SQL Tutorial

Www.w3schools.com, 2024. — 299 p.

Contents:

SQL INNER JOIN
SQL LEFT JOIN Keyword
SQL RIGHT JOIN Keyword
SQL FULL OUTER JOIN Keyword
SQL Self Join
SQL UNION Operator
SQL GROUP BY Statement
SQL HAVING Clause
SQL EXISTS Operator
SQL ANY and ALL Operators
SQL SELECT INTO Statement
SQL INSERT INTO SELECT Statement
SQL CASE Expression
SQL NULL Functions
SQL Stored Procedures for SQL Server
SQL Comments
SQL Operators
SQL DATABASE
SQL CREATE DATABASE Statement
SQL DROP DATABASE Statement
SQL BACKUP DATABASE for SQL Server
SQL CREATE TABLE Statement
SQL DROP TABLE Statement
SQL ALTER TABLE Statement
SQL Constraints
SQL NOT NULL Constraint
SQL UNIQUE Constraint
SQL PRIMARY KEY Constraint
SQL FOREIGN KEY Constraint
SQL CHECK Constraint
SQL DEFAULT Constraint
SQL CREATE INDEX Statement
SQL AUTO INCREMENT Field
SQL Working With Dates

SQL Views
SQL Injection
SQL Hosting
SQL Data Types for MySQL, SQL Server, and MS Access
SQL Keywords Reference
SQL ADD Keyword
SQL ADD CONSTRAINT Keyword
SQL ALL Keyword
SQL ALTER Keyword
SQL ALTER COLUMN Keyword
SQL ALTER TABLE Keyword
SQL AND Keyword
SQL ANY Keyword
SQL AS Keyword
SQL ASC Keyword
SQL BACKUP DATABASE Keyword
SQL BETWEEN Keyword
SQL CASE Keyword
SQL CHECK Keyword
SQL COLUMN Keyword
SQL CONSTRAINT Keyword
SQL CREATE Keyword
SQL CREATE DATABASE Keyword
SQL CREATE INDEX Keyword
SQL CREATE OR REPLACE VIEW Keyword
SQL CREATE TABLE Keyword
SQL CREATE PROCEDURE Keyword
SQL CREATE UNIQUE INDEX Keyword
SQL CREATE VIEW Keyword
SQL DATABASE Keyword
SQL DEFAULT Keyword
SQL DELETE Keyword
SQL DESC Keyword
SQL SELECT DISTINCT Keyword

SQL DROP Keyword
SQL DROP COLUMN Keyword
SQL DROP CONSTRAINT Keyword
SQL DROP DATABASE Keyword
SQL DROP DEFAULT Keyword
SQL DROP INDEX Keyword
SQL DROP TABLE and TRUNCATE TABLE Keywords
SQL DROP VIEW Keyword
SQL EXEC Keyword
SQL EXISTS Keyword
SQL FOREIGN KEY Keyword
SQL FROM Keyword
SQL FULL OUTER JOIN Keyword
SQL GROUP BY Keyword
SQL HAVING Keyword
SQL IN Keyword
SQL INDEX Keyword
SQL INNER JOIN Keyword
SQL INSERT INTO Keyword
SQL INSERT INTO SELECT Keyword
SQL IS NULL Keyword
SQL IS NOT NULL Keyword
SQL JOIN Keyword
SQL LEFT JOIN Keyword
SQL LIKE Keyword
SQL SELECT TOP, LIMIT and ROWNUM Keywords
SQL SELECT TOP, LIMIT and ROWNUM Keywords
SQL NOT NULL Keyword
SQL OR Keyword
SQL ORDER BY Keyword
SQL FULL OUTER JOIN Keyword
SQL PRIMARY KEY Keyword
SQL CREATE PROCEDURE Keyword
SQL RIGHT JOIN Keyword

SQL is a standard language for storing, manipulating and retrieving data in databases.

Our SQL tutorial will teach you how to use SQL in: MySQL, SQL Server, MS Access, Oracle, Sybase, Informix, Postgres, and other database systems.

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 became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987

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/ISO 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, PostalCode and Country. 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

SQL Statements

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

SQL statements consists of keywords that are easy to understand.

The following SQL statement returns all records from a table named "Customers":

Example

Select all records from the Customers table:

```
SELECT * FROM Customers;
```

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

Database Tables

A database most often contains one or more tables. Each table is identified by a name (e.g. "Customers" or "Orders"), and 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 used in the examples:

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).

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

SQL SELECT Statement

The SQL SELECT Statement

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

Example

Return data from the Customers table:

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

Syntax

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

Here, column1, column2, ... are the *field names* of the table you want to select data from.

The table_name represents the name of the *table* you want to select data from.

Demo Database

Below is a selection from the **Customers** table used in the examples:

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 ALL columns

If you want to return all columns, without specifying every column name, you can use the `SELECT *` syntax:

Example

Return all the columns from the Customers table:

```
SELECT * FROM Customers;
```

Test Yourself With Exercises

Exercise:

Insert the missing statement to get all the columns from the `Customers` table.

_____ * FROM Customers;

Submit Answer »

```
SELECT * FROM Customers;
```

SQL SELECT DISTINCT Statement

The SQL SELECT DISTINCT Statement

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

Example

Select all the different countries from the "Customers" table:

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

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

Syntax

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

Demo Database

Below is a selection from the **Customers** table used in the examples:

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 Without DISTINCT

If you omit the `DISTINCT` keyword, the SQL statement returns the "Country" value from all the records of the "Customers" table:

Example

```
SELECT Country FROM Customers;
```

Count Distinct

By using the `DISTINCT` keyword in a function called `COUNT`, we can return the number of different countries.

Example

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

Note: The COUNT(DISTINCT *column_name*) is not supported in Microsoft Access databases.

Here is a workaround for MS Access:

Example

```
SELECT Count(*) AS DistinctCountries
FROM (SELECT DISTINCT Country FROM Customers);
```

You will learn about the COUNT function later in this tutorial.

Test Yourself With Exercises

Exercise:

Select all the different values from the `Country` column in the `Customers` table.

_____._____ Country FROM Customers;

Submit Answer »

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

SQL WHERE Clause

The SQL WHERE Clause

The `WHERE` clause is used to filter records.

It is used to extract only those records that fulfill a specified condition.

Example

Select all customers from Mexico:

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

Syntax

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

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

Demo Database

Below is a selection from the **Customers** table used in the examples:

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

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

You can use other operators than the = operator to filter the search.

Example

Select all customers with a CustomerID greater than 80:

```
SELECT * FROM Customers
WHERE CustomerID > 80;
```

The following operators can be used in the `WHERE` clause:

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

Test Yourself With Exercises

Exercise:

Select all records where the `City` column has the value "Berlin".

```
SELECT * FROM Customers
_____._____ = _____;
```

Submit Answer »

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

SQL ORDER BY Keyword

The SQL ORDER BY

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

Example

Sort the products by price:

```
SELECT * FROM Products
ORDER BY Price;
```

Syntax

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

Demo Database

Below is a selection from the **Products** table used in the examples:

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	22
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35

DESC

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

Example

Sort the products from highest to lowest price:

```
SELECT * FROM Products
ORDER BY Price DESC;
```

Order Alphabetically

For string values the `ORDER BY` keyword will order alphabetically:

Example

Sort the products alphabetically by ProductName:

```
SELECT * FROM Products
ORDER BY ProductName;
```

Alphabetically DESC

To sort the table reverse alphabetically, use the `DESC` keyword:

Example

Sort the products by ProductName in reverse order:

```
SELECT * FROM Products
ORDER BY ProductName DESC;
```

ORDER BY Several Columns

The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" and the "CustomerName" column. This means that it orders by Country, but if some rows have the same Country, it orders them by CustomerName:

Example

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

Using Both ASC and DESC

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;
```

Test Yourself With Exercises

Exercise:

Select all records from the `Customers` table, sort the result alphabetically by the column `City`.

```
SELECT * FROM Customers
_____________ _____________;
```

Submit Answer »

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

SQL AND Operator

The SQL AND Operator

The `WHERE` clause can contain one or many `AND` operators.

The `AND` operator is used to filter records based on more than one condition, like if you want to return all customers from Spain that starts with the letter 'G':

Example

Select all customers from Spain that starts with the letter 'G':

```
SELECT *
FROM Customers
WHERE Country = 'Spain' AND CustomerName LIKE 'G%';
```

Syntax

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

AND vs OR

The `AND` operator displays a record if *all* the conditions are TRUE.

The `OR` operator displays a record if *any* of the conditions are TRUE.

Demo Database

Below is a selection from the **Customers** table used in the examples:

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

All Conditions Must Be True

The following SQL statement selects all fields from `Customers` where `Country` is "Germany" AND `City` is "Berlin" AND `PostalCode` is higher than 12000:

Example

```
SELECT * FROM Customers
WHERE Country = 'Germany'
AND City = 'Berlin'
AND PostalCode > 12000;
```

Combining AND and OR

You can combine the `AND` and `OR` operators.

The following SQL statement selects all customers from Spain that starts with a "G" or an "R".

Make sure you use parenthesis to get the correct result.

Example

Select all Spanish customers that starts with either "G" or "R":

```
SELECT * FROM Customers
WHERE Country
= 'Spain' AND (CustomerName LIKE 'G%' OR CustomerName LIKE 'R%');
```

Without parenthesis, the select statement will return all customers from Spain that starts with a "G", *plus* all customers that starts with an "R", regardless of the country value:

Example

Select all customers that either:
are from Spain and starts with either "G", *or*
starts with the letter "R":

```
SELECT * FROM Customers
WHERE Country
= 'Spain' AND CustomerName LIKE 'G%' OR CustomerName LIKE 'R%';
```

Test Yourself With Exercises

Exercise:

Select all records where the `City` column has the value 'Berlin' and the `PostalCode` column has the value 12209.

_____ * FROM Customers

```
_____ City = 'Berlin'
_____._____ = '12209';
```

Submit Answer »

```
SELECT * FROM Customers
WHERE City = 'Berlin'
AND PostalCode = '12209';
```

SQL OR Operator

The SQL OR Operator

The `WHERE` clause can contain one or more `OR` operators.

The `OR` operator is used to filter records based on more than one condition, like if you want to return all customers from Germany but also those from Spain:

Example

Select all customers from Germany or Spain:

```
SELECT *
FROM Customers
WHERE Country = 'Germany' OR Country = 'Spain';
```

Syntax

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

OR vs AND

The `OR` operator displays a record if *any* of the conditions are TRUE.

The `AND` operator displays a record if *all* the conditions are TRUE.

Demo Database

Below is a selection from the **Customers** table used in the examples:

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

At Least One Condition Must Be True

The following SQL statement selects all fields from Customers where either `City` is "Berlin", `CustomerName` starts with the letter "G" or `Country` is "Norway":

Example

```
SELECT * FROM Customers
WHERE City = 'Berlin' OR CustomerName LIKE 'G%' OR Country = 'Norway';
```

Combining AND and OR

You can combine the `AND` and `OR` operators.

The following SQL statement selects all customers from Spain that starts with a "G" or an "R".

Make sure you use parenthesis to get the correct result.

Example

Select all Spanish customers that starts with either "G" or "R":

```
SELECT * FROM Customers
WHERE Country
= 'Spain' AND (CustomerName LIKE 'G%' OR CustomerName LIKE 'R%');
```

Without parenthesis, the select statement will return all customers from Spain that starts with a "G", *plus* all customers that starts with an "R", regardless of the country value:

Example

Select all customers that either:
are from Spain and starts with either "G", *or*
starts with the letter "R":

```
SELECT * FROM Customers
WHERE Country
= 'Spain' AND CustomerName LIKE 'G%' OR CustomerName LIKE 'R%';
```

Test Yourself With Exercises

Exercise:

Select all records where the `City` column has the value 'Berlin' or 'London'.

_____ * FROM Customers

_____ City = 'Berlin'

_____._____ = '_____';

Submit Answer »

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

SQL NOT Operator

The NOT Operator

The `NOT` operator is used in combination with other operators to give the opposite result, also called the negative result.

In the select statement below we want to return all customers that are NOT from Spain:

Example

Select only the customers that are NOT from Spain:

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

In the example above, the `NOT` operator is used in combination with the `=` operator, but it can be used in combination with other comparison and/or logical operators. See examples below.

Syntax

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

Demo Database

Below is a selection from the **Customers** table used in the examples:

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

NOT LIKE

Example

Select customers that does not start with the letter 'A':

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

NOT BETWEEN

Example

Select customers with a customerID not between 10 and 60:

```
SELECT * FROM Customers
WHERE CustomerID NOT BETWEEN 10 AND 60;
```

NOT IN

Example

Select customers that are not from Paris or London:

```
SELECT * FROM Customers
WHERE City NOT IN ('Paris', 'London');
```

NOT Greater Than

Example

Select customers with a CustomerId not greater than 50:

```
SELECT * FROM Customers
WHERE NOT CustomerID > 50;
```

Note: There is a not-greater-than operator: `!>` that would give you the same result.

NOT Less Than

Example

Select customers with a CustomerID not less than 50:

```
SELECT * FROM Customers
WHERE NOT CustomerId < 50;
```

Note: There is a not-less-than operator: `!<` that would give you the same result.

Test Yourself With Exercises

Exercise:

Use the `NOT` keyword to select all records where `City` is NOT "Berlin".

```
SELECT * FROM Customers
_____________ = '_____________';
```

Submit Answer »

```
SELECT * FROM Customers
WHERE NOT City = 'Berlin';
```

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:

1. Specify both the column names and the values to be inserted:

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

2. 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. Here, 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 used in the examples:

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

Insert Multiple Rows

It is also possible to insert multiple rows in one statement.

To insert multiple rows of data, we use the same `INSERT INTO` statement, but with multiple values:

Example

```
INSERT INTO Customers (CustomerName, ContactName, Address, City,
PostalCode, Country)
VALUES
('Cardinal', 'Tom B. Erichsen', 'Skagen
21', 'Stavanger', '4006', 'Norway'),
('Greasy Burger', 'Per Olsen', 'Gateveien
15', 'Sandnes', '4306', 'Norway'),
('Tasty Tee', 'Finn Egan', 'Streetroad 19B', 'Liverpool', 'L1
0AA', 'UK');
```

Make sure you separate each set of values with a comma `,`.

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
93	Greasy Burger	Per Olsen	Gateveien 15	Sandnes	4306	Norway
94	Tasty Tee	Finn Egan	Streetroad 19B	Liverpool	L1 0AA	UK

Test Yourself With Exercises

Exercise:

Insert a new record in the `Customers` table.

