
6. (4, '2009-09-08', 101, 1800);

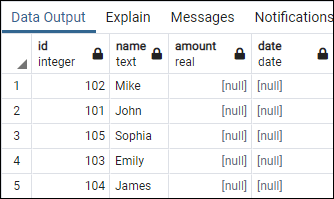
Next, we will use a SELECT statement to verify the records. See the below image:



Let's look into an example for SQL EXCEPT using these tables. Suppose we want to join these tables in our SELECT statement as shown below:

1. **SELECT**  id, **name**, amount, **date**
2. **FROM** customer
3. LEFT JOIN orders
4. **ON** customer.id = orders. order\_id
5. **EXCEPT**
6. **SELECT**  id, **name**, amount, **date**
7. **FROM** customer
8. RIGHT JOIN orders
9. **ON** customer.id = orders. order\_id;

It will produce the below output:

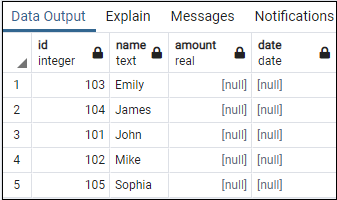


### **EXCEPT with ORDER BY Clause**

If we want to sort the result set obtained by the EXCEPT operator, we need to add the [ORDER BY clause](https://www.javatpoint.com/sql-order-by) in the query. For example, the following example joins both tables and sorts the result set by their name in ascending order:

1. **SELECT**  id, **name**, amount, **date**
2. **FROM** customer
3. LEFT JOIN orders
4. **ON** customer.id = orders. order\_id
5. **EXCEPT**
6. **SELECT**  id, **name**, amount, **date**
7. **FROM** customer
8. RIGHT JOIN orders
9. **ON** customer.id = orders. order\_id
10. **ORDER** **BY** **name**;

It will produce the below output:



### **EXCEPT statements in a single table**

Generally, we use the EXCEPT statements in two tables, but we can also use them to filter records from a single table. For example, the following EXCEPT statement will return all the records from the customer table where the age is greater than 21:

1. **SELECT** id, **name**, age, salary **FROM** customer
2. **EXCEPT**
3. **SELECT** id, **name**, age, salary **FROM** customer **WHERE** age > 21;

In this script, the first SELECT query returns all records from the customer table, and the second query all those records whose age is greater than 21. Next, the EXCEPT statement filters the records using both SELECT statements and returns only those rows whose age is greater than 21.

### **How is EXCEPT different from NOT IN Clause?**

EXCEPT is different from the NOT IN clause in the following manner:

* EXCEPT clause removes all duplicates in the result, set automatically whereas NOT IN does not remove duplicate records.
* EXCEPT clause can perform comparison in single or multiple columns. Whereas the NOT IN clause can perform comparison in a single column only.

# Types of SQL JOIN

## **SQL JOIN**

A [SQL Join](https://www.javatpoint.com/sql-join) is used to fetch or combine data (rows or columns) from two or more tables based on the defined conditions.

**Table 1: Order**

|  |  |  |  |
| --- | --- | --- | --- |
| **OrderID** | **CustomerID** | **OrderName** | **ProductName** |
| 12025 | 101 | Peter | ABC |
| 12030 | 105 | Robert | XYX |
| 12032 | 110 | James | XYZ |
| 12034 | 115 | Andrew | PQR |
| 12035 | 120 | Mathew | AAA |

**Table 2: Customer**

|  |  |  |
| --- | --- | --- |
| **CustomerID** | **CustomerName** | **Country** |
| 100 | Messy | Maxico |
| 101 | Prince | Taiwan |
| 103 | Maria Fernandez | Turkey |
| 105 | Jasmine | Paris |
| 110 | Faf Weasel | Indonesia |
| 120 | Romen Rocket | Russia |

Now, we have two tables **Order** and the **Customer**. There is a **CustomerID** column common in both tables. So, write the SQL query to define the general relationship to select the matches' records from both tables.

