

## 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.

### 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

### 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.

### 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

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;
```

#### Keep in Mind That...

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

### 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.

## 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

### SQL SELECT Statement

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

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

### SQL SELECT Syntax

```
SELECT column_name,column_name
```

```
FROM table_name; and
```

```
SELECT * 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

### SELECT Column Example

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

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

#### Output

Number of Records: 91

Alfreds Futterkiste	Berlin
Ana Trujillo Emparedados y helados	México D.F.

### SELECT \* Example

The following SQL statement selects all the columns from the "Customers" table: `SELECT * FROM Customers;`

Output:

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

### SQL SELECT DISTINCT Statement

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

#### The SQL SELECT DISTINCT Statement

In a table, a column may contain many duplicate values; and sometimes you only want to list the different (distinct) values.

The DISTINCT keyword can be used to return only distinct (different) values.

### SQL SELECT DISTINCT Syntax

```
SELECT DISTINCT column_name,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

	helados		2222	D.F.		
--	---------	--	------	------	--	--

### SELECT DISTINCT Example

The following SQL statement selects only the distinct values from the "City" columns from the "Customers" table:

#### Example

SELECT DISTINCT City FROM Customers;

**Output:** Number of Records: 69

City
Aachen
Albuquerque

### SQL WHERE Clause

The WHERE clause is used to filter records.

#### The SQL WHERE Clause

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

#### SQL WHERE Syntax

SELECT *column\_name,column\_name*

FROM *table\_name*

WHERE *column\_name operator value;*

#### 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

### WHERE Clause Example

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

SELECT \* FROM Customers

WHERE Country='Mexico'

**Output:**Number of Records: 5

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

### 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;

**Output:** Number of Records: 1

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany

### Operators in The WHERE Clause

The following operators can be used in the WHERE clause:

Operator	Description
=	Equal
<>	Not equal. <b>Note:</b> 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 Operators

The AND & OR operators are used to filter records based on more than one condition.

The AND operator displays a record if both the first condition AND the second condition are true.

The OR operator displays a record if either the first condition OR the second condition is true.

### 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

### AND Operator Example

The following SQL statement selects all customers from the country "Germany" AND the city "Berlin", in the "Customers" table:

#### Example

```
SELECT * FROM Customers
```

```
WHERE Country='Germany'
```

```
AND City='Berlin';
```

**Output:** Number of Records: 1

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany

### OR Operator Example

The following SQL statement selects all customers from the city "Berlin" OR "München", in the "Customers" table:

```
SELECT * FROM Customers
```

```
WHERE City='Berlin'
```

```
OR City='München';
```

**Output:** Number of Records: 2

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
25	Frankenversand	Peter Franken	Berliner Platz 43	München	80805	Germany

### Combining AND & OR

You can also combine AND and OR (use parenthesis to form complex expressions).

The following SQL statement selects all customers from the country "Germany" AND the city must be equal to "Berlin" OR "München", in the "Customers" table:

#### Example

```
SELECT * FROM Customers
```

```
WHERE Country='Germany'
```

```
AND (City='Berlin' OR City='München');
```

**Output:** Number of Records: 2

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
25	Frankenversand	Peter Franken	Berliner Platz 43	München	80805	Germany

### SQL ORDER BY Keyword

The ORDER BY keyword is used to sort the result-set.

The ORDER BY keyword is used to sort the result-set by one or more columns.

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

## SQL ORDER BY Syntax

```
SELECT column_name,column_name
FROM table_name
ORDER BY column_name,column_name ASC|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

### ORDER BY Example

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

**Output:** Number of Records: 91

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
64	Rancho grande	Sergio Gutiérrez	Av. del Libertador 900	Buenos Aires	1010	Argentina
54	Océano Atlántico Ltda.	Yvonne Moncada	Ing. Gustavo Moncada 8585 Piso 20-A	Buenos Aires	1010	Argentina

### ORDER BY DESC Example

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

**Output:** Number of Records: 91

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
47	LINO-Delicatesses	Felipe Izquierdo	Ave. 5 de Mayo Porlamar	I. de Margarita	4980	Venezuela
33	GROSELLA-Restaurante	Manuel Pereira	5ª Ave. Los Palos Grandes	Caracas	1081	Venezuela

### ORDER BY Several Columns Example

```
SELECT * FROM Customers
ORDER BY Country, Customer Name;
```

**Output:** Number of Records: 91

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
12	Cactus Comidas para llevar	Patricio Simpson	Cerrito 333	Buenos Aires	1010	Argentina
54	Océano Atlántico Ltda.	Yvonne Moncada	Ing. Gustavo Moncada 8585 Piso 20-A	Buenos Aires	1010	Argentina