```
_____ Customers _____
CustomerName,
Address,
City,
PostalCode,
Country _____
_____.______
'Hekkan Burger',
```

```
'Gateveien 15',
'Sandnes',
'4306',
'Norway' _____;
```

Submit Answer »

```
INSERT INTO Customers (
CustomerName,
Address,
City,
PostalCode,
Country)
VALUES (
'Hekkan Burger',
'Gateveien 15',
'Sandnes',
'4306',
'Norway');
```

SQL NULL Values

What is a NULL Value?

A field with a NULL value is a field with no value.

If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

Note: A NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation!

How to 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.

IS NULL Syntax

```
SELECT column_names
FROM table_name
WHERE column_name IS NULL;
```

IS NOT NULL Syntax

```
SELECT column_names
FROM table_name
WHERE column_name IS NOT NULL;
```

Demo Database

Below is a selection from the **Customers** table used in the examples:

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 IS NULL Operator

The `IS NULL` operator is used to test for empty values (NULL values).

The following SQL lists all customers with a NULL value in the "Address" field:

Example

```
SELECT CustomerName, ContactName, Address
FROM Customers
WHERE Address IS NULL;
```

Tip: Always use IS NULL to look for NULL values.

The IS NOT NULL Operator

The `IS NOT NULL` operator is used to test for non-empty values (NOT NULL values).

The following SQL lists all customers with a value in the "Address" field:

Example

```
SELECT CustomerName, ContactName, Address
FROM Customers
WHERE Address IS NOT NULL;
```

Test Yourself With Exercises

Exercise:

Select all records from the `Customers` where the `PostalCode` column is empty.

```
SELECT * FROM Customers
WHERE _____._____._____;
```

Submit Answer »

```
SELECT * FROM Customers
WHERE PostalCode IS NULL;
```

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 used in the examples:

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 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

Test Yourself With Exercises

Exercise:

Update the `City` column of all records in the `Customers` table.

```
_____________ Customers
_____________ City = 'Oslo';
```

Submit Answer »

```
UPDATE Customers
SET City = 'Oslo';
```

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) 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 used in the examples:

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;
```

The following SQL statement deletes all rows in the "Customers" table, without deleting the table:

Example

```
DELETE FROM Customers;
```

Delete a Table

To delete the table completely, use the `DROP TABLE` statement:

Example

Remove the Customers table:

```
DROP TABLE Customers;
```

Test Yourself With Exercises

Exercise:

Delete all the records from the `Customers` table where the `Country` value is 'Norway'.

```
_____ Customers
_____ Country = 'Norway';
```

Submit Answer »

```
DELETE FROM Customers
WHERE Country = 'Norway';
```

SQL TOP, LIMIT, FETCH FIRST 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 performance.

Example

Select only the first 3 records of the Customers table:

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

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 `FETCH FIRST n ROWS ONLY` and `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 12 Syntax:

```
SELECT column_name(s)
FROM table_name
ORDER BY column_name(s)
FETCH FIRST number ROWS ONLY;
```

Older Oracle Syntax:

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

Older Oracle Syntax (with ORDER BY):

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

Demo Database

Below is a selection from the **Customers** table used in the examples:

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

LIMIT

The following SQL statement shows the equivalent example for MySQL:

Example

Select the first 3 records of the Customers table:

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

FETCH FIRST

The following SQL statement shows the equivalent example for Oracle:

Example

Select the first 3 records of the Customers table:

```
SELECT * FROM Customers
FETCH FIRST 3 ROWS ONLY;
```

SQL TOP PERCENT Example

The following SQL statement selects the first 50% of the records from the "Customers" table (for SQL Server/MS Access):

Example

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

The following SQL statement shows the equivalent example for Oracle:

Example

```
SELECT * FROM Customers
FETCH FIRST 50 PERCENT ROWS ONLY;
```

ADD a WHERE CLAUSE

The following SQL statement selects the first three records from the "Customers" table, where the country is "Germany" (for SQL Server/MS Access):

Example

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

The following SQL statement shows the equivalent example for MySQL:

Example

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

The following SQL statement shows the equivalent example for Oracle:

Example

```
SELECT * FROM Customers
WHERE Country='Germany'
FETCH FIRST 3 ROWS ONLY;
```

ADD the ORDER BY Keyword

Add the `ORDER BY` keyword when you want to sort the result, and return the first 3 records of the sorted result.

For SQL Server and MS Access:

Example

Sort the result reverse alphabetically by CustomerName, and return the first 3 records:

```
SELECT TOP 3 * FROM Customers
ORDER BY CustomerName DESC;
```

The following SQL statement shows the equivalent example for MySQL:

Example

```
SELECT * FROM Customers
ORDER BY CustomerName DESC
LIMIT 3;
```

The following SQL statement shows the equivalent example for Oracle:

Example

```
SELECT * FROM Customers
ORDER BY CustomerName DESC
FETCH FIRST 3 ROWS ONLY;
```

SQL Aggregate Functions

SQL Aggregate Functions

An aggregate function is a function that performs a calculation on a set of values, and returns a single value.

Aggregate functions are often used with the `GROUP BY` clause of the `SELECT` statement. The `GROUP BY` clause splits the result-set into groups of values and the aggregate function can be used to return a single value for each group.

The most commonly used SQL aggregate functions are:

- `MIN()` - returns the smallest value within the selected column
- `MAX()` - returns the largest value within the selected column
- `COUNT()` - returns the number of rows in a set
- `SUM()` - returns the total sum of a numerical column
- `AVG()` - returns the average value of a numerical column

Aggregate functions ignore null values (except for `COUNT()`).

We will go through the aggregate functions above in the next chapters.

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 Example

Find the lowest price in the Price column:

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

MAX Example

Find the highest price in the Price column:

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

Syntax

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

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

Demo Database

Below is a selection from the **Products** table used in the examples:

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	22
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35

Set Column Name (Alias)

When you use `MIN()` or `MAX()`, the returned column will not have a descriptive name. To give the column a descriptive name, use the `AS` keyword:

Example

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

Use MIN() with GROUP BY

Here we use the `MIN()` function and the `GROUP BY` clause, to return the smallest price for each category in the Products table:

Example

```
SELECT MIN(Price) AS SmallestPrice, CategoryID
FROM Products
GROUP BY CategoryID;
```

You will learn more about the GROUP BY clause later in this tutorial.

Test Yourself With Exercises

Exercise:

Use the `MIN` function to select the record with the smallest value of the `Price` column.

```
SELECT _____
FROM Products;
```

Submit Answer »

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

SQL COUNT() Function

The SQL COUNT() Function

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

Example

Find the total number of rows in the `Products` table:

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

Syntax

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

Demo Database

Below is a selection from the **Products** table used in the examples:

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	22
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35

Specify Column

You can specify a column name instead of the asterix symbol `(*)`.

If you specify a column name instead of `(*)`, NULL values will not be counted.

Example

Find the number of products where the `ProductName` is not null:

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

Add a WHERE Clause

You can add a `WHERE` clause to specify conditions:

Example

Find the number of products where `Price` is higher than 20:

```
SELECT COUNT(ProductID)
FROM Products
WHERE Price > 20;
```

Ignore Duplicates

You can ignore duplicates by using the `DISTINCT` keyword in the `COUNT()` function.

If `DISTINCT` is specified, rows with the same value for the specified column will be counted as one.

Example

How many *different* prices are there in the `Products` table:

```
SELECT COUNT(DISTINCT Price)
FROM Products;
```

Use an Alias

Give the counted column a name by using the `AS` keyword.

Example

Name the column "Number of records":

```
SELECT COUNT(*) AS [Number of records]
FROM Products;
```

Use COUNT() with GROUP BY

Here we use the `COUNT()` function and the `GROUP BY` clause, to return the number of records for each category in the Products table:

Example

```
SELECT COUNT(*) AS [Number of records], CategoryID
FROM Products
GROUP BY CategoryID;
```

You will learn more about the `GROUP BY` clause later in this tutorial.

Test Yourself With Exercises

Exercise:

Use the correct function to return the number of records that have the `Price` value set to 18.

```
SELECT _____ (*)
FROM Products
_____ Price = 18;
```

Submit Answer »

```
SELECT COUNT(*)
FROM Products
WHERE Price = 18;
```

SQL SUM() Function

The SQL SUM() Function

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

Example

Return the sum of all `Quantity` fields in the `OrderDetails` table:

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

Syntax

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

Demo Database

Below is a selection from the **OrderDetails** table used in the examples:

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

Add a WHERE Clause

You can add a `WHERE` clause to specify conditions:

Example

Return the sum of the `Quantity` field for the product with `ProductID` 11:

```
SELECT SUM(Quantity)
FROM OrderDetails
WHERE ProductId = 11;
```

Use an Alias

Give the summarized column a name by using the `AS` keyword.

Example

Name the column "total":

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

Use SUM() with GROUP BY

Here we use the `SUM()` function and the `GROUP BY` clause, to return the `Quantity` for each `OrderID` in the OrderDetails table:

Example

```
SELECT OrderID, SUM(Quantity) AS [Total Quantity]
FROM OrderDetails
GROUP BY OrderID;
```

You will learn more about the `GROUP BY` clause later in this tutorial.

SUM() With an Expression

The parameter inside the `SUM()` function can also be an expression.

If we assume that each product in the `OrderDetails` column costs 10 dollars, we can find the total earnings in dollars by multiply each quantity with 10:

Example

Use an expression inside the `SUM()` function:

```
SELECT SUM(Quantity * 10)
FROM OrderDetails;
```

We can also join the `OrderDetails` table to the `Products` table to find the actual amount, instead of assuming it is 10 dollars:

Example

Join `OrderDetails` with `Products`, and use `SUM()` to find the total amount:

```
SELECT SUM(Price * Quantity)
FROM OrderDetails
LEFT JOIN Products ON OrderDetails.ProductID = Products.ProductID;
```

You will learn more about Joins later in this tutorial.

Test Yourself With Exercises

Exercise:

Use an SQL function to calculate the sum of all the `Price` column values in the `Products` table.

```
SELECT _____
FROM Products;
```

Submit Answer »

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

SQL AVG() Function

The SQL AVG() Function

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

Example

Find the average price of all products:

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

Note: NULL values are ignored.

Syntax

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

Demo Database

Below is a selection from the **Products** table used in the examples:

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	22

5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35

Add a WHERE Clause

You can add a `WHERE` clause to specify conditions:

Example

Return the average price of products in category 1:

```
SELECT AVG(Price)
FROM Products
WHERE CategoryID = 1;
```

Use an Alias

Give the AVG column a name by using the `AS` keyword.

Example

Name the column "average price":

```
SELECT AVG(Price) AS [average price]
FROM Products;
```

Higher Than Average

To list all records with a higher price than average, we can use the `AVG()` function in a sub query:

Example

Return all products with a higher price than the average price:

```
SELECT * FROM Products
WHERE price > (SELECT AVG(price) FROM Products);
```

Use AVG() with GROUP BY

Here we use the `AVG()` function and the `GROUP BY` clause, to return the average price for each category in the Products table:

Example

```
SELECT AVG(Price) AS AveragePrice, CategoryID
FROM Products
GROUP BY CategoryID;
```

You will learn more about the GROUP BY clause later in this tutorial.

Test Yourself With Exercises

Exercise:

Use an SQL function to calculate the average price of all products.

```
SELECT _____
FROM Products;
```

Submit Answer »

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

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 often used in conjunction with the `LIKE` operator:

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

You will learn more about wildcards in the next chapter.

Example

Select all customers that starts with the letter "a":

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

Syntax

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

Demo Database

Below is a selection from the **Customers** table used in the examples:

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 _ Wildcard

The _ wildcard represents a single character.

It can be any character or number, but each _ represents one, and only one, character.

Example

Return all customers from a city that starts with 'L' followed by one wildcard character, then 'nd' and then two wildcard characters:

```
SELECT * FROM Customers
WHERE city LIKE 'L_nd__';
```

The % Wildcard

The `%` wildcard represents any number of characters, even zero characters.

Example

Return all customers from a city that *contains* the letter 'L':

```
SELECT * FROM Customers
WHERE city LIKE '%L%';
```

Starts With

To return records that starts with a specific letter or phrase, add the `%` at the end of the letter or phrase.

Example

Return all customers that starts with 'La':

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

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

Example

Return all customers that starts with 'a' or starts with 'b':

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

Ends With

To return records that ends with a specific letter or phrase, add the `%` at the beginning of the letter or phrase.

Example

Return all customers that ends with 'a':

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

Tip: You can also combine "starts with" and "ends with":

Example

Return all customers that starts with "b" and ends with "s":

```
SELECT * FROM Customers
WHERE CustomerName LIKE 'b%s';
```

Contains

To return records that contains a specific letter or phrase, add the `%` both before and after the letter or phrase.

Example

Return all customers that contains the phrase 'or'

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

Combine Wildcards

Any wildcard, like `%` and `_` , can be used in combination with other wildcards.

Example

Return all customers that starts with "a" and are at least 3 characters in length:

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

Example

Return all customers that have "r" in the second position:

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

Without Wildcard

If no wildcard is specified, the phrase has to have an exact match to return a result.

Example

Return all customers from Spain:

```
SELECT * FROM Customers
WHERE Country LIKE 'Spain';
```

Test Yourself With Exercises

Exercise:

Select all records where the value of the `City` column starts with the letter "a".

```
SELECT * FROM Customers
_____________;
```

Submit Answer »

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

SQL Wildcards

SQL Wildcard Characters

A wildcard character is used to substitute one or more characters in a string.

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

Example

Return all customers that starts with the letter 'a':

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

Wildcard Characters

Symbol	Description
%	Represents zero or more characters
_	Represents a single character
[]	Represents any single character within the brackets *
^	Represents any character not in the brackets *
-	Represents any single character within the specified range *

{}	Represents any escaped character **

* Not supported in PostgreSQL and MySQL databases.

** Supported only in Oracle databases.

Demo Database

Below is a selection from the **Customers** table used in the examples:

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 `%` wildcard represents any number of characters, even zero characters.

Example

Return all customers that ends with the pattern 'es':

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

Example

Return all customers that *contains* the pattern 'mer':

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

Using the _ Wildcard

The _ wildcard represents a single character.

It can be any character or number, but each _ represents one, and only one, character.

Example

Return all customers with a `City` starting with any character, followed by "ondon":

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

Example

Return all customers with a `City` starting with "L", followed by any 3 characters, ending with "on":

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

Using the [] Wildcard

The `[]` wildcard returns a result if *any* of the characters inside gets a match.

Example

Return all customers starting with either "b", "s", or "p":

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

Using the - Wildcard

The `-` wildcard allows you to specify a range of characters inside the `[]` wildcard.

