w3schools.com

THE WORLD'S LARGEST WEB DEVELOPER SITE

# Introduction to SQL

SQL is a standard language for accessing and manipulating databases.

## What is SQL?

- SQL stands for Structured Query Language
- SQL lets you access and manipulate databases
- SQL is an ANSI (American National Standards Institute) standard

## What Can SQL do?

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

## SQL is a Standard - BUT....

Although SQL is an ANSI (American National Standards Institute) standard, there are different versions of the SQL language.

However, to be compliant with the ANSI standard, they all support at least the major commands (such as SELECT, UPDATE, DELETE, INSERT, WHERE) in a similar manner.

> **Note:** Most of the SQL database programs also have their own proprietary extensions in addition to the SQL standard!

# Using SQL in Your Web Site

To build a web site that shows data from a database, you will need:

- An RDBMS database program (i.e. MS Access, SQL Server, MySQL)
- To use a server-side scripting language, like PHP or ASP
- To use SQL to get the data you want
- To use HTML / CSS to style the page

---

# RDBMS

RDBMS stands for Relational Database Management System.

RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

The data in RDBMS is stored in database objects called tables. A table is a collection of related data entries and it consists of columns and rows.

Look at the "Customers" table:

## Example

```
SELECT * FROM Customers;
```

Every table is broken up into smaller entities called fields. The fields in the Customers table consist of CustomerID, CustomerName, ContactName, Address, City and PostalCode. A field is a column in a table that is designed to maintain specific information about every record in the table.

A record, also called a row, is each individual entry that exists in a table. For example, there are 91 records in the above Customers table. A record is a horizontal entity in a table.

A column is a vertical entity in a table that contains all information associated with a specific field in a table.

# SQL Syntax

## Database Tables

A database most often contains one or more tables. Each table is identified by a name (e.g. "Customers" or "Orders"). Tables contain records (rows) with data.

In this tutorial we will use the well-known Northwind sample database (included in MS Access and MS SQL Server).

Below is a selection from the "Customers" table:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

The table above contains five records (one for each customer) and seven columns (CustomerID, CustomerName, ContactName, Address, City, PostalCode, and Country).

# SQL Statements

Most of the actions you need to perform on a database are done with SQL statements.

The following SQL statement selects all the records in the "Customers" table:

## Example

```
SELECT * FROM Customers;
```

In this tutorial we will teach you all about the different SQL statements.

---

# Keep in Mind That...

- SQL keywords are NOT case sensitive: select is the same as SELECT

In this tutorial we will write all SQL keywords in upper-case.

---

# Semicolon after SQL Statements?

Some database systems require a semicolon at the end of each SQL statement.

Semicolon is the standard way to separate each SQL statement in database systems that allow more than one SQL statement to be executed in the same call to the server.

In this tutorial, we will use semicolon at the end of each SQL statement.

---

# Some of The Most Important SQL Commands

- **SELECT** - extracts data from a database
- **UPDATE** - updates data in a database
- **DELETE** - deletes data from a database
- **INSERT INTO** - inserts new data into a database
- **CREATE DATABASE** - creates a new database
- **ALTER DATABASE** - modifies a database
- **CREATE TABLE** - creates a new table
- **ALTER TABLE** - modifies a table
- **DROP TABLE** - deletes a table
- **CREATE INDEX** - creates an index (search key)
- **DROP INDEX** - deletes an index

w3schools.com

THE WORLD'S LARGEST WEB DEVELOPER SITE

# SQL SELECT Statement

## The SQL SELECT Statement

The SELECT statement is used to select data from a database.

The data returned is stored in a result table, called the result-set.

### SELECT Syntax

```
SELECT column1, column2, ...
FROM table_name;
```

Here, column1, column2, ... are the field names of the table you want to select data from. If you want to select all the fields available in the table, use the following syntax:

```
SELECT * FROM table_name;
```

## Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# SELECT Column Example

The following SQL statement selects the "CustomerName" and "City" columns from the "Customers" table:

## Example

```
SELECT CustomerName, City FROM Customers;
```

# SELECT * Example

The following SQL statement selects all the columns from the "Customers" table:

## Example

```
SELECT * FROM Customers;
```

# SQL SELECT DISTINCT Statement

## The SQL SELECT DISTINCT Statement

The SELECT DISTINCT statement is used to return only distinct (different) values.

Inside a table, a column often contains many duplicate values; and sometimes you only want to list the different (distinct) values.

The SELECT DISTINCT statement is used to return only distinct (different) values.

### SELECT DISTINCT Syntax

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

## Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# SELECT Example

The following SQL statement selects all (and duplicate) values from the "Country" column in the "Customers" table:

## Example

```
SELECT Country FROM Customers;
```

Now, let us use the DISTINCT keyword with the above SELECT statement and see the result.

---

# SELECT DISTINCT Examples

The following SQL statement selects only the DISTINCT values from the "Country" column in the "Customers" table:

## Example

```
SELECT DISTINCT Country FROM Customers;
```

The following SQL statement lists the number of customers countries:

## Example

```
SELECT COUNT(DISTINCT Country) FROM Customers;
```

w3schools.com

THE WORLD'S LARGEST WEB DEVELOPER SITE

# SQL WHERE Clause

## The SQL WHERE Clause

The WHERE clause is used to filter records.

The WHERE clause is used to extract only those records that fulfill a specified condition.

### WHERE Syntax

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

**Note:** The WHERE clause is not only used in SELECT statement, it is also used in UPDATE, DELETE statement, etc.!

## Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# WHERE Clause Example

The following SQL statement selects all the customers from the country "Mexico", in the "Customers" table:

## Example

```
SELECT * FROM Customers
WHERE Country='Mexico';
```

# Text Fields vs. Numeric Fields

SQL requires single quotes around text values (most database systems will also allow double quotes).

However, numeric fields should not be enclosed in quotes:

## Example

```
SELECT * FROM Customers
WHERE CustomerID=1;
```

# Operators in The WHERE Clause

The following operators can be used in the WHERE clause:

| Operator | Description |
|---|---|
| = | Equal |
| <> | Not equal. **Note:** In some versions of SQL this operator may be written as != |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |
| BETWEEN | Between an inclusive range |
| LIKE | Search for a pattern |
| IN | To specify multiple possible values for a column |

# SQL AND, OR and NOT Operators

## The SQL AND, OR and NOT Operators

The WHERE clause can be combined with AND, OR, and NOT operators.

The AND and OR operators are used to filter records based on more than one condition:

- The AND operator displays a record if all the conditions separated by AND is TRUE.
- The OR operator displays a record if any of the conditions separated by OR is TRUE.

The NOT operator displays a record if the condition(s) is NOT TRUE.

### AND Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 AND condition2 AND condition3 ...;
```

### OR Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE condition1 OR condition2 OR condition3 ...;
```

### NOT Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE NOT condition;
```

# Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# AND Example

The following SQL statement selects all fields from "Customers" where country is "Germany" AND city is "Berlin":

## Example

```
SELECT * FROM Customers
WHERE Country='Germany' AND City='Berlin';
```

# OR Example

The following SQL statement selects all fields from "Customers" where city is "Berlin" OR "München":

## Example

```
SELECT * FROM Customers
WHERE City='Berlin' OR City='München';
```

# NOT Example

The following SQL statement selects all fields from "Customers" where country is NOT "Germany":

## Example

```
SELECT * FROM Customers
WHERE NOT Country='Germany';
```

---

# Combining AND, OR and NOT

You can also combine the AND, OR and NOT operators.

The following SQL statement selects all fields from "Customers" where country is "Germany" AND city must be "Berlin" OR "München" (use parenthesis to form complex expressions):

## Example

```
SELECT * FROM Customers
WHERE Country='Germany' AND (City='Berlin' OR City='München');
```

The following SQL statement selects all fields from "Customers" where country is NOT "Germany" and NOT "USA":

## Example

```
SELECT * FROM Customers
WHERE NOT Country='Germany' AND NOT Country='USA';
```

w3schools.com

THE WORLD'S LARGEST WEB DEVELOPER SITE

# SQL ORDER BY Keyword

## The SQL ORDER BY Keyword

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

### ORDER BY Syntax

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

## Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# ORDER BY Example

The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" column:

## Example

```
SELECT * FROM Customers
ORDER BY Country;
```

# ORDER BY DESC Example

The following SQL statement selects all customers from the "Customers" table, sorted DESCENDING by the "Country" column:

## Example

```
SELECT * FROM Customers
ORDER BY Country DESC;
```

# ORDER BY Several Columns Example

The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" and the "CustomerName" column:

## Example

```
SELECT * FROM Customers
ORDER BY Country, CustomerName;
```

# ORDER BY Several Columns Example 2

The following SQL statement selects all customers from the "Customers" table, sorted ascending by the "Country" and descending by the "CustomerName" column:

## Example

```
SELECT * FROM Customers
ORDER BY Country ASC, CustomerName DESC;
```

w3schools.com

THE WORLD'S LARGEST WEB DEVELOPER SITE

# SQL INSERT INTO Statement

## The SQL INSERT INTO Statement

The INSERT INTO statement is used to insert new records in a table.

### INSERT INTO Syntax

It is possible to write the INSERT INTO statement in two ways.

The first way specifies both the column names and the values to be inserted:

```
INSERT INTO table_name (column1, column2, column3, ...)
VALUES (value1, value2, value3, ...);
```

If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. The INSERT INTO syntax would be as follows:

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

## Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 89 | White Clover Markets | Karl Jablonski | 305 - 14th Ave. S. Suite 3B | Seattle | 98128 | USA |
| 90 | Wilman Kala | Matti Karttunen | Keskuskatu 45 | Helsinki | 21240 | Finland |
| 91 | Wolski | Zbyszek | ul. Filtrowa 68 | Walla | 01-012 | Poland |

# INSERT INTO Example

The following SQL statement inserts a new record in the "Customers" table:

## Example

```
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode,
Country)
VALUES ('Cardinal','Tom B. Erichsen','Skagen 21','Stavanger','4006','Norway');
```

The selection from the "Customers" table will now look like this:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 89 | White Clover Markets | Karl Jablonski | 305 - 14th Ave. S. Suite 3B | Seattle | 98128 | USA |
| 90 | Wilman Kala | Matti Karttunen | Keskuskatu 45 | Helsinki | 21240 | Finland |
| 91 | Wolski | Zbyszek | ul. Filtrowa 68 | Walla | 01-012 | Poland |
| 92 | Cardinal | Tom B. Erichsen | Skagen 21 | Stavanger | 4006 | Norway |

**Did you notice that we did not insert any number into the CustomerID field?**
The CustomerID column is an auto-increment field and will be generated automatically when a new record is inserted into the table.

# Insert Data Only in Specified Columns

It is also possible to only insert data in specific columns.

The following SQL statement will insert a new record, but only insert data in the "CustomerName", "City", and "Country" columns (CustomerID will be updated automatically):

## Example

```
INSERT INTO Customers (CustomerName, City, Country)
VALUES ('Cardinal', 'Stavanger', 'Norway');
```

The selection from the "Customers" table will now look like this:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 89 | White Clover Markets | Karl Jablonski | 305 - 14th Ave. S. Suite 3B | Seattle | 98128 | USA |
| 90 | Wilman Kala | Matti Karttunen | Keskuskatu 45 | Helsinki | 21240 | Finland |
| 91 | Wolski | Zbyszek | ul. Filtrowa 68 | Walla | 01-012 | Poland |
| 92 | Cardinal | null | null | Stavanger | null | Norway |

w3schools.com

THE WORLD'S LARGEST WEB DEVELOPER SITE

# SQL UPDATE Statement

## The SQL UPDATE Statement

The UPDATE statement is used to modify the existing records in a table.

### UPDATE Syntax

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

**Note:** Be careful when updating records in a table! Notice the WHERE clause in the UPDATE statement. The WHERE clause specifies which record(s) that should be updated. If you omit the WHERE clause, all records in the table will be updated!

## Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# UPDATE Table

The following SQL statement updates the first customer (CustomerID = 1) with a new contact person *and* a new city.

## Example

```
UPDATE Customers
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'
WHERE CustomerID = 1;
```

The selection from the "Customers" table will now look like this:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Alfred Schmidt | Obere Str. 57 | Frankfurt | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# UPDATE Multiple Records

It is the WHERE clause that determines how many records that will be updated.

The following SQL statement will update the contactname to "Juan" for all records where country is "Mexico":

## Example

```
UPDATE Customers
SET ContactName='Juan'
WHERE Country='Mexico';
```

The selection from the "Customers" table will now look like this:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Alfred Schmidt | Obere Str. 57 | Frankfurt | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Juan | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Juan | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# Update Warning!

Be careful when updating records. If you omit the WHERE clause, ALL records will be updated!

## Example

```
UPDATE Customers
SET ContactName='Juan';
```

The selection from the "Customers" table will now look like this:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Juan | Obere Str. 57 | Frankfurt | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Juan | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Juan | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Juan | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Juan | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# SQL DELETE Statement

## The SQL DELETE Statement

The DELETE statement is used to delete existing records in a table.

### DELETE Syntax

```
DELETE FROM table_name
WHERE condition;
```

**Note:** Be careful when deleting records in a table! Notice the WHERE clause in the DELETE statement. The WHERE clause specifies which record(s) that should be deleted. If you omit the WHERE clause, all records in the table will be deleted!

## Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# SQL DELETE Example

The following SQL statement deletes the customer "Alfreds Futterkiste" from the "Customers" table:

## Example

```
DELETE FROM Customers
WHERE CustomerName='Alfreds Futterkiste';
```

The "Customers" table will now look like this:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# Delete All Records

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

```
DELETE FROM table_name;
```

or:

```
DELETE * FROM table_name;
```

# SQL TOP, LIMIT or ROWNUM Clause

## The SQL SELECT TOP Clause

The SELECT TOP clause is used to specify the number of records to return.

The SELECT TOP clause is useful on large tables with thousands of records. Returning a large number of records can impact on performance.

> **Note:** Not all database systems support the SELECT TOP clause. MySQL supports the LIMIT clause to select a limited number of records, while Oracle uses ROWNUM.

**SQL Server / MS Access Syntax:**

```
SELECT TOP number|percent column_name(s)
FROM table_name
WHERE condition;
```

**MySQL Syntax:**

```
SELECT column_name(s)
FROM table_name
WHERE condition
LIMIT number;
```

**Oracle Syntax:**

```
SELECT column_name(s)
FROM table_name
WHERE ROWNUM <= number;
```

# Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# SQL TOP, LIMIT and ROWNUM Examples

The following SQL statement selects the first three records from the "Customers" table:

## Example

```
SELECT TOP 3 * FROM Customers;
```

The following SQL statement shows the equivalent example using the LIMIT clause:

## Example

```
SELECT * FROM Customers
LIMIT 3;
```

The following SQL statement shows the equivalent example using ROWNUM:

## Example

```
SELECT * FROM Customers
WHERE ROWNUM <= 3;
```

# SQL TOP PERCENT Example

The following SQL statement selects the first 50% of the records from the "Customers" table:

## Example

```
SELECT TOP 50 PERCENT * FROM Customers;
```

# ADD a WHERE CLAUSE

The following SQL statement selects the first three records from the "Customers" table, where the country is "Germany":

## Example

```
SELECT TOP 3 * FROM Customers
WHERE Country='Germany';
```

The following SQL statement shows the equivalent example using the LIMIT clause:

## Example

```
SELECT * FROM Customers
WHERE Country='Germany'
LIMIT 3;
```

The following SQL statement shows the equivalent example using ROWNUM:

## Example

```
SELECT * FROM Customers
WHERE Country='Germany' AND ROWNUM <= 3;
```

# SQL MIN() and MAX() Functions

## The SQL MIN() and MAX() Functions

The MIN() function returns the smallest value of the selected column.

The MAX() function returns the largest value of the selected column.

### MIN() Syntax

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

### MAX() Syntax

```
SELECT MAX(column_name)
FROM table_name
WHERE condition;
```

## Demo Database

Below is a selection from the "Products" table in the Northwind sample database:

| ProductID | ProductName | SupplierID | CategoryID | Unit | Price |
|---|---|---|---|---|---|
| 1 | Chais | 1 | 1 | 10 boxes x 20 bags | 18 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottles | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 12 - 550 ml bottles | 10 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 2 | 48 - 6 oz jars | 21.35 |
| 5 | Chef Anton's Gumbo Mix | 2 | 2 | 36 boxes | 25 |

# MIN() Example

The following SQL statement finds the price of the cheapest product:

## Example

```
SELECT MIN(Price) AS SmallestPrice
FROM Products;
```

# MAX() Example

The following SQL statement finds the price of the most expensive product:

## Example