## SQL INSERT INTO Statement

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

### SQL INSERT INTO Syntax

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

The first form does not specify the column names where the data will be inserted, only their values:

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

The second form specifies both the column names and the values to be inserted:

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

### 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
------------	--------------	-------------	---------	------	------------	---------

87	Wartian Herkku	Pirkko Koskitalo	Torikatu 38	Oulu	90110	Finland
88	Wellington Importadora	Paula Parente	Rua do Mercado, 12	Resende	08737-363	Brazil

### INSERT INTO Example

Assume we wish to insert a new row in the "Customers" table.

We can use the following SQL statement:

#### Example

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

**Output:** Operation must use an updateable query.

**Note:** Your browser does not support WebSQL, so you are now using a light-version of our Try-SQL Editor, with a read-only Database.

If you use a browser with WebSQL support (Chrome, Safari, or Opera), you can try any SQL statement, and play with the Database as much as you like.

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

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
87	Wartian Herkku	Pirkko Koskitalo	Torikatu 38	Oulu	90110	Finland
88	Wellington Importadora	Paula Parente	Rua do Mercado, 12	Resende	08737-363	Brazil

### 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 row, but only insert data in the "CustomerName", "City", and "Country" columns (and the CustomerID field will of course also 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
87	Wartian Herkku	Pirkko Koskitalo	Torikatu 38	Oulu	90110	Finland
88	Wellington Importadora	Paula Parente	Rua do Mercado, 12	Resende	08737-363	Brazil

### SQL UPDATE Statement

The UPDATE statement is used to update records in a table

#### SQL UPDATE Syntax

```
UPDATE table_name
```

```
SET column1=value1,column2=value2,...
```

```
WHERE some_column=some_value;
```

#### 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

### SQL UPDATE Example

Assume we wish to update the customer "Alfreds Futterkiste" with a new contact person and city.

We use the following SQL statement:

#### Example

```
UPDATE Customers
SET ContactName='Alfred Schmidt', City='Hamburg'
WHERE CustomerName='Alfreds Futterkiste';
```

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	Hamburg	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico

### Update Warning!

Be careful when updating records. If we had omitted the WHERE clause, in the example above, like this:

```
UPDATE Customers
```

```
SET ContactName='Alfred Schmidt', City='Hamburg';
```

The "Customers" table would have looked like this:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Alfred Schmidt	Obere Str. 57	Hamburg	12209	Germany
2	Ana Trujillo Emparedados y helados	Alfred Schmidt	Avda. de la Constitución 2222	Hamburg	05021	Mexico

### SQL DELETE Statement

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

### SQL DELETE Syntax

```
DELETE FROM table_name
```

```
WHERE some_column=some_value;
```

### Demo 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

### SQL DELETE Example

Assume we wish to delete the customer "Alfreds Futterkiste" from the "Customers" table.

We use the following SQL statement:

### Example

```
DELETE FROM Customers
```

```
WHERE CustomerName='Alfreds Futterkiste' AND ContactName='Maria Anders';
```

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

### Delete All Data

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 Injection

An SQL Injection can destroy your database.

### SQL in Web Pages

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 statements, via web page input. Injected SQL commands can alter SQL statement and compromises 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:

#### 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:

Password:

#### Server Code

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

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

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.

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

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 Customers table, and then delete the table called Suppliers. If we had the following server code:

#### Server Code

```
txtUserId = getQueryString("UserId");  
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;  
And the following input: User id:
```

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.

### PHP INSERT INTO

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



```
$stmt->bindParam(':cit', $txtCit);
```

```
$stmt->execute();
```

### The SQL SELECT TOP Clause

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

The SELECT TOP clause can be very 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.

### SQL Server / MS Access Syntax

```
SELECT TOP number|percent column_name(s)
```

```
FROM table_name;
```

### MySQL Syntax

```
SELECT column_name(s)
```

```
FROM table_name
```

```
LIMIT number;
```

### Example

```
SELECT *
```

```
FROM Persons
```

```
LIMIT 5;
```

### 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

### SQL SELECT TOP Example

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

```
SELECT TOP 2 * FROM Customers;
```

Output: Number of Records: 2

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

### SQL SELECT TOP PERCENT Example

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

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

Number of Records: 46

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

### The SQL LIKE Operator

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

### SQL LIKE Syntax

```
SELECT column_name(s)
```

```
FROM table_name
```

```
WHERE column_name LIKE pattern;
```

### 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

## SQL LIKE Operator Examples

The following SQL statement selects all customers with a City starting with the letter "s":

### Example

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

Number of Records: 12

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
7	Blondel père et fils	Frédérique Citeaux	24, place Kléber	Strasbourg	67000	France
15	Comércio Mineiro	Pedro Afonso	Av. dos Lusíadas, 23	São Paulo	05432-043	Brazil

### SQL Wildcards

A wildcard character can be used to substitute for any other character(s) in a string.