Example

Return all customers starting with "a", "b", "c", "d", "e" or "f":

```
SELECT * FROM Customers
WHERE CustomerName LIKE '[a-f]%';
```

Combine Wildcards

Any wildcard, like `%` and `_` , can be used in combination with other wildcards.

Example

Return all customers that starts with "a" and are at least 3 characters in length:

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

Example

Return all customers that have "r" in the second position:

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

Without Wildcard

If no wildcard is specified, the phrase has to have an exact match to return a result.

Example

Return all customers from Spain:

```
SELECT * FROM Customers
WHERE Country LIKE 'Spain';
```

Microsoft Access Wildcards

The Microsoft Access Database has some other wildcards:

Symbol	Description	Example
*	Represents zero or more characters	bl* finds bl, black, blue, and blob
?	Represents a single character	h?t finds hot, hat, and hit
[]	Represents any single character within the brackets	h[oa]t finds hot and hat, but not hit

!	Represents any character not in the brackets	h[!oa]t finds hit, but not hot and hat
-	Represents any single character within the specified range	c[a-b]t finds cat and cbt
#	Represents any single numeric character	2#5 finds 205, 215, 225, 235, 245, 255, 265, 275, 285, and 295

Test Yourself With Exercises

Exercise:

Select all records where the second letter of the `City` is an "a".

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

Submit Answer »

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

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.

Example

Return all customers from 'Germany', 'France', or 'UK'

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

Syntax

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

Demo Database

Below is a selection from the **Customers** table used in the examples:

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

NOT IN

By using the `NOT` keyword in front of the `IN` operator, you return all records that are NOT any of the values in the list.

Example

Return all customers that are NOT from 'Germany', 'France', or 'UK':

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

IN (SELECT)

You can also use `IN` with a subquery in the `WHERE` clause.

With a subquery you can return all records from the main query that are present in the result of the subquery.

Example

Return all customers that have an order in the **Orders** table:

```
SELECT * FROM Customers
WHERE CustomerID IN (SELECT CustomerID FROM Orders);
```

NOT IN (SELECT)

The result in the example above returned 74 records, that means that there are 17 customers that haven't placed any orders.

Let us check if that is correct, by using the `NOT IN` operator.

Example

Return all customers that have NOT placed any orders in the **Orders** table:

```
SELECT * FROM Customers
WHERE CustomerID NOT IN (SELECT CustomerID FROM Orders);
```

Test Yourself With Exercises

Exercise:

Use the `IN` operator to select all the records where `Country` is either "Norway" or "France".

```
SELECT * FROM Customers
_____ _____ 'France'_____;
```

Submit Answer »

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

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.

Example

Selects all products with a price between 10 and 20:

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

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 used in the examples:

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	22
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35

NOT BETWEEN

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

The following SQL statement selects all products with a price between 10 and 20. In addition, the CategoryID must be either 1,2, or 3:

Example

```
SELECT * FROM Products
WHERE Price BETWEEN 10 AND 20
AND CategoryID IN (1,2,3);
```

BETWEEN Text Values

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

Example

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

The following SQL statement selects all products with a ProductName between Carnarvon Tigers and Chef Anton's Cajun Seasoning:

Example

```
SELECT * FROM Products
WHERE ProductName BETWEEN "Carnarvon Tigers" AND "Chef Anton's Cajun
Seasoning"
ORDER BY ProductName;
```

NOT BETWEEN Text Values

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;
```

BETWEEN Dates

The following SQL statement selects all orders with an OrderDate between '01-July-1996' and '31-July-1996':

Example

```
SELECT * FROM Orders
WHERE OrderDate BETWEEN #07/01/1996# AND #07/31/1996#;
```

OR:

Example

```
SELECT * FROM Orders
WHERE OrderDate BETWEEN '1996-07-01' AND '1996-07-31';
```

Sample Table

Below is a selection from the **Orders** table used in the examples:

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

Test Yourself With Exercises

Exercise:

Use the `BETWEEN` operator to select all the records where the value of the `Price` column is between 10 and 20.

```
SELECT * FROM Products
WHERE Price _____________;
```

Submit Answer »

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

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 that query.

An alias is created with the `AS` keyword.

Example

```
SELECT CustomerID AS ID
FROM Customers;
```

AS is Optional

Actually, in most database languages, you can skip the AS keyword and get the same result:

Example

```
SELECT CustomerID ID
FROM Customers;
```

Syntax

When alias is used on column:

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

When alias is used on table:

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

Demo Database

Below is a selection from the **Customers** and **Orders** tables used in the examples:

Customers

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

Orders

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

Alias for Columns

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;
```

Using Aliases With a Space Character

If you want your alias to contain one or more spaces, like "`My Great Products`", surround your alias with square brackets or double quotes.

Example

Using [square brackets] for aliases with space characters:

```
SELECT ProductName AS [My Great Products]
FROM Products;
```

Example

Using "double quotes" for aliases with space characters:

```
SELECT ProductName AS "My Great Products"
FROM Products;
```

Note: Some database systems allows both [] and "", and some only allows one of them.

Concatenate Columns

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:

MySQL Example

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

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

Oracle Example

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

Alias for Tables

The same rules applies when you want to use an alias for a table.

Example

Refer to the Customers table as Persons instead:

```
SELECT * FROM Customers AS Persons;
```

It might seem useless to use aliases on tables, but when you are using more than one table in your queries, it can make the SQL statements shorter.

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

Test Yourself With Exercises

Exercise:

When displaying the `Customers` table, make an ALIAS of the `PostalCode` column, the column should be called `Pno` instead.

```
SELECT CustomerName,
Address,
PostalCode _____________
FROM Customers;
```

Submit Answer »

```
SELECT CustomerName,
Address,
PostalCode AS Pno
FROM Customers;
```

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`: Returns all records from the left table, and the matched records from the right table
- `RIGHT (OUTER) JOIN`: Returns all records from the right table, and the matched records from the left table
- `FULL (OUTER) JOIN`: Returns all records when there is a match in either left or right table

Test Yourself With Exercises

Exercise:

Insert the missing parts in the `JOIN` clause to join the two tables `Orders` and `Customers`, using the `CustomerID` field in both tables as the relationship between the two tables.

```
SELECT *
FROM Orders
LEFT JOIN Customers
_____=_____;
```

Submit Answer »

```
SELECT *
FROM Orders
LEFT JOIN Customers
ON Orders.CustomerID=Customers.CustomerID;
```

SQL INNER JOIN

INNER JOIN

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

Let's look at a selection of the **Products** table:

ProductID	ProductName	CategoryID	Price

1	Chais	1	18
2	Chang	1	19
3	Aniseed Syrup	2	10

And a selection of the **Categories** table:

CategoryID	CategoryName	Description
1	Beverages	Soft drinks, coffees, teas, beers, and ales
2	Condiments	Sweet and savory sauces, relishes, spreads, and seasonings
3	Confections	Desserts, candies, and sweet breads

We will join the Products table with the Categories table, by using the `CategoryID` field from both tables:

Example

Join Products and Categories with the INNER JOIN keyword:

```
SELECT ProductID, ProductName, CategoryName
FROM Products
INNER JOIN Categories ON Products.CategoryID = Categories.CategoryID;
```

Note: The `INNER JOIN` keyword returns only rows with a match in both tables. Which means that if you have a product with no CategoryID, or with a CategoryID that is not present in the Categories table, that record would not be returned in the result.

Syntax

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

Naming the Columns

It is a good practice to include the table name when specifying columns in the SQL statement.

Example

Specify the table names:

```
SELECT Products.ProductID, Products.ProductName, Categories.CategoryName
FROM Products
INNER JOIN Categories ON Products.CategoryID = Categories.CategoryID;
```

The example above works without specifying table names, because none of the specified column names are present in both tables. If you try to include `CategoryID` in the `SELECT` statement, you will get an error if you do not specify the table name (because `CategoryID` is present in both tables).

JOIN or INNER JOIN

`JOIN` and `INNER JOIN` will return the same result.

`INNER` is the default join type for `JOIN`, so when you write `JOIN` the parser actually writes `INNER JOIN`.

Example

JOIN is the same as INNER JOIN:

```
SELECT Products.ProductID, Products.ProductName, Categories.CategoryName
FROM Products
JOIN Categories ON Products.CategoryID = Categories.CategoryID;
```

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);
```

Test Yourself With Exercises

Exercise:

Choose the correct `JOIN` clause to select all records from the two tables where there is a match in both tables.

```
SELECT *
FROM Orders
_____________
ON Orders.CustomerID=Customers.CustomerID;
```

Submit Answer »

```
SELECT *
FROM Orders
INNER JOIN Customers
ON Orders.CustomerID=Customers.CustomerID;
```

SQL LEFT JOIN Keyword

SQL LEFT JOIN Keyword

The `LEFT JOIN` keyword returns all records from the left table (table1), and the matching records from the right table (table2). The result is 0 records 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 matching records from the left table (table1). The result is 0 records from the left side, if 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 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).

Test Yourself With Exercises

Exercise:

Choose the correct `JOIN` clause to select all the records from the `Customers` table plus all the matches in the `Orders` table.

```
SELECT *
FROM Orders
_____________
ON Orders.CustomerID=Customers.CustomerID;
```

Submit Answer »

```
SELECT *
FROM Orders
RIGHT JOIN Customers
ON Orders.CustomerID=Customers.CustomerID;
```

SQL FULL OUTER JOIN Keyword

SQL FULL OUTER JOIN Keyword

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

Tip: `FULL OUTER JOIN` and `FULL JOIN` are the same.

FULL OUTER JOIN Syntax

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

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

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
Null	10309
Null	10310
Alfreds Futterkiste	*Null*
Ana Trujillo Emparedados y helados	10308

Antonio Moreno Taquería	*Null*

Note: The `FULL OUTER JOIN` keyword returns all matching records from both tables whether the other table matches or not. So, 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

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;
```

T1 and *T2* are different table aliases for the same table.

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 SQL UNION Operator

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

- Every `SELECT` statement within `UNION` must have the same number of columns
- The columns must also have similar data types
- The columns in every `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.

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 returns the cities (only distinct values) from both the "Customers" and the "Suppliers" table:

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 returns the cities (duplicate values also) from both the "Customers" and the "Suppliers" table:

Example

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

SQL UNION With WHERE

The following SQL statement returns the German cities (only distinct values) from both the "Customers" and the "Suppliers" table:

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 returns the German cities (duplicate values also) from both the "Customers" and the "Suppliers" table:

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;
```

Notice the "AS Type" above - it is an alias. SQL Aliases are used to give a table or a column a temporary name. An alias only exists for the duration of the query. So, here we have created a temporary column named "Type", that list whether the contact person is a "Customer" or a "Supplier".

SQL GROUP BY Statement

The SQL GROUP BY Statement

The `GROUP BY` statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

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;
```

Test Yourself With Exercises

Exercise:

List the number of customers in each country.

```
SELECT _____ (CustomerID),
Country
FROM Customers
_____;
```

Submit Answer »

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

SQL HAVING Clause

The SQL HAVING Clause

The `HAVING` clause was added to SQL because the `WHERE` keyword cannot 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 EXISTS Operator

The SQL EXISTS Operator

The `EXISTS` operator is used to test for the existence of any record in a subquery.

The `EXISTS` operator returns TRUE if the subquery returns one or more records.

EXISTS Syntax

```
SELECT column_name(s)
FROM table_name
WHERE EXISTS
(SELECT 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	22
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35

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
4	Tokyo Traders	Yoshi Nagase	9-8 Sekimai Musashino-shi	Tokyo	100	Japan

SQL EXISTS Examples

The following SQL statement returns TRUE and lists the suppliers with a product price less than 20:

Example

```
SELECT SupplierName
FROM Suppliers
WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID
= Suppliers.supplierID AND Price < 20);
```

The following SQL statement returns TRUE and lists the suppliers with a product price equal to 22:

Example

```
SELECT SupplierName
FROM Suppliers
WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID
= Suppliers.supplierID AND Price = 22);
```

SQL ANY and ALL Operators

The SQL ANY and ALL Operators

The ANY and ALL operators allow you to perform a comparison between a single column value and a range of other values.

The SQL ANY Operator

The ANY operator:

- returns a boolean value as a result
- returns TRUE if ANY of the subquery values meet the condition

ANY means that the condition will be true if the operation is true for any of the values in the range.

ANY Syntax

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ANY
  (SELECT column_name
  FROM table_name
  WHERE condition);
```

Note: The *operator* must be a standard comparison operator (=, <>, !=, >, >=, <, or <=).

The SQL ALL Operator

The `ALL` operator:

- returns a boolean value as a result
- returns TRUE if ALL of the subquery values meet the condition
- is used with `SELECT`, `WHERE` and `HAVING` statements

`ALL` means that the condition will be true only if the operation is true for all values in the range.

ALL Syntax With SELECT

```
SELECT ALL column_name(s)
FROM table_name
WHERE condition;
```

ALL Syntax With WHERE or HAVING

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator ALL
  (SELECT column_name
  FROM table_name
  WHERE condition);
```

Note: The *operator* must be a standard comparison operator (=, <>, !=, >, >=, <, or <=).

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	22
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25
7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30
8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97

And 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
6	10250	41	10
7	10250	51	35
8	10250	65	15
9	10251	22	6
10	10251	57	15

SQL ANY Examples

The following SQL statement lists the ProductName if it finds ANY records in the OrderDetails table has Quantity equal to 10 (this will return TRUE because the Quantity column has some values of 10):

Example

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY
  (SELECT ProductID
  FROM OrderDetails
  WHERE Quantity = 10);
```

The following SQL statement lists the ProductName if it finds ANY records in the OrderDetails table has Quantity larger than 99 (this will return TRUE because the Quantity column has some values larger than 99):

Example

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY
  (SELECT ProductID
  FROM OrderDetails
  WHERE Quantity > 99);
```

The following SQL statement lists the ProductName if it finds ANY records in the OrderDetails table has Quantity larger than 1000 (this will return FALSE because the Quantity column has no values larger than 1000):

Example

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY
  (SELECT ProductID
  FROM OrderDetails
  WHERE Quantity > 1000);
```

SQL ALL Examples

The following SQL statement lists ALL the product names:

Example

```
SELECT ALL ProductName
FROM Products
WHERE TRUE;
```

The following SQL statement lists the ProductName if ALL the records in the OrderDetails table has Quantity equal to 10. This will of course return FALSE because the Quantity column has many different values (not only the value of 10):

Example

```
SELECT ProductName
FROM Products
WHERE ProductID = ALL
  (SELECT ProductID
  FROM OrderDetails
  WHERE Quantity = 10);
```

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.