```
SELECT MAX(Price) AS LargestPrice
FROM Products;
```

# SQL COUNT(), AVG() and SUM() Functions

## The SQL COUNT(), AVG() and SUM() Functions

The COUNT() function returns the number of rows that matches a specified criteria.

The AVG() function returns the average value of a numeric column.

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

### COUNT() Syntax

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

### AVG() Syntax

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

### SUM() Syntax

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

# Demo Database

Below is a selection from the "Products" table in the Northwind sample database:

| ProductID | ProductName | SupplierID | CategoryID | Unit | Price |
|---|---|---|---|---|---|
| 1 | Chais | 1 | 1 | 10 boxes x 20 bags | 18 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottles | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 12 - 550 ml bottles | 10 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 2 | 48 - 6 oz jars | 21.35 |
| 5 | Chef Anton's Gumbo Mix | 2 | 2 | 36 boxes | 25 |

# COUNT() Example

The following SQL statement finds the number of products:

## Example

```
SELECT COUNT(ProductID)
FROM Products;
```

# AVG() Example

The following SQL statement finds the average price of all products:

## Example

```
SELECT AVG(Price)
FROM Products;
```

# Demo Database

Below is a selection from the "OrderDetails" table in the Northwind sample database:

| OrderDetailID | OrderID | ProductID | Quantity |
|---|---|---|---|
| 1 | 10248 | 11 | 12 |
| 2 | 10248 | 42 | 10 |
| 3 | 10248 | 72 | 5 |
| 4 | 10249 | 14 | 9 |
| 5 | 10249 | 51 | 40 |

# SUM() Example

The following SQL statement finds the sum of the "Quantity" fields in the "OrderDetails" table:

## Example

```
SELECT SUM(Quantity)
FROM OrderDetails;
```

# SQL LIKE Operator

## The SQL LIKE Operator

The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards used in conjunction with the LIKE operator:

- % - The percent sign represents zero, one, or multiple characters
- _ - The underscore represents a single character

**Note:** MS Access uses a question mark (?) instead of the underscore (_).

The percent sign and the underscore can also be used in combinations!

### LIKE Syntax

```
SELECT column1, column2, ...
FROM table_name
WHERE columnN LIKE pattern;
```

**Tip:** You can also combine any number of conditions using AND or OR operators.

Here are some examples showing different LIKE operators with '%' and '_' wildcards:

| LIKE Operator | Description |
|---|---|
| WHERE CustomerName LIKE 'a%' | Finds any values that starts with "a" |
| WHERE CustomerName LIKE '%a' | Finds any values that ends with "a" |
| WHERE CustomerName LIKE '%or%' | Finds any values that have "or" in any position |
| WHERE CustomerName LIKE '_r%' | Finds any values that have "r" in the second position |
| WHERE CustomerName LIKE 'a_%_%' | Finds any values that starts with "a" and are at least 3 characters in length |
| WHERE ContactName LIKE 'a%o' | Finds any values that starts with "a" and ends with "o" |

# Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# SQL LIKE Examples

The following SQL statement selects all customers with a CustomerName starting with "a":

## Example

```
SELECT * FROM Customers
WHERE CustomerName LIKE 'a%';
```

The following SQL statement selects all customers with a CustomerName ending with "a":

## Example

```
SELECT * FROM Customers
WHERE CustomerName LIKE '%a';
```

The following SQL statement selects all customers with a CustomerName that have "or" in any position:

## Example

```
SELECT * FROM Customers
WHERE CustomerName LIKE '%or%';
```

The following SQL statement selects all customers with a CustomerName that have "r" in the second position:

## Example

```
SELECT * FROM Customers
WHERE CustomerName LIKE '_r%';
```

The following SQL statement selects all customers with a CustomerName that starts with "a" and are at least 3 characters in length:

## Example

```
SELECT * FROM Customers
WHERE CustomerName LIKE 'a_%_%';
```

The following SQL statement selects all customers with a CustomerName that starts with "a" and ends with "o":

## Example

```
SELECT * FROM Customers
WHERE ContactName LIKE 'a%o';
```

The following SQL statement selects all customers with a CustomerName that NOT starts with "a":

## Example

```
SELECT * FROM Customers
WHERE CustomerName NOT LIKE 'a%';
```

# SQL Wildcards

## SQL Wildcard Characters

A wildcard character is used to substitute any other character(s) in a string.

Wildcard characters are used with the SQL LIKE operator. The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.

There are two wildcards used in conjunction with the LIKE operator:

- % - The percent sign represents zero, one, or multiple characters
- _ - The underscore represents a single character

> **Note:** MS Access uses a question mark (?) instead of the underscore (_).

In MS Access and SQL Server you can also use:

- [*charlist*] - Defines sets and ranges of characters to match
- [^*charlist*] or [!*charlist*] - Defines sets and ranges of characters NOT to match

The wildcards can also be used in combinations!

Here are some examples showing different LIKE operators with '%' and '_' wildcards:

| LIKE Operator | Description |
|---|---|
| WHERE CustomerName LIKE 'a%' | Finds any values that starts with "a" |
| WHERE CustomerName LIKE '%a' | Finds any values that ends with "a" |
| WHERE CustomerName LIKE '%or%' | Finds any values that have "or" in any position |
| WHERE CustomerName LIKE '_r%' | Finds any values that have "r" in the second position |
| WHERE CustomerName LIKE 'a_%_%' | Finds any values that starts with "a" and are at least 3 characters in length |
| WHERE ContactName LIKE 'a%o' | Finds any values that starts with "a" and ends with "o" |

# Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# Using the % Wildcard

The following SQL statement selects all customers with a City starting with "ber":

## Example

```
SELECT * FROM Customers
WHERE City LIKE 'ber%';
```

The following SQL statement selects all customers with a City containing the pattern "es":

## Example

```
SELECT * FROM Customers
WHERE City LIKE '%es%';
```

# Using the _ Wildcard

The following SQL statement selects all customers with a City starting with any character, followed by "erlin":

## Example

```
SELECT * FROM Customers
WHERE City LIKE '_erlin';
```

The following SQL statement selects all customers with a City starting with "L", followed by any character, followed by "n", followed by any character, followed by "on":

## Example

```
SELECT * FROM Customers
WHERE City LIKE 'L_n_on';
```

# Using the [charlist] Wildcard

The following SQL statement selects all customers with a City starting with "b", "s", or "p":

## Example

```
SELECT * FROM Customers
WHERE City LIKE '[bsp]%';
```

The following SQL statement selects all customers with a City starting with "a", "b", or "c":

## Example

```
SELECT * FROM Customers
WHERE City LIKE '[a-c]%';
```

# Using the [!charlist] Wildcard

The two following SQL statements selects all customers with a City NOT starting with "b", "s", or "p":

## Example

```
SELECT * FROM Customers
WHERE City LIKE '[!bsp]%';
```

Or:

## Example

```
SELECT * FROM Customers
WHERE City NOT LIKE '[bsp]%';
```

# SQL IN Operator

## The SQL IN Operator

The IN operator allows you to specify multiple values in a WHERE clause.

The IN operator is a shorthand for multiple OR conditions.

### IN Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1, value2, ...);
```

or:

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (SELECT STATEMENT);
```

## Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# IN Operator Examples

The following SQL statement selects all customers that are located in "Germany", "France" and "UK":

## Example

```
SELECT * FROM Customers
WHERE Country IN ('Germany', 'France', 'UK');
```

The following SQL statement selects all customers that are NOT located in "Germany", "France" or "UK":

## Example

```
SELECT * FROM Customers
WHERE Country NOT IN ('Germany', 'France', 'UK');
```

The following SQL statement selects all customers that are from the same countries as the suppliers:

## Example

```
SELECT * FROM Customers
WHERE Country IN (SELECT Country FROM Suppliers);
```

w3schools.com

THE WORLD'S LARGEST WEB DEVELOPER SITE

# SQL BETWEEN Operator

## The SQL BETWEEN Operator

The BETWEEN operator selects values within a given range. The values can be numbers, text, or dates.

The BETWEEN operator is inclusive: begin and end values are included.

### BETWEEN Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

## Demo Database

Below is a selection from the "Products" table in the Northwind sample database:

| ProductID | ProductName | SupplierID | CategoryID | Unit | Price |
|---|---|---|---|---|---|
| 1 | Chais | 1 | 1 | 10 boxes x 20 bags | 18 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottles | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 12 - 550 ml bottles | 10 |
| 4 | Chef Anton's Cajun Seasoning | 1 | 2 | 48 - 6 oz jars | 22 |
| 5 | Chef Anton's Gumbo Mix | 1 | 2 | 36 boxes | 21.35 |

## BETWEEN Example

The following SQL statement selects all products with a price BETWEEN 10 and 20:

### Example

```
SELECT * FROM Products
WHERE Price BETWEEN 10 AND 20;
```

# NOT BETWEEN Example

To display the products outside the range of the previous example, use NOT BETWEEN:

## Example

```
SELECT * FROM Products
WHERE Price NOT BETWEEN 10 AND 20;
```

# BETWEEN with IN Example

The following SQL statement selects all products with a price BETWEEN 10 and 20. In addition; do not show products with a CategoryID of 1,2, or 3:

## Example

```
SELECT * FROM Products
WHERE (Price BETWEEN 10 AND 20)
AND NOT CategoryID IN (1,2,3);
```

# BETWEEN Text Values Example

The following SQL statement selects all products with a ProductName BETWEEN 'Carnarvon Tigers' and 'Mozzarella di Giovanni':

## Example

```
SELECT * FROM Products
WHERE ProductName BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'
ORDER BY ProductName;
```

# NOT BETWEEN Text Values Example

The following SQL statement selects all products with a ProductName NOT BETWEEN 'Carnarvon Tigers' and 'Mozzarella di Giovanni':

## Example

```
SELECT * FROM Products
WHERE ProductName NOT BETWEEN 'Carnarvon Tigers' AND 'Mozzarella di Giovanni'
ORDER BY ProductName;
```

# Sample Table

Below is a selection from the "Orders" table in the Northwind sample database:

| OrderID | CustomerID | EmployeeID | OrderDate | ShipperID |
|---|---|---|---|---|
| 10248 | 90 | 5 | 7/4/1996 | 3 |
| 10249 | 81 | 6 | 7/5/1996 | 1 |
| 10250 | 34 | 4 | 7/8/1996 | 2 |
| 10251 | 84 | 3 | 7/9/1996 | 1 |
| 10252 | 76 | 4 | 7/10/1996 | 2 |

# BETWEEN Dates Example

The following SQL statement selects all orders with an OrderDate BETWEEN '04-July-1996' and '09-July-1996':

## Example

```
SELECT * FROM Orders
WHERE OrderDate BETWEEN #07/04/1996# AND #07/09/1996#;
```

w3schools.com

THE WORLD'S LARGEST WEB DEVELOPER SITE

# SQL Aliases

## SQL Aliases

SQL aliases are used to give a table, or a column in a table, a temporary name.

Aliases are often used to make column names more readable.

An alias only exists for the duration of the query.

### Alias Column Syntax

```
SELECT column_name AS alias_name
FROM table_name;
```

### Alias Table Syntax

```
SELECT column_name(s)
FROM table_name AS alias_name;
```

---

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |

And a selection from the "Orders" table:

| OrderID | CustomerID | EmployeeID | OrderDate | ShipperID |
|---|---|---|---|---|
| 10354 | 58 | 8 | 1996-11-14 | 3 |
| 10355 | 4 | 6 | 1996-11-15 | 1 |
| 10356 | 86 | 6 | 1996-11-18 | 2 |

# Alias for Columns Examples

The following SQL statement creates two aliases, one for the CustomerID column and one for the CustomerName column:

## Example

```
SELECT CustomerID as ID, CustomerName AS Customer
FROM Customers;
```

The following SQL statement creates two aliases, one for the CustomerName column and one for the ContactName column. **Note:** It requires double quotation marks or square brackets if the alias name contains spaces:

## Example

```
SELECT CustomerName AS Customer, ContactName AS [Contact Person]
FROM Customers;
```

The following SQL statement creates an alias named "Address" that combine four columns (Address, PostalCode, City and Country):

## Example

```
SELECT CustomerName, Address + ', ' + PostalCode + ' ' + City + ', ' + Country AS
Address
FROM Customers;
```

**Note:** To get the SQL statement above to work in MySQL use the following:

```
SELECT CustomerName, CONCAT(Address,', ',PostalCode,', ',City,', ',Country) AS
Address
FROM Customers;
```

# Alias for Tables Example

The following SQL statement selects all the orders from the customer with CustomerID=4 (Around the Horn). We use the "Customers" and "Orders" tables, and give them the table aliases of "c" and "o" respectively (Here we use aliases to make the SQL shorter):

## Example

```
SELECT o.OrderID, o.OrderDate, c.CustomerName
FROM Customers AS c, Orders AS o
WHERE c.CustomerName="Around the Horn" AND c.CustomerID=o.CustomerID;
```

The following SQL statement is the same as above, but without aliases:

## Example

```
SELECT Orders.OrderID, Orders.OrderDate, Customers.CustomerName
FROM Customers, Orders
WHERE Customers.CustomerName="Around the Horn" AND
Customers.CustomerID=Orders.CustomerID;
```

Aliases can be useful when:

- There are more than one table involved in a query
- Functions are used in the query
- Column names are big or not very readable
- Two or more columns are combined together

# SQL Joins

## SQL JOIN

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Let's look at a selection from the "Orders" table:

| OrderID | CustomerID | OrderDate |
|---|---|---|
| 10308 | 2 | 1996-09-18 |
| 10309 | 37 | 1996-09-19 |
| 10310 | 77 | 1996-09-20 |

Then, look at a selection from the "Customers" table:

| CustomerID | CustomerName | ContactName | Country |
|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mexico |

Notice that the "CustomerID" column in the "Orders" table refers to the "CustomerID" in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.

Then, we can create the following SQL statement (that contains an INNER JOIN), that selects records that have matching values in both tables:

### Example

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

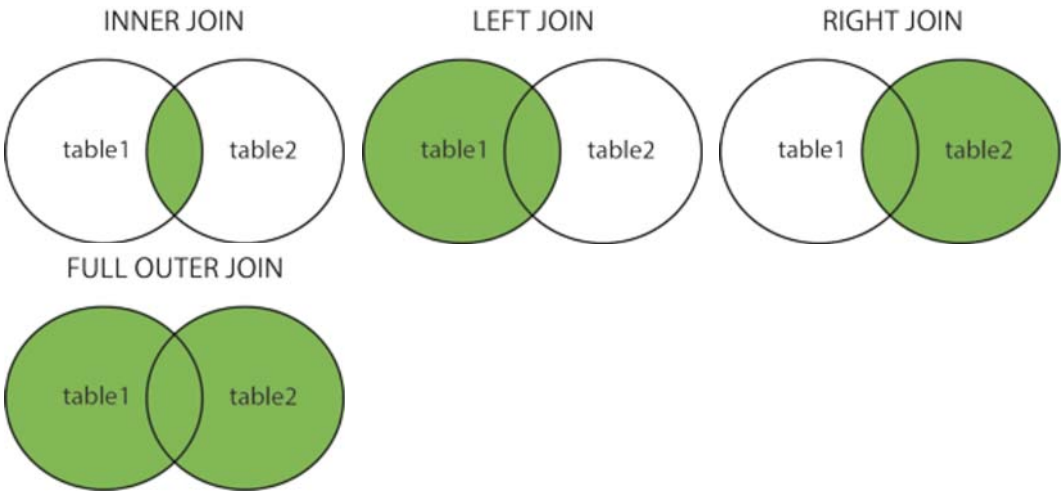
and it will produce something like this:

| OrderID | CustomerName | OrderDate |
|---|---|---|
| 10308 | Ana Trujillo Emparedados y helados | 9/18/1996 |
| 10365 | Antonio Moreno Taquería | 11/27/1996 |
| 10383 | Around the Horn | 12/16/1996 |
| 10355 | Around the Horn | 11/15/1996 |
| 10278 | Berglunds snabbköp | 8/12/1996 |

# Different Types of SQL JOINs

Here are the different types of the JOINs in SQL:

- **(INNER) JOIN**: Returns records that have matching values in both tables
- **LEFT (OUTER) JOIN**: Return all records from the left table, and the matched records from the right table
- **RIGHT (OUTER) JOIN**: Return all records from the right table, and the matched records from the left table
- **FULL (OUTER) JOIN**: Return all records when there is a match in either left or right table



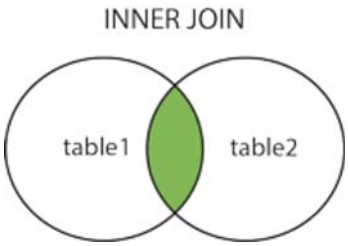
# SQL INNER JOIN Keyword

## SQL INNER JOIN Keyword

The INNER JOIN keyword selects records that have matching values in both tables.

### INNER JOIN Syntax

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




## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Orders" table:

| OrderID | CustomerID | EmployeeID | OrderDate | ShipperID |
|---|---|---|---|---|
| 10308 | 2 | 7 | 1996-09-18 | 3 |
| 10309 | 37 | 3 | 1996-09-19 | 1 |
| 10310 | 77 | 8 | 1996-09-20 | 2 |

And a selection from the "Customers" table:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |

# SQL INNER JOIN Example

The following SQL statement selects all orders with customer information:

## Example

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

**Note:** The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns. If there are records in the "Orders" table that do not have matches in "Customers", these orders will not show!

# JOIN Three Tables

The following SQL statement selects all orders with customer and shipper information:

## Example