### SQL Wildcard Characters

In SQL, wildcard characters are used with the SQL LIKE operator.

SQL wildcards are used to search for data within a table.

With SQL, the wildcards are:

Wildcard	Description
%	A substitute for zero or more characters
_	A substitute for a single character
[charlist]	Sets and ranges of characters to match
[^charlist] or [!charlist]	Matches only a character NOT specified within the brackets

### 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

### Using the SQL % Wildcard

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

### Example

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

Number of Records: 3

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
14	Chop-suey Chinese	Yang Wang	Hauptstr. 29	Bern	3012	Switzerland

### The IN Operator

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

### SQL IN Syntax

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

### 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

### IN Operator Example

The following SQL statement selects all customers with a City of "Paris" or "London":

### Example

```
SELECT * FROM Customers
WHERE City IN ('Paris','London');
```

Number of Records: 8

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
11	B's Beverages	Victoria Ashworth	Fauntleroy Circus	London	EC2 5NT	UK

### The SQL BETWEEN Operator

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

#### SQL BETWEEN Syntax

```
SELECT column_name(s)
```

```
FROM table_name
```

```
WHERE column_name BETWEEN value1 AND value2;
```

#### 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

#### BETWEEN Operator 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;
```

Number of Records: 29

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

#### SQL Aliases

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

Basically aliases are created to make column names more readable.

#### SQL Alias Syntax for Columns

```
SELECT column_name AS alias_name
```

```
FROM table_name;
```

#### SQL Alias Syntax for Tables

```
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

#### Alias Example for Table Columns

The following SQL statement specifies two aliases, one for the CustomerName column and one for the ContactName column. **Tip:**

It require double quotation marks or square brackets if the column name contains spaces:

#### Example

```
SELECT CustomerName AS Customer, ContactName AS [Contact Person]
```

```
FROM Customers;
```

Number of Records: 91

Customer	Contact Person
Alfreds Futterkiste	María Anders
Ana Trujillo Emparedados y helados	Ana Trujillo

#### Alias Example for Tables

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 have used 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;
Number of Records: 2
```

OrderID	OrderDate	CustomerName
10383	12/16/1996	Around the Horn
10355	11/15/1996	Around the Horn

### SQL JOIN

An SQL JOIN clause is used to combine rows from two or more tables, based on a common field between them.

The most common type of join is: **SQL INNER JOIN (simple join)**. An SQL INNER JOIN return all rows from multiple tables where the join condition is met.

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

OrderID	CustomerID	OrderDate
10308	2	1996-09-18
10309	37	1996-09-19

Then, have a 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 customer in the "Customers" table. The relationship between the two tables above is the "CustomerID" column.

Then, if we run the following SQL statement (that contains an INNER JOIN):

### Example

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers
ON Orders.CustomerID=Customers.CustomerID;
Number of Records: 196
```

OrderID	CustomerName	OrderDate
10308	Ana Trujillo Emparedados y helados	9/18/1996
10365	Antonio Moreno Taquería	11/27/1996

### Different SQL JOINS

Before we continue with examples, we will list the types the different SQL JOINS you can use:

- **INNER JOIN:** Returns all rows when there is at least one match in BOTH tables
- **LEFT JOIN:** Return all rows from the left table, and the matched rows from the right table
- **RIGHT JOIN:** Return all rows from the right table, and the matched rows from the left table
- **FULL JOIN:** Return all rows when there is a match in ONE of the tables

### SQL INNER JOIN Keyword

The INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns in both tables.

### SQL INNER JOIN Syntax

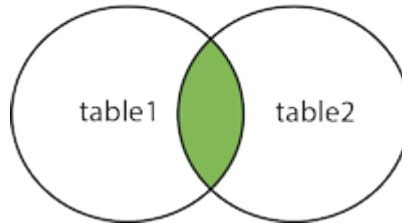
```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name=table2.column_name;
```

or:

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

**PS!** INNER JOIN is the same as JOIN.

## INNER 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

### SQL INNER JOIN Example

The following SQL statement will return all customers with orders:

#### Example

```
SELECT Customers.CustomerName, Orders.OrderID
```

```
FROM Customers
```

```
INNER JOIN Orders
```

```
ON Customers.CustomerID=Orders.CustomerID
```

```
ORDER BY Customers.CustomerName;
```

Number of Records: 196

CustomerName	OrderID
Ana Trujillo Emparedados y helados	10308
Antonio Moreno Taquería	10365

### SQL LEFT JOIN Keyword

The LEFT JOIN keyword returns all rows from the left table (table1), with the matching rows in the right table (table2). The result is NULL in the right side when there is no match.