The `INSERT INTO SELECT` statement requires that the data types in source and target tables match.

Note: 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
1	Exotic Liquid	Charlotte Cooper	49 Gilbert St.	Londona	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 INSERT INTO SELECT Examples

Example

Copy "Suppliers" into "Customers" (the columns that are not filled with data, will contain NULL):

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

Example

Copy "Suppliers" into "Customers" (fill all columns):

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

Example

Copy only the German suppliers into "Customers":

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

SQL CASE Expression

The SQL CASE Expression

The `CASE` expression goes through conditions and returns a value when the first condition is met (like an if-then-else statement). So, once a condition is true, it will stop reading and return the result. If no conditions are true, it returns the value in the `ELSE` clause.

If there is no `ELSE` part and no conditions are true, it returns NULL.

CASE Syntax

```
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
```

```
    WHEN conditionN THEN resultN
    ELSE result
END;
```

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

SQL CASE Examples

The following SQL goes through conditions and returns a value when the first condition is met:

Example

```
SELECT OrderID, Quantity,
CASE
    WHEN Quantity > 30 THEN 'The quantity is greater than 30'
    WHEN Quantity = 30 THEN 'The quantity is 30'
    ELSE 'The quantity is under 30'
END AS QuantityText
FROM OrderDetails;
```

The following SQL will order the customers by City. However, if City is NULL, then order by Country:

Example

```
SELECT CustomerName, City, Country
FROM Customers
ORDER BY
(CASE
    WHEN City IS NULL THEN Country
    ELSE City
END);
```

SQL NULL Functions

SQL IFNULL(), ISNULL(), COALESCE(), and NVL() 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.

Look at 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 will be NULL.

Solutions

MySQL

The MySQL IFNULL() function lets you return an alternative value if an expression is NULL:

```
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 Server

The SQL Server ISNULL() function lets you return an alternative value when an expression is NULL:

```
SELECT ProductName, UnitPrice * (UnitsInStock + ISNULL(UnitsOnOrder, 0))
FROM Products;
```

or we can use the COALESCE() function, like this:

```
SELECT ProductName, UnitPrice * (UnitsInStock
+ COALESCE(UnitsOnOrder, 0))
FROM Products;
```

MS Access

The MS Access IsNull() function returns TRUE (-1) if the expression is a null value, otherwise FALSE (0):

```
SELECT ProductName, UnitPrice * (UnitsInStock +
IIF(IsNull(UnitsOnOrder), 0, UnitsOnOrder))
FROM Products;
```

Oracle

The Oracle NVL() function achieves the same result:

```
SELECT ProductName, UnitPrice * (UnitsInStock + NVL(UnitsOnOrder, 0))
FROM Products;
```

or we can use the COALESCE() function, like this:

```
SELECT ProductName, UnitPrice * (UnitsInStock
+ COALESCE(UnitsOnOrder, 0))
FROM Products;
```

SQL Stored Procedures for SQL Server

What is a Stored Procedure?

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

So if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it.

You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

Stored Procedure Syntax

```
CREATE PROCEDURE procedure_name
AS
sql_statement
GO;
```

Execute a Stored Procedure

```
EXEC procedure_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

Stored Procedure Example

The following SQL statement creates a stored procedure named "SelectAllCustomers" that selects all records from the "Customers" table:

Example

```
CREATE PROCEDURE SelectAllCustomers
AS
SELECT * FROM Customers
GO;
```

Execute the stored procedure above as follows:

Example

```
EXEC SelectAllCustomers;
```

Stored Procedure With One Parameter

The following SQL statement creates a stored procedure that selects Customers from a particular City from the "Customers" table:

Example

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30)
AS
SELECT * FROM Customers WHERE City = @City
GO;
```

Execute the stored procedure above as follows:

Example

```
EXEC SelectAllCustomers @City = 'London';
```

Stored Procedure With Multiple Parameters

Setting up multiple parameters is very easy. Just list each parameter and the data type separated by a comma as shown below.

The following SQL statement creates a stored procedure that selects Customers from a particular City with a particular PostalCode from the "Customers" table:

Example

```
CREATE PROCEDURE SelectAllCustomers @City nvarchar(30), @PostalCode
nvarchar(10)
AS
SELECT * FROM Customers WHERE City = @City AND PostalCode = @PostalCode
GO;
```

Execute the stored procedure above as follows:

Example

```
EXEC SelectAllCustomers @City = 'London', @PostalCode = 'WA1 1DP';
```

SQL Comments

SQL Comments

Comments are used to explain sections of SQL statements, or to prevent execution of SQL statements.

Note: Comments are not supported in Microsoft Access databases!

Single Line Comments

Single line comments start with --.

Any text between -- and the end of the line will be ignored (will not be executed).

The following example uses a single-line comment as an explanation:

Example

```
-- Select all:
SELECT * FROM Customers;
```

The following example uses a single-line comment to ignore the end of a line:

Example

```
SELECT * FROM Customers -- WHERE City='Berlin';
```

The following example uses a single-line comment to ignore a statement:

Example

```
-- SELECT * FROM Customers;
SELECT * FROM Products;
```

Multi-line Comments

Multi-line comments start with `/*` and end with `*/`.

Any text between /* and */ will be ignored.

The following example uses a multi-line comment as an explanation:

Example

```
/*Select all the columns
of all the records
in the Customers table:*/
SELECT * FROM Customers;
```

The following example uses a multi-line comment to ignore many statements:

Example

```
/*SELECT * FROM Customers;
SELECT * FROM Products;
SELECT * FROM Orders;
```

```
SELECT * FROM Categories;*/
SELECT * FROM Suppliers;
```

To ignore just a part of a statement, also use the /* */ comment.

The following example uses a comment to ignore part of a line:

Example

```
SELECT CustomerName, /*City,*/ Country FROM Customers;
```

The following example uses a comment to ignore part of a statement:

Example

```
SELECT * FROM Customers WHERE (CustomerName LIKE 'L%'
OR CustomerName LIKE 'R%' /*OR CustomerName LIKE 'S%'
OR CustomerName LIKE 'T%'*/ OR CustomerName LIKE 'W%')
AND Country='USA'
ORDER BY CustomerName;
```

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

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 DATABASE

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;`

Test Yourself With Exercises

Exercise:

Write the correct SQL statement to create a new database called `testDB`.

_____;

Submit Answer »

```
CREATE DATABASE testDB;
```

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;`

Exercise:

Write the correct SQL statement to delete a database named `testDB`.

_____;

Submit Answer »

DROP DATABASE testDB

SQL BACKUP DATABASE for SQL Server

The SQL BACKUP DATABASE Statement

The `BACKUP DATABASE` statement is used in SQL Server to create a full back up of an existing SQL database.

Syntax

```
BACKUP DATABASE databasename
TO DISK = 'filepath';
```

The SQL BACKUP WITH DIFFERENTIAL Statement

A differential back up only backs up the parts of the database that have changed since the last full database backup.

Syntax

```
BACKUP DATABASE databasename
TO DISK = 'filepath'
WITH DIFFERENTIAL;
```

BACKUP DATABASE Example

The following SQL statement creates a full back up of the existing database "testDB" to the D disk:

Example

```
BACKUP DATABASE testDB
TO DISK = 'D:\backups\testDB.bak';
```

Tip: Always back up the database to a different drive than the actual database. Then, if you get a disk crash, you will not lose your backup file along with the database.

BACKUP WITH DIFFERENTIAL Example

The following SQL statement creates a differential back up of the database "testDB":

Example

```
BACKUP DATABASE testDB
TO DISK = 'D:\backups\testDB.bak'
WITH DIFFERENTIAL;
```

Tip: A differential back up reduces the back up time (since only the changes are backed up).

SQL CREATE TABLE Statement

The 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.).

Tip: For an overview of the available data types, go to our complete Data Types Reference.

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 also be created using `CREATE TABLE`.

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 ....;
```

The following SQL creates a new table called "TestTables" (which is a copy of the "Customers" table):

Example

```
CREATE TABLE TestTable AS
SELECT customername, contactname
FROM customers;
```

Test Yourself With Exercises

Exercise:

Write the correct SQL statement to create a new table called `Persons`.

```
_____ (
  PersonID int,
  LastName varchar(255),
  FirstName varchar(255),
  Address varchar(255),
  City varchar(255)
);
```

Submit Answer »

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

SQL DROP TABLE Statement

The 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;
```

Test Yourself With Exercises

Exercise:

Write the correct SQL statement to delete a table called `Persons`.

_____ Persons;

Submit Answer »

```
DROP TABLE Persons;
```

SQL ALTER TABLE Statement

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;
```

The following SQL adds an "Email" column to the "Customers" table:

Example

```
ALTER TABLE Customers
ADD Email varchar(255);
```

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;
```

The following SQL deletes the "Email" column from the "Customers" table:

Example

```
ALTER TABLE Customers
DROP COLUMN Email;
```

ALTER TABLE - RENAME COLUMN

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

```
ALTER TABLE table_name
RENAME COLUMN old_name to new_name;
```

To rename a column in a table in SQL Server, use the following syntax:

SQL Server:

```
EXEC sp_rename 'table_name.old_name',  'new_name', 'COLUMN';
```

ALTER TABLE - ALTER/MODIFY DATATYPE

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

Test Yourself With Exercises

Exercise:

Add a column of type `DATE` called `Birthday`.

_____ Persons

_____;

Submit Answer »

```
ALTER TABLE Persons
ADD Birthday DATE;
```

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` - Prevents actions that would destroy links between tables
- `CHECK` - Ensures that the values in a column satisfies a specific condition
- `DEFAULT` - Sets a default value for a column if no value is specified
- `CREATE INDEX` - Used 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.

SQL NOT NULL on CREATE TABLE

The following SQL ensures that the "ID", "LastName", and "FirstName" columns will NOT accept NULL values when the "Persons" table is created:

Example

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

SQL NOT NULL on ALTER TABLE

To create a `NOT NULL` constraint on the "Age" column when the "Persons" table is already created, use the following SQL:

SQL Server / MS Access:

```
ALTER TABLE Persons
ALTER COLUMN Age int NOT NULL;
```

My SQL / Oracle (prior version 10G):

```
ALTER TABLE Persons
MODIFY COLUMN Age int NOT NULL;
```

Oracle 10G and later:

```
ALTER TABLE Persons
MODIFY Age int NOT NULL;
```

SQL UNIQUE Constraint

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

SQL PRIMARY KEY Constraint

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

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

A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (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 ALTER TABLE to add a primary key, the primary key column(s) must 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

SQL FOREIGN KEY Constraint

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

A `FOREIGN KEY` is a field (or collection of fields) in one table, that refers to the `PRIMARY KEY` in another table.

The table with the foreign key is called the child table, and the table with the primary key is called the referenced or parent 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 prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the parent table.

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

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 column it will allow 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 the age of a person must be 18, or older:

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

SQL DEFAULT Constraint

The `DEFAULT` constraint is used to set 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:

```
ALTER TABLE Persons
ADD CONSTRAINT df_City
DEFAULT 'Sandnes' FOR City;
```

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 Server:

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

SQL CREATE INDEX Statement

SQL CREATE INDEX Statement

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

Indexes are used to retrieve data from the database more quickly than otherwise. 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 "Personid" column to be an auto-increment primary key field in the "Persons" table:

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

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 "Personid" 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 "Personid" 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 "Personid" column to be an auto-increment primary key field in the "Persons" table:

```
CREATE TABLE Persons (
    Personid 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 "Personid" 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 "Personid" 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 "Personid" 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 "Personid" column to be an auto-increment primary key field in the "Persons" table:

```
CREATE TABLE Persons (
    Personid AUTOINCREMENT PRIMARY KEY,
    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 "Personid" 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 "Personid" 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 "Personid" 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 (Personid,FirstName,LastName)
VALUES (seq_person.nextval,'Lars','Monsen');
```

The SQL statement above would insert a new record into the "Persons" table. The "Personid" 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 Working With Dates

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!

SQL Working with Dates

Look at the following table:

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

Note: Two dates can easily be compared if there is no time component involved!

Now, assume that the "Orders" table looks like this (notice the added 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: To keep your queries simple and easy to maintain, do not use time-components in your dates, unless you have to!

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 statements and functions to a view and present the data as if the data were coming from one single table.

A view is created with the `CREATE VIEW` statement.

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 view, every time a user queries it.

SQL CREATE VIEW Examples

The following SQL creates a view that shows all customers from Brazil:

Example

```
CREATE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName
FROM Customers
WHERE Country = 'Brazil';
```

We can query the view above as follows:

Example

```
SELECT * FROM [Brazil Customers];
```

The following SQL creates a view that selects every product in the "Products" table with a price higher than the average price:

Example

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

We can query the view above as follows:

Example

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

SQL Updating a View

A view can be updated with the `CREATE OR REPLACE VIEW` statement.

SQL CREATE OR REPLACE VIEW Syntax

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

The following SQL adds the "City" column to the "Brazil Customers" view:

Example

```
CREATE OR REPLACE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName, City
FROM Customers
WHERE Country = 'Brazil';
```

SQL Dropping a View

A view is deleted with the `DROP VIEW` statement.

SQL DROP VIEW Syntax

```
DROP VIEW view_name;
```

The following SQL drops the "Brazil Customers" view:

Example

```
DROP VIEW [Brazil Customers];
```

SQL Injection

SQL Injection

SQL injection is a code injection technique that might destroy your database.

SQL injection is one of the most common web hacking techniques.

SQL injection is the placement of malicious code in SQL statements, via web page input.

SQL in Web Pages

SQL injection usually occurs when you ask a user for input, like their username/userid, and instead of a name/id, the user gives you an SQL statement that you will **unknowingly** run on your database.

Look at the following example which creates a `SELECT` statement by adding a variable (txtUserId) to a select string. The variable is fetched from user input (getRequestString):

Example

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

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

SQL Injection Based on 1=1 is Always True

Look at the example above again. 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=

Then, the SQL statement will look like this:

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

The SQL above is valid and will return ALL rows from the "Users" table, since **OR 1=1** is always TRUE.

Does the example above look 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 hacker might get access to all the user names and passwords in a database, by simply inserting 105 OR 1=1 into the input field.