```
SELECT Orders.OrderID, Customers.CustomerName, Shippers.ShipperName
FROM ((Orders
INNER JOIN Customers ON Orders.CustomerID = Customers.CustomerID)
INNER JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID);
```

# SQL LEFT JOIN Keyword

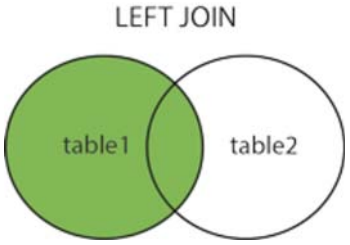
## SQL LEFT JOIN Keyword

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

### LEFT JOIN Syntax

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

**Note:** In some databases LEFT JOIN is called LEFT OUTER JOIN.




## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |

And a selection from the "Orders" table:

| OrderID | CustomerID | EmployeeID | OrderDate | ShipperID |
|---|---|---|---|---|
| 10308 | 2 | 7 | 1996-09-18 | 3 |
| 10309 | 37 | 3 | 1996-09-19 | 1 |
| 10310 | 77 | 8 | 1996-09-20 | 2 |

# SQL LEFT JOIN Example

The following SQL statement will select all customers, and any orders they might have:

## Example

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

**Note:** The LEFT JOIN keyword returns all records from the left table (Customers), even if there are no matches in the right table (Orders).

# SQL RIGHT JOIN Keyword

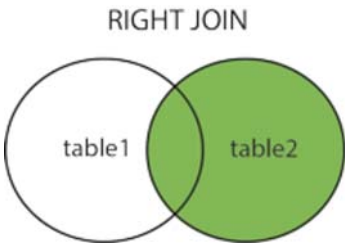
## SQL RIGHT JOIN Keyword

The RIGHT JOIN keyword returns all records from the right table (table2), and the matched records from the left table (table1). The result is NULL from the left side, when there is no match.

### RIGHT JOIN Syntax

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

**Note:** In some databases RIGHT JOIN is called RIGHT OUTER JOIN.




## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Orders" table:

| OrderID | CustomerID | EmployeeID | OrderDate | ShipperID |
|---|---|---|---|---|
| 10308 | 2 | 7 | 1996-09-18 | 3 |
| 10309 | 37 | 3 | 1996-09-19 | 1 |
| 10310 | 77 | 8 | 1996-09-20 | 2 |

And a selection from the "Employees" table:

| EmployeeID | LastName | FirstName | BirthDate | Photo |
|---|---|---|---|---|
| 1 | Davolio | Nancy | 12/8/1968 | EmpID1.pic |
| 2 | Fuller | Andrew | 2/19/1952 | EmpID2.pic |
| 3 | Leverling | Janet | 8/30/1963 | EmpID3.pic |

# SQL RIGHT JOIN Example

The following SQL statement will return all employees, and any orders they might have have placed:

## Example

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

**Note:** The RIGHT JOIN keyword returns all records from the right table (Employees), even if there are no matches in the left table (Orders).

# SQL FULL OUTER JOIN Keyword

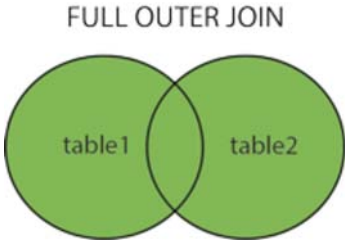
## SQL FULL OUTER JOIN Keyword

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

**Note:** FULL OUTER JOIN can potentially return very large result-sets!

### FULL OUTER JOIN Syntax

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




## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |

And a selection from the "Orders" table:

| OrderID | CustomerID | EmployeeID | OrderDate | ShipperID |
|---|---|---|---|---|
| 10308 | 2 | 7 | 1996-09-18 | 3 |
| 10309 | 37 | 3 | 1996-09-19 | 1 |
| 10310 | 77 | 8 | 1996-09-20 | 2 |

# SQL FULL OUTER JOIN Example

The following SQL statement selects all customers, and all orders:

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

A selection from the result set may look like this:

| CustomerName | OrderID |
|---|---|
| Alfreds Futterkiste | |
| Ana Trujillo Emparedados y helados | 10308 |
| Antonio Moreno Taquería | 10365 |
| | 10382 |
| | 10351 |

**Note:** The FULL OUTER JOIN keyword returns all the rows from the left table (Customers), and all the rows from the right table (Orders). If there are rows in "Customers" that do not have matches in "Orders", or if there are rows in "Orders" that do not have matches in "Customers", those rows will be listed as well.

# SQL Self JOIN

A self JOIN is a regular join, but the table is joined with itself.

## Self JOIN Syntax

```
SELECT column_name(s)
FROM table1 T1, table1 T2
WHERE condition;
```

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |

## SQL Self JOIN Example

The following SQL statement matches customers that are from the same city:

### Example

```
SELECT A.CustomerName AS CustomerName1, B.CustomerName AS CustomerName2, A.City
FROM Customers A, Customers B
WHERE A.CustomerID <> B.CustomerID
AND A.City = B.City
ORDER BY A.City;
```

# SQL UNION Operator

The UNION operator is used to combine the result-set of two or more SELECT statements.

- Each SELECT statement within UNION must have the same number of columns
- The columns must also have similar data types
- The columns in each SELECT statement must also be in the same order

## UNION Syntax

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

## UNION ALL Syntax

The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL:

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

**Note:** The column names in the result-set are usually equal to the column names in the first SELECT statement in the UNION.

# Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |

And a selection from the "Suppliers" table:

| SupplierID | SupplierName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Exotic Liquid | Charlotte Cooper | 49 Gilbert St. | London | EC1 4SD | UK |
| 2 | New Orleans Cajun Delights | Shelley Burke | P.O. Box 78934 | New Orleans | 70117 | USA |
| 3 | Grandma Kelly's Homestead | Regina Murphy | 707 Oxford Rd. | Ann Arbor | 48104 | USA |

# SQL UNION Example

The following SQL statement selects all the different cities (only distinct values) from "Customers" and "Suppliers":

## Example

```
SELECT City FROM Customers
UNION
SELECT City FROM Suppliers
ORDER BY City;
```

**Note:** If some customers or suppliers have the same city, each city will only be listed once, because UNION selects only distinct values. Use UNION ALL to also select duplicate values!

# SQL UNION ALL Example

The following SQL statement selects all cities (duplicate values also) from "Customers" and "Suppliers":

## Example

```
SELECT City FROM Customers
UNION ALL
SELECT City FROM Suppliers
ORDER BY City;
```

# SQL UNION With WHERE

The following SQL statement selects all the different German cities (only distinct values) from "Customers" and "Suppliers":

## Example

```
SELECT City, Country FROM Customers
WHERE Country='Germany'
UNION
SELECT City, Country FROM Suppliers
WHERE Country='Germany'
ORDER BY City;
```

# SQL UNION ALL With WHERE

The following SQL statement selects all German cities (duplicate values also) from "Customers" and "Suppliers":

## Example

```
SELECT City, Country FROM Customers
WHERE Country='Germany'
UNION ALL
SELECT City, Country FROM Suppliers
WHERE Country='Germany'
ORDER BY City;
```

# Another UNION Example

The following SQL statement lists all customers and suppliers:

## Example

```
SELECT 'Customer' As Type, ContactName, City, Country
FROM Customers
UNION
SELECT 'Supplier', ContactName, City, Country
FROM Suppliers;
```

# SQL GROUP BY Statement

## The SQL GROUP BY Statement

The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

### GROUP BY Syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

## Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# SQL GROUP BY Examples

The following SQL statement lists the number of customers in each country:

## Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country;
```

The following SQL statement lists the number of customers in each country, sorted high to low:

## Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
ORDER BY COUNT(CustomerID) DESC;
```

# Demo Database

Below is a selection from the "Orders" table in the Northwind sample database:

| OrderID | CustomerID | EmployeeID | OrderDate | ShipperID |
|---|---|---|---|---|
| 10248 | 90 | 5 | 1996-07-04 | 3 |
| 10249 | 81 | 6 | 1996-07-05 | 1 |
| 10250 | 34 | 4 | 1996-07-08 | 2 |

And a selection from the "Shippers" table:

| ShipperID | ShipperName |
|---|---|
| 1 | Speedy Express |
| 2 | United Package |
| 3 | Federal Shipping |

# GROUP BY With JOIN Example

The following SQL statement lists the number of orders sent by each shipper:

## Example

```
SELECT Shippers.ShipperName, COUNT(Orders.OrderID) AS NumberOfOrders FROM Orders
LEFT JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID
GROUP BY ShipperName;
```

# SQL HAVING Clause

## The SQL HAVING Clause

The HAVING clause was added to SQL because the WHERE keyword could not be used with aggregate functions.

### HAVING Syntax

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

## Demo Database

Below is a selection from the "Customers" table in the Northwind sample database:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# SQL HAVING Examples

The following SQL statement lists the number of customers in each country. Only include countries with more than 5 customers:

## Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5;
```

The following SQL statement lists the number of customers in each country, sorted high to low (Only include countries with more than 5 customers):

## Example

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5
ORDER BY COUNT(CustomerID) DESC;
```

# Demo Database

Below is a selection from the "Orders" table in the Northwind sample database:

| OrderID | CustomerID | EmployeeID | OrderDate | ShipperID |
|---|---|---|---|---|
| 10248 | 90 | 5 | 1996-07-04 | 3 |
| 10249 | 81 | 6 | 1996-07-05 | 1 |
| 10250 | 34 | 4 | 1996-07-08 | 2 |

And a selection from the "Employees" table:

| EmployeeID | LastName | FirstName | BirthDate | Photo | Notes |
|---|---|---|---|---|---|
| 1 | Davolio | Nancy | 1968-12-08 | EmpID1.pic | Education includes a BA.... |
| 2 | Fuller | Andrew | 1952-02-19 | EmpID2.pic | Andrew received his BTS.... |
| 3 | Leverling | Janet | 1963-08-30 | EmpID3.pic | Janet has a BS degree.... |

# More HAVING Examples

The following SQL statement lists the employees that have registered more than 10 orders:

## Example

```
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders
FROM (Orders
INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID)
GROUP BY LastName
HAVING COUNT(Orders.OrderID) > 10;
```

The following SQL statement lists if the employees "Davolio" or "Fuller" have registered more than 25 orders:

## Example

```
SELECT Employees.LastName, COUNT(Orders.OrderID) AS NumberOfOrders
FROM Orders
INNER JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID
WHERE LastName = 'Davolio' OR LastName = 'Fuller'
GROUP BY LastName
HAVING COUNT(Orders.OrderID) > 25;
```

# SQL SELECT INTO Statement

## The SQL SELECT INTO Statement

The SELECT INTO statement copies data from one table into a new table.

### SELECT INTO Syntax

Copy all columns into a new table:

```
SELECT *
INTO newtable [IN externaldb]
FROM oldtable
WHERE condition;
```

Copy only some columns into a new table:

```
SELECT column1, column2, column3, ...
INTO newtable [IN externaldb]
FROM oldtable
WHERE condition;
```

The new table will be created with the column-names and types as defined in the old table. You can create new column names using the AS clause.

## SQL SELECT INTO Examples

The following SQL statement creates a backup copy of Customers:

```
SELECT * INTO CustomersBackup2017
FROM Customers;
```

The following SQL statement uses the IN clause to copy the table into a new table in another database:

```
SELECT * INTO CustomersBackup2017 IN 'Backup.mdb'
FROM Customers;
```

The following SQL statement copies only a few columns into a new table:

```
SELECT CustomerName, ContactName INTO CustomersBackup2017
FROM Customers;
```

The following SQL statement copies only the German customers into a new table:

```
SELECT * INTO CustomersGermany
FROM Customers
WHERE Country = 'Germany';
```

The following SQL statement copies data from more than one table into a new table:

```
SELECT Customers.CustomerName, Orders.OrderID
INTO CustomersOrderBackup2017
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

**Tip:** SELECT INTO can also be used to create a new, empty table using the schema of another. Just add a WHERE clause that causes the query to return no data:

```
SELECT * INTO newtable
FROM oldtable
WHERE 1 = 0;
```

# SQL INSERT INTO SELECT Statement

## The SQL INSERT INTO SELECT Statement

The INSERT INTO SELECT statement copies data from one table and inserts it into another table.

- INSERT INTO SELECT requires that data types in source and target tables match
- The existing records in the target table are unaffected

### INSERT INTO SELECT Syntax

Copy all columns from one table to another table:

```
INSERT INTO table2
SELECT * FROM table1
WHERE condition;
```

Copy only some columns from one table into another table:

```
INSERT INTO table2 (column1, column2, column3, ...)
SELECT column1, column2, column3, ...
FROM table1
WHERE condition;
```

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |

And a selection from the "Suppliers" table:

| SupplierID | SupplierName | ContactName | Address | City | Postal Code | Country | Phone |
|---|---|---|---|---|---|---|---|
| 1 | Exotic Liquid | Charlotte Cooper | 49 Gilbert St. | Londona | EC1 4SD | UK | (171) 555-2222 |
| 2 | New Orleans Cajun Delights | Shelley Burke | P.O. Box 78934 | New Orleans | 70117 | USA | (100) 555-4822 |
| 3 | Grandma Kelly's Homestead | Regina Murphy | 707 Oxford Rd. | Ann Arbor | 48104 | USA | (313) 555-5735 |

# SQL INSERT INTO SELECT Examples

The following SQL statement copies "Suppliers" into "Customers" (the columns that are not filled with data, will contain NULL):

## Example

```
INSERT INTO Customers (CustomerName, City, Country)
SELECT SupplierName, City, Country FROM Suppliers;
```

The following SQL statement copies "Suppliers" into "Customers" (fill all columns):

## Example

```
INSERT INTO Customers (CustomerName, ContactName, Address, City, PostalCode,
Country)
SELECT SupplierName, ContactName, Address, City, PostalCode, Country FROM
Suppliers;
```

The following SQL statement copies only the German suppliers into "Customers":

## Example

```
INSERT INTO Customers (CustomerName, City, Country)
SELECT SupplierName, City, Country FROM Suppliers
WHERE Country='Germany';
```

# SQL CREATE DATABASE Statement

## The SQL CREATE DATABASE Statement

The CREATE DATABASE statement is used to create a new SQL database.

### Syntax

```
CREATE DATABASE databasename;
```

---

## CREATE DATABASE Example

The following SQL statement creates a database called "testDB":

### Example

```
CREATE DATABASE testDB;
```

**Tip:** Make sure you have admin privilege before creating any database. Once a database is created, you can check it in the list of databases with the following SQL command: SHOW DATABASES;

# SQL DROP DATABASE Statement

## The SQL DROP DATABASE Statement

The DROP DATABASE statement is used to drop an existing SQL database.

### Syntax

```
DROP DATABASE databasename;
```

**Note:** Be careful before dropping a database. Deleting a database will result in loss of complete information stored in the database!

---

## DROP DATABASE Example

The following SQL statement drops the existing database "testDB":

### Example

```
DROP DATABASE testDB;
```

**Tip:** Make sure you have admin privilege before dropping any database. Once a database is dropped, you can check it in the list of databases with the following SQL command: SHOW DATABASES;

# SQL CREATE TABLE Statement

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

## Syntax

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

The column parameters specify the names of the columns of the table.

The datatype parameter specifies the type of data the column can hold (e.g. varchar, integer, date, etc.).

# SQL CREATE TABLE Example

The following example creates a table called "Persons" that contains five columns: PersonID, LastName, FirstName, Address, and City:

## Example

```
CREATE TABLE Persons (
    PersonID int,
    LastName varchar(255),
    FirstName varchar(255),
    Address varchar(255),
    City varchar(255)
);
```

The PersonID column is of type int and will hold an integer.

The LastName, FirstName, Address, and City columns are of type varchar and will hold characters, and the maximum length for these fields is 255 characters.

The empty "Persons" table will now look like this:

| PersonID | LastName | FirstName | Address | City |
|---|---|---|---|---|
| | | | | |

**Tip:** The empty "Persons" table can now be filled with data with the SQL INSERT INTO statement.

# Create Table Using Another Table

A copy of an existing table can be created using a combination of the CREATE TABLE statement and the SELECT statement.

The new table gets the same column definitions. All columns or specific columns can be selected.