1. Select Order.OrderID, Customer.CustomerName, Customer.Country, Order.ProductName from Order INNER JOIN Customer ON Order.CustomerID = Customer.CustomerID;

After executing the above [SQL](https://www.javatpoint.com/sql-tutorial) queries, it produces the following output:

|  |  |  |  |
| --- | --- | --- | --- |
| **OrderID** | **CustomerName** | **Country** | **ProductName** |
| 12025 | Prince | Taiwan | ABC |
| 12030 | Jasmine | Paris | XYX |
| 12032 | Faf Weasel | Indonesia | XYZ |
| 12035 | Romen Rocket | Russia | AAA |

## **Types of SQL Join**

There are different types of joins used in SQL:

1. [Inner Join / Simple Join](https://www.javatpoint.com/types-of-sql-join#Inner)
2. [Left Outer Join / Left Join](https://www.javatpoint.com/types-of-sql-join#Left)
3. [Right Outer Join / Right Join](https://www.javatpoint.com/types-of-sql-join#Right)
4. [Full Outer Join](https://www.javatpoint.com/types-of-sql-join#Full)
5. [Cross Join](https://www.javatpoint.com/types-of-sql-join#Cross)
6. [Self Join](https://www.javatpoint.com/types-of-sql-join#Self)

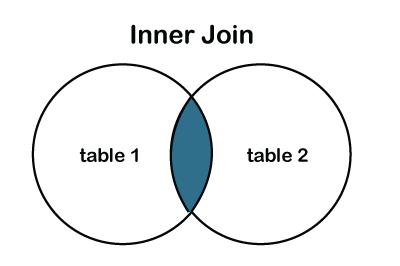
### **Inner Join**

The inner join is used to select all matching rows or columns in both tables or as long as the defined condition is valid in SQL.

**Syntax:**

1. Select column\_1, column\_2, column\_3 FROM table\_1 INNER JOIN table\_2 ON table\_1.column = table\_2.column;

We can represent the inner join through the Venn diagram, as follows:



**Table 1: Students**

|  |  |  |  |
| --- | --- | --- | --- |
| **Student\_ID** | **StudentName** | **Subject** | **TeacherID** |
| 101 | Alexandra | Computer Science | T201 |
| 102 | Charles | Economics | T202 |
| 103 | Tom Cruise | Computer Science | T201 |
| 104 | Aron Finch | Electronics | T203 |
| 105 | Siemen Bajoff | Web designing | T204 |
| 106 | Christopher | English Literature | T205 |
| 107 | Denim | Fashion Designer | T206 |

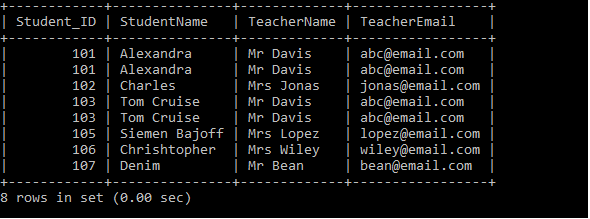
**Table 2: Teachers**

|  |  |  |
| --- | --- | --- |
| **TeacherID** | **TeacherName** | **TeacherEmail** |
| T201 | Mr Davis | abc@email.com |
| T202 | Mrs Jonas | jonas@email.com |
| T201 | Mr Davis | abc@email.com |
| T204 | Mrs Lopez | lopez@email.com |
| T205 | Mrs Wiley | wiley@email.com |
| T206 | Mr Bean | bean@email.com |

We have two tables: **Students** and the **Teachers** Tables. Let's write the SQL Queries to join the table using the **INNER JOIN** as follows:

1. Select Student\_ID, StudentName, TeacherName, TeacherEmail FROM Students INNER JOIN Teachers ON Students.TeacherID = Teachers.TeacherID;

After executing the query, it produces the below table.



### **Natural Join**

It is a type of inner type that joins two or more tables based on the same column name and has the same data type present on both tables.

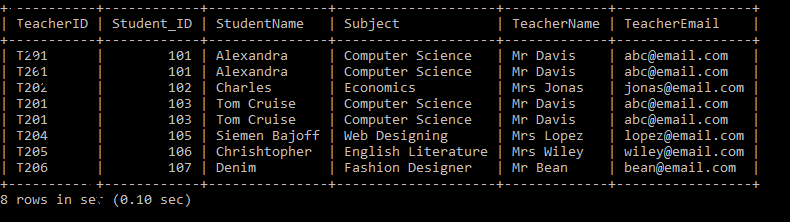
**Syntax:**

1. Select \* from tablename1 Natural JOIN tablename\_2;

We have two tables: **Students** and the **Teachers** Tables. Let's write the SQL Queries to join the table using the **Natural JOIN** as follows:

1. Select \* from Students Natural JOIN Teachers;

After executing the above query, it produces the following table.