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

Here is an example of a user login on a web site:

Username:
John Doe

Password:
myPass

Example

```
uName = getRequestString("username");
uPass = getRequestString("userpassword");

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

Result

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

A 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:

Password:

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

Result

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

The SQL above is valid and will return all rows from the "Users" table, since **OR ""=""** is always TRUE.

SQL Injection Based on Batched SQL Statements

Most databases support batched SQL statement.

A batch of SQL statements is a group of two or more SQL statements, separated by semicolons.

The SQL statement below will return all rows from the "Users" table, then delete the "Suppliers" table.

Example

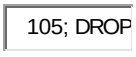
```
SELECT * FROM Users; DROP TABLE Suppliers
```

Look at the following example:

Example

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

And the following input:

User id: 

The valid SQL statement would look like this:

Result

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

Use SQL Parameters for Protection

To protect a web site from SQL injection, you can 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);
```

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

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.

MS Access

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

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

SQL Data Types for MySQL, SQL Server, and MS Access

The data type of a column defines what value the column can hold: integer, character, money, date and time, binary, and so on.

SQL Data Types

Each column in a database table is required to have a name and a data type.

An SQL developer must decide what type of data that will be stored inside each column when creating a table. The data type is 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.

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!**

MySQL Data Types (Version 8.0)

In MySQL there are three main data types: string, numeric, and date and time.

String Data Types

Data type	Description
CHAR(size)	A FIXED length string (can contain letters, numbers, and special characters). The *size* parameter specifies the column length in characters - can be from 0 to 255. Default is 1
VARCHAR(size)	A VARIABLE length string (can contain letters, numbers, and special characters). The *size* parameter specifies the maximum string length in characters - can be from 0 to 65535
BINARY(size)	Equal to CHAR(), but stores binary byte strings. The *size* parameter specifies the column length in bytes. Default is 1
VARBINARY(size)	Equal to VARCHAR(), but stores binary byte strings. The *size* parameter specifies the maximum column length in bytes.
TINYBLOB	For BLOBs (Binary Large Objects). Max length: 255 bytes
TINYTEXT	Holds a string with a maximum length of 255 characters
TEXT(size)	Holds a string with a maximum length of 65,535 bytes

BLOB(size)	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(val1, val2, val3, ...)	A string object that can have only one value, chosen from 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. The values are sorted in the order you enter them
SET(val1, val2, val3, ...)	A string object that can have 0 or more values, chosen from a list of possible values. You can list up to 64 values in a SET list

Numeric Data Types

Data type	**Description**
BIT(*size*)	A bit-value type. The number of bits per value is specified in *size*. The *size* parameter can hold a value from 1 to 64. The default value for *size* is 1.

TINYINT(*size*)	A very small integer. Signed range is from -128 to 127. Unsigned range is from 0 to 255. The *size* parameter specifies the maximum display width (which is 255)
BOOL	Zero is considered as false, nonzero values are considered as true.
BOOLEAN	Equal to BOOL
SMALLINT(*size*)	A small integer. Signed range is from -32768 to 32767. Unsigned range is from 0 to 65535. The *size* parameter specifies the maximum display width (which is 255)
MEDIUMINT(*size*)	A medium integer. Signed range is from -8388608 to 8388607. Unsigned range is from 0 to 16777215. The *size* parameter specifies the maximum display width (which is 255)
INT(*size*)	A medium integer. Signed range is from -2147483648 to 2147483647. Unsigned range is from 0 to 4294967295. The *size* parameter specifies the maximum display width (which is 255)
INTEGER(*size*)	Equal to INT(size)
BIGINT(*size*)	A large integer. Signed range is from -9223372036854775808 to 9223372036854775807. Unsigned range is from 0 to 18446744073709551615. The *size* parameter specifies the maximum display width (which is 255)
FLOAT(*size*, *d*)	A floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. This syntax is deprecated in MySQL 8.0.17, and it will be removed in future MySQL versions

FLOAT(*p*)	A floating point number. MySQL uses the *p* value to determine whether to use FLOAT or DOUBLE for the resulting data type. If *p* is from 0 to 24, the data type becomes FLOAT(). If *p* is from 25 to 53, the data type becomes DOUBLE()
DOUBLE(*size*, *d*)	A normal-size floating point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter
DOUBLE PRECISION(*size*, *d*)	
DECIMAL(*size*, *d*)	An exact fixed-point number. The total number of digits is specified in *size*. The number of digits after the decimal point is specified in the *d* parameter. The maximum number for *size* is 65. The maximum number for *d* is 30. The default value for *size* is 10. The default value for *d* is 0.
DEC(*size*, *d*)	Equal to DECIMAL(size,d)

Note: All the numeric data types may have an extra option: UNSIGNED or ZEROFILL. If you add the UNSIGNED option, MySQL disallows negative values for the column. If you add the ZEROFILL option, MySQL automatically also adds the UNSIGNED attribute to the column.

Date and Time Data Types

Data type	Description
DATE	A date. Format: YYYY-MM-DD. The supported range is from '1000-01-01' to '9999-12-31'
DATETIME(*fsp*)	A date and time combination. Format: YYYY-MM-DD hh:mm:ss. The supported range is from '1000-01-01

	00:00:00' to '9999-12-31 23:59:59'. Adding DEFAULT and ON UPDATE in the column definition to get automatic initialization and updating to the current date and time
TIMESTAMP(*fsp*)	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. The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC. Automatic initialization and updating to the current date and time can be specified using DEFAULT CURRENT_TIMESTAMP and ON UPDATE CURRENT_TIMESTAMP in the column definition
TIME(*fsp*)	A time. Format: hh:mm:ss. The supported range is from '-838:59:59' to '838:59:59'
YEAR	A year in four-digit format. Values allowed in four-digit format: 1901 to 2155, and 0000. MySQL 8.0 does not support year in two-digit format.

SQL Server Data Types

String Data Types

Data type	Description	Max size	Storage
char(n)	Fixed width character string	8,000 characters	Defined width

varchar(n)	Variable width character string	8,000 characters	2 bytes + number of chars
varchar(max)	Variable width character string	1,073,741,824 characters	2 bytes + number of chars
text	Variable width character string	2GB of text data	4 bytes + number of chars
nchar	Fixed width Unicode string	4,000 characters	Defined width x 2
nvarchar	Variable width Unicode string	4,000 characters	
nvarchar(max)	Variable width Unicode string	536,870,912 characters	
ntext	Variable width Unicode string	2GB of text data	
binary(n)	Fixed width binary string	8,000 bytes	
varbinary	Variable width binary string	8,000 bytes	
varbinary(max)	Variable width binary string	2GB	

image	Variable width binary string	2GB

Numeric Data Types

Data type	Description	Storage
bit	Integer that can be 0, 1, or NULL	
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.	5-17 bytes

	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	
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 and Time Data 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

MS 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

SQL Keywords Reference

This SQL keywords reference contains the reserved words in SQL.

SQL Keywords

Keyword	Description
ADD	Adds a column in an existing table
ADD CONSTRAINT	Adds a constraint after a table is already created

ALL	Returns true if all of the subquery values meet the condition
ALTER	Adds, deletes, or modifies columns in a table, or changes the data type of a column in a table
ALTER COLUMN	Changes the data type of a column in a table
ALTER TABLE	Adds, deletes, or modifies columns in a table
AND	Only includes rows where both conditions is true
ANY	Returns true if any of the subquery values meet the condition
AS	Renames a column or table with an alias
ASC	Sorts the result set in ascending order
BACKUP DATABASE	Creates a back up of an existing database
BETWEEN	Selects values within a given range
CASE	Creates different outputs based on conditions

CHECK	A constraint that limits the value that can be placed in a column
COLUMN	Changes the data type of a column or deletes a column in a table
CONSTRAINT	Adds or deletes a constraint
CREATE	Creates a database, index, view, table, or procedure
CREATE DATABASE	Creates a new SQL database
CREATE INDEX	Creates an index on a table (allows duplicate values)
CREATE OR REPLACE VIEW	Updates a view
CREATE TABLE	Creates a new table in the database
CREATE PROCEDURE	Creates a stored procedure
CREATE UNIQUE INDEX	Creates a unique index on a table (no duplicate values)
CREATE VIEW	Creates a view based on the result set of a SELECT statement

DATABASE	Creates or deletes an SQL database
DEFAULT	A constraint that provides a default value for a column
DELETE	Deletes rows from a table
DESC	Sorts the result set in descending order
DISTINCT	Selects only distinct (different) values
DROP	Deletes a column, constraint, database, index, table, or view
DROP COLUMN	Deletes a column in a table
DROP CONSTRAINT	Deletes a UNIQUE, PRIMARY KEY, FOREIGN KEY, or CHECK constraint
DROP DATABASE	Deletes an existing SQL database
DROP DEFAULT	Deletes a DEFAULT constraint
DROP INDEX	Deletes an index in a table
DROP TABLE	Deletes an existing table in the database

DROP VIEW	Deletes a view
EXEC	Executes a stored procedure
EXISTS	Tests for the existence of any record in a subquery
FOREIGN KEY	A constraint that is a key used to link two tables together
FROM	Specifies which table to select or delete data from
FULL OUTER JOIN	Returns all rows when there is a match in either left table or right table
GROUP BY	Groups the result set (used with aggregate functions: COUNT, MAX, MIN, SUM, AVG)
HAVING	Used instead of WHERE with aggregate functions
IN	Allows you to specify multiple values in a WHERE clause
INDEX	Creates or deletes an index in a table
INNER JOIN	Returns rows that have matching values in both tables

INSERT INTO	Inserts new rows in a table
INSERT INTO SELECT	Copies data from one table into another table
IS NULL	Tests for empty values
IS NOT NULL	Tests for non-empty values
JOIN	Joins tables
LEFT JOIN	Returns all rows from the left table, and the matching rows from the right table
LIKE	Searches for a specified pattern in a column
LIMIT	Specifies the number of records to return in the result set
NOT	Only includes rows where a condition is not true
NOT NULL	A constraint that enforces a column to not accept NULL values
OR	Includes rows where either condition is true

ORDER BY	Sorts the result set in ascending or descending order
OUTER JOIN	Returns all rows when there is a match in either left table or right table
PRIMARY KEY	A constraint that uniquely identifies each record in a database table
PROCEDURE	A stored procedure
RIGHT JOIN	Returns all rows from the right table, and the matching rows from the left table
ROWNUM	Specifies the number of records to return in the result set
SELECT	Selects data from a database
SELECT DISTINCT	Selects only distinct (different) values
SELECT INTO	Copies data from one table into a new table
SELECT TOP	Specifies the number of records to return in the result set
SET	Specifies which columns and values that should be updated in a table

TABLE	Creates a table, or adds, deletes, or modifies columns in a table, or deletes a table or data inside a table
TOP	Specifies the number of records to return in the result set
TRUNCATE TABLE	Deletes the data inside a table, but not the table itself
UNION	Combines the result set of two or more SELECT statements (only distinct values)
UNION ALL	Combines the result set of two or more SELECT statements (allows duplicate values)
UNIQUE	A constraint that ensures that all values in a column are unique
UPDATE	Updates existing rows in a table
VALUES	Specifies the values of an INSERT INTO statement
VIEW	Creates, updates, or deletes a view
WHERE	Filters a result set to include only records that fulfill a specified condition

SQL ADD Keyword

ADD

The ADD command is used to add a column in an existing table.

Example

Add an "Email" column to the "Customers" table:

```
ALTER TABLE Customers
ADD Email varchar(255);
```

SQL ADD CONSTRAINT Keyword

ADD CONSTRAINT

The ADD CONSTRAINT command is used to create a constraint after a table is already created.

The following SQL adds a constraint named "PK_Person" that is a PRIMARY KEY constraint on multiple columns (ID and LastName):

Example

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

SQL ALL Keyword

ALL

The ALL command returns true if all of the subquery values meet the condition.

The following SQL statement returns TRUE and lists the productnames if ALL the records in the OrderDetails table has quantity = 10:

Example

```
SELECT ProductName
FROM Products
WHERE ProductID = ALL (SELECT ProductID FROM OrderDetails WHERE Quantity
= 10);
```

SQL ALTER Keyword

ALTER TABLE

The `ALTER TABLE` command adds, deletes, or modifies columns in a table.

The `ALTER TABLE` command also adds and deletes various constraints in a table.

The following SQL adds an "Email" column to the "Customers" table:

Example

```
ALTER TABLE Customers
ADD Email varchar(255);
```

The following SQL deletes the "Email" column from the "Customers" table:

Example

```
ALTER TABLE Customers
DROP COLUMN Email;
```

ALTER COLUMN

The `ALTER COLUMN` command is used to change the data type of a column in a table.

The following SQL changes the data type of the column named "BirthDate" in the "Employees" table to type year:

Example

```
ALTER TABLE Employees
ALTER COLUMN BirthDate year;
```

SQL ALTER COLUMN Keyword

ALTER COLUMN

The `ALTER COLUMN` command is used to change the data type of a column in a table.

The following SQL changes the data type of the column named "BirthDate" in the "Employees" table to type year:

Example

```
ALTER TABLE Employees
ALTER COLUMN BirthDate year;
```

SQL ALTER TABLE Keyword

ALTER TABLE

The `ALTER TABLE` command adds, deletes, or modifies columns in a table.

The `ALTER TABLE` command also adds and deletes various constraints in a table.

The following SQL adds an "Email" column to the "Customers" table:

Example

```
ALTER TABLE Customers
ADD Email varchar(255);
```

The following SQL deletes the "Email" column from the "Customers" table:

Example

```
ALTER TABLE Customers
DROP COLUMN Email;
```

SQL AND Keyword

AND

The AND command is used with WHERE to only include rows where both conditions is true.

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';
```

SQL ANY Keyword

ANY

The ANY command returns true if any of the subquery values meet the condition.

The following SQL statement returns TRUE and lists the productnames if it finds ANY records in the OrderDetails table where quantity = 10:

Example

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity
= 10);
```

The following SQL statement returns TRUE and lists the productnames if it finds ANY records in the OrderDetails table where quantity > 99:

Example

```
SELECT ProductName
FROM Products
WHERE ProductID = ANY (SELECT ProductID FROM OrderDetails WHERE Quantity
> 99);
```

SQL AS Keyword

AS

The AS command is used to rename a column or table with an alias.

An alias only exists for the duration of the query.