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

## Syntax

```
CREATE TABLE new_table_name AS
    SELECT column1, column2,...
    FROM existing_table_name
    WHERE ....;
```

# SQL DROP TABLE Statement

The DROP TABLE statement is used to drop an existing table in a database.

## Syntax

```
DROP TABLE table_name;
```

**Note:** Be careful before dropping a table. Deleting a table will result in loss of complete information stored in the table!

## SQL DROP TABLE Example

The following SQL statement drops the existing table "Shippers":

### Example

```
DROP TABLE Shippers;
```

## SQL TRUNCATE TABLE

The TRUNCATE TABLE statement is used to delete the data inside a table, but not the table itself.

## Syntax

```
TRUNCATE TABLE table_name;
```

# SQL ALTER TABLE Statement

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

The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

## ALTER TABLE - ADD Column

To add a column in a table, use the following syntax:

```
ALTER TABLE table_name
ADD column_name datatype;
```

## ALTER TABLE - DROP COLUMN

To delete a column in a table, use the following syntax (notice that some database systems don't allow deleting a column):

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

---

## ALTER TABLE - ALTER/MODIFY COLUMN

To change the data type of a column in a table, use the following syntax:

**SQL Server / MS Access:**

```
ALTER TABLE table_name
ALTER COLUMN column_name datatype;
```

**My SQL / Oracle (prior version 10G):**

```
ALTER TABLE table_name
MODIFY COLUMN column_name datatype;
```

**Oracle 10G and later:**

```
ALTER TABLE table_name
MODIFY column_name datatype;
```

# SQL ALTER TABLE Example

Look at the "Persons" table:

| ID | LastName | FirstName | Address | City |
|---|---|---|---|---|
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

Now we want to add a column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

```
ALTER TABLE Persons
ADD DateOfBirth date;
```

Notice that the new column, "DateOfBirth", is of type date and is going to hold a date. The data type specifies what type of data the column can hold. For a complete reference of all the data types available in MS Access, MySQL, and SQL Server, go to our complete Data Types reference.

The "Persons" table will now look like this:

| ID | LastName | FirstName | Address | City | DateOfBirth |
|---|---|---|---|---|---|
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes | |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes | |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger | |

# Change Data Type Example

Now we want to change the data type of the column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

```
ALTER TABLE Persons
ALTER COLUMN DateOfBirth year;
```

Notice that the "DateOfBirth" column is now of type year and is going to hold a year in a two- or four-digit format.

---

# DROP COLUMN Example

Next, we want to delete the column named "DateOfBirth" in the "Persons" table.

We use the following SQL statement:

```
ALTER TABLE Persons
DROP COLUMN DateOfBirth;
```

The "Persons" table will now look like this:

| ID | LastName | FirstName | Address | City |
|---|---|---|---|---|
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

# SQL Constraints

SQL constraints are used to specify rules for data in a table.

## SQL Create Constraints

Constraints can be specified when the table is created with the CREATE TABLE statement, or after the table is created with the ALTER TABLE statement.

### Syntax

```
CREATE TABLE table_name (
    column1 datatype constraint,
    column2 datatype constraint,
    column3 datatype constraint,
    ....
);
```

## SQL Constraints

SQL constraints are used to specify rules for the data in a table.

Constraints are used to limit the type of data that can go into a table. This ensures the accuracy and reliability of the data in the table. If there is any violation between the constraint and the data action, the action is aborted.

Constraints can be column level or table level. Column level constraints apply to a column, and table level constraints apply to the whole table.

The following constraints are commonly used in SQL:

- **NOT NULL** - Ensures that a column cannot have a NULL value
- **UNIQUE** - Ensures that all values in a column are different
- **PRIMARY KEY** - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table
- **FOREIGN KEY** - Uniquely identifies a row/record in another table
- **CHECK** - Ensures that all values in a column satisfies a specific condition
- **DEFAULT** - Sets a default value for a column when no value is specified
- **INDEX** - Use to create and retrieve data from the database very quickly

# SQL NOT NULL Constraint

## SQL NOT NULL Constraint

By default, a column can hold NULL values.

The NOT NULL constraint enforces a column to NOT accept NULL values.

This enforces a field to always contain a value, which means that you cannot insert a new record, or update a record without adding a value to this field.

The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values:

### Example

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int
);
```

**Tip:** If the table has already been created, you can add a NOT NULL constraint to a column with the ALTER TABLE statement.

# SQL UNIQUE Constraint

The UNIQUE constraint ensures that all values in a column are different.

Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.

A PRIMARY KEY constraint automatically has a UNIQUE constraint.

However, you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

## SQL UNIQUE Constraint on CREATE TABLE

The following SQL creates a UNIQUE constraint on the "ID" column when the "Persons" table is created:

**SQL Server / Oracle / MS Access:**

```
CREATE TABLE Persons (
    ID int NOT NULL UNIQUE,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
);
```

**MySQL:**

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    UNIQUE (ID)
);
```

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

**MySQL / SQL Server / Oracle / MS Access:**

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT UC_Person UNIQUE (ID,LastName)
);
```

# SQL UNIQUE Constraint on ALTER TABLE

To create a UNIQUE constraint on the "ID" column when the table is already created, use the following SQL:

**MySQL / SQL Server / Oracle / MS Access:**

```
ALTER TABLE Persons
ADD UNIQUE (ID);
```

To name a UNIQUE constraint, and to define a UNIQUE constraint on multiple columns, use the following SQL syntax:

**MySQL / SQL Server / Oracle / MS Access:**

```
ALTER TABLE Persons
ADD CONSTRAINT UC_Person UNIQUE (ID,LastName);
```

# DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL:

**MySQL:**

```
ALTER TABLE Persons
DROP INDEX UC_Person;
```

**SQL Server / Oracle / MS Access:**

```
ALTER TABLE Persons
DROP CONSTRAINT UC_Person;
```

# SQL PRIMARY KEY Constraint

The PRIMARY KEY constraint uniquely identifies each record in a database table.

Primary keys must contain UNIQUE values, and cannot contain NULL values.

A table can have only one primary key, which may consist of single or multiple fields.

---

## SQL PRIMARY KEY on CREATE TABLE

The following SQL creates a PRIMARY KEY on the "ID" column when the "Persons" table is created:

**MySQL:**

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    PRIMARY KEY (ID)
);
```

**SQL Server / Oracle / MS Access:**

```
CREATE TABLE Persons (
    ID int NOT NULL PRIMARY KEY,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
);
```

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

**MySQL / SQL Server / Oracle / MS Access:**

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CONSTRAINT PK_Person PRIMARY KEY (ID,LastName)
);
```

**Note:** In the example above there is only ONE PRIMARY KEY (PK_Person). However, the VALUE of the primary key is made up of TWO COLUMNS (ID + LastName).

---

# SQL PRIMARY KEY on ALTER TABLE

To create a PRIMARY KEY constraint on the "ID" column when the table is already created, use the following SQL:

**MySQL / SQL Server / Oracle / MS Access:**

```
ALTER TABLE Persons
ADD PRIMARY KEY (ID);
```

To allow naming of a PRIMARY KEY constraint, and for defining a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

**MySQL / SQL Server / Oracle / MS Access:**

```
ALTER TABLE Persons
ADD CONSTRAINT PK_Person PRIMARY KEY (ID,LastName);
```

**Note:** If you use the ALTER TABLE statement to add a primary key, the primary key column(s) must already have been declared to not contain NULL values (when the table was first created).

---

# DROP a PRIMARY KEY Constraint

To drop a PRIMARY KEY constraint, use the following SQL:

**MySQL:**

```
ALTER TABLE Persons
DROP PRIMARY KEY;
```

**SQL Server / Oracle / MS Access:**

```
ALTER TABLE Persons
DROP CONSTRAINT PK_Person;
```

# SQL FOREIGN KEY Constraint

A FOREIGN KEY is a key used to link two tables together.

A FOREIGN KEY in a table points to a PRIMARY KEY in another table.

Look at the following two tables:

"Persons" table:

| PersonID | LastName | FirstName | Age |
|---|---|---|---|
| 1 | Hansen | Ola | 30 |
| 2 | Svendson | Tove | 23 |
| 3 | Pettersen | Kari | 20 |

"Orders" table:

| OrderID | OrderNumber | PersonID |
|---|---|---|
| 1 | 77895 | 3 |
| 2 | 44678 | 3 |
| 3 | 22456 | 2 |
| 4 | 24562 | 1 |

Notice that the "PersonID" column in the "Orders" table points to the "PersonID" column in the "Persons" table.

The "PersonID" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.

The "PersonID" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.

The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.

The FOREIGN KEY constraint also prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the table it points to.

# SQL FOREIGN KEY on CREATE TABLE

The following SQL creates a FOREIGN KEY on the "PersonID" column when the "Orders" table is created:

**MySQL:**

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

**SQL Server / Oracle / MS Access:**

```
CREATE TABLE Orders (
    OrderID int NOT NULL PRIMARY KEY,
    OrderNumber int NOT NULL,
    PersonID int FOREIGN KEY REFERENCES Persons(PersonID)
);
```

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

**MySQL / SQL Server / Oracle / MS Access:**

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID)
    REFERENCES Persons(PersonID)
);
```

# SQL FOREIGN KEY on ALTER TABLE

To create a FOREIGN KEY constraint on the "PersonID" column when the "Orders" table is already created, use the following SQL:

**MySQL / SQL Server / Oracle / MS Access:**

```
ALTER TABLE Orders
ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

To allow naming of a FOREIGN KEY constraint, and for defining a FOREIGN KEY constraint on multiple columns, use the following SQL syntax:

**MySQL / SQL Server / Oracle / MS Access:**

```
ALTER TABLE Orders
ADD CONSTRAINT FK_PersonOrder
FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

---

# DROP a FOREIGN KEY Constraint

To drop a FOREIGN KEY constraint, use the following SQL:

**MySQL:**

```
ALTER TABLE Orders
DROP FOREIGN KEY FK_PersonOrder;
```

**SQL Server / Oracle / MS Access:**

```
ALTER TABLE Orders
DROP CONSTRAINT FK_PersonOrder;
```

# SQL CHECK Constraint

The CHECK constraint is used to limit the value range that can be placed in a column.

If you define a CHECK constraint on a single column it allows only certain values for this column.

If you define a CHECK constraint on a table it can limit the values in certain columns based on values in other columns in the row.

## SQL CHECK on CREATE TABLE

The following SQL creates a CHECK constraint on the "Age" column when the "Persons" table is created. The CHECK constraint ensures that you can not have any person below 18 years:

**MySQL:**

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    CHECK (Age>=18)
);
```

**SQL Server / Oracle / MS Access:**

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int CHECK (Age>=18)
);
```

To allow naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns, use the following SQL syntax:

**MySQL / SQL Server / Oracle / MS Access:**

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255),
    CONSTRAINT CHK_Person CHECK (Age>=18 AND City='Sandnes')
);
```

# SQL CHECK on ALTER TABLE

To create a CHECK constraint on the "Age" column when the table is already created, use the following SQL:

**MySQL / SQL Server / Oracle / MS Access:**

```
ALTER TABLE Persons
ADD CHECK (Age>=18);
```

To allow naming of a CHECK constraint, and for defining a CHECK constraint on multiple columns, use the following SQL syntax:

**MySQL / SQL Server / Oracle / MS Access:**

```
ALTER TABLE Persons
ADD CONSTRAINT CHK_PersonAge CHECK (Age>=18 AND City='Sandnes');
```

# DROP a CHECK Constraint

To drop a CHECK constraint, use the following SQL:

**SQL Server / Oracle / MS Access:**

```
ALTER TABLE Persons
DROP CONSTRAINT CHK_PersonAge;
```

**MySQL:**

```
ALTER TABLE Persons
DROP CHECK CHK_PersonAge;
```

# SQL DEFAULT Constraint

The DEFAULT constraint is used to provide a default value for a column.

The default value will be added to all new records IF no other value is specified.

## SQL DEFAULT on CREATE TABLE

The following SQL sets a DEFAULT value for the "City" column when the "Persons" table is created:

**My SQL / SQL Server / Oracle / MS Access:**

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255) DEFAULT 'Sandnes'
);
```

The DEFAULT constraint can also be used to insert system values, by using functions like GETDATE():

```
CREATE TABLE Orders (
    ID int NOT NULL,
    OrderNumber int NOT NULL,
    OrderDate date DEFAULT GETDATE()
);
```

# SQL DEFAULT on ALTER TABLE

To create a DEFAULT constraint on the "City" column when the table is already created, use the following SQL:

**MySQL:**

```
ALTER TABLE Persons
ALTER City SET DEFAULT 'Sandnes';
```

**SQL Server / MS Access:**

```
ALTER TABLE Persons
ALTER COLUMN City SET DEFAULT 'Sandnes';
```

**Oracle:**

```
ALTER TABLE Persons
MODIFY City DEFAULT 'Sandnes';
```

# DROP a DEFAULT Constraint

To drop a DEFAULT constraint, use the following SQL:

**MySQL:**

```
ALTER TABLE Persons
ALTER City DROP DEFAULT;
```

**SQL Server / Oracle / MS Access:**

```
ALTER TABLE Persons
ALTER COLUMN City DROP DEFAULT;
```

# SQL CREATE INDEX Statement

The CREATE INDEX statement is used to create indexes in tables.

Indexes are used to retrieve data from the database very fast. The users cannot see the indexes, they are just used to speed up searches/queries.

> **Note:** Updating a table with indexes takes more time than updating a table without (because the indexes also need an update). So, only create indexes on columns that will be frequently searched against.

## CREATE INDEX Syntax

Creates an index on a table. Duplicate values are allowed:

```
CREATE INDEX index_name
ON table_name (column1, column2, ...);
```

## CREATE UNIQUE INDEX Syntax

Creates a unique index on a table. Duplicate values are not allowed:

```
CREATE UNIQUE INDEX index_name
ON table_name (column1, column2, ...);
```

**Note:** The syntax for creating indexes varies among different databases. Therefore: Check the syntax for creating indexes in your database.

# CREATE INDEX Example

The SQL statement below creates an index named "idx_lastname" on the "LastName" column in the "Persons" table:

```
CREATE INDEX idx_lastname
ON Persons (LastName);
```

If you want to create an index on a combination of columns, you can list the column names within the parentheses, separated by commas:

```
CREATE INDEX idx_pname
ON Persons (LastName, FirstName);
```

# DROP INDEX Statement

The DROP INDEX statement is used to delete an index in a table.

**MS Access:**

```
DROP INDEX index_name ON table_name;
```

**SQL Server:**

```
DROP INDEX table_name.index_name;
```

**DB2 / Oracle:**

```
DROP INDEX index_name;
```

**MySQL:**

```
ALTER TABLE table_name
DROP INDEX index_name;
```

# SQL AUTO INCREMENT Field

## AUTO INCREMENT Field

Auto-increment allows a unique number to be generated automatically when a new record is inserted into a table.

Often this is the primary key field that we would like to be created automatically every time a new record is inserted.

---

## Syntax for MySQL

The following SQL statement defines the "ID" column to be an auto-increment primary key field in the "Persons" table:

```
CREATE TABLE Persons (
    ID int NOT NULL AUTO_INCREMENT,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    PRIMARY KEY (ID)
);
```

MySQL uses the AUTO_INCREMENT keyword to perform an auto-increment feature.

By default, the starting value for AUTO_INCREMENT is 1, and it will increment by 1 for each new record.

To let the AUTO_INCREMENT sequence start with another value, use the following SQL statement:

```
ALTER TABLE Persons AUTO_INCREMENT=100;
```

To insert a new record into the "Persons" table, we will NOT have to specify a value for the "ID" column (a unique value will be added automatically):

```
INSERT INTO Persons (FirstName,LastName)
VALUES ('Lars','Monsen');
```

The SQL statement above would insert a new record into the "Persons" table. The "ID" column would be assigned a unique value. The "FirstName" column would be set to "Lars" and the "LastName" column would be set to "Monsen".

# Syntax for SQL Server

The following SQL statement defines the "ID" column to be an auto-increment primary key field in the "Persons" table:

```
CREATE TABLE Persons (
    ID int IDENTITY(1,1) PRIMARY KEY,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
);
```

The MS SQL Server uses the IDENTITY keyword to perform an auto-increment feature.

In the example above, the starting value for IDENTITY is 1, and it will increment by 1 for each new record.

**Tip:** To specify that the "ID" column should start at value 10 and increment by 5, change it to IDENTITY(10,5).

To insert a new record into the "Persons" table, we will NOT have to specify a value for the "ID" column (a unique value will be added automatically):

```
INSERT INTO Persons (FirstName,LastName)
VALUES ('Lars','Monsen');
```

The SQL statement above would insert a new record into the "Persons" table. The "ID" column would be assigned a unique value. The "FirstName" column would be set to "Lars" and the "LastName" column would be set to "Monsen".