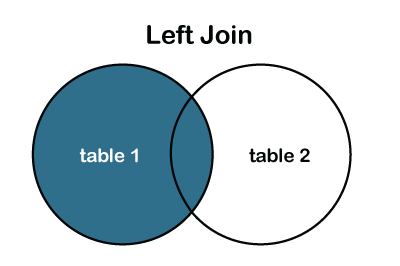
### **LEFT JOIN**

The [**LEFT JOIN**](https://www.javatpoint.com/sql-left-join) is used to retrieve all records from the left table (table1) and the matched rows or columns from the right table (table2). If both tables do not contain any matched rows or columns, it returns the NULL.

**Syntax:**

1. Select column\_1, column\_2, column(s) FROM table\_1 LEFT JOIN table\_2 ON table\_1.column\_name = table\_2.column\_name;

We can also represent the left join through the Venn diagram, as follows:



#### **Note: In some databases, LEFT JOIN is also known as LEFT OUTER JOIN.**

**Table 1: Product\_Details**

|  |  |  |
| --- | --- | --- |
| **ProductID** | **ProductName** | **Amount** |
| Pro101 | Laptop | 56000 |
| Pro102 | Mobile | 38000 |
| Pro103 | Headphones | 5000 |
| Pro104 | Television | 25000 |
| Pro105 | iPad | 60000 |

**Table 2: Customer\_Details**

|  |  |  |  |
| --- | --- | --- | --- |
| **CustomerName** | **CustomerAddress** | **CustomerAge** | **ProductID** |
| Martin Guptill | San Francisco, USA | 26 | Pro101 |
| James | Australia | 29 | Pro103 |
| Ambati Williamson | New Zealand | 27 | Pro102 |
| Jofra Archer | South Africa | 24 | Pro105 |
| Kate Wiley | Australia | 20 | Pro103 |

We have two tables: **Product\_Details** and the **Customer\_Details** Tables. Let's write the SQL Queries to join the table using the **LEFT JOIN** as follows:

1. Select ID, ProductName, CustomerName, CustomerAddress, Amount FROM Product\_Details LEFT JOIN Customer\_Details ON Product\_Details.ID = Customer\_Details.ProductID;

After executing the query, it produces the following table.



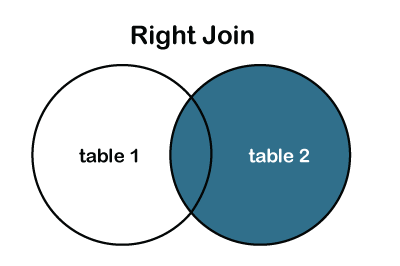
### **RIGHT JOIN or RIGHT Outer JOIN:**

The [**RIGHT JOIN**](https://www.javatpoint.com/sql-right-join) is used to retrieve all records from the right table (table2) and the matched rows or columns from the left table (table1). If both tables do not contain any matched rows or columns, it returns the NULL.

**Syntax:**

1. Select column\_1, column\_2, column(s) FROM table\_1 RIGHT JOIN table\_2 ON table\_1.column\_name = table\_2.column\_name;

We can also represent the right join through the Venn diagram, as follows:



#### **Note: In some databases, the RIGHT JOIN is also known as the RIGHT OUTER JOIN.**

**Table 1: Product\_Details**

|  |  |  |
| --- | --- | --- |
| **ID** | **ProductName** | **Amount** |
| Pro101 | Laptop | 56000 |
| Pro102 | Mobile | 38000 |
| Pro103 | Headphones | 5000 |
| Pro104 | Television | 25000 |
| Pro105 | iPad | 60000 |

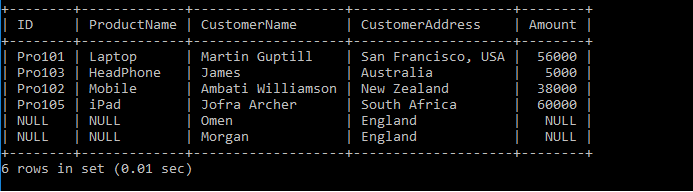
**Table 2: Customer\_Details**

|  |  |  |  |
| --- | --- | --- | --- |
| **CustomerName** | **CustomerAddress** | **CustomerAge** | **ProductID** |
| Martin Guptill | San Francisco, USA | 26 | Pro101 |
| James | Australia | 29 | Pro103 |
| Ambati Williamson | New Zealand | 27 | Pro102 |
| Jofra Archer | South Africa | 24 | Pro105 |
| Omen | England | 29 | Pro107 |
| Morgan | England | 20 | Pro108 |

