1. 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.

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.

2. 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.

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).

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

The SQL SELECT Statement

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

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.

Select ALL columns

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

The SQL SELECT DISTINCT Statement

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

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

Syntax

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

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

Count Distinct

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

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

Note: The example above will not work in Firefox!

Because COUNT(DISTINCT *column_name*) is not supported in Microsoft Access databases. Firefox is using Microsoft Access in our examples.

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.

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.!

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

The SQL ORDER BY

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

Syntax

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

DESC

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

Order Alphabetically

For string values the ORDER BY keyword will order alphabetically:

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;
```

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':

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.

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%';
```

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.

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%';
```

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;
```

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;
```

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;
```

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-then operator: !< that would give you the same result.

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, ...);
```

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');
```

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');
```

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 ,.**

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;
```

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;
```

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!

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;
```

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';
```

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 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!

SQL DELETE Example

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

Example

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

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;
```

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;
```

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;
```

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:

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

MAX Example

Find the highest price:

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

Syntax

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

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

Set Column Name (Alias)

When you use MIN() or MAX(), the returned column will be named MIN(*field*) or MAX(*field*) by default. To give the column a new name, use the AS keyword:

Example

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

The SQL COUNT() Function

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

Example

Find the total number of products in the Products table:

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

Syntax

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

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;
```

Specify Column

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

If you specify a column 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;
```

Unfortunately the Products table does not have any NULL values, but we can fix that. Run the SQL statement below, and re-run the example above to see the result.

Example

Insert a NULL value to better understand the example above:

```
UPDATE Products
SET ProductName = NULL
WHERE ProductName = 'Chang';
```

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;
```

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;
```

Add a Where Clause

You can add a WHERE clause to specify conditions:

Example

Return the number of orders made for the product with ProductID 11:

```
SELECT SUM(Quantity)
FROM OrderDetails
WHERE ProdictId = 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;
```

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() parenthesis:
```
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.

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;
```

Add a Where Clause

You can add a WHERE clause to specify conditions:

Example

Return the avarege 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);
```

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
```

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';
```

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.

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

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, ...);
```

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', ot '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);
```

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;
```

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';
```

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;
```

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

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;
```

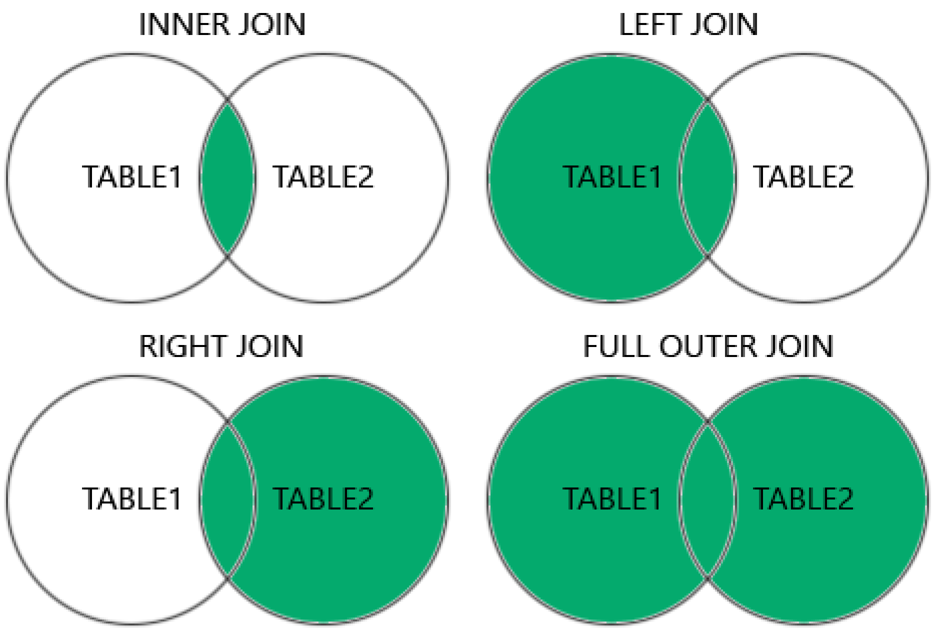
and it will produce something like this:

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



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

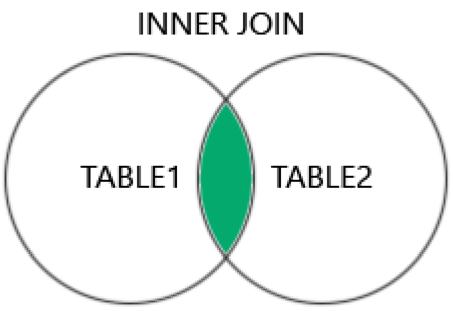
And a selection of the **Categories** table:

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);
```

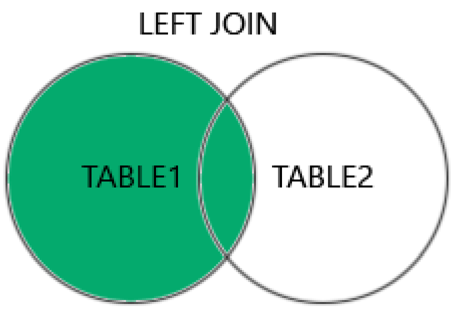
SQL LEFT JOIN Keyword

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.



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).

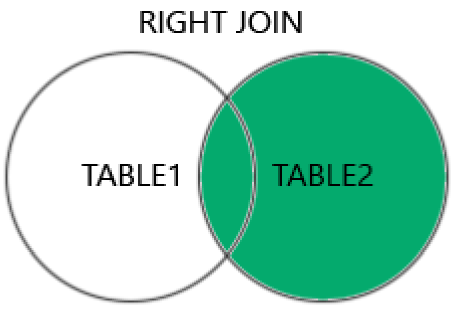
SQL RIGHT JOIN Keyword

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.



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).

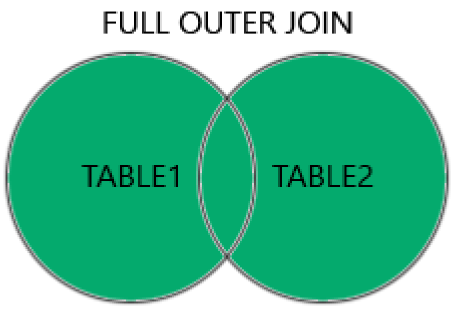
SQL FULL OUTER JOIN Keyword

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!

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

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.

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;
```

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.

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".

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);
```

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;
```

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 FR
OM Orders
LEFT JOIN Shippers ON Orders.ShipperID = Shippers.ShipperID
GROUP BY ShipperName;
```

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);
```

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;
```

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;
```

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);
```

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.Suppli
erID = 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.Suppli
erID = Suppliers.supplierID AND Price = 22);
```

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 <=).

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);
```

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;
```

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;
```

SQL INSERT INTO SELECT Examples

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

Example

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

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

Example

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

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

Example

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

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;
```

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;
```

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

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

Note: The examples in this chapter will not work in Firefox and Microsoft Edge!

Comments are not supported in Microsoft Access databases. Firefox and Microsoft Edge are using Microsoft Access database in our examples.

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;
```

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;
```

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