# Syntax for Access

The following SQL statement defines the "ID" column to be an auto-increment primary key field in the "Persons" table:

```
CREATE TABLE Persons (
    ID Integer PRIMARY KEY AUTOINCREMENT,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
);
```

The MS Access uses the AUTOINCREMENT keyword to perform an auto-increment feature.

By default, the starting value for AUTOINCREMENT is 1, and it will increment by 1 for each new record.

**Tip:** To specify that the "ID" column should start at value 10 and increment by 5, change the autoincrement to AUTOINCREMENT(10,5).

To insert a new record into the "Persons" table, we will NOT have to specify a value for the "ID" column (a unique value will be added automatically):

```
INSERT INTO Persons (FirstName,LastName)
VALUES ('Lars','Monsen');
```

The SQL statement above would insert a new record into the "Persons" table. The "P_Id" column would be assigned a unique value. The "FirstName" column would be set to "Lars" and the "LastName" column would be set to "Monsen".

# Syntax for Oracle

In Oracle the code is a little bit more tricky.

You will have to create an auto-increment field with the sequence object (this object generates a number sequence).

Use the following CREATE SEQUENCE syntax:

```
CREATE SEQUENCE seq_person
MINVALUE 1
START WITH 1
INCREMENT BY 1
CACHE 10;
```

The code above creates a sequence object called seq_person, that starts with 1 and will increment by 1. It will also cache up to 10 values for performance. The cache option specifies how many sequence values will be stored in memory for faster access.

To insert a new record into the "Persons" table, we will have to use the nextval function (this function retrieves the next value from seq_person sequence):

```
INSERT INTO Persons (ID,FirstName,LastName)
VALUES (seq_person.nextval,'Lars','Monsen');
```

The SQL statement above would insert a new record into the "Persons" table. The "ID" column would be assigned the next number from the seq_person sequence. The "FirstName" column would be set to "Lars" and the "LastName" column would be set to "Monsen".

# SQL Views

## SQL CREATE VIEW Statement

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

### CREATE VIEW Syntax

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

**Note:** A view always shows up-to-date data! The database engine recreates the data, using the view's SQL statement, every time a user queries a view.

## SQL CREATE VIEW Examples

If you have the Northwind database you can see that it has several views installed by default.

The view "Current Product List" lists all active products (products that are not discontinued) from the "Products" table. The view is created with the following SQL:

```
CREATE VIEW [Current Product List] AS
SELECT ProductID, ProductName
FROM Products
WHERE Discontinued = No;
```

Then, we can query the view as follows:

```
SELECT * FROM [Current Product List];
```

Another view in the Northwind sample database selects every product in the "Products" table with a unit price higher than the average unit price:

```
CREATE VIEW [Products Above Average Price] AS
SELECT ProductName, UnitPrice
FROM Products
WHERE UnitPrice > (SELECT AVG(UnitPrice) FROM Products);
```

We can query the view above as follows:

```
SELECT * FROM [Products Above Average Price];
```

Another view in the Northwind database calculates the total sale for each category in 1997. Note that this view selects its data from another view called "Product Sales for 1997":

```
CREATE VIEW [Category Sales For 1997] AS
SELECT DISTINCT CategoryName, Sum(ProductSales) AS CategorySales
FROM [Product Sales for 1997]
GROUP BY CategoryName;
```

We can query the view above as follows:

```
SELECT * FROM [Category Sales For 1997];
```

We can also add a condition to the query. Let's see the total sale only for the category "Beverages":

```
SELECT * FROM [Category Sales For 1997]
WHERE CategoryName = 'Beverages';
```

# SQL Updating a View

You can update a view by using the following syntax:

## SQL CREATE OR REPLACE VIEW Syntax

```
CREATE OR REPLACE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

Now we want to add the "Category" column to the "Current Product List" view. We will update the view with the following SQL:

```
CREATE OR REPLACE VIEW [Current Product List] AS
SELECT ProductID, ProductName, Category
FROM Products
WHERE Discontinued = No;
```

# SQL Dropping a View

You can delete a view with the DROP VIEW command.

## SQL DROP VIEW Syntax

```
DROP VIEW view_name;
```

# SQL Injection

An SQL Injection can destroy your database.

## SQL in Web Pages

In the previous chapters, you have learned to retrieve (and update) database data, using SQL.

When SQL is used to display data on a web page, it is common to let web users input their own search values.

Since SQL statements are text only, it is easy, with a little piece of computer code, to dynamically change SQL statements to provide the user with selected data:

### Server Code

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

The example above, creates a select statement by adding a variable (txtUserId) to a select string. The variable is fetched from the user input (Request) to the page.

The rest of this chapter describes the potential dangers of using user input in SQL statements.

## SQL Injection

SQL injection is a technique where malicious users can inject SQL commands into an SQL statement, via web page input.

Injected SQL commands can alter SQL statement and compromise the security of a web application.

# SQL Injection Based on 1=1 is Always True

Look at the example above, one more time.

Let's say that the original purpose of the code was to create an SQL statement to select a user with a given user id.

If there is nothing to prevent a user from entering "wrong" input, the user can enter some "smart" input like this:

UserId:
105 or 1=1

## Server Result

```
SELECT * FROM Users WHERE UserId = 105 or 1=1;
```

The SQL above is valid. It will return all rows from the table Users, since **WHERE 1=1** is always true.

Does the example above seem dangerous? What if the Users table contains names and passwords?

The SQL statement above is much the same as this:

```
SELECT UserId, Name, Password FROM Users WHERE UserId = 105 or 1=1;
```

A smart hacker might get access to all the user names and passwords in a database by simply inserting 105 or 1=1 into the input box.

---

# SQL Injection Based on ""="" is Always True

Here is a common construction, used to verify user login to a web site:

User Name:
John Doe

Password:
myPass

## Server Code

```
uName = getRequestString("UserName");
uPass = getRequestString("UserPass");

sql = 'SELECT * FROM Users WHERE Name ="' + uName + '" AND Pass ="' +
uPass + '"'
```

## Result

```
SELECT * FROM Users WHERE Name ="John Doe" AND Pass ="myPass"
```

A smart hacker might get access to user names and passwords in a database by simply inserting " or ""=" into the user name or password text box:

User Name:

```
" or ""="
```

Password:

```
" or ""="
```

The code at the server will create a valid SQL statement like this:

## Result

```
SELECT * FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""
```

The result SQL is valid. It will return all rows from the table Users, since **WHERE ""=""** is always true.

# SQL Injection Based on Batched SQL Statements

Most databases support batched SQL statement, separated by semicolon.

## Example

```
SELECT * FROM Users; DROP TABLE Suppliers
```

The SQL above will return all rows in the Users table, and then delete the table called Suppliers.

If we had the following server code:

## Server Code

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

And the following input:

User id:
105; DROP TABLE Suppliers

The code at the server would create a valid SQL statement like this:

## Result

```
SELECT * FROM Users WHERE UserId = 105; DROP TABLE Suppliers
```

---

# Parameters for Protection

Some web developers use a "blacklist" of words or characters to search for in SQL input, to prevent SQL injection attacks.

This is not a very good idea. Many of these words (like delete or drop) and characters (like semicolons and quotation marks), are used in common language, and should be allowed in many types of input.

(In fact it should be perfectly legal to input an SQL statement in a database field.)

The only proven way to protect a web site from SQL injection attacks, is to use SQL parameters.

SQL parameters are values that are added to an SQL query at execution time, in a controlled manner.

### ASP.NET Razor Example

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = @0";
db.Execute(txtSQL,txtUserId);
```

Note that parameters are represented in the SQL statement by a @ marker.

The SQL engine checks each parameter to ensure that it is correct for its column and are treated literally, and not as part of the SQL to be executed.

### Another Example

```
txtNam = getRequestString("CustomerName");
txtAdd = getRequestString("Address");
txtCit = getRequestString("City");
txtSQL = "INSERT INTO Customers (CustomerName,Address,City)
Values(@0,@1,@2)";
db.Execute(txtSQL,txtNam,txtAdd,txtCit);
```

> You have just learned to avoid SQL injection. One of the top website vulnerabilities.

---

## Examples

The following examples shows how to build parameterized queries in some common web languages.

SELECT STATEMENT IN ASP.NET:

```
txtUserId = getRequestString("UserId");
sql = "SELECT * FROM Customers WHERE CustomerId = @0";
command = new SqlCommand(sql);
command.Parameters.AddWithValue("@0",txtUserID);
command.ExecuteReader();
```

INSERT INTO STATEMENT IN ASP.NET:

```
txtNam = getRequestString("CustomerName");
txtAdd = getRequestString("Address");
txtCit = getRequestString("City");
txtSQL = "INSERT INTO Customers (CustomerName,Address,City)
Values(@0,@1,@2)";
command = new SqlCommand(txtSQL);
command.Parameters.AddWithValue("@0",txtNam);
command.Parameters.AddWithValue("@1",txtAdd);
command.Parameters.AddWithValue("@2",txtCit);
command.ExecuteNonQuery();
```

INSERT INTO STATEMENT IN PHP:

```
$stmt = $dbh->prepare("INSERT INTO Customers
(CustomerName,Address,City)
VALUES (:nam, :add, :cit)");
$stmt->bindParam(':nam', $txtNam);
$stmt->bindParam(':add', $txtAdd);
$stmt->bindParam(':cit', $txtCit);
$stmt->execute();
```

# SQL Hosting

If you want your web site to be able to store and retrieve data from a database, your web server should have access to a database-system that uses the SQL language.

If your web server is hosted by an Internet Service Provider (ISP), you will have to look for SQL hosting plans.

The most common SQL hosting databases are MS SQL Server, Oracle, MySQL, and MS Access.

## MS SQL Server

Microsoft's SQL Server is a popular database software for database-driven web sites with high traffic.

SQL Server is a very powerful, robust and full featured SQL database system.

## Oracle

Oracle is also a popular database software for database-driven web sites with high traffic.

Oracle is a very powerful, robust and full featured SQL database system.

## MySQL

MySQL is also a popular database software for web sites.

MySQL is a very powerful, robust and full featured SQL database system.

MySQL is an inexpensive alternative to the expensive Microsoft and Oracle solutions.

## Access

When a web site requires only a simple database, Microsoft Access can be a solution.

Access is not well suited for very high-traffic, and not as powerful as MySQL, SQL Server, or Oracle.

# SQL Functions

SQL has many built-in functions for performing calculations on data.

## SQL Aggregate Functions

SQL aggregate functions return a single value, calculated from values in a column.

| Function | Description |
|---|---|
| AVG() | Returns the average value |
| COUNT() | Returns the number of rows |
| FIRST() | Returns the first value |
| LAST() | Returns the last value |
| MAX() | Returns the largest value |
| MIN() | Returns the smallest value |
| ROUND() | Rounds a numeric field to the number of decimals specified |
| SUM() | Returns the sum |

# SQL String Functions

| Function | Description |
|---|---|
| CHARINDEX | Searches an expression in a string expression and returns its starting position if found |
| CONCAT() | |
| LEFT() | |
| LEN() / LENGTH() | Returns the length of the value in a text field |
| LOWER() / LCASE() | Converts character data to lower case |
| LTRIM() | |
| SUBSTRING() / MID() | Extract characters from a text field |
| PATINDEX() | |
| REPLACE() | |
| RIGHT() | |
| RTRIM() | |
| UPPER() / UCASE() | Converts character data to upper case |

# SQL AVG() Function

The AVG() function returns the average value of a numeric column.

## SQL AVG() Syntax

```
SELECT AVG(column_name) FROM table_name
```

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Products" table:

| ProductID | ProductName | SupplierID | CategoryID | Unit | Price |
|---|---|---|---|---|---|
| 1 | Chais | 1 | 1 | 10 boxes x 20 bags | 18 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottles | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 12 - 550 ml bottles | 10 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 2 | 48 - 6 oz jars | 21.35 |
| 5 | Chef Anton's Gumbo Mix | 2 | 2 | 36 boxes | 25 |

## SQL AVG() Example

The following SQL statement gets the average value of the "Price" column from the "Products" table:

### Example

```
SELECT AVG(Price) AS PriceAverage FROM Products;
```

The following SQL statement selects the "ProductName" and "Price" records that have an above average price:

## Example

```
SELECT ProductName, Price FROM Products
WHERE Price>(SELECT AVG(Price) FROM Products);
```

# SQL COUNT() Function

The COUNT() function returns the number of rows that matches a specified criteria.

## SQL COUNT(column_name) Syntax

The COUNT(column_name) function returns the number of values (NULL values will not be counted) of the specified column:

```
SELECT COUNT(column_name) FROM table_name;
```

## SQL COUNT(*) Syntax

The COUNT(*) function returns the number of records in a table:

```
SELECT COUNT(*) FROM table_name;
```

## SQL COUNT(DISTINCT column_name) Syntax

The COUNT(DISTINCT column_name) function returns the number of distinct values of the specified column:

```
SELECT COUNT(DISTINCT column_name) FROM table_name;
```

**Note:** COUNT(DISTINCT) works with ORACLE and Microsoft SQL Server, but not with Microsoft Access.

# Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Orders" table:

| OrderID | CustomerID | EmployeeID | OrderDate | ShipperID |
|---|---|---|---|---|
| 10265 | 7 | 2 | 1996-07-25 | 1 |
| 10266 | 87 | 3 | 1996-07-26 | 3 |
| 10267 | 25 | 4 | 1996-07-29 | 1 |

# SQL COUNT(column_name) Example

The following SQL statement counts the number of orders from "CustomerID"=7 from the "Orders" table:

## Example

```
SELECT COUNT(CustomerID) AS OrdersFromCustomerID7 FROM Orders
WHERE CustomerID=7;
```

# SQL COUNT(*) Example

The following SQL statement counts the total number of orders in the "Orders" table:

## Example

```
SELECT COUNT(*) AS NumberOfOrders FROM Orders;
```

# SQL COUNT(DISTINCT column_name) Example

The following SQL statement counts the number of unique customers in the "Orders" table:

## Example

```
SELECT COUNT(DISTINCT CustomerID) AS NumberOfCustomers FROM Orders;
```

# SQL FIRST() Function

The FIRST() function returns the first value of the selected column.

## SQL FIRST() Syntax

```
SELECT FIRST(column_name) FROM table_name;
```

**Note:** The FIRST() function is only supported in MS Access.

# SQL FIRST() Workaround in SQL Server, MySQL and Oracle

## SQL Server Syntax

```
SELECT TOP 1 column_name FROM table_name
ORDER BY column_name ASC;
```

## Example

```
SELECT TOP 1 CustomerName FROM Customers
ORDER BY CustomerID ASC;
```

## MySQL Syntax

```
SELECT column_name FROM table_name
ORDER BY column_name ASC
LIMIT 1;
```

## Example

```
SELECT CustomerName FROM Customers
ORDER BY CustomerID ASC
LIMIT 1;
```

## Oracle Syntax

```
SELECT column_name FROM table_name
WHERE ROWNUM <=1
ORDER BY column_name ASC;
```

### Example

```
SELECT CustomerName FROM Customers
WHERE ROWNUM <=1
ORDER BY CustomerID ASC;
```

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

## SQL FIRST() Example

The following SQL statement selects the first value of the "CustomerName" column from the "Customers" table:

### Example

```
SELECT FIRST(CustomerName) AS FirstCustomer FROM Customers;
```

# SQL LAST() Function

The LAST() function returns the last value of the selected column.

## SQL LAST() Syntax

```
SELECT LAST(column_name) FROM table_name;
```

**Note:** The LAST() function is only supported in MS Access.

---

# SQL LAST() Workaround in SQL Server, MySQL and Oracle

## SQL Server Syntax

```
SELECT TOP 1 column_name FROM table_name
ORDER BY column_name DESC;
```

## Example

```
SELECT TOP 1 CustomerName FROM Customers
ORDER BY CustomerID DESC;
```

## MySQL Syntax

```
SELECT column_name FROM table_name
ORDER BY column_name DESC
LIMIT 1;
```

## Example

```
SELECT CustomerName FROM Customers
ORDER BY CustomerID DESC
LIMIT 1;
```

## Oracle Syntax

```
SELECT column_name FROM table_name
WHERE ROWNUM <=1
ORDER BY column_name DESC;
```

### Example

```
SELECT CustomerName FROM Customers
WHERE ROWNUM <=1
ORDER BY CustomerID DESC;
```

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

## SQL LAST() Example

The following SQL statement selects the last value of the "CustomerName" column from the "Customers" table:

### Example

```
SELECT LAST(CustomerName) AS LastCustomer FROM Customers;
```

# SQL MAX() Function

The MAX() function returns the largest value of the selected column.