We have two tables: **Product\_Details** and the **Customer\_Details** Tables. Let's write the SQL Queries to join the table using the **RIGHT JOIN** as follows:

1. Select ID, ProductName, CustomerName, CustomerAddress, Amount FROM Product\_Details LEFT JOIN Customer\_Details ON Product\_Details.ID = Customer\_Details.ProductID;

After executing the query, it produces the below table.



### **FULL JOIN or FULL Outer JOIN:**

It is a combination result set of both **LEFT JOIN** and **RIGHT JOIN**. The joined tables return all records from both the tables and if no matches are found in the table, it places NULL. It is also called a [**FULL OUTER JOIN**](https://www.javatpoint.com/sql-full-join).

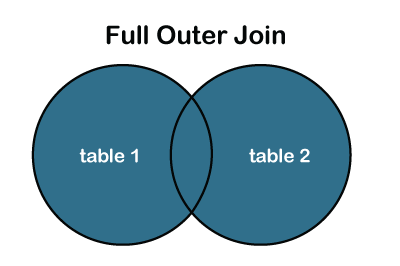
**Syntax:**

1. Select column\_1, column\_2, column(s) FROM table\_1 FULL JOIN table\_2 ON table\_1.column\_name = table\_2.column\_name;

Or, **FULL OUTER JOIN**

1. Select column\_1, column\_2, column(s) FROM table\_1 FULL OUTER JOIN table\_2 ON table\_1.column\_name = table\_2.column\_name;

We can also represent the full outer join through the Venn diagram, as follows:



**Table 1: Product\_Details**

|  |  |  |
| --- | --- | --- |
| **ID** | **ProductName** | **Amount** |
| Pro101 | Laptop | 56000 |
| Pro102 | Mobile | 38000 |
| Pro103 | Headphones | 5000 |
| Pro104 | Television | 25000 |
| Pro105 | iPad | 60000 |

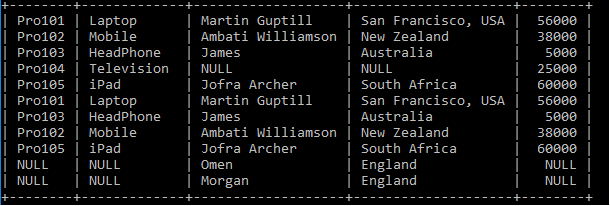
**Table 2: Customer\_Details**

|  |  |  |  |
| --- | --- | --- | --- |
| **CustomerName** | **CustomerAddress** | **CustomerAge** | **ProductID** |
| Martin Guptill | San Francisco, USA | 26 | Pro101 |
| James | Australia | 29 | Pro103 |
| Ambati Williamson | New Zealand | 27 | Pro102 |
| Jofra Archer | South Africa | 24 | Pro105 |
| Omen | England | 29 | Pro107 |
| Morgan | England | 20 | Pro108 |

We have two tables: **Product\_Details** and the **Customer\_Details** Tables. Let's write the SQL Queries to join the table using the **FULL** JOIN as follows:

1. Select ID, ProductName, CustomerName, CustomerAddress, Amount FROM Product\_Details FULL JOIN Customer\_Details ON Product\_Details.ID = Customer\_Details.ProductID;

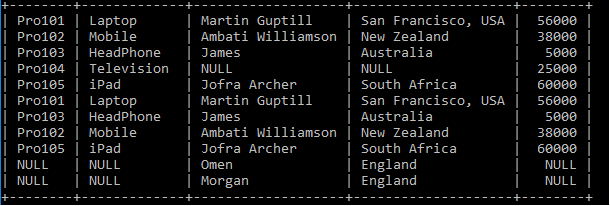
After executing the query, it produces the below table.



#### **Note: MySQL does not support FULL JOIN concepts, so we can use UNION ALL clause to combine both tables.**

Here is the Syntax for **UNION ALL** Clause to combine the tables.