#### SQL LEFT JOIN Syntax

```
SELECT column_name(s)
```

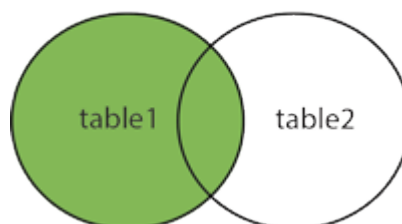
```
FROM table1
```

```
LEFT JOIN table2
```

```
ON table1.column_name=table2.column_name;
```

**PS!** In some databases LEFT JOIN is called LEFT OUTER JOIN.

## LEFT JOIN



### SQL LEFT JOIN Example

The following SQL statement will return 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;
```

### SQL RIGHT JOIN Keyword

The RIGHT JOIN keyword returns all rows from the right table (table2), with the matching rows in the left table (table1). The result is NULL in the left side when there is no match.

### SQL RIGHT JOIN Example

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

#### Example

```
SELECT Orders.OrderID, Employees.FirstName
FROM Orders
RIGHT JOIN Employees
ON Orders.EmployeeID=Employees.EmployeeID
ORDER BY Orders.OrderID;
Number of Records: 197
```

OrderID	FirstName
	Adam
10248	Steven

### SQL FULL OUTER JOIN Keyword

The FULL OUTER JOIN keyword returns all rows from the left table (table1) and from the right table (table2). The FULL OUTER JOIN keyword combines the result of both LEFT and RIGHT joins.

#### SQL FULL OUTER JOIN Syntax

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

### 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

### The SQL UNION Operator

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

Notice that each SELECT statement within the UNION must have the same number of columns. The columns must also have similar data types. Also, the columns in each SELECT statement must be in the same order.

#### SQL UNION Syntax

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

### SQL UNION Example

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

#### Example

```
SELECT City FROM Customers
UNION
SELECT City FROM Suppliers
ORDER BY City;
Number of Records: 94
```

City
Aachen
Albuquerque

### The SQL SELECT INTO Statement

The SELECT INTO statement selects data from one table and inserts it into a new table.

#### SQL SELECT INTO Syntax

We can copy all columns into the new table:

```
SELECT *
INTO newtable [IN externaldb]
FROM table1;
```

Or we can copy only the columns we want into the new table:

```
SELECT column_name(s)
INTO newtable [IN externaldb]
FROM table1;
```

### SQL SELECT INTO Examples

Create a backup copy of Customers:

```
SELECT *
INTO CustomersBackup2013
FROM Customers;
```

Use the IN clause to copy the table into another database:

```
SELECT *
INTO CustomersBackup2013 IN 'Backup.mdb'
FROM Customers;
```

Copy only a few columns into the new table:

```
SELECT CustomerName, ContactName
INTO CustomersBackup2013
FROM Customers;
```

Copy only the German customers into the new table:

```
SELECT *
INTO CustomersBackup2013
FROM Customers
WHERE Country='Germany';
```

Copy data from more than one table into the new table:

```
SELECT Customers.CustomerName, Orders.OrderID
INTO CustomersOrderBackup2013
FROM Customers
LEFT JOIN Orders
ON Customers.CustomerID=Orders.CustomerID;
```

**Tip:** The SELECT INTO statement 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 table1
WHERE 1=0;
```

### The SQL INSERT INTO SELECT Statement

The INSERT INTO SELECT statement selects data from one table and inserts it into an existing table. Any existing rows in the target table are unaffected.

### SQL INSERT INTO SELECT Syntax

We can copy all columns from one table to another, existing table:

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

Or we can copy only the columns we want to into another, existing table:

```
INSERT INTO table2
(column_name(s))
SELECT column_name(s)
FROM table1;
```

### SQL INSERT INTO SELECT Examples

Copy only a few columns from "Suppliers" into "Customers":

#### Example

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

Copy only the German suppliers into "Customers":

### Example

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

### The SQL CREATE DATABASE Statement

The CREATE DATABASE statement is used to create a database.