## SQL MAX() Syntax

```
SELECT MAX(column_name) FROM table_name;
```

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Products" table:

| ProductID | ProductName | SupplierID | CategoryID | Unit | Price |
|---|---|---|---|---|---|
| 1 | Chais | 1 | 1 | 10 boxes x 20 bags | 18 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottles | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 12 - 550 ml bottles | 10 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 2 | 48 - 6 oz jars | 21.35 |
| 5 | Chef Anton's Gumbo Mix | 2 | 2 | 36 boxes | 25 |

# SQL MAX() Example

The following SQL statement gets the largest value of the "Price" column from the "Products" table:

## Example

```
SELECT MAX(Price) AS HighestPrice FROM Products;
```

# SQL MIN() Function

The MIN() function returns the smallest value of the selected column.

## SQL MIN() Syntax

```
SELECT MIN(column_name) FROM table_name;
```

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Products" table:

| ProductID | ProductName | SupplierID | CategoryID | Unit | Price |
|---|---|---|---|---|---|
| 1 | Chais | 1 | 1 | 10 boxes x 20 bags | 18 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottles | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 12 - 550 ml bottles | 10 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 2 | 48 - 6 oz jars | 21.35 |
| 5 | Chef Anton's Gumbo Mix | 2 | 2 | 36 boxes | 25 |

## SQL MIN() Example

The following SQL statement gets the smallest value of the "Price" column from the "Products" table:

### Example

```
SELECT MIN(Price) AS SmallestOrderPrice FROM Products;
```

# SQL ROUND() Function

## The ROUND() Function

The ROUND() function is used to round a numeric field to the number of decimals specified.

**Note:** Many database systems do rounding differently than you might expect. When rounding a number with a fractional part to an integer, our school teachers told us to round .1 through .4 DOWN to the next lower integer, and .5 through .9 UP to the next higher integer. But if all the digits 1 through 9 are equally likely, this introduces a slight bias towards infinity, since we always round .5 up. Many database systems have adopted the IEEE 754 standard for arithmetic operations, according to which the default rounding behavior is "round half to even." In this scheme, .5 is rounded to the nearest even integer. So, both 11.5 and 12.5 would be rounded to 12.

### SQL ROUND() Syntax

```
SELECT ROUND(column_name,decimals) FROM table_name;
```

| Parameter | Description |
|---|---|
| column_name | Required. The field to round. |
| decimals | Required. Specifies the number of decimals to be returned. |

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Products" table:

| ProductID | ProductName | SupplierID | CategoryID | Unit | Price |
|---|---|---|---|---|---|
| 1 | Chais | 1 | 1 | 10 boxes x 20 bags | 18 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottles | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 12 - 550 ml bottles | 10 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 2 | 48 - 6 oz jars | 21.35 |
| 5 | Chef Anton's Gumbo Mix | 2 | 2 | 36 boxes | 25 |

# SQL ROUND() Example

The following SQL statement selects the product name and rounds the price in the "Products" table:

## Example

```
SELECT ProductName, ROUND(Price,0) AS RoundedPrice
FROM Products;
```

# SQL SUM() Function

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

## SQL SUM() Syntax

```
SELECT SUM(column_name) FROM table_name;
```

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "OrderDetails" table:

| OrderDetailID | OrderID | ProductID | Quantity |
|---|---|---|---|
| 1 | 10248 | 11 | 12 |
| 2 | 10248 | 42 | 10 |
| 3 | 10248 | 72 | 5 |
| 4 | 10249 | 14 | 9 |
| 5 | 10249 | 51 | 40 |

## SQL SUM() Example

The following SQL statement finds the sum of all the "Quantity" fields for the "OrderDetails" table:

### Example

```
SELECT SUM(Quantity) AS TotalItemsOrdered FROM OrderDetails;
```

# SQL LEN() Function

## The LEN() Function

The LEN() function returns the length of the value in a text field.

### SQL LEN() Syntax

```
SELECT LEN(column_name) FROM table_name;
```

### Syntax for Oracle

```
SELECT LENGTH(column_name) FROM table_name;
```

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# SQL LEN() Example

The following SQL statement selects the "CustomerName" and the length of the values in the "Address" column from the "Customers" table:

## Example

```
SELECT CustomerName,LEN(Address) as LengthOfAddress
FROM Customers;
```

# SQL LCASE() Function

## The LCASE() Function

The LCASE() function converts the value of a field to lowercase.

### SQL LCASE() Syntax

```
SELECT LCASE(column_name) FROM table_name;
```

### Syntax for SQL Server

```
SELECT LOWER(column_name) FROM table_name;
```

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# SQL LCASE() Example

The following SQL statement selects the "CustomerName" and "City" columns from the "Customers" table, and converts the "CustomerName" column to lowercase:

## Example

```
SELECT LCASE(CustomerName) AS Customer, City
FROM Customers;
```

# SQL MID() Function

## The MID() Function

The MID() function is used to extract characters from a text field.

### SQL MID() Syntax

```
SELECT MID(column_name,start,length) AS some_name FROM table_name;
```

| Parameter | Description |
|---|---|
| column_name | Required. The field to extract characters from |
| start | Required. Specifies the starting position (starts at 1) |
| length | Optional. The number of characters to return. If omitted, the MID() function returns the rest of the text |

**Note:** The equivalent function for SQL Server is SUBSTRING():

```
SELECT SUBSTRING(column_name,start,length) AS some_name FROM table_name;
```

**Note:** The equivalent function for Oracle is SUBSTR():

```
SELECT SUBSTR(column_name,start,length) AS some_name FROM table_name;
```

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# SQL MID() Example

The following SQL statement selects the first four characters from the "City" column from the "Customers" table:

## Example

```
SELECT MID(City,1,4) AS ShortCity
FROM Customers;
```

# SQL UCASE() Function

## The UCASE() Function

The UCASE() function converts the value of a field to uppercase.

### SQL UCASE() Syntax

```
SELECT UCASE(column_name) FROM table_name;
```

### Syntax for SQL Server

```
SELECT UPPER(column_name) FROM table_name;
```

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Customers" table:

| CustomerID | CustomerName | ContactName | Address | City | PostalCode | Country |
|---|---|---|---|---|---|---|
| 1 | Alfreds Futterkiste | Maria Anders | Obere Str. 57 | Berlin | 12209 | Germany |
| 2 | Ana Trujillo Emparedados y helados | Ana Trujillo | Avda. de la Constitución 2222 | México D.F. | 05021 | Mexico |
| 3 | Antonio Moreno Taquería | Antonio Moreno | Mataderos 2312 | México D.F. | 05023 | Mexico |
| 4 | Around the Horn | Thomas Hardy | 120 Hanover Sq. | London | WA1 1DP | UK |
| 5 | Berglunds snabbköp | Christina Berglund | Berguvsvägen 8 | Luleå | S-958 22 | Sweden |

# SQL UCASE() Example

The following SQL statement selects the "CustomerName" and "City" columns from the "Customers" table, and converts the "CustomerName" column to uppercase:

## Example

```
SELECT UCASE(CustomerName) AS Customer, City
FROM Customers;
```

# MySQL Date Functions

The following table lists the most important built-in date functions in MySQL:

| Function | Description |
|---|---|
| NOW() | Returns the current date and time |
| CURDATE() | Returns the current date |
| CURTIME() | Returns the current time |
| DATE() | Extracts the date part of a date or date/time expression |
| EXTRACT() | Returns a single part of a date/time |
| DATE_ADD() | Adds a specified time interval to a date |
| DATE_SUB() | Subtracts a specified time interval from a date |
| DATEDIFF() | Returns the number of days between two dates |
| DATE_FORMAT() | Displays date/time data in different formats |

## SQL Server Date Functions

The following table lists the most important built-in date functions in SQL Server:

| Function | Description |
|---|---|
| GETDATE() | Returns the current date and time |
| DATEPART() | Returns a single part of a date/time |
| DATEADD() | Adds or subtracts a specified time interval from a date |
| DATEDIFF() | Returns the time between two dates |
| CONVERT() | Displays date/time data in different formats |

## SQL Date and Time Data Types and Functions

| Function | Description |
|---|---|
| FORMAT() | Formats how a field is to be displayed |
| NOW() | Returns the current system date and time |

# SQL Dates

> The most difficult part when working with dates is to be sure that the format of the date you are trying to insert, matches the format of the date column in the database.

As long as your data contains only the date portion, your queries will work as expected. However, if a time portion is involved, it gets more complicated.

# SQL Date Data Types

**MySQL** comes with the following data types for storing a date or a date/time value in the database:

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

**SQL Server** comes with the following data types for storing a date or a date/time value in the database:

- DATE - format YYYY-MM-DD
- DATETIME - format: YYYY-MM-DD HH:MI:SS
- SMALLDATETIME - format: YYYY-MM-DD HH:MI:SS
- TIMESTAMP - format: a unique number

**Note:** The date types are chosen for a column when you create a new table in your database!

For an overview of all data types available, go to our complete Data Types reference.

# SQL Working with Dates

> You can compare two dates easily if there is no time component involved!

Assume we have the following "Orders" table:

| OrderId | ProductName | OrderDate |
|---|---|---|
| 1 | Geitost | 2008-11-11 |
| 2 | Camembert Pierrot | 2008-11-09 |
| 3 | Mozzarella di Giovanni | 2008-11-11 |
| 4 | Mascarpone Fabioli | 2008-10-29 |

Now we want to select the records with an OrderDate of "2008-11-11" from the table above.

We use the following SELECT statement:

```
SELECT * FROM Orders WHERE OrderDate='2008-11-11'
```

The result-set will look like this:

| OrderId | ProductName | OrderDate |
|---|---|---|
| 1 | Geitost | 2008-11-11 |
| 3 | Mozzarella di Giovanni | 2008-11-11 |

Now, assume that the "Orders" table looks like this (notice the time component in the "OrderDate" column):

| OrderId | ProductName | OrderDate |
|---|---|---|
| 1 | Geitost | 2008-11-11 13:23:44 |
| 2 | Camembert Pierrot | 2008-11-09 15:45:21 |
| 3 | Mozzarella di Giovanni | 2008-11-11 11:12:01 |
| 4 | Mascarpone Fabioli | 2008-10-29 14:56:59 |

If we use the same SELECT statement as above:

```
SELECT * FROM Orders WHERE OrderDate='2008-11-11'
```

we will get no result! This is because the query is looking only for dates with no time portion.

**Tip:** If you want to keep your queries simple and easy to maintain, do not allow time components in your dates!

w3schools.com

THE WORLD'S LARGEST WEB DEVELOPER SITE

# MySQL NOW() Function

## Definition and Usage

NOW() returns the current date and time.

### Syntax

```
NOW()
```

## Example

The following SELECT statement:

```
SELECT NOW(),CURDATE(),CURTIME()
```

will result in something like this:

| NOW() | CURDATE() | CURTIME() |
|---|---|---|
| 2014-11-22 12:45:34 | 2014-11-22 | 12:45:34 |

## Example

The following SQL creates an "Orders" table with a datetime column (OrderDate):

```
CREATE TABLE Orders
(
OrderId int NOT NULL,
ProductName varchar(50) NOT NULL,
OrderDate datetime NOT NULL DEFAULT NOW(),
PRIMARY KEY (OrderId)
)
```

Notice that the OrderDate column specifies NOW() as the default value. As a result, when you insert a row into the table, the current date and time are automatically inserted into the column.

Now we want to insert a record into the "Orders" table:

```
INSERT INTO Orders (ProductName) VALUES ('Jarlsberg Cheese')
```

The "Orders" table will now look something like this:

| OrderId | ProductName | OrderDate |
|---|---|---|
| 1 | Jarlsberg Cheese | 2014-11-22 13:23:44.657 |

w3schools.com

THE WORLD'S LARGEST WEB DEVELOPER SITE

# MySQL CURDATE() Function

## Definition and Usage

CURDATE() returns the current date.

### Syntax

```
CURDATE()
```

### Example

The following SELECT statement:

```
SELECT NOW(),CURDATE(),CURTIME()
```

will result in something like this:

| NOW() | CURDATE() | CURTIME() |
|---|---|---|
| 2014-11-22 12:45:34 | 2014-11-22 | 12:45:34 |

### Example

The following SQL creates an "Orders" table with a datetime column (OrderDate):

```
CREATE TABLE Orders
(
OrderId int NOT NULL,
ProductName varchar(50) NOT NULL,
OrderDate datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
PRIMARY KEY (OrderId)
)
```

Notice that the OrderDate column specifies CURRENT_TIMESTAMP as the default value. As a result, when you insert a row into the table, the current date and time are automatically inserted into the column.

Now we want to insert a record into the "Orders" table:

```
INSERT INTO Orders (ProductName) VALUES ('Jarlsberg Cheese')
```

The "Orders" table will now look something like this:

| OrderId | ProductName | OrderDate |
|---|---|---|
| 1 | Jarlsberg Cheese | 2014-11-22 12:45:34 |

# MySQL CURTIME() Function

## Definition and Usage

CURTIME() returns the current time.

### Syntax

```
CURTIME()
```

## Example

The following SELECT statement:

```
SELECT NOW(),CURDATE(),CURTIME()
```

will result in something like this:

| NOW() | CURDATE() | CURTIME() |
|---|---|---|
| 2014-11-22 12:45:34 | 2014-11-22 | 12:45:34 |

# MySQL DATE() Function

## Definition and Usage

The DATE() function extracts the date part of a date or date/time expression.

### Syntax

```
DATE(date)
```

Where date is a valid date expression.

## Example

Assume we have the following "Orders" table:

| OrderId | ProductName | OrderDate |
|---|---|---|
| 1 | Jarlsberg Cheese | 2014-11-22 13:23:44.657 |

The following SELECT statement:

```
SELECT ProductName, DATE(OrderDate) AS OrderDate
FROM Orders
WHERE OrderId=1
```

will result in this:

| ProductName | OrderDate |
|---|---|
| Jarlsberg Cheese | 2014-11-22 |

# MySQL EXTRACT() Function

## Definition and Usage

The EXTRACT() function is used to return a single part of a date/time, such as year, month, day, hour, minute, etc.

### Syntax

```
EXTRACT(unit FROM date)
```

Where date is a valid date expression and unit can be one of the following:

| Unit Value |
|---|
| MICROSECOND |
| SECOND |
| MINUTE |
| HOUR |
| DAY |
| WEEK |
| MONTH |
| QUARTER |
| YEAR |
| SECOND_MICROSECOND |
| MINUTE_MICROSECOND |
| MINUTE_SECOND |

| |
|---|
| HOUR_MICROSECOND |
| HOUR_SECOND |
| HOUR_MINUTE |
| DAY_MICROSECOND |
| DAY_SECOND |
| DAY_MINUTE |
| DAY_HOUR |
| YEAR_MONTH |

## Example

Assume we have the following "Orders" table:

| OrderId | ProductName | OrderDate |
|---|---|---|
| 1 | Jarlsberg Cheese | 2014-11-22 13:23:44.657 |

The following SELECT statement:

```
SELECT EXTRACT(YEAR FROM OrderDate) AS OrderYear,
EXTRACT(MONTH FROM OrderDate) AS OrderMonth,
EXTRACT(DAY FROM OrderDate) AS OrderDay
FROM Orders
WHERE OrderId=1
```

will result in this:

| OrderYear | OrderMonth | OrderDay |
|---|---|---|
| 2014 | 11 | 22 |

# MySQL DATE_ADD() Function

## Definition and Usage

The DATE_ADD() function adds a specified time interval to a date.

### Syntax

```
DATE_ADD(date,INTERVAL expr type)
```

Where date is a valid date expression and expr is the number of interval you want to add.

type can be one of the following:

| Type Value |
|---|
| MICROSECOND |
| SECOND |
| MINUTE |
| HOUR |
| DAY |
| WEEK |
| MONTH |
| QUARTER |
| YEAR |
| SECOND_MICROSECOND |
| MINUTE_MICROSECOND |

| |
|---|
| MINUTE_SECOND |
| HOUR_MICROSECOND |
| HOUR_SECOND |
| HOUR_MINUTE |
| DAY_MICROSECOND |
| DAY_SECOND |
| DAY_MINUTE |
| DAY_HOUR |
| YEAR_MONTH |

## Example

Assume we have the following "Orders" table:

| OrderId | ProductName | OrderDate |
|---|---|---|
| 1 | Jarlsberg Cheese | 2014-11-22 13:23:44.657 |

Now we want to add 30 days to the "OrderDate", to find the payment date.

We use the following SELECT statement:

```
SELECT OrderId,DATE_ADD(OrderDate,INTERVAL 30 DAY) AS OrderPayDate
FROM Orders
```

Result:

| OrderId | OrderPayDate |
|---|---|
| 1 | 2014-12-22 13:23:44.657 |

# MySQL DATE_SUB() Function

## Definition and Usage

The DATE_SUB() function subtracts a specified time interval from a date.