1. Select ID, ProductName, CustomerName, CustomerAddress, Amount FROM Product\_Details LEFT JOIN Customer\_Details ON Product\_Details.ID = Customer\_Details.ProductID
2. UNION ALL
3. Select ID, ProductName, CustomerName, CustomerAddress, Amount FROM Product\_Details RIGHT JOIN Customer\_Details ON Product\_Details.ID = Customer\_Details.ProductID



### **CROSS JOIN**

It is also known as **CARTESIAN JOIN**, which returns the Cartesian product of two or more joined tables. The [**CROSS JOIN**](https://www.javatpoint.com/sql-cross-join) produces a table that merges each row from the first table with each second table row. It is not required to include any condition in CROSS JOIN.

**Syntax:**

1. Select \* from table\_1 cross join table\_2;

Or,

1. Select column1, column2, column3 FROM table\_1, table\_2;

**Table 1: Product\_Details**

|  |  |  |
| --- | --- | --- |
| **ID** | **ProductName** | **Amount** |
| Pro101 | Laptop | 56000 |
| Pro102 | Mobile | 38000 |
| Pro103 | Headphones | 5000 |
| Pro104 | Television | 25000 |
| Pro105 | iPad | 60000 |

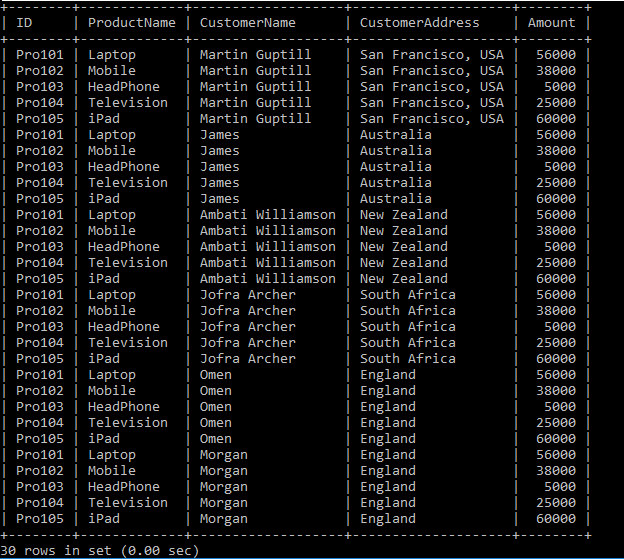
**Table 2: Customer\_Details**

|  |  |  |  |
| --- | --- | --- | --- |
| **CustomerName** | **CustomerAddress** | **CustomerAge** | **ProductID** |
| Martin Guptill | San Francisco, USA | 26 | Pro101 |
| James | Australia | 29 | Pro103 |
| Ambati Williamson | New Zealand | 27 | Pro102 |
| Jofra Archer | South Africa | 24 | Pro105 |
| Omen | England | 29 | Pro107 |
| Morgan | England | 20 | Pro108 |

We have two tables: **Product\_Details** and the **Customer\_Details** Tables. Let's write the SQL Queries to join the table using the **FULL** JOIN as follows:

1. Select ID, ProductName, CustomerName, CustomerAddress, Amount FROM Product\_Details, Customer\_Details;

After executing the query, it produces the below table.



### **SELF JOIN**

It is a SELF JOIN used to create a table by joining itself as there were two tables. It makes temporary naming of at least one table in an SQL statement.

**Syntax:**

1. Select column1, column2, column(s) FROM table\_1 Tbl1, table\_2 Tbl2 WHERE condition;

**Tbl1** and **Tbl2** are two different table aliases for the same table.

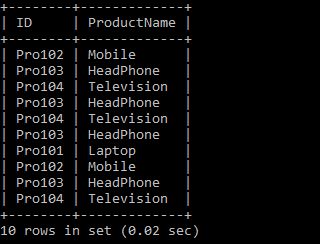
**Table 1: Product\_Details**

|  |  |  |
| --- | --- | --- |
| **ID** | **ProductName** | **Amount** |
| Pro101 | Laptop | 56000 |
| Pro102 | Mobile | 38000 |
| Pro103 | Headphones | 5000 |
| Pro104 | Television | 25000 |
| Pro105 | iPad | 60000 |

Let's write the SQL Queries to join the table using the **SELF JOIN** as follows:

1. Select TB.ID, TB.ProductName FROM Product\_Details TB, Product\_Details TB2
2. WHERE TB.AMOUNT **<** **TB2.AMOUNT**;

After executing the query, it produces the below table.