### SQL CREATE DATABASE Syntax

```
CREATE DATABASE dbname;
```

### SQL CREATE DATABASE Example

The following SQL statement creates a database called "my\_db":

```
CREATE DATABASE my_db;
```

### The SQL CREATE TABLE Statement

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

Tables are organized into rows and columns; and each table must have a name.

### SQL CREATE TABLE Syntax

```
CREATE TABLE table_name
(
  column_name1 data_type(size),
  column_name2 data_type(size),
  column_name3 data_type(size),
  ....
);
```

The *column\_name* parameters specify the names of the columns of the table.  
The *data\_type* parameter specifies what type of data the column can hold (e.g. varchar, integer, decimal, date, etc.).  
The size parameter specifies the maximum length of the column of the table.

### SQL CREATE TABLE Example

Now we want to create a table called "Persons" that contains five columns: PersonID, LastName, FirstName, Address, and City.  
We use the following CREATE TABLE statement:

### Example

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

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

PersonID	LastName	FirstName	Address	City

### SQL Constraints

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

If there is any violation between the constraint and the data action, the action is aborted by the constraint.

Constraints can be specified when the table is created (inside the CREATE TABLE statement) or after the table is created (inside the ALTER TABLE statement).

### SQL CREATE TABLE + CONSTRAINT Syntax

```
CREATE TABLE table_name
(
  column_name1 data_type(size) constraint_name,
  column_name2 data_type(size) constraint_name,
  column_name3 data_type(size) constraint_name,
  ....
);
```

In SQL, we have the following constraints:

- **NOT NULL** - Indicates that a column cannot store NULL value
- **UNIQUE** - Ensures that each row for a column must have a unique value



- **PRIMARY KEY** - A combination of a NOT NULL and UNIQUE. Ensures that a column (or combination of two or more columns) have an unique identity which helps to find a particular record in a table more easily and quickly
- **FOREIGN KEY** - Ensure the referential integrity of the data in one table to match values in another table
- **CHECK** - Ensures that the value in a column meets a specific condition
- **DEFAULT** - Specifies a default value when specified none for this column

### SQL NOT NULL Constraint

By default, a table column can hold NULL values.

#### SQL NOT NULL Constraint

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

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

The following SQL enforces the "P\_Id" column and the "LastName" column to not accept NULL values:

#### Example

```
CREATE TABLE PersonsNotNull
(
P_Id int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255)
)
```

#### SQL UNIQUE Constraint

The UNIQUE constraint uniquely identifies each record in a database table.

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

A PRIMARY KEY constraint automatically has a UNIQUE constraint defined on it.

Note that 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 "P\_Id" column when the "Persons" table is created:

#### MySQL:

```
CREATE TABLE PersonsUnique
(
P_Id int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
UNIQUE (P_Id)
)
```

#### SQL UNIQUE Constraint on ALTER TABLE

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

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

```
ALTER TABLE Persons
ADD UNIQUE (P_Id)
```

#### To DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL:

#### MySQL:

```
ALTER TABLE Persons
DROP INDEX uc_PersonID
```

#### SQL PRIMARY KEY Constraint

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

Primary keys must contain unique values.

A primary key column cannot contain NULL values.

Each table should have a primary key, and each table can have only ONE primary key.

#### SQL PRIMARY KEY Constraint on CREATE TABLE

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

#### MySQL:

```
CREATE TABLE Persons
(
P_Id int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
PRIMARY KEY (P_Id)
)
```

### SQL PRIMARY KEY Constraint on ALTER TABLE

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

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

```
ALTER TABLE Persons
ADD PRIMARY KEY (P_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_PersonID PRIMARY KEY (P_Id,LastName)
```

### To DROP a PRIMARY KEY Constraint

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

**MySQL:**

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

### SQL FOREIGN KEY Constraint

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

Let's illustrate the foreign key with an example. Look at the following two tables:

The "Persons" table:

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

### SQL FOREIGN KEY Constraint on CREATE TABLE

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

**MySQL:**

```
CREATE TABLE Orders
(
O_Id int NOT NULL,
OrderNo int NOT NULL,
P_Id int,
PRIMARY KEY (O_Id),
FOREIGN KEY (P_Id) REFERENCES Persons(P_Id)
)
```

### SQL FOREIGN KEY Constraint on ALTER TABLE

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

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

```
ALTER TABLE Orders
ADD FOREIGN KEY (P_Id)
REFERENCES Persons(P_Id)
```

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_PerOrders
FOREIGN KEY (P_Id)
REFERENCES Persons(P_Id)
```

### To DROP a FOREIGN KEY Constraint

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

#### MySQL:

```
ALTER TABLE Orders  
DROP FOREIGN KEY fk_PerOrders
```

#### SQL Server / Oracle / MS Access:

```
ALTER TABLE Orders  
DROP CONSTRAINT fk_PerOrders
```

### 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 Constraint on CREATE TABLE

The following SQL creates a CHECK constraint on the "P\_Id" column when the "Persons" table is created. The CHECK constraint specifies that the column "P\_Id" must only include integers greater than 0.

#### MySQL:

```
CREATE TABLE Persons  
(  
P_Id int NOT NULL,  
LastName varchar(255) NOT NULL,  
FirstName varchar(255),  
Address varchar(255),  
City varchar(255),  
CHECK (P_Id>0)  
)
```

#### SQL CHECK Constraint on ALTER TABLE

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

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

```
ALTER TABLE Persons  
ADD CHECK (P_Id>0)
```

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_Person CHECK (P_Id>0 AND City='Sandnes')
```

### To DROP a CHECK Constraint

To drop a CHECK constraint, use the following SQL:

#### SQL Server / Oracle / MS Access:

```
ALTER TABLE Persons  
DROP CONSTRAINT chk_Person
```

#### MySQL:

```
ALTER TABLE Persons  
DROP CHECK chk_Person
```

### SQL DEFAULT Constraint

The DEFAULT constraint is used to insert a default value into a column.

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

#### SQL DEFAULT Constraint on CREATE TABLE

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

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

```
CREATE TABLE Orders  
(  
O_Id int NOT NULL,  
OrderNo int NOT NULL,  
P_Id int,  
OrderDate date DEFAULT GETDATE()  
)
```

## SQL DEFAULT Constraint 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'
```

---

## To 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 allow the database application to find data fast; without reading the whole table.

### Indexes

An index can be created in a table to find data more quickly and efficiently.

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 you should only create indexes on columns (and tables) that will be frequently searched against.

### SQL CREATE INDEX Syntax

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

```
CREATE INDEX index_name  
ON table_name (column_name)
```

### SQL CREATE UNIQUE INDEX Syntax

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

```
CREATE UNIQUE INDEX index_name  
ON table_name (column_name)
```

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

### CREATE INDEX Example

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

```
CREATE INDEX PIndex  
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 PIndex  
ON Persons (LastName, FirstName)
```

### The DROP INDEX Statement

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

### DROP INDEX Syntax for MySQL:

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

### The DROP TABLE Statement

The DROP TABLE statement is used to delete a table.

```
DROP TABLE table_name
```

### The DROP DATABASE Statement

The DROP DATABASE statement is used to delete a database.

```
DROP DATABASE database_name
```

### The TRUNCATE TABLE Statement

What if we only want to delete the data inside the table, and not the table itself?

Then, use the TRUNCATE TABLE statement:

```
TRUNCATE TABLE table_name
```

### The ALTER TABLE Statement

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

#### SQL ALTER TABLE Syntax

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

```
ALTER TABLE table_name
```

```
ADD column_name datatype
```

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
```

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:

```
ALTER TABLE table_name
```

```
MODIFY COLUMN column_name datatype
```

#### SQL ALTER TABLE Example

Look at the "Persons" table:

P_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 like this:

P_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-digit 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 like this:

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

## AUTO INCREMENT a Field

Very often we would like the value of the primary key field to be created automatically every time a new record is inserted. We would like to create an auto-increment field in a table.

### 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),
  Address varchar(255),
  City varchar(255),
  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),
  Address varchar(255),
  City varchar(255)
)
```

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".

## SQL Views

A view is a virtual table.

This chapter shows how to create, update, and delete a view.

### 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.

### SQL CREATE VIEW Syntax

```
CREATE VIEW view_name AS
SELECT column_name(s)
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
```

We can query the view above 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. Now we want to 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 column_name(s)
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 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
```

### MySQL Date Functions

The following table lists the most important built-in date functions in MySQL:

Function	Description
<a href="#">NOW()</a>	Returns the current date and time
<a href="#">CURDATE()</a>	Returns the current date
<a href="#">CURTIME()</a>	Returns the current time
<a href="#">DATE()</a>	Extracts the date part of a date or date/time expression
<a href="#">EXTRACT()</a>	Returns a single part of a date/time
<a href="#">DATE_ADD()</a>	Adds a specified time interval to a date

<a href="#">DATE_SUB()</a>	Subtracts a specified time interval from a date
<a href="#">DATEDIFF()</a>	Returns the number of days between two dates
<a href="#">DATE_FORMAT()</a>	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
<a href="#">GETDATE()</a>	Returns the current date and time
<a href="#">DATEPART()</a>	Returns a single part of a date/time
<a href="#">DATEADD()</a>	Adds or subtracts a specified time interval from a date
<a href="#">DATEDIFF()</a>	Returns the time between two dates
<a href="#">CONVERT()</a>	Displays date/time data in different formats

### 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:MM:SS
- TIMESTAMP - format: YYYY-MM-DD HH:MM: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:MM:SS
- SMALLDATETIME - format: YYYY-MM-DD HH:MM: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](#).