Alias for Columns

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. Notice that 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

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;
```

SQL ASC Keyword

ASC

The ASC command is used to sort the data returned in ascending order.

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

Example

```
SELECT * FROM Customers
ORDER BY CustomerName ASC;
```

SQL BACKUP DATABASE Keyword

BACKUP DATABASE

The `BACKUP DATABASE` command is used in SQL Server to create a full back up of an existing SQL database.

The following SQL statement creates a full back up of the existing database "testDB" to the D disk:

Example

```
BACKUP DATABASE testDB
TO DISK = 'D:\backups\testDB.bak';
```

Tip: Always back up the database to a different drive than the actual database. If you get a disk crash, you will not lose your backup file along with the database.

A differential back up only backs up the parts of the database that have changed since the last full database backup.

The following SQL statement creates a differential back up of the database "testDB":

Example

```
BACKUP DATABASE testDB
TO DISK = 'D:\backups\testDB.bak'
WITH DIFFERENTIAL;
```

Tip: A differential back up reduces the back up time (since only the changes are backed up).

SQL BETWEEN Keyword

BETWEEN

The BETWEEN command is used to select values within a given range. The values can be numbers, text, or dates.

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

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

Example

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

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;
```

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;
```

SQL CASE Keyword

CASE

The CASE command is used is to create different output based on conditions.

The following SQL goes through several conditions and returns a value when the specified condition is met:

Example

```
SELECT OrderID, Quantity,
CASE
    WHEN Quantity > 30 THEN 'The quantity is greater than 30'
    WHEN Quantity = 30 THEN 'The quantity is 30'
    ELSE 'The quantity is under 30'
END
FROM OrderDetails;
```

The following SQL will order the customers by City. However, if City is NULL, then order by Country:

Example

```
SELECT CustomerName, City, Country
FROM Customers
ORDER BY
(CASE
    WHEN City IS NULL THEN Country
    ELSE City
END);
```

SQL CHECK Keyword

CHECK

The CHECK constraint limits the value that can be placed in a column.

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 (
    Age int,
    CHECK (Age>=18)
);
```

SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons (
    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 (
    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 COLUMN Keyword

ALTER COLUMN

The `ALTER COLUMN` command is used to change the data type of a column in a table.

The following SQL changes the data type of the column named "BirthDate" in the "Employees" table to type year:

Example

```
ALTER TABLE Employees
ALTER COLUMN BirthDate year;
```

DROP COLUMN

The `DROP COLUMN` command is used to delete a column in an existing table.

The following SQL deletes the "ContactName" column from the "Customers" table:

Example

```
ALTER TABLE Customers
DROP COLUMN ContactName;
```

SQL CONSTRAINT Keyword

ADD CONSTRAINT

The ADD CONSTRAINT command is used to create a constraint after a table is already created.

The following SQL adds a constraint named "PK_Person" that is a PRIMARY KEY constraint on multiple columns (ID and LastName):

Example

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

DROP CONSTRAINT

The DROP CONSTRAINT command is used to delete a UNIQUE, PRIMARY KEY, FOREIGN KEY, or CHECK constraint.

DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL:

SQL Server / Oracle / MS Access:

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

MySQL:

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

DROP a PRIMARY KEY Constraint

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

SQL Server / Oracle / MS Access:

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

MySQL:

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

DROP a FOREIGN KEY Constraint

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

SQL Server / Oracle / MS Access:

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

MySQL:

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

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 CREATE Keyword

CREATE DATABASE

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

The following SQL 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;

CREATE TABLE

The `CREATE TABLE` command creates a new table in the database.

The following SQL 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)
);
```

CREATE TABLE Using Another Table

The following SQL creates a new table called "TestTables" (which is a copy of two columns of the "Customers" table):

Example

```
CREATE TABLE TestTable AS
SELECT customername, contactname
FROM customers;
```

CREATE INDEX

The `CREATE INDEX` command is used to create indexes in tables (allows duplicate values).

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.

The following SQL 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);
```

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

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 UNIQUE INDEX

The `CREATE UNIQUE INDEX` command creates a unique index on a table (no duplicate values allowed)

The following SQL creates an index named "uidx_pid" on the "PersonID" column in the "Persons" table:

```
CREATE UNIQUE INDEX uidx_pid
ON Persons (PersonID);
```

CREATE VIEW

The `CREATE VIEW` command creates a view.

A view is a virtual table based on the result set of an SQL statement.

The following SQL creates a view that selects all customers from Brazil:

Example

```
CREATE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName
FROM Customers
WHERE Country = "Brazil";
```

CREATE OR REPLACE VIEW

The `CREATE OR REPLACE VIEW` command updates a view.

The following SQL adds the "City" column to the "Brazil Customers" view:

Example

```
CREATE OR REPLACE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName, City
FROM Customers
WHERE Country = "Brazil";
```

Query The View

We can query the view above as follows:

Example

```
SELECT * FROM [Brazil Customers];
```

CREATE PROCEDURE

The `CREATE PROCEDURE` command is used to create a stored procedure.

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

The following SQL creates a stored procedure named "SelectAllCustomers" that selects all records from the "Customers" table:

Example

```
CREATE PROCEDURE SelectAllCustomers
AS
SELECT * FROM Customers
GO;
```

Execute the stored procedure above as follows:

Example

```
EXEC SelectAllCustomers;
```

SQL CREATE DATABASE Keyword

CREATE DATABASE

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

The following SQL 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 CREATE INDEX Keyword

CREATE INDEX

The `CREATE INDEX` command is used to create indexes in tables (allows duplicate values).

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.

The following SQL 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);
```

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

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.

SQL CREATE OR REPLACE VIEW Keyword

CREATE OR REPLACE VIEW

The `CREATE OR REPLACE VIEW` command updates a view.

The following SQL adds the "City" column to the "Brazil Customers" view:

Example

```
CREATE OR REPLACE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName, City
FROM Customers
WHERE Country = "Brazil";
```

Query The View

We can query the view above as follows:

Example

```
SELECT * FROM [Brazil Customers];
```

SQL CREATE TABLE Keyword

CREATE TABLE

The `CREATE TABLE` command creates a new table in the database.

The following SQL 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)
);
```

CREATE TABLE Using Another Table

The following SQL creates a new table called "TestTables" (which is a copy of two columns of the "Customers" table):

Example

```
CREATE TABLE TestTable AS
SELECT customername, contactname
FROM customers;
```

SQL CREATE PROCEDURE Keyword

CREATE PROCEDURE

The `CREATE PROCEDURE` command is used to create a stored procedure.

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

The following SQL creates a stored procedure named "SelectAllCustomers" that selects all records from the "Customers" table:

Example

```
CREATE PROCEDURE SelectAllCustomers
AS
SELECT * FROM Customers
GO;
```

Execute the stored procedure above as follows:

Example

```
EXEC SelectAllCustomers;
```

SQL CREATE UNIQUE INDEX Keyword

CREATE UNIQUE INDEX

The `CREATE UNIQUE INDEX` command creates a unique index on a table (no duplicate values allowed)

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.

The following SQL creates an index named "uidx_pid" on the "PersonID" column in the "Persons" table:

```
CREATE UNIQUE INDEX uidx_pid
ON Persons (PersonID);
```

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

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.

SQL CREATE VIEW Keyword

CREATE VIEW

The `CREATE VIEW` command creates a view.

A view is a virtual table based on the result set of an SQL statement.

The following SQL creates a view that selects all customers from Brazil:

Example

```
CREATE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName
FROM Customers
WHERE Country = "Brazil";
```

Query The View

We can query the view above as follows:

Example

```
SELECT * FROM [Brazil Customers];
```

SQL DATABASE Keyword

CREATE DATABASE

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

The following SQL 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;

DROP DATABASE

The `DROP DATABASE` command is used is to delete an existing SQL database.

The following SQL drops a database named "testDB":

Example

```
DROP DATABASE testDB;
```

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

SQL DEFAULT Keyword

DEFAULT

The DEFAULT constraint provides 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 (
    City varchar(255) DEFAULT 'Sandnes'
);
```

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

```
CREATE TABLE Orders (
    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:

```
ALTER TABLE Persons
ADD CONSTRAINT df_City
DEFAULT 'Sandnes' FOR City;
```

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 DELETE Keyword

DELETE

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

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

Example

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

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

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:

The following SQL statement deletes all rows in the "Customers" table, without deleting the table. This means that the table structure, attributes, and indexes will be intact:

Example

```
DELETE FROM Customers;
```

SQL DESC Keyword

DESC

The `DESC` command is used to sort the data returned in descending order.

The following SQL statement selects all the columns from the "Customers" table, sorted descending by the "CustomerName" column:

Example

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

SQL SELECT DISTINCT Keyword

SELECT DISTINCT

The `SELECT DISTINCT` command returns only distinct (different) values in the result set.

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

Example

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

SQL DROP Keyword

DROP COLUMN

The `DROP COLUMN` command is used to delete a column in an existing table.

The following SQL deletes the "ContactName" column from the "Customers" table:

Example

```
ALTER TABLE Customers
DROP COLUMN ContactName;
```

DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL:

SQL Server / Oracle / MS Access:

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

MySQL:

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

DROP a PRIMARY KEY Constraint

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

SQL Server / Oracle / MS Access:

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

MySQL:

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

DROP a FOREIGN KEY Constraint

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

SQL Server / Oracle / MS Access:

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

MySQL:

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

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;
```

DROP DEFAULT

The `DROP DEFAULT` command is used to delete a DEFAULT constraint.

To drop a DEFAULT constraint, use the following SQL:

SQL Server / Oracle / MS Access:

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

MySQL:

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

DROP INDEX

The DROP INDEX command 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;
```

DROP DATABASE

The DROP DATABASE command is used is to delete an existing SQL database.

The following SQL drops a database named "testDB":

Example

```
DROP DATABASE testDB;
```

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

DROP TABLE

The DROP TABLE command deletes a table in the database.

The following SQL deletes the table "Shippers":

Example

```
DROP TABLE Shippers;
```

Note: Be careful before deleting a table. Deleting a table results in loss of all information stored in the table!

DROP VIEW

The DROP VIEW command deletes a view.

The following SQL drops the "Brazil Customers" view:

Example

```
DROP VIEW [Brazil Customers];
```

SQL DROP COLUMN Keyword

DROP COLUMN

The DROP COLUMN command is used to delete a column in an existing table.

The following SQL deletes the "ContactName" column from the "Customers" table:

Example

```
ALTER TABLE Customers
DROP COLUMN ContactName;
```

SQL DROP CONSTRAINT Keyword

DROP CONSTRAINT

The `DROP CONSTRAINT` command is used to delete a UNIQUE, PRIMARY KEY, FOREIGN KEY, or CHECK constraint.

DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL:

SQL Server / Oracle / MS Access:

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

MySQL:

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

DROP a PRIMARY KEY Constraint

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

SQL Server / Oracle / MS Access:

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

MySQL:

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

DROP a FOREIGN KEY Constraint

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

SQL Server / Oracle / MS Access:

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

MySQL:

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

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 DROP DATABASE Keyword

DROP DATABASE

The `DROP DATABASE` command is used to delete an existing SQL database.

The following SQL drops a database named "testDB":

Example

```
DROP DATABASE testDB;
```

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

SQL DROP DEFAULT Keyword

DROP DEFAULT

The DROP DEFAULT command is used to delete a DEFAULT constraint.

To drop a DEFAULT constraint, use the following SQL:

SQL Server / Oracle / MS Access:

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

MySQL:

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

SQL DROP INDEX Keyword

DROP INDEX

The DROP INDEX command 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 DROP TABLE and TRUNCATE TABLE Keywords

DROP TABLE

The `DROP TABLE` command deletes a table in the database.

The following SQL deletes the table "Shippers":

Example

```
DROP TABLE Shippers;
```

Note: Be careful before deleting a table. Deleting a table results in loss of all information stored in the table!

TRUNCATE TABLE

The `TRUNCATE TABLE` command deletes the data inside a table, but not the table itself.

The following SQL truncates the table "Categories":

Example

```
TRUNCATE TABLE Categories;
```

SQL DROP VIEW Keyword

DROP VIEW

The DROP VIEW command deletes a view.

The following SQL drops the "Brazil Customers" view:

Example

```
DROP VIEW [Brazil Customers];
```

SQL EXEC Keyword

EXEC

The EXEC command is used to execute a stored procedure.

The following SQL executes a stored procedure named "SelectAllCustomers":

Example

```
EXEC SelectAllCustomers;
```

SQL EXISTS Keyword

EXISTS

The EXISTS command tests for the existence of any record in a subquery, and returns true if the subquery returns one or more records.

The following SQL lists the suppliers with a product price less than 20:

Example

```
SELECT SupplierName
FROM Suppliers
WHERE EXISTS (SELECT ProductName FROM Products WHERE SupplierId =
Suppliers.supplierId AND Price < 20);
```

The following SQL lists the suppliers with a product price equal to 22:

Example

```
SELECT SupplierName
FROM Suppliers
WHERE EXISTS (SELECT ProductName FROM Products WHERE SupplierId =
Suppliers.supplierId AND Price = 22);
```

SQL FOREIGN KEY Keyword

FOREIGN KEY

The `FOREIGN KEY` constraint is a key used to link two tables together.

A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table.

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 FROM Keyword

FROM

The FROM command is used to specify which table to select or delete data from.

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

Example

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

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

Example

```
SELECT * FROM Customers;
```

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

Example

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

SQL FULL OUTER JOIN Keyword

FULL OUTER JOIN

The `FULL OUTER JOIN` command returns all rows when there is a match in either left table or right table.

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;
```

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 GROUP BY Keyword

GROUP BY

The `GROUP BY` command is used to group the result set (used with aggregate functions: COUNT, MAX, MIN, SUM, AVG).

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

Example

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

The following SQL 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;
```

SQL HAVING Keyword

HAVING

The HAVING command is used instead of WHERE with aggregate functions.

The following SQL 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 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;
```

SQL IN Keyword

IN

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

The IN operator is a shorthand for multiple OR conditions.

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

Example

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

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

Example

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

SQL INDEX Keyword

CREATE INDEX

The `CREATE INDEX` command is used to create indexes in tables (allows duplicate values).

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.

The following SQL 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);
```

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

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.

DROP INDEX

The DROP INDEX command 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 INNER JOIN Keyword

INNER JOIN

The INNER JOIN command returns rows that have matching values in both tables.

The following SQL selects all orders with customer information:

Example

```
SELECT 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 be shown!

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 INSERT INTO Keyword

INSERT INTO

The `INSERT INTO` command is used to insert new rows in a table.

The following SQL 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 following SQL 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');
```

SQL INSERT INTO SELECT Keyword

INSERT INTO SELECT

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

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

Example

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

SQL IS NULL Keyword

IS NULL

The `IS NULL` command is used to test for empty values (NULL values).

The following SQL lists all customers with a NULL value in the "Address" field:

Example

```
SELECT CustomerName, ContactName, Address
FROM Customers
WHERE Address IS NULL;
```

Note: A NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation!

Tip: Always use IS NULL to look for NULL values.

SQL IS NOT NULL Keyword

IS NOT NULL

The `IS NOT NULL` command is used to test for non-empty values (NOT NULL values).