# Change datatype of column in SQL

SQL being a dynamically manipulating database query language lets you play with your data-set that may be **organized** or **unorganized**. Such data may be presented in the form of different types depending upon your requirements. There are various methods to change the types of data present in the rows or columns of your database. Here, we will discuss the method to change the datatype of column in [SQL](https://www.javatpoint.com/sql-tutorial).

## **Using SQL server**

1. Open the SQL server. In the **Object Explorer** option, right-click the column you want to change and click on Design.
2. You need to select the column whose data type you want to modify.
3. In the Column Properties, you need to click the grid cell to change the Data Type property and then choose the data type from the appeared drop-down list.
4. Now, click **Savetable** on the File menu to save the changes.

#### **Note: Whenever you modify the column data type in the SQL server, the option Table Designer applies the changes related to the length of the selected data type. You may always need to specify the length of the data type along with desired specified value after the data type.1.**

## **Using ALTER TABLE**

The ALTER TABLE command in SQL lets you **delete, add** or **modify columns** present in your database table. It is also used for other purposes like adding or dropping constraints on your existing database table. Create the sample database shown in the below examples. Proceed with the below steps to understand how the data type is changed.

### **Syntax:**

1. **ALTER** **TABLE** table\_name
2. **ADD** column\_name datatype;

Example:

1. **ALTER** **TABLE** Students
2. **ADD** **name** **varchar**(100);

To modify the datatype of the column:

Syntax:

1. **ALTER** **TABLE** table\_name
2. **ADD** column\_name datatype;

Example:

1. **ALTER** **TABLE** Employees
2. **ADD** employee\_name string;

Also, using the ALTER COLUMN option in SQL, you can easily modify the data type of the given column as shown. The below query changes the datatype of the column named **DateofBirth** to the type year.

1. **ALTER** **TABLE** Employees
2. **ALTER** **COLUMN** DateofBirth year;

The main purpose of the alter command is not just to delete or add the columns present in your database but to modify and change it too. In the above examples, you have seen the simple and easy syntax of ALTER TABLE command in SQL. There might also arise a situation when you want to modify multiple columns in the database. To do that, you simply need to assign the column's name along with the datatype conversion you want in your newly modified column. Consider the below example.

1. **ALTER** **TABLE** table\_name
2. **ADD** (column\_1 column\_definition,
3. column\_2 column\_definition,
4. ...
5. column\_n column\_definition);

## **Using other databases**

For Oracle, MySQL, MariaDB:

1. **ALTER** **TABLE** table\_name
2. **MODIFY** column\_name column\_type;

For POSTgreSQL:

1. **ALTER** **TABLE** table\_name
2. **ALTER** **COLUMN** column\_name TYPE column\_definition;

Also, if you do not want to lose data while changing the datatype of the respective column, you might see the below example for reference.

1. **Create** **table** Employees
2. (
3. ID **int** **primary** **key** ID,
4. **Name** **varchar**(50),
5. Sex **varchar**(50),
6. Incentives nvarchar(50)
7. )

To know what datatype your column is, you need to type the below command which tells you the data type of the column you want to change.

### **Syntax**

1. **SELECT** datatype **from** **Table**.COLUMS
2. **WHERE** **Table**.**schema** = "Your\_database\_name"
3. AND table\_name = "Your\_table\_name"

To understand this is quite a depth, let's create a database to observe how datatypes of columns can be brought out.

1. In MySQL
3. **create** **table** DataTypeDemo
4. (
5. Id **int**,
6. Venue **varchar**(100),
7. Amount  **decimal**(9,3)
8. );

Query:

1. **SELECT** datatype **from** **Table**.COLUMNS
2. **WHERE** table\_schema = "Company"
3. AND table\_name = "Attendance"

In the above example, the output of the query will roll out the datatype of the respective columns. We used MySQL since the syntax is quite familiar and easy to understand.

## **Summary**

In this article, you learned how you can easily change the data types of your desired columns in SQL, MySQL, or any other databases you might be using. There are no such hard and fast rules to write the queries in capital or small letter provided some data types are case-sensitive and should be used only with prior knowledge. If you're working with huge amounts of data, rolling out all the data types back to previous data types is not an easy task; rather you would find it more difficult to arrange them after converting. Thus, one should carefully figure out the fragile measures before opting to change the data types of the columns in your desired database table.