### Syntax

```
DATE_SUB(date,INTERVAL expr type)
```

Where date is a valid date expression and expr is the number of interval you want to subtract.

type can be one of the following:

| Type Value |
|---|
| MICROSECOND |
| SECOND |
| MINUTE |
| HOUR |
| DAY |
| WEEK |
| MONTH |
| QUARTER |
| YEAR |
| SECOND_MICROSECOND |
| MINUTE_MICROSECOND |

| |
|---|
| MINUTE_SECOND |
| HOUR_MICROSECOND |
| HOUR_SECOND |
| HOUR_MINUTE |
| DAY_MICROSECOND |
| DAY_SECOND |
| DAY_MINUTE |
| DAY_HOUR |
| YEAR_MONTH |

## Example

Assume we have the following "Orders" table:

| OrderId | ProductName | OrderDate |
|---|---|---|
| 1 | Jarlsberg Cheese | 2014-11-22 13:23:44.657 |

Now we want to subtract 5 days from the "OrderDate" date.

We use the following SELECT statement:

```
SELECT OrderId,DATE_SUB(OrderDate,INTERVAL 5 DAY) AS SubtractDate
FROM Orders
```

Result:

| OrderId | SubtractDate |
|---|---|
| 1 | 2014-11-17 13:23:44.657 |

# MySQL DATEDIFF() Function

## Definition and Usage

The DATEDIFF() function returns the time between two dates.

### Syntax

```
DATEDIFF(date1,date2)
```

Where date1 and date2 are valid date or date/time expressions.

**Note:** Only the date parts of the values are used in the calculation.

### Example

The following SELECT statement:

```
SELECT DATEDIFF('2014-11-30','2014-11-29') AS DiffDate
```

will result in this:

| DiffDate |
| --- |
| 1 |

The following SELECT statement:

```
SELECT DATEDIFF('2014-11-29','2014-11-30') AS DiffDate
```

will result in this:

| DiffDate |
| --- |
| -1 |

# MySQL DATE_FORMAT() Function

## Definition and Usage

The DATE_FORMAT() function is used to display date/time data in different formats.

### Syntax

```
DATE_FORMAT(date,format)
```

Where date is a valid date and format specifies the output format for the date/time.

The formats that can be used are:

| Format | Description |
|---|---|
| %a | Abbreviated weekday name (Sun-Sat) |
| %b | Abbreviated month name (Jan-Dec) |
| %c | Month, numeric (0-12) |
| %D | Day of month with English suffix (0th, 1st, 2nd, 3rd, ) |
| %d | Day of month, numeric (00-31) |
| %e | Day of month, numeric (0-31) |
| %f | Microseconds (000000-999999) |
| %H | Hour (00-23) |
| %h | Hour (01-12) |
| %I | Hour (01-12) |
| %i | Minutes, numeric (00-59) |
| %j | Day of year (001-366) |

| | |
|---|---|
| %k | Hour (0-23) |
| %l | Hour (1-12) |
| %M | Month name (January-December) |
| %m | Month, numeric (00-12) |
| %p | AM or PM |
| %r | Time, 12-hour (hh:mm:ss followed by AM or PM) |
| %S | Seconds (00-59) |
| %s | Seconds (00-59) |
| %T | Time, 24-hour (hh:mm:ss) |
| %U | Week (00-53) where Sunday is the first day of week |
| %u | Week (00-53) where Monday is the first day of week |
| %V | Week (01-53) where Sunday is the first day of week, used with %X |
| %v | Week (01-53) where Monday is the first day of week, used with %x |
| %W | Weekday name (Sunday-Saturday) |
| %w | Day of the week (0=Sunday, 6=Saturday) |
| %X | Year for the week where Sunday is the first day of week, four digits, used with %V |
| %x | Year for the week where Monday is the first day of week, four digits, used with %v |
| %Y | Year, numeric, four digits |
| %y | Year, numeric, two digits |

# Example

The following script uses the DATE_FORMAT() function to display different formats. We will use the NOW() function to get the current date/time:

```
DATE_FORMAT(NOW(),'%b %d %Y %h:%i %p')
DATE_FORMAT(NOW(),'%m-%d-%Y')
DATE_FORMAT(NOW(),'%d %b %y')
DATE_FORMAT(NOW(),'%d %b %Y %T:%f')
```

The result would look something like this:

```
Nov 04 2014 11:45 PM
11-04-2014
04 Nov 14
04 Nov 2014 11:45:34:243
```

# SQL Server GETDATE() Function

## Definition and Usage

The GETDATE() function returns the current date and time from the SQL Server.

### Syntax

```
GETDATE()
```

### Example

The following SELECT statement:

```
SELECT GETDATE() AS CurrentDateTime
```

will result in something like this:

| CurrentDateTime |
| --- |
| 2014-11-22 12:45:34.243 |

**Note:** The time part above goes all the way to milliseconds.

### Example

The following SQL creates an "Orders" table with a datetime column (OrderDate):

```
CREATE TABLE Orders
(
OrderId int NOT NULL PRIMARY KEY,
ProductName varchar(50) NOT NULL,
OrderDate datetime NOT NULL DEFAULT GETDATE()
)
```

Notice that the OrderDate column specifies GETDATE() as the default value. As a result, when you insert a row into the table, the current date and time are automatically inserted into the column.

Now we want to insert a record into the "Orders" table:

```
INSERT INTO Orders (ProductName) VALUES ('Jarlsberg Cheese')
```

The "Orders" table will now look something like this:

| OrderId | ProductName | OrderDate |
|---|---|---|
| 1 | Jarlsberg Cheese | 2014-11-22 13:23:44.657 |

# SQL Server DATEPART() Function

## Definition and Usage

The DATEPART() function is used to return a single part of a date/time, such as year, month, day, hour, minute, etc.

### Syntax

```
DATEPART(datepart,date)
```

Where date is a valid date expression and datepart can be one of the following:

| datepart | Abbreviation |
|---|---|
| year | yy, yyyy |
| quarter | qq, q |
| month | mm, m |
| dayofyear | dy, y |
| day | dd, d |
| week | wk, ww |
| weekday | dw, w |
| hour | hh |
| minute | mi, n |
| second | ss, s |
| millisecond | ms |
| microsecond | mcs |
| nanosecond | ns |

# Example

Assume we have the following "Orders" table:

| OrderId | ProductName | OrderDate |
|---|---|---|
| 1 | Jarlsberg Cheese | 2014-11-22 13:23:44.657 |

The following SELECT statement:

```
SELECT DATEPART(yyyy,OrderDate) AS OrderYear,
DATEPART(mm,OrderDate) AS OrderMonth,
DATEPART(dd,OrderDate) AS OrderDay
FROM Orders
WHERE OrderId=1
```

will result in this:

| OrderYear | OrderMonth | OrderDay |
|---|---|---|
| 2014 | 11 | 22 |

# SQL Server DATEADD() Function

## Definition and Usage

The DATEADD() function adds or subtracts a specified time interval from a date.

### Syntax

```
DATEADD(datepart,number,date)
```

Where date is a valid date expression and number is the number of interval you want to add. The number can either be positive, for dates in the future, or negative, for dates in the past.

datepart can be one of the following:

| datepart | Abbreviation |
|---|---|
| year | yy, yyyy |
| quarter | qq, q |
| month | mm, m |
| dayofyear | dy, y |
| day | dd, d |
| week | wk, ww |
| weekday | dw, w |
| hour | hh |
| minute | mi, n |
| second | ss, s |
| millisecond | ms |

| | |
|---|---|
| microsecond | mcs |
| nanosecond | ns |

## Example

Assume we have the following "Orders" table:

| OrderId | ProductName | OrderDate |
|---|---|---|
| 1 | Jarlsberg Cheese | 2014-11-22 13:23:44.657 |

Now we want to add 30 days to the "OrderDate", to find the payment date.

We use the following SELECT statement:

```
SELECT OrderId,DATEADD(day,30,OrderDate) AS OrderPayDate
FROM Orders
```

Result:

| OrderId | OrderPayDate |
|---|---|
| 1 | 2014-12-22 13:23:44.657 |

# SQL Server DATEDIFF() Function

## Definition and Usage

The DATEDIFF() function returns the time between two dates.

### Syntax

```
DATEDIFF(datepart,startdate,enddate)
```

Where startdate and enddate are valid date expressions and datepart can be one of the following:

| datepart | Abbreviation |
|---|---|
| year | yy, yyyy |
| quarter | qq, q |
| month | mm, m |
| dayofyear | dy, y |
| day | dd, d |
| week | wk, ww |
| weekday | dw, w |
| hour | hh |
| minute | mi, n |
| second | ss, s |
| millisecond | ms |
| microsecond | mcs |
| nanosecond | ns |

## Example

Now we want to get the number of days between two dates.

We use the following SELECT statement:

```
SELECT DATEDIFF(day,'2014-06-05','2014-08-05') AS DiffDate
```

Result:

| DiffDate |
| --- |
| 61 |

## Example

Now we want to get the number of days between two dates (notice that the second date is "earlier" than the first date, and will result in a negative number).

We use the following SELECT statement:

```
SELECT DATEDIFF(day,'2014-08-05','2014-06-05') AS DiffDate
```

Result:

| DiffDate |
| --- |
| -61 |

# SQL Server CONVERT() Function

## Definition and Usage

The CONVERT() function is a general function that converts an expression of one data type to another.

The CONVERT() function can be used to display date/time data in different formats.

## Syntax

```
CONVERT(data_type(length),expression,style)
```

| Value | Description |
|---|---|
| *data_type(length)* | Specifies the target data type (with an optional length) |
| *expression* | Specifies the value to be converted |
| *style* | Specifies the output format for the date/time (see table below) |

The *style* value can be one of the following values:

| Without century | With century | Input/Output | Standard |
|---|---|---|---|
| - | 0 or 100 | mon dd yyyy hh:miAM (or PM) | Default |
| 1 | 101 | 1 = mm/dd/yy<br>101 = mm/dd/yyyy | USA |

| | | | |
|---|---|---|---|
| 2 | 102 | 2 = yy.mm.dd<br>102 = yyyy.mm.dd | ANSI |
| 3 | 103 | 3 = dd/mm/yy<br>103 = dd/mm/yyyy | British/French |
| 4 | 104 | 4 = dd.mm.yy<br>104 = dd.mm.yyyy | German |
| 5 | 105 | 5 = dd-mm-yy<br>105 = dd-mm-yyyy | Italian |
| 6 | 106 | 6 = dd mon yy<br>106 = dd mon yyyy | - |
| 7 | 107 | 7 = Mon dd, yy<br>107 = Mon dd, yyyy | - |
| 8 | 108 | hh:mm:ss | - |
| - | 9 or 109 | mon dd yyyy<br>hh:mi:ss:mmmAM (or PM) | Default + millisec |
| 10 | 110 | 10 = mm-dd-yy<br>110 = mm-dd-yyyy | USA |
| 11 | 111 | 11 = yy/mm/dd<br>111 = yyyy/mm/dd | Japan |
| 12 | 112 | 12 = yymmdd<br>112 = yyyymmdd | ISO |
| - | 13 or 113 | dd mon yyyy<br>hh:mi:ss:mmm (24h) | Europe default + millisec |
| 14 | 114 | hh:mi:ss:mmm (24h) | - |
| - | 20 or 120 | yyyy-mm-dd hh:mi:ss (24h) | ODBC canonical |
| - | 21 or 121 | yyyy-mm-dd<br>hh:mi:ss.mmm (24h) | ODBC canonical (with milliseconds) default for time, date, datetime2, and datetimeoffset |
| - | 126 | yyyy-mm-ddThh:mi:ss.mmm (no | ISO8601 |

| | | | |
|---|---|---|---|
| | | spaces) | |
| - | 127 | yyyy-mm-ddThh:mi:ss.mmmZ (no spaces) | ISO8601 with time zone Z |
| - | 130 | dd mon yyyy hh:mi:ss:mmmAM | Hijiri |
| - | 131 | dd/mm/yy hh:mi:ss:mmmAM | Hijiri |

## Example

The following script uses the CONVERT() function to display different formats. We will use the GETDATE() function to get the current date/time:

```
CONVERT(VARCHAR(19),GETDATE())
CONVERT(VARCHAR(10),GETDATE(),10)
CONVERT(VARCHAR(10),GETDATE(),110)
CONVERT(VARCHAR(11),GETDATE(),6)
CONVERT(VARCHAR(11),GETDATE(),106)
CONVERT(VARCHAR(24),GETDATE(),113)
```

The result would look something like this:

```
Nov 04 2014 11:45 PM
11-04-14
11-04-2014
04 Nov 14
04 Nov 2014
04 Nov 2014 11:45:34:243
```

# SQL FORMAT() Function

## The FORMAT() Function

The FORMAT() function is used to format how a field is to be displayed.

### SQL FORMAT() Syntax

```
SELECT FORMAT(column_name,format) FROM table_name;
```

| Parameter | Description |
|---|---|
| column_name | Required. The field to be formatted. |
| format | Required. Specifies the format. |

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Products" table:

| ProductID | ProductName | SupplierID | CategoryID | Unit | Price |
|---|---|---|---|---|---|
| 1 | Chais | 1 | 1 | 10 boxes x 20 bags | 18 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottles | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 12 - 550 ml bottles | 10 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 2 | 48 - 6 oz jars | 21.35 |
| 5 | Chef Anton's Gumbo Mix | 2 | 2 | 36 boxes | 25 |

# SQL FORMAT() Example

The following SQL statement selects the product name, and price for today (formatted like YYYY-MM-DD) from the "Products" table:

## Example

```
SELECT ProductName, Price, FORMAT(Now(),'YYYY-MM-DD') AS PerDate
FROM Products;
```

w3schools.com

THE WORLD'S LARGEST WEB DEVELOPER SITE

# SQL NOW() Function

## The NOW() Function

The NOW() function returns the current system date and time.

### SQL NOW() Syntax

```
SELECT NOW() FROM table_name;
```

## Demo Database

In this tutorial we will use the well-known Northwind sample database.

Below is a selection from the "Products" table:

| ProductID | ProductName | SupplierID | CategoryID | Unit | Price |
|---|---|---|---|---|---|
| 1 | Chais | 1 | 1 | 10 boxes x 20 bags | 18 |
| 2 | Chang | 1 | 1 | 24 - 12 oz bottles | 19 |
| 3 | Aniseed Syrup | 1 | 2 | 12 - 550 ml bottles | 10 |
| 4 | Chef Anton's Cajun Seasoning | 2 | 2 | 48 - 6 oz jars | 21.35 |
| 5 | Chef Anton's Gumbo Mix | 2 | 2 | 36 boxes | 25 |

# SQL NOW() Example

The following SQL statement selects the product name, and price for today from the "Products" table:

## Example

```
SELECT ProductName, Price, Now() AS PerDate
FROM Products;
```

# SQL NULL Functions

## SQL ISNULL(), NVL(), IFNULL() and COALESCE() Functions

Look at the following "Products" table:

| P_Id | ProductName | UnitPrice | UnitsInStock | UnitsOnOrder |
|---|---|---|---|---|
| 1 | Jarlsberg | 10.45 | 16 | 15 |
| 2 | Mascarpone | 32.56 | 23 | |
| 3 | Gorgonzola | 15.67 | 9 | 20 |

Suppose that the "UnitsOnOrder" column is optional, and may contain NULL values.

We have the following SELECT statement:

```
SELECT ProductName,UnitPrice*(UnitsInStock+UnitsOnOrder)
FROM Products
```

In the example above, if any of the "UnitsOnOrder" values are NULL, the result is NULL.

Microsoft's ISNULL() function is used to specify how we want to treat NULL values.

The NVL(), IFNULL(), and COALESCE() functions can also be used to achieve the same result.

In this case we want NULL values to be zero.

Below, if "UnitsOnOrder" is NULL it will not harm the calculation, because ISNULL() returns a zero if the value is NULL:

### MS Access

```
SELECT ProductName,UnitPrice*
(UnitsInStock+IIF(ISNULL(UnitsOnOrder),0,UnitsOnOrder))
FROM Products
```

### SQL Server

```
SELECT ProductName,UnitPrice*(UnitsInStock+ISNULL(UnitsOnOrder,0))
FROM Products
```

### Oracle

Oracle does not have an ISNULL() function. However, we can use the NVL() function to achieve the same result:

```
SELECT ProductName,UnitPrice*(UnitsInStock+NVL(UnitsOnOrder,0))
FROM Products
```

### MySQL

MySQL does have an ISNULL() function. However, it works a little bit different from Microsoft's ISNULL() function.