The following SQL lists all customers with a value in the "Address" field:

Example

```
SELECT CustomerName, ContactName, Address
FROM Customers
WHERE Address IS NOT NULL;
```

SQL JOIN Keyword

INNER JOIN

The INNER JOIN command returns rows that have matching values in both tables.

The following SQL selects all orders with customer information:

Example

```
SELECT 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 be shown!

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);
```

LEFT JOIN

The LEFT JOIN command returns all rows from the left table, and the matching rows from the right table. The result is NULL from the right side, if there is no match.

The following SQL 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).

RIGHT JOIN

The `RIGHT JOIN` command returns all rows from the right table, and the matching records from the left table. The result is NULL from the left side, when there is no match.

The following SQL will return all employees, and any orders they might 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).

FULL OUTER JOIN

The `FULL OUTER JOIN` command returns all rows when there is a match in either left table or right table.

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;
```

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 LEFT JOIN Keyword

LEFT JOIN

The `LEFT JOIN` command returns all rows from the left table, and the matching rows from the right table. The result is NULL from the right side, if there is no match.

The following SQL 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 LIKE Keyword

LIKE

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

You can use two wildcards with `LIKE`:

- % - Represents zero, one, or multiple characters
- _ - Represents a single character (MS Access uses a question mark (?) instead)

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

Example

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

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

Example

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

The following SQL 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 starts with "a" and are at least 3 characters in length:

Example

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

SQL SELECT TOP, LIMIT and ROWNUM Keywords

SELECT TOP, LIMIT and ROWNUM

The `LIMIT`, `SELECT TOP` or `ROWNUM` command is used to specify the number of records to return.

Note: SQL Server uses `SELECT TOP`. MySQL uses `LIMIT`, and Oracle uses `ROWNUM`.

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

Example

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

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

Example

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

The following SQL statement shows the equivalent example using ROWNUM (Oracle):

Example

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

SQL SELECT TOP, LIMIT and ROWNUM Keywords

SELECT TOP, LIMIT and ROWNUM

The `LIMIT`, `SELECT TOP` or `ROWNUM` command is used to specify the number of records to return.

Note: SQL Server uses `SELECT TOP`. MySQL uses `LIMIT`, and Oracle uses `ROWNUM`.

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

Example

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

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

Example

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

The following SQL statement shows the equivalent example using ROWNUM (Oracle):

Example

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

SQL NOT NULL Keyword

NOT NULL

The NOT NULL constraint enforces a column to not accept NULL values, which means that you cannot insert 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
);
```

The following SQL creates a NOT NULL constraint on the "Age" column when the "Persons" table is already created:

```
ALTER TABLE Persons
MODIFY Age int NOT NULL;
```

SQL OR Keyword

OR

The OR command is used with WHERE to include rows where either condition is true.

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

Example

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

SQL ORDER BY Keyword

ORDER BY

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

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

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

Example

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

ASC

The ASC command is used to sort the data returned in ascending order.

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

Example

```
SELECT * FROM Customers
ORDER BY CustomerName ASC;
```

DESC

The DESC command is used to sort the data returned in descending order.

The following SQL statement selects all the columns from the "Customers" table, sorted descending by the "CustomerName" column:

Example

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

SQL FULL OUTER JOIN Keyword

FULL OUTER JOIN

The FULL OUTER JOIN command returns all rows when there is a match in either left table or right table.

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;
```

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 PRIMARY KEY Keyword

PRIMARY KEY

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

A table can have only one primary key, which may consist of one single or of 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 CREATE PROCEDURE Keyword

CREATE PROCEDURE

The `CREATE PROCEDURE` command is used to create a stored procedure.

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again.

The following SQL creates a stored procedure named "SelectAllCustomers" that selects all records from the "Customers" table:

Example

```
CREATE PROCEDURE SelectAllCustomers
AS
SELECT * FROM Customers
GO;
```

Execute the stored procedure above as follows:

Example

```
EXEC SelectAllCustomers;
```

SQL RIGHT JOIN Keyword

RIGHT JOIN

The `RIGHT JOIN` command returns all rows from the right table, and the matching records from the left table. The result is NULL from the left side, when there is no match.

The following SQL will return all employees, and any orders they might 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 SELECT TOP, LIMIT and ROWNUM Keywords

SELECT TOP, LIMIT and ROWNUM

The `LIMIT`, `SELECT TOP` or `ROWNUM` command is used to specify the number of records to return.

Note: SQL Server uses `SELECT TOP`. MySQL uses `LIMIT`, and Oracle uses `ROWNUM`.

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

Example

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

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

Example

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

The following SQL statement shows the equivalent example using ROWNUM (Oracle):

Example

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

SQL SELECT Keyword

SELECT

The SELECT command is used to select data from a database. The data returned is stored in a result table, called the result set.

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

Example

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

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

Example

```
SELECT * FROM Customers;
```

SQL SELECT DISTINCT Keyword

SELECT DISTINCT

The SELECT DISTINCT command returns only distinct (different) values in the result set.

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

Example

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

SQL SELECT INTO Keyword

SELECT INTO

The `SELECT INTO` command copies data from one table and inserts it into a new table.

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;
```

SQL SELECT TOP, LIMIT and ROWNUM Keywords

SELECT TOP, LIMIT and ROWNUM

The `LIMIT`, `SELECT TOP` or `ROWNUM` command is used to specify the number of records to return.

Note: SQL Server uses `SELECT TOP`. MySQL uses `LIMIT`, and Oracle uses `ROWNUM`.

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

Example

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

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

Example

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

The following SQL statement shows the equivalent example using ROWNUM (Oracle):

Example

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

SQL SET Keyword

SET

The SET command is used with UPDATE to specify which columns and values that should be updated in a table.

The following SQL updates the first customer (CustomerID = 1) with a new ContactName *and* a new City:

Example

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

The following SQL will update the "ContactName" field to "Juan" for all records where Country is "Mexico":

Example

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

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!

SQL TABLE Keyword

CREATE TABLE

The CREATE TABLE command creates a new table in the database.

The following SQL 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)
);
```

CREATE TABLE Using Another Table

A copy of an existing table can also be created using `CREATE TABLE`.

The following SQL creates a new table called "TestTables" (which is a copy of the "Customers" table):

Example

```
CREATE TABLE TestTable AS
SELECT customername, contactname
FROM customers;
```

ALTER TABLE

The `ALTER TABLE` command adds, deletes, or modifies columns in a table.

The `ALTER TABLE` command also adds and deletes various constraints in a table.

The following SQL adds an "Email" column to the "Customers" table:

Example

```
ALTER TABLE Customers
ADD Email varchar(255);
```

The following SQL deletes the "Email" column from the "Customers" table:

Example

```
ALTER TABLE Customers
DROP COLUMN Email;
```

DROP TABLE

The `DROP TABLE` command deletes a table in the database.

The following SQL deletes the table "Shippers":

Example

```
DROP TABLE Shippers;
```

Note: Be careful before deleting a table. Deleting a table results in loss of all information stored in the table!

TRUNCATE TABLE

The `TRUNCATE TABLE` command deletes the data inside a table, but not the table itself.

The following SQL truncates the table "Categories":

Example

```
TRUNCATE TABLE Categories;
```

SQL SELECT TOP, LIMIT and ROWNUM Keywords

SELECT TOP, LIMIT and ROWNUM

The `LIMIT`, `SELECT TOP` or `ROWNUM` command is used to specify the number of records to return.

Note: SQL Server uses `SELECT TOP`. MySQL uses `LIMIT`, and Oracle uses `ROWNUM`.

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

Example

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

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

Example

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

The following SQL statement shows the equivalent example using ROWNUM (Oracle):

Example

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

SQL DROP TABLE and TRUNCATE TABLE Keywords

DROP TABLE

The `DROP TABLE` command deletes a table in the database.

The following SQL deletes the table "Shippers":

Example

```
DROP TABLE Shippers;
```

Note: Be careful before deleting a table. Deleting a table results in loss of all information stored in the table!

TRUNCATE TABLE

The `TRUNCATE TABLE` command deletes the data inside a table, but not the table itself.

The following SQL truncates the table "Categories":

Example

```
TRUNCATE TABLE Categories;
```

SQL UNION Keyword

UNION

The UNION command combines the result set of two or more SELECT statements (only distinct values)

The following SQL statement returns the cities (only distinct values) from both the "Customers" and the "Suppliers" table:

Example

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

SQL UNION ALL Keyword

UNION ALL

The UNION ALL command combines the result set of two or more SELECT statements (allows duplicate values).

The following SQL statement returns the cities (duplicate values also) from both the "Customers" and the "Suppliers" table:

Example

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

SQL UNIQUE Keyword

UNIQUE

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

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 UPDATE Keyword

UPDATE

The UPDATE command is used to update existing rows in a 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 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';
```

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!

SQL VALUES Keyword

VALUES

The VALUES command specifies the values of an INSERT INTO statement.

The following SQL 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 following SQL 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');
```

SQL VIEW Keyword

CREATE VIEW

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

The `CREATE VIEW` command creates a view.

The following SQL creates a view that selects all customers from Brazil:

Example

```
CREATE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName
FROM Customers
WHERE Country = "Brazil";
```

Query The View

We can query the view above as follows:

Example

```
SELECT * FROM [Brazil Customers];
```

CREATE OR REPLACE VIEW

The `CREATE OR REPLACE VIEW` command updates a view.

The following SQL adds the "City" column to the "Brazil Customers" view:

Example

```
CREATE OR REPLACE VIEW [Brazil Customers] AS
SELECT CustomerName, ContactName, City
FROM Customers
WHERE Country = "Brazil";
```

DROP VIEW

The DROP VIEW command deletes a view.

The following SQL drops the "Brazil Customers" view:

Example

```
DROP VIEW [Brazil Customers];
```

SQL WHERE Keyword

SELECT

The WHERE command filters a result set to include only records that fulfill a specified condition.

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

Example

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

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;
```

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

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 a certain range
LIKE	Search for a pattern

IN	To specify multiple possible values for a column

MySQL Functions

MySQL has many built-in functions.

This reference contains string, numeric, date, and some advanced functions in MySQL.

MySQL String Functions

Function	Description
ASCII	Returns the ASCII value for the specific character
CHAR_LENGTH	Returns the length of a string (in characters)
CHARACTER_LENGTH	Returns the length of a string (in characters)
CONCAT	Adds two or more expressions together
CONCAT_WS	Adds two or more expressions together with a separator

FIELD	Returns the index position of a value in a list of values
FIND_IN_SET	Returns the position of a string within a list of strings
FORMAT	Formats a number to a format like "#,###,###.##", rounded to a specified number of decimal places
INSERT	Inserts a string within a string at the specified position and for a certain number of characters
INSTR	Returns the position of the first occurrence of a string in another string
LCASE	Converts a string to lower-case
LEFT	Extracts a number of characters from a string (starting from left)
LENGTH	Returns the length of a string (in bytes)
LOCATE	Returns the position of the first occurrence of a substring in a string
LOWER	Converts a string to lower-case

LPAD	Left-pads a string with another string, to a certain length
LTRIM	Removes leading spaces from a string
MID	Extracts a substring from a string (starting at any position)
POSITION	Returns the position of the first occurrence of a substring in a string
REPEAT	Repeats a string as many times as specified
REPLACE	Replaces all occurrences of a substring within a string, with a new substring
REVERSE	Reverses a string and returns the result
RIGHT	Extracts a number of characters from a string (starting from right)
RPAD	Right-pads a string with another string, to a certain length
RTRIM	Removes trailing spaces from a string
SPACE	Returns a string of the specified number of space characters

STRCMP	Compares two strings
SUBSTR	Extracts a substring from a string (starting at any position)
SUBSTRING	Extracts a substring from a string (starting at any position)
SUBSTRING_INDEX	Returns a substring of a string before a specified number of delimiter occurs
TRIM	Removes leading and trailing spaces from a string
UCASE	Converts a string to upper-case
UPPER	Converts a string to upper-case

MySQL Numeric Functions

Function	Description
ABS	Returns the absolute value of a number

ACOS	Returns the arc cosine of a number
ASIN	Returns the arc sine of a number
ATAN	Returns the arc tangent of one or two numbers
ATAN2	Returns the arc tangent of two numbers
AVG	Returns the average value of an expression
CEIL	Returns the smallest integer value that is >= to a number
CEILING	Returns the smallest integer value that is >= to a number
COS	Returns the cosine of a number
COT	Returns the cotangent of a number
COUNT	Returns the number of records returned by a select query
DEGREES	Converts a value in radians to degrees
DIV	Used for integer division

EXP	Returns e raised to the power of a specified number
FLOOR	Returns the largest integer value that is <= to a number
GREATEST	Returns the greatest value of the list of arguments
LEAST	Returns the smallest value of the list of arguments
LN	Returns the natural logarithm of a number
LOG	Returns the natural logarithm of a number, or the logarithm of a number to a specified base
LOG10	Returns the natural logarithm of a number to base 10
LOG2	Returns the natural logarithm of a number to base 2
MAX	Returns the maximum value in a set of values
MIN	Returns the minimum value in a set of values
MOD	Returns the remainder of a number divided by another number
PI	Returns the value of PI

POW	Returns the value of a number raised to the power of another number
POWER	Returns the value of a number raised to the power of another number
RADIANS	Converts a degree value into radians
RAND	Returns a random number
ROUND	Rounds a number to a specified number of decimal places
SIGN	Returns the sign of a number
SIN	Returns the sine of a number
SQRT	Returns the square root of a number
SUM	Calculates the sum of a set of values
TAN	Returns the tangent of a number
TRUNCATE	Truncates a number to the specified number of decimal places

MySQL Date Functions

Function	Description
ADDDATE	Adds a time/date interval to a date and then returns the date
ADDTIME	Adds a time interval to a time/datetime and then returns the time/datetime
CURDATE	Returns the current date
CURRENT_DATE	Returns the current date
CURRENT_TIME	Returns the current time
CURRENT_TIMESTAMP	Returns the current date and time
CURTIME	Returns the current time
DATE	Extracts the date part from a datetime expression
DATEDIFF	Returns the number of days between two date values
DATE_ADD	Adds a time/date interval to a date and then returns the date

DATE_FORMAT	Formats a date
DATE_SUB	Subtracts a time/date interval from a date and then returns the date
DAY	Returns the day of the month for a given date
DAYNAME	Returns the weekday name for a given date
DAYOFMONTH	Returns the day of the month for a given date
DAYOFWEEK	Returns the weekday index for a given date
DAYOFYEAR	Returns the day of the year for a given date
EXTRACT	Extracts a part from a given date
FROM_DAYS	Returns a date from a numeric datevalue
HOUR	Returns the hour part for a given date
LAST_DAY	Extracts the last day of the month for a given date
LOCALTIME	Returns the current date and time