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!

### SQL NULL Values

If a column in a table is optional, we can insert a new record or update an existing record without adding a value to this column.

This means that the field will be saved with a NULL value.

NULL values are treated differently from other values.

NULL is used as a placeholder for unknown or inapplicable values.



## SQL Working with NULL Values

Look at the following "Persons" table:

P_Id	LastName	FirstName	Address	City
1	Hansen	Ola		Sandnes
2	Svendson	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari		Stavanger

Suppose that the "Address" column in the "Persons" table is optional. This means that if we insert a record with no value for the "Address" column, the "Address" column will be saved with a NULL value.

How can we test for NULL values?

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the IS NULL and IS NOT NULL operators instead.

### SQL IS NULL

How do we select only the records with NULL values in the "Address" column?

We will have to use the IS NULL operator:

```
SELECT LastName,FirstName,Address FROM Persons
WHERE Address IS NULL
```

The result-set will look like this:

LastName	FirstName	Address
Hansen	Ola	
Pettersen	Kari	

### SQL IS NOT NULL

How do we select only the records with no NULL values in the "Address" column?

We will have to use the IS NOT NULL operator:

```
SELECT LastName,FirstName,Address FROM Persons
WHERE Address IS NOT NULL
```

The result-set will look like this:

LastName	FirstName	Address
Svendson	Tove	Borgvn 23

In the next chapter we will look at the ISNULL(), NVL(), IFNULL() and COALESCE() functions.

## 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:

### 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 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
<i>boolean</i>	Yes/No	Bit	Byte	N/A	Boolean
<i>integer</i>	Number (integer)	Int	Number	Int Integer	Int Integer
<i>float</i>	Number (single)	Float Real	Number	Float	Numeric
<i>currency</i>	Currency	Money	N/A	N/A	Money
<i>string (fixed)</i>	N/A	Char	Char	Char	Char
<i>string (variable)</i>	Text (<256) Memo (65k+)	Varchar	Varchar Varchar2	Varchar	Varchar
<i>binary object</i>	OLE Object Memo	Binary (fixed up to 8K) Varbinary (<8K) Image (<2GB)	Long Raw	Blob Text	Binary Varbinary

### SQL Data Types for Various DBs

#### MySQL Data Types

In MySQL there are three main 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. <b>Note:</b> 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. <b>Note:</b> The values are sorted in the order you enter them. 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 <b>Note:</b> The supported range is from '1000-01-01' to '9999-12-31'
DATETIME()	*A date and time combination. Format: YYYY-MM-DD HH:MM:SS <b>Note:</b> 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:MM:SS <b>Note:</b> 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:MM:SS <b>Note:</b> The supported range is from '-838:59:59' to '838:59:59'
YEAR()	A year in two-digit or four-digit format. <b>Note:</b> 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 YYYYMMDDHHMMSS, YYMMDDHHMMSS, 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. Allows numbers from $-10^{38} + 1$ to $10^{38} - 1$ . 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. 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. Allows numbers from $-10^{38} + 1$ to $10^{38} - 1$ . 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. 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 $1.79E + 308$ . 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

### 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.

Useful aggregate functions:

- 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
- SUM() - Returns the sum

#### SQL Scalar functions

SQL scalar functions return a single value, based on the input value.

Useful scalar functions:

- UCASE() - Converts a field to upper case
- LCASE() - Converts a field to lower case
- MID() - Extract characters from a text field
- LEN() - Returns the length of a text field
- ROUND() - Rounds a numeric field to the number of decimals specified
- NOW() - Returns the current system date and time
- FORMAT() - Formats how a field is to be displayed

#### The 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

#### 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;
```

Number of Records: 1

PriceAverage
28.8664

#### 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;
```

Number of Records: 1

OrdersFromCustomerID7
4

### 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 Basic

[SQL HOME](#) [SQL Intro](#) [SQL Syntax](#) [SQL SELECT](#) [SQL SELECT DISTINCT](#) [SQL WHERE](#) [SQL AND & OR](#) [SQL ORDER BY](#)  
[SQL INSERT INTO](#) [SQL UPDATE](#) [SQL DELETE](#) [SQL Injection](#)