In MySQL we can use the IFNULL() function, like this:

```
SELECT ProductName,UnitPrice*(UnitsInStock+IFNULL(UnitsOnOrder,0))
FROM Products
```

or we can use the COALESCE() function, like this:

```
SELECT ProductName,UnitPrice*(UnitsInStock+COALESCE(UnitsOnOrder,0))
FROM Products
```

# SQL Operators

## SQL Arithmetic Operators

| Operator | Description |
|---|---|
| + | Add |
| - | Subtract |
| * | Multiply |
| / | Divide |
| % | Modulo |

## SQL Bitwise Operators

| Operator | Description |
|---|---|
| & | Bitwise AND |
| \| | Bitwise OR |
| ^ | Bitwise exclusive OR |

## SQL Comparison Operators

| Operator | Description |
|---|---|
| = | Equal to |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal to |
| <= | Less than or equal to |
| <> | Not equal to |

# SQL Compound Operators

| Operator | Description |
|---|---|
| += | Add equals |
| -= | Subtract equals |
| *= | Multiply equals |
| /= | Divide equals |
| %= | Modulo equals |
| &= | Bitwise AND equals |
| ^-= | Bitwise exclusive equals |
| \|*= | Bitwise OR equals |

# SQL Logical Operators

**Note:** ALL and ANY are not supported in Web SQL databases. Chrome, Safari and Opera are using Web SQL in our examples.

| Operator | Description |
|---|---|
| ALL | TRUE if all of the subquery values meet the condition |
| AND | TRUE if all the conditions separated by AND is TRUE |
| ANY | TRUE if any of the subquery values meet the condition |
| BETWEEN | TRUE if the operand is within the range of comparisons |
| EXISTS | TRUE if the subquery returns one or more records |
| IN | TRUE if the operand is equal to one of a list of expressions |
| LIKE | TRUE if the operand matches a pattern |
| NOT | Displays a record if the condition(s) is NOT TRUE |
| OR | TRUE if any of the conditions separated by OR is TRUE |
| SOME | TRUE if any of the subquery values meet the condition |

# SQL General Data Types

A data type defines what kind of value a column can contain.

## SQL General Data Types

Each column in a database table is required to have a name and a data type.

SQL developers have to decide what types of data will be stored inside each and every table column when creating a SQL table. The data type is a label and a guideline for SQL to understand what type of data is expected inside of each column, and it also identifies how SQL will interact with the stored data.

The following table lists the general data types in SQL:

| Data type | Description |
|---|---|
| CHARACTER(n) | Character string. Fixed-length n |
| VARCHAR(n) or CHARACTER VARYING(n) | Character string. Variable length. Maximum length n |
| BINARY(n) | Binary string. Fixed-length n |
| BOOLEAN | Stores TRUE or FALSE values |
| VARBINARY(n) or BINARY VARYING(n) | Binary string. Variable length. Maximum length n |
| INTEGER(p) | Integer numerical (no decimal). Precision p |
| SMALLINT | Integer numerical (no decimal). Precision 5 |
| INTEGER | Integer numerical (no decimal). Precision 10 |
| BIGINT | Integer numerical (no decimal). Precision 19 |

| | |
|---|---|
| DECIMAL(p,s) | Exact numerical, precision p, scale s. Example: decimal(5,2) is a number that has 3 digits before the decimal and 2 digits after the decimal |
| NUMERIC(p,s) | Exact numerical, precision p, scale s. (Same as DECIMAL) |
| FLOAT(p) | Approximate numerical, mantissa precision p. A floating number in base 10 exponential notation. The size argument for this type consists of a single number specifying the minimum precision |
| REAL | Approximate numerical, mantissa precision 7 |
| FLOAT | Approximate numerical, mantissa precision 16 |
| DOUBLE PRECISION | Approximate numerical, mantissa precision 16 |
| DATE | Stores year, month, and day values |
| TIME | Stores hour, minute, and second values |
| TIMESTAMP | Stores year, month, day, hour, minute, and second values |
| INTERVAL | Composed of a number of integer fields, representing a period of time, depending on the type of interval |
| ARRAY | A set-length and ordered collection of elements |
| MULTISET | A variable-length and unordered collection of elements |
| XML | Stores XML data |

# SQL Data Type Quick Reference

However, different databases offer different choices for the data type definition.

The following table shows some of the common names of data types between the various database platforms:

| Data type | Access | SQLServer | Oracle | MySQL | PostgreSQL |
|---|---|---|---|---|---|
| *boolean* | Yes/No | Bit | Byte | N/A | Boolean |
| *integer* | Number (integer) | Int | Number | Int<br>Integer | Int<br>Integer |
| *float* | Number (single) | Float<br>Real | Number | Float | Numeric |
| *currency* | Currency | Money | N/A | N/A | Money |
| *string (fixed)* | N/A | Char | Char | Char | Char |
| *string (variable)* | Text (<256)<br>Memo (65k+) | Varchar | Varchar<br>Varchar2 | Varchar | Varchar |
| *binary object* | OLE Object<br>Memo | Binary (fixed up to 8K)<br>Varbinary (<8K)<br>Image (<2GB) | Long Raw | Blob<br>Text | Binary<br>Varbinary |

**Note:** Data types might have different names in different database. And even if the name is the same, the size and other details may be different! **Always check the documentation!**

# SQL Data Types for Various DBs

Data types and ranges for Microsoft Access, MySQL and SQL Server.

## Microsoft Access Data Types

| Data type | Description | Storage |
|---|---|---|
| Text | Use for text or combinations of text and numbers. 255 characters maximum | |
| Memo | Memo is used for larger amounts of text. Stores up to 65,536 characters. **Note:** You cannot sort a memo field. However, they are searchable | |
| Byte | Allows whole numbers from 0 to 255 | 1 byte |
| Integer | Allows whole numbers between -32,768 and 32,767 | 2 bytes |
| Long | Allows whole numbers between -2,147,483,648 and 2,147,483,647 | 4 bytes |
| Single | Single precision floating-point. Will handle most decimals | 4 bytes |
| Double | Double precision floating-point. Will handle most decimals | 8 bytes |
| Currency | Use for currency. Holds up to 15 digits of whole dollars, plus 4 decimal places. **Tip:** You can choose which country's currency to use | 8 bytes |
| AutoNumber | AutoNumber fields automatically give each record its own number, usually starting at 1 | 4 bytes |
| Date/Time | Use for dates and times | 8 bytes |
| Yes/No | A logical field can be displayed as Yes/No, True/False, or On/Off. In code, use the constants True and False (equivalent to -1 and 0). **Note:** Null values are not allowed in Yes/No fields | 1 bit |

| | | |
|---|---|---|
| Ole Object | Can store pictures, audio, video, or other BLOBs (Binary Large OBjects) | up to 1GB |
| Hyperlink | Contain links to other files, including web pages | |
| Lookup Wizard | Let you type a list of options, which can then be chosen from a drop-down list | 4 bytes |

# MySQL Data Types

In MySQL there are three main data types : text, number, and Date/Time types.

## Text types:

| Data type | Description |
|---|---|
| CHAR(size) | Holds a fixed length string (can contain letters, numbers, and special characters). The fixed size is specified in parenthesis. Can store up to 255 characters |
| VARCHAR(size) | Holds a variable length string (can contain letters, numbers, and special characters). The maximum size is specified in parenthesis. Can store up to 255 characters. **Note:** If you put a greater value than 255 it will be converted to a TEXT type |
| TINYTEXT | Holds a string with a maximum length of 255 characters |
| TEXT | Holds a string with a maximum length of 65,535 characters |
| BLOB | For BLOBs (Binary Large OBjects). Holds up to 65,535 bytes of data |
| MEDIUMTEXT | Holds a string with a maximum length of 16,777,215 characters |
| MEDIUMBLOB | For BLOBs (Binary Large OBjects). Holds up to 16,777,215 bytes of data |
| LONGTEXT | Holds a string with a maximum length of 4,294,967,295 characters |
| LONGBLOB | For BLOBs (Binary Large OBjects). Holds up to 4,294,967,295 bytes of data |
| ENUM(x,y,z,etc.) | Let you enter a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted. |

| | |
|---|---|
| | **Note:** The values are sorted in the order you enter them.<br><br>You enter the possible values in this format: ENUM('X','Y','Z') |
| SET | Similar to ENUM except that SET may contain up to 64 list items and can store more than one choice |

## Number types:

| Data type | Description |
|---|---|
| TINYINT(size) | -128 to 127 normal. 0 to 255 UNSIGNED*. The maximum number of digits may be specified in parenthesis |
| SMALLINT(size) | -32768 to 32767 normal. 0 to 65535 UNSIGNED*. The maximum number of digits may be specified in parenthesis |
| MEDIUMINT(size) | -8388608 to 8388607 normal. 0 to 16777215 UNSIGNED*. The maximum number of digits may be specified in parenthesis |
| INT(size) | -2147483648 to 2147483647 normal. 0 to 4294967295 UNSIGNED*. The maximum number of digits may be specified in parenthesis |
| BIGINT(size) | -9223372036854775808 to 9223372036854775807 normal. 0 to 18446744073709551615 UNSIGNED*. The maximum number of digits may be specified in parenthesis |
| FLOAT(size,d) | A small number with a floating decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter |
| DOUBLE(size,d) | A large number with a floating decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter |
| DECIMAL(size,d) | A DOUBLE stored as a string , allowing for a fixed decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter |

*The integer types have an extra option called UNSIGNED. Normally, the integer goes from an negative to positive value. Adding the UNSIGNED attribute will move that range up so it starts at zero instead of a negative number.

# Date types:

| Data type | Description |
|---|---|
| DATE() | A date. Format: YYYY-MM-DD<br><br>**Note:** The supported range is from '1000-01-01' to '9999-12-31' |
| DATETIME() | *A date and time combination. Format: YYYY-MM-DD HH:MI:SS<br><br>**Note:** The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59' |
| TIMESTAMP() | *A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD HH:MI:SS<br><br>**Note:** The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC |
| TIME() | A time. Format: HH:MI:SS<br><br>**Note:** The supported range is from '-838:59:59' to '838:59:59' |
| YEAR() | A year in two-digit or four-digit format.<br><br>**Note:** Values allowed in four-digit format: 1901 to 2155. Values allowed in two-digit format: 70 to 69, representing years from 1970 to 2069 |

*Even if DATETIME and TIMESTAMP return the same format, they work very differently. In an INSERT or UPDATE query, the TIMESTAMP automatically set itself to the current date and time. TIMESTAMP also accepts various formats, like YYYYMMDDHHMISS, YYMMDDHHMISS, YYYYMMDD, or YYMMDD.

# SQL Server Data Types

## String types:

| Data type | Description | Storage |
|---|---|---|
| char(n) | Fixed width character string. Maximum 8,000 characters | Defined width |
| varchar(n) | Variable width character string. Maximum 8,000 characters | 2 bytes + number of chars |
| varchar(max) | Variable width character string. Maximum 1,073,741,824 characters | 2 bytes + number of chars |
| text | Variable width character string. Maximum 2GB of text data | 4 bytes + number of chars |
| nchar | Fixed width Unicode string. Maximum 4,000 characters | Defined width x 2 |
| nvarchar | Variable width Unicode string. Maximum 4,000 characters | |
| nvarchar(max) | Variable width Unicode string. Maximum 536,870,912 characters | |
| ntext | Variable width Unicode string. Maximum 2GB of text data | |
| bit | Allows 0, 1, or NULL | |
| binary(n) | Fixed width binary string. Maximum 8,000 bytes | |
| varbinary | Variable width binary string. Maximum 8,000 bytes | |
| varbinary(max) | Variable width binary string. Maximum 2GB | |
| image | Variable width binary string. Maximum 2GB | |

# Number types:

| Data type | Description | Storage |
|---|---|---|
| tinyint | Allows whole numbers from 0 to 255 | 1 byte |
| smallint | Allows whole numbers between -32,768 and 32,767 | 2 bytes |
| int | Allows whole numbers between -2,147,483,648 and 2,147,483,647 | 4 bytes |
| bigint | Allows whole numbers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 | 8 bytes |
| decimal(p,s) | Fixed precision and scale numbers.<br>Allows numbers from -10^38 +1 to 10^38 –1.<br>The p parameter indicates the maximum total number of digits that can be stored (both to the left and to the right of the decimal point). p must be a value from 1 to 38. Default is 18.<br>The s parameter indicates the maximum number of digits stored to the right of the decimal point. s must be a value from 0 to p. Default value is 0 | 5-17 bytes |
| numeric(p,s) | Fixed precision and scale numbers.<br>Allows numbers from -10^38 +1 to 10^38 –1.<br>The p parameter indicates the maximum total number of digits that can be stored (both to the left and to the right of the decimal point). p must be a value from 1 to 38. Default is 18.<br>The s parameter indicates the maximum number of digits stored to the right of the decimal point. s must be a value from 0 to p. Default value is 0 | 5-17 bytes |
| smallmoney | Monetary data from -214,748.3648 to 214,748.3647 | 4 bytes |
| money | Monetary data from -922,337,203,685,477.5808 to 922,337,203,685,477.5807 | 8 bytes |

| | | |
|---|---|---|
| float(n) | Floating precision number data from -1.79E + 308 to 1.79E + 308.<br><br>The n parameter indicates whether the field should hold 4 or 8 bytes. float(24) holds a 4-byte field and float(53) holds an 8-byte field. Default value of n is 53. | 4 or 8 bytes |
| real | Floating precision number data from -3.40E + 38 to 3.40E + 38 | 4 bytes |

## Date types:

| Data type | Description | Storage |
|---|---|---|
| datetime | From January 1, 1753 to December 31, 9999 with an accuracy of 3.33 milliseconds | 8 bytes |
| datetime2 | From January 1, 0001 to December 31, 9999 with an accuracy of 100 nanoseconds | 6-8 bytes |
| smalldatetime | From January 1, 1900 to June 6, 2079 with an accuracy of 1 minute | 4 bytes |
| date | Store a date only. From January 1, 0001 to December 31, 9999 | 3 bytes |
| time | Store a time only to an accuracy of 100 nanoseconds | 3-5 bytes |
| datetimeoffset | The same as datetime2 with the addition of a time zone offset | 8-10 bytes |
| timestamp | Stores a unique number that gets updated every time a row gets created or modified. The timestamp value is based upon an internal clock and does not correspond to real time. Each table may have only one timestamp variable | |

# Other data types:

| Data type | Description |
|---|---|
| sql_variant | Stores up to 8,000 bytes of data of various data types, except text, ntext, and timestamp |
| uniqueidentifier | Stores a globally unique identifier (GUID) |
| xml | Stores XML formatted data. Maximum 2GB |
| cursor | Stores a reference to a cursor used for database operations |
| table | Stores a result-set for later processing |

w3schools.com

THE WORLD'S LARGEST WEB DEVELOPER SITE

# SQL Quick Reference From W3Schools

| SQL Statement | Syntax |
|---|---|
| AND / OR | SELECT column_name(s)<br>FROM table_name<br>WHERE condition<br>AND\|OR condition |
| ALTER TABLE | ALTER TABLE table_name<br>ADD column_name datatype<br><br>or<br><br>ALTER TABLE table_name<br>DROP COLUMN column_name |
| AS (alias) | SELECT column_name AS column_alias<br>FROM table_name<br><br>or<br><br>SELECT column_name<br>FROM table_name AS table_alias |
| BETWEEN | SELECT column_name(s)<br>FROM table_name<br>WHERE column_name<br>BETWEEN value1 AND value2 |
| CREATE DATABASE | CREATE DATABASE database_name |
| CREATE TABLE | CREATE TABLE table_name<br>(<br>column_name1 data_type,<br>column_name2 data_type,<br>column_name3 data_type,<br>...<br>) |

| | |
|---|---|
| CREATE INDEX | CREATE INDEX index_name<br>ON table_name (column_name)<br><br>or<br><br>CREATE UNIQUE INDEX index_name<br>ON table_name (column_name) |
| CREATE VIEW | CREATE VIEW view_name AS<br>SELECT column_name(s)<br>FROM table_name<br>WHERE condition |
| DELETE | DELETE FROM table_name<br>WHERE some_column=some_value<br><br>or<br><br>DELETE FROM table_name<br>(**Note:** Deletes the entire table!!)<br><br>DELETE * FROM table_name<br>(**Note:** Deletes the entire table!!) |
| DROP DATABASE | DROP DATABASE database_name |
| DROP INDEX | DROP INDEX table_name.index_name (SQL Server)<br>DROP INDEX index_name ON table_name (MS Access)<br>DROP INDEX index_name (DB2/Oracle)<br>ALTER TABLE table_name<br>DROP INDEX index_name (MySQL) |
| DROP TABLE | DROP TABLE table_name |
| EXISTS | IF EXISTS (SELECT * FROM table_name WHERE id = ?)<br>BEGIN<br>--do what needs to be done if exists<br>END<br>ELSE<br>BEGIN<br>--do what needs to be done if not<br>END |
| GROUP BY | SELECT column_name, aggregate_function(column_name)<br>FROM table_name<br>WHERE column_name operator value<br>GROUP BY column_name |