LOCALTIMESTAMP	Returns the current date and time
MAKEDATE	Creates and returns a date based on a year and a number of days value
MAKETIME	Creates and returns a time based on an hour, minute, and second value
MICROSECOND	Returns the microsecond part of a time/datetime
MINUTE	Returns the minute part of a time/datetime
MONTH	Returns the month part for a given date
MONTHNAME	Returns the name of the month for a given date
NOW	Returns the current date and time
PERIOD_ADD	Adds a specified number of months to a period
PERIOD_DIFF	Returns the difference between two periods
QUARTER	Returns the quarter of the year for a given date value
SECOND	Returns the seconds part of a time/datetime

SEC_TO_TIME	Returns a time value based on the specified seconds
STR_TO_DATE	Returns a date based on a string and a format
SUBDATE	Subtracts a time/date interval from a date and then returns the date
SUBTIME	Subtracts a time interval from a datetime and then returns the time/datetime
SYSDATE	Returns the current date and time
TIME	Extracts the time part from a given time/datetime
TIME_FORMAT	Formats a time by a specified format
TIME_TO_SEC	Converts a time value into seconds
TIMEDIFF	Returns the difference between two time/datetime expressions
TIMESTAMP	Returns a datetime value based on a date or datetime value
TO_DAYS	Returns the number of days between a date and date "0000-00-00"

WEEK	Returns the week number for a given date
WEEKDAY	Returns the weekday number for a given date
WEEKOFYEAR	Returns the week number for a given date
YEAR	Returns the year part for a given date
YEARWEEK	Returns the year and week number for a given date

MySQL Advanced Functions

Function	Description
BIN	Returns a binary representation of a number
BINARY	Converts a value to a binary string
CASE	Goes through conditions and return a value when the first condition is met
CAST	Converts a value (of any type) into a specified datatype
COALESCE	Returns the first non-null value in a list

CONNECTION_ID	Returns the unique connection ID for the current connection
CONV	Converts a number from one numeric base system to another
CONVERT	Converts a value into the specified datatype or character set
CURRENT_USER	Returns the user name and host name for the MySQL account that the server used to authenticate the current client
DATABASE	Returns the name of the current database
IF	Returns a value if a condition is TRUE, or another value if a condition is FALSE
IFNULL	Return a specified value if the expression is NULL, otherwise return the expression
ISNULL	Returns 1 or 0 depending on whether an expression is NULL
LAST_INSERT_ID	Returns the AUTO_INCREMENT id of the last row that has been inserted in a table
NULLIF	Compares two expressions and returns NULL if they are equal. Otherwise, the first expression is returned

SESSION_USER	Returns the current MySQL user name and host name
SYSTEM_USER	Returns the current MySQL user name and host name
USER	Returns the current MySQL user name and host name
VERSION	Returns the current version of the MySQL database

SQL Server Functions

SQL Server has many built-in functions.

This reference contains string, numeric, date, conversion, and some advanced functions in SQL Server.

SQL Server String Functions

Function	Description
ASCII	Returns the ASCII value for the specific character
CHAR	Returns the character based on the ASCII code
CHARINDEX	Returns the position of a substring in a string

CONCAT	Adds two or more strings together
Concat with +	Adds two or more strings together
CONCAT_WS	Adds two or more strings together with a separator
DATALENGTH	Returns the number of bytes used to represent an expression
DIFFERENCE	Compares two SOUNDEX values, and returns an integer value
FORMAT	Formats a value with the specified format
LEFT	Extracts a number of characters from a string (starting from left)
LEN	Returns the length of a string
LOWER	Converts a string to lower-case
LTRIM	Removes leading spaces from a string
NCHAR	Returns the Unicode character based on the number code
PATINDEX	Returns the position of a pattern in a string

QUOTENAME	Returns a Unicode string with delimiters added to make the string a valid SQL Server delimited identifier
REPLACE	Replaces all occurrences of a substring within a string, with a new substring
REPLICATE	Repeats a string a specified number of times
REVERSE	Reverses a string and returns the result
RIGHT	Extracts a number of characters from a string (starting from right)
RTRIM	Removes trailing spaces from a string
SOUNDEX	Returns a four-character code to evaluate the similarity of two strings
SPACE	Returns a string of the specified number of space characters
STR	Returns a number as string
STUFF	Deletes a part of a string and then inserts another part into the string, starting at a specified position
SUBSTRING	Extracts some characters from a string

TRANSLATE	Returns the string from the first argument after the characters specified in the second argument are translated into the characters specified in the third argument.
TRIM	Removes leading and trailing spaces (or other specified characters) from a string
UNICODE	Returns the Unicode value for the first character of the input expression
UPPER	Converts a string to upper-case

SQL Server Math/Numeric Functions

Function	Description
ABS	Returns the absolute value of a number
ACOS	Returns the arc cosine of a number
ASIN	Returns the arc sine of a number
ATAN	Returns the arc tangent of a number

ATN2	Returns the arc tangent of two numbers
AVG	Returns the average value of an expression
CEILING	Returns the smallest integer value that is >= a number
COUNT	Returns the number of records returned by a select query
COS	Returns the cosine of a number
COT	Returns the cotangent of a number
DEGREES	Converts a value in radians to degrees
EXP	Returns e raised to the power of a specified number
FLOOR	Returns the largest integer value that is <= to a number
LOG	Returns the natural logarithm of a number, or the logarithm of a number to a specified base
LOG10	Returns the natural logarithm of a number to base 10
MAX	Returns the maximum value in a set of values

MIN	Returns the minimum value in a set of values
PI	Returns the value of PI
POWER	Returns the value of a number raised to the power of another number
RADIANS	Converts a degree value into radians
RAND	Returns a random number
ROUND	Rounds a number to a specified number of decimal places
SIGN	Returns the sign of a number
SIN	Returns the sine of a number
SQRT	Returns the square root of a number
SQUARE	Returns the square of a number
SUM	Calculates the sum of a set of values
TAN	Returns the tangent of a number

SQL Server Date Functions

Function	Description
CURRENT_TIMESTAMP	Returns the current date and time
DATEADD	Adds a time/date interval to a date and then returns the date
DATEDIFF	Returns the difference between two dates
DATEFROMPARTS	Returns a date from the specified parts (year, month, and day values)
DATENAME	Returns a specified part of a date (as string)
DATEPART	Returns a specified part of a date (as integer)
DAY	Returns the day of the month for a specified date
GETDATE	Returns the current database system date and time
GETUTCDATE	Returns the current database system UTC date and time
ISDATE	Checks an expression and returns 1 if it is a valid date, otherwise 0

MONTH	Returns the month part for a specified date (a number from 1 to 12)
SYSDATETIME	Returns the date and time of the SQL Server
YEAR	Returns the year part for a specified date

SQL Server Advanced Functions

Function	Description
CAST	Converts a value (of any type) into a specified datatype
COALESCE	Returns the first non-null value in a list
CONVERT	Converts a value (of any type) into a specified datatype
CURRENT_USER	Returns the name of the current user in the SQL Server database
IIF	Returns a value if a condition is TRUE, or another value if a condition is FALSE

ISNULL	Return a specified value if the expression is NULL, otherwise return the expression
ISNUMERIC	Tests whether an expression is numeric
NULLIF	Returns NULL if two expressions are equal
SESSION_USER	Returns the name of the current user in the SQL Server database
SESSIONPROPERTY	Returns the session settings for a specified option
SYSTEM_USER	Returns the login name for the current user
USER_NAME	Returns the database user name based on the specified id

MS Access Functions

MS Access has many built-in functions.

This reference contains the string, numeric, and date functions in MS Access.

MS Access String Functions

Function	Description
Asc	Returns the ASCII value for the specific character
Chr	Returns the character for the specified ASCII number code
Concat with &	Adds two or more strings together
CurDir	Returns the full path for a specified drive
Format	Formats a value with the specified format
InStr	Gets the position of the first occurrence of a string in another
InstrRev	Gets the position of the first occurrence of a string in another, from the end of string
LCase	Converts a string to lower-case
Left	Extracts a number of characters from a string (starting from left)
Len	Returns the length of a string

LTrim	Removes leading spaces from a string
Mid	Extracts some characters from a string (starting at any position)
Replace	Replaces a substring within a string, with another substring, a specified number of times
Right	Extracts a number of characters from a string (starting from right)
RTrim	Removes trailing spaces from a string
Space	Returns a string of the specified number of space characters
Split	Splits a string into an array of substrings
Str	Returns a number as string
StrComp	Compares two strings
StrConv	Returns a converted string
StrReverse	Reverses a string and returns the result

Trim	Removes both leading and trailing spaces from a string
UCase	Converts a string to upper-case

MS Access Numeric Functions

Function	Description
Abs	Returns the absolute value of a number
Atn	Returns the arc tangent of a number
Avg	Returns the average value of an expression
Cos	Returns the cosine of an angle
Count	Returns the number of records returned by a select query
Exp	Returns e raised to the power of a specified number
Fix	Returns the integer part of a number

Format	Formats a numeric value with the specified format
Int	Returns the integer part of a number
Max	Returns the maximum value in a set of values
Min	Returns the minimum value in a set of values
Randomize	Initializes the random number generator (used by Rnd()) with a seed
Rnd	Returns a random number
Round	Rounds a number to a specified number of decimal places
Sgn	Returns the sign of a number
Sqr	Returns the square root of a number
Sum	Calculates the sum of a set of values
Val	Reads a string and returns the numbers found in the string

MS Access Date Functions

Function	Description
Date	Returns the current system date
DateAdd	Adds a time/date interval to a date and then returns the date
DateDiff	Returns the difference between two dates
DatePart	Returns a specified part of a date (as an integer)
DateSerial	Returns a date from the specified parts (year, month, and day values)
DateValue	Returns a date based on a string
Day	Returns the day of the month for a given date
Format	Formats a date value with the specified format
Hour	Returns the hour part of a time/datetime
Minute	Returns the minute part of a time/datetime

Month	Returns the month part of a given date
MonthName	Returns the name of the month based on a number
Now	Returns the current date and time based on the computer's system date and time
Second	Returns the seconds part of a time/datetime
Time	Returns the current system time
TimeSerial	Returns a time from the specified parts (hour, minute, and second value)
TimeValue	Returns a time based on a string
Weekday	Returns the weekday number for a given date
WeekdayName	Returns the weekday name based on a number
Year	Returns the year part of a given date

MS Access Some Other Functions

Function	Description
CurrentUser	Returns the name of the current database user
Environ	Returns a string that contains the value of an operating system environment variable
IsDate	Checks whether an expression can be converted to a date
IsNull	Checks whether an expression contains Null (no data)
IsNumeric	Checks whether an expression is a valid number

SQL Quick Reference from W3Schools

SQL Statement	Syntax
AND / OR	SELECT column_name(s) FROM table_name

	WHERE condition AND|OR condition
ALTER TABLE	ALTER TABLE table_name ADD column_name datatype or ALTER TABLE table_name DROP COLUMN column_name
AS (alias)	SELECT column_name AS column_alias FROM table_name or SELECT column_name FROM table_name AS table_alias
BETWEEN	SELECT column_name(s) FROM table_name WHERE column_name BETWEEN value1 AND value2
CREATE DATABASE	CREATE DATABASE database_name
CREATE TABLE	CREATE TABLE table_name (column_name1 data_type, column_name2 data_type, column_name3 data_type, ...)
CREATE INDEX	CREATE INDEX index_name ON table_name (column_name) or CREATE UNIQUE INDEX index_name ON table_name (column_name)
CREATE VIEW	CREATE VIEW view_name AS SELECT column_name(s) FROM table_name WHERE condition

DELETE	DELETE FROM table_name WHERE some_column=some_value or DELETE FROM table_name (**Note:** Deletes the entire table!!) DELETE * FROM table_name (**Note:** Deletes the entire table!!)
DROP DATABASE	DROP DATABASE database_name
DROP INDEX	DROP INDEX table_name.index_name (SQL Server) DROP INDEX index_name ON table_name (MS Access) DROP INDEX index_name (DB2/Oracle) ALTER TABLE table_name DROP INDEX index_name (MySQL)
DROP TABLE	DROP TABLE table_name

EXISTS	IF EXISTS (SELECT * FROM table_name WHERE id = ?)
	BEGIN
	--do what needs to be done if exists
	END
	ELSE
	BEGIN
	--do what needs to be done if not
	END

GROUP BY	SELECT column_name, aggregate_function(column_name) FROM table_name WHERE column_name operator value GROUP BY column_name
HAVING	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
IN	SELECT column_name(s) FROM table_name WHERE column_name IN (value1,value2,..)

INSERT INTO	INSERT INTO table_name VALUES (value1, value2, value3,....) *or* INSERT INTO table_name (column1, column2, column3,...) VALUES (value1, value2, value3,....)
INNER JOIN	SELECT column_name(s) FROM table_name1 INNER JOIN table_name2 ON table_name1.column_name=table_name2.column_name
LEFT JOIN	SELECT column_name(s) FROM table_name1 LEFT JOIN table_name2 ON table_name1.column_name=table_name2.column_name
RIGHT JOIN	SELECT column_name(s) FROM table_name1 RIGHT JOIN table_name2 ON table_name1.column_name=table_name2.column_name
FULL JOIN	SELECT column_name(s) FROM table_name1 FULL JOIN table_name2 ON table_name1.column_name=table_name2.column_name
LIKE	SELECT column_name(s) FROM table_name WHERE column_name LIKE pattern
ORDER BY	SELECT column_name(s) FROM table_name ORDER BY column_name [ASC|DESC]
SELECT	SELECT column_name(s) FROM table_name
SELECT *	SELECT * FROM table_name
SELECT DISTINCT	SELECT DISTINCT column_name(s) FROM table_name

SELECT INTO	SELECT * INTO new_table_name [IN externaldatabase] FROM old_table_name *or* SELECT column_name(s) INTO new_table_name [IN externaldatabase] FROM old_table_name
SELECT TOP	SELECT TOP number|percent column_name(s) FROM table_name
TRUNCATE TABLE	TRUNCATE TABLE table_name
UNION	SELECT column_name(s) FROM table_name1 UNION SELECT column_name(s) FROM table_name2
UNION ALL	SELECT column_name(s) FROM table_name1 UNION ALL SELECT column_name(s) FROM table_name2
UPDATE	UPDATE table_name SET column1=value, column2=value,... WHERE some_column=some_value
WHERE	SELECT column_name(s) FROM table_name WHERE column_name operator value