### SQL FIRST() Function

#### The 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;
```

#### 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;
```

Number of Records: 1

<b>FirstCustomer</b>
----------------------

Alfreds Futterkiste
---------------------

### The 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;
```

#### 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;
```

### The 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;
```

### The MIN() Function

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

#### SQL MIN() Syntax

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

#### 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;
```

Number of Records: 1

<b>SmallestOrderPrice</b>
2.5

### The 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

### 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;
```

Number of Records: 1

### TotalItemsOrdered

12743

### The GROUP BY Statement

The GROUP BY statement is used in conjunction with the aggregate functions to group the result-set by one or more columns.

### SQL GROUP BY Syntax

```
SELECT column_name, aggregate_function(column_name)
```

```
FROM table_name
```

```
WHERE column_name operator value
```

```
GROUP BY 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
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	Phone
1	Speedy Express	(503) 555-9831
2	United Package	(503) 555-3199
3	Federal Shipping	(503) 555-9931

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....

### SQL GROUP BY Example

Now we want to find the number of orders sent by each shipper.

The following SQL statement counts as orders grouped by shippers:

### Example

```
SELECT Shippers.ShipperName,COUNT(Orders.OrderID) AS NumberOfOrders FROM Orders
```

```
LEFT JOIN Shippers
```

```
ON Orders.ShipperID=Shippers.ShipperID
```

```
GROUP BY ShipperName;
```

Number of Records: 3

ShipperName	NumberOfOrders
Federal Shipping	68



Speedy Express	54
United Package	74

### The HAVING Clause

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

### SQL HAVING Syntax

```
SELECT column_name, aggregate_function(column_name)
```

```
FROM table_name
```

```
WHERE column_name operator value
```

```
GROUP BY column_name
```

```
HAVING aggregate_function(column_name) operator value;
```

### 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
10248	90	5	1996-07-04	3
10249	81	6	1996-07-05	1
10250	34	4	1996-07-08	2

### SQL HAVING Example

Now we want to find if any of the customers have a total order of less than 2000.

We use the following SQL statement:

The following SQL statement finds if any of the employees has 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;
```

Number of Records: 8

LastName	NumberOfOrders
Buchanan	11
Callahan	27
Davolio	29

### 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;
```

#### 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;
```

Number of Records: 91

Customer	City
ALFREDS FUTTERKISTE	Berlin
ANA TRUJILLO EMPAREDADOS Y HELADOS	México D.F.
ANTONIO MORENO TAQUERÍA	México D.F.

### 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;
```

### 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;
Number of Records: 91
```

Customer	City
alfreds futterkiste	Berlin
ana trujillo emparedados y helados	México D.F.
antonio moreno taquería	México D.F.

### The MID() Function

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

#### SQL MID() Syntax

```
SELECT MID(column_name,start[,length]) 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

### 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;
Number of Records: 91
```

ShortCity
Berl
Méxi
Méxi

### 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;
```

### 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;
Number of Records: 91
```

CustomerName	LengthOfAddress
Alfreds Futterkiste	13
Ana Trujillo Emparedados y helados	29

### The ROUND() Function

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

#### 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.

### SQL ROUND() Example

The following SQL statement selects the product name and the price rounded to the nearest integer from the "Products" table:

#### 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

#### Example

```
SELECT ProductName, ROUND(Price,0) AS RoundedPrice
```

```
FROM Products;
```

Number of Records: 77

ProductName	RoundedPrice
Chais	18
Chang	19
Aniseed Syrup	10

#### 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

#### 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;
```

Number of Records: 77

ProductName	Price	PerDate
Chais	18	11/20/2013 5:51:21 AM
Chang	19	11/20/2013 5:51:21 AM
Aniseed Syrup	10	11/20/2013 5:51:21 AM

#### 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

#### 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;
```

Number of Records: 77

<b>ProductName</b>	<b>Price</b>	<b>PerDate</b>
Chais	18	2013-11-20
Chang	19	2013-11-20
Aniseed Syrup	10	2013-11-20

### **SQL Hosting**

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

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

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

You can have SQL databases on both Windows and Linux/UNIX operating systems.

Below is an overview of which database system that runs on which OS.

#### **MS SQL Server**

Runs only on Windows OS.

#### **MySQL**

Runs on Windows, Mac OS X, and Linux/UNIX operating systems.

**MS Access** (recommended only for small websites)

Runs only on Windows OS.