SQL BASICS AND ADVANCED



# INTRODUCTION

SQL is a standard language for accessing databases. how to use SQL to access and manipulate data in:

MySQL, SQL Server, Access, Oracle, Sybase, DB2, and other database systems.

### SQL Syntax:

SELECT Company, Country FROM Customers WHERE Country <> 'USA'

### SQL Result:

|  |  |
| --- | --- |
| **Company** | **Country** |
| Island Trading | UK |
| Galería del gastrónomo | Spain |
| Laughing Bacchus Wine Cellars | Canada |
| Paris spécialités | France |
| Simons bistro | Denmark |
| Wolski Zajazd | Poland |

**SQL is a “standard language for accessing and manipulating databases”. What is SQL?**

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

### What Can SQL do?

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

### RDBMS

* RDBMS stands for Relational Database Management System.
* RDBMS is the basis for SQL, and for all modern database systems like 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 collections of related data entries and it consists of columns and rows.

# SQL BASICS

### Database Tables

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

Below is an example of a table called "Persons":

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

The table above contains three records (one for each person) and five columns (P\_Id, LastName, FirstName, Address, and City).

### SQL Statements

Most of the actions you need to perform on a database are done with SQL statements. The following SQL statement will select all the records in the "Persons" table:

SELECT \* FROM Persons

### Note:

* SQL is not case sensitive
* 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.
* We are using MS Access and SQL Server 2000 and we do not have to put a semicolon after each

SQL statement, but some database programs force you to use it.

### SQL DML and DDL

SQL can be divided into two parts: The Data Manipulation Language (DML) and the Data Definition Language (DDL).

The query and update commands form the **DML part of SQL**:

**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

The **DDL part of SQL** permits database tables to be created or deleted. It also define indexes (keys), specify links between tables, and impose constraints between tables. The most important DDL statements in SQL are:

**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.
* The result is stored in a result table, called the result-set.
* SQL SELECT Syntax

SELECT column\_name(s) FROM table\_name

And

SELECT \* FROM table\_name

**Note:** SQL is not case sensitive. SELECT is the same as select.

## An SQL SELECT Example

The "Persons" table:

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

Now we want to select the content of the columns named "LastName" and "FirstName" from the table above.

We use the following SELECT statement:

SELECT LastName,FirstName FROM Persons The result-set will look like this:

|  |  |
| --- | --- |
| **LastName** | **FirstName** |
| Hansen | Ola |
| Svendson | Tove |
| Pettersen | Kari |

## SELECT \* Example

Now we want to select all the columns from the "Persons" table. We use the following SELECT statement:

SELECT \* From Persons

## The SQL SELECT DISTINCT Statement

* In a table, some of the columns may contain duplicate values. This is not a problem, however, sometimes you will want to list only the different (distinct) values in a table.
* The DISTINCT keyword can be used to return only distinct (different) values.
* SQL SELECT DISTINCT Syntax

SELECT DISTINCT column\_name(s) FROM table\_name

## SELECT DISTINCT Example

The "Persons" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

Now we want to select only the distinct values from the column named "City" from the table above. We use the following SELECT statement:

SELECT DISTINCT City FROM Persons The result-set will look like this:

|  |
| --- |
| **City** |
| Sandnes |
| Stavanger |

## The WHERE Clause

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

SELECT column\_name(s) FROM table\_name

WHERE column\_name operator value

## WHERE Clause Example

The "Persons" table:

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

Now we want to select only the persons living in the city "Sandnes" from the table above. We use the following SELECT statement:

SELECT \* FROM Persons WHERE City='Sandnes'

The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |

## Quotes Around Text Fields

SQL uses single quotes around text values (most database systems will also accept double quotes). Although, numeric values should not be enclosed in quotes.

For text values:

This is correct:

SELECT \* FROM Persons WHERE FirstName='Tove' This is wrong:

SELECT \* FROM Persons WHERE FirstName=Tove For numeric values:

This is correct:

SELECT \* FROM Persons WHERE Year=1965 This is wrong:

SELECT \* FROM Persons WHERE Year='1965'

## Operators Allowed in the WHERE Clause

With the WHERE clause, the following operators can be used:

|  |  |
| --- | --- |
| **Operator** | **Description** |
| = | Equal |
| <> | Not equal |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |
| BETWEEN | Between an inclusive range |
| LIKE | Search for a pattern |
| IN | If you know the exact value you want to return for at least one of the columns |

**Note:** In some versions of SQL the <> operator may be written as !=

## The AND & OR Operators

* The AND & OR operators are used to filter records based on more than one condition.
* The AND operator displays a record if both the first condition and the second condition is true.
* The OR operator displays a record if either the first condition or the second condition is true.

## AND Operator Example

The "Persons" table:

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

Now we want to select only the persons with the first name equal to "Tove" AND the last name equal to "Svendson":

We use the following SELECT statement:

SELECT \* FROM Persons

WHERE FirstName='Tove'AND LastName='Svendson' The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |

## OR Operator Example

Now we want to select only the persons with the first name equal to "Tove" OR the first name equal to "Ola":

We use the following SELECT statement: SELECT \* FROM Persons

WHERE FirstName='Tove' OR FirstName='Ola'

The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |

## Combining AND & OR

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

Now we want to select only the persons with the last name equal to "Svendson" AND the first name equal to "Tove" OR to "Ola":

We use the following SELECT statement:

SELECT \* FROM Persons

WHERE LastName='Svendson' AND (FirstName='Tove' OR FirstName='Ola') The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |

## The ORDER BY Keyword

* The ORDER BY keyword is used to sort the result-set.
* The ORDER BY keyword is used to sort the result-set by a specified column.
* The ORDER BY keyword sort the records in ascending order by default.
* If you want to sort the records in a descending order, you can use the DESC keyword.
* SQL ORDER BY Syntax

SELECT column\_name(s)

FROM table\_name ORDER BY column\_name(s) ASC|DESC

## ORDER BY Example

The "Persons" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |
| 4 | Nilsen | Tom | Vingvn 23 | Stavanger |

Now we want to select all the persons from the table above, however, we want to sort the persons by their last name.

We use the following SELECT statement: SELECT \* FROM Persons ORDER BY LastName

The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 4 | Nilsen | Tom | Vingvn 23 | Stavanger |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |

## ORDER BY DESC Example

Now we want to select all the persons from the table above, however, we want to sort the persons descending by their last name.

We use the following SELECT statement:

SELECT \* FROM Persons ORDER BY LastName DESC The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |
| 4 | Nilsen | Tom | Vingvn 23 | Stavanger |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |

## The INSERT INTO Statement

* The INSERT INTO statement is used to insert new records in a table.
* The INSERT INTO statement is used to insert a new row in a table.
* SQL INSERT INTO Syntax
* It is possible to write the INSERT INTO statement in two forms.
* The first form doesn't specify the column names where the data will be inserted, only their values: INSERT INTO table\_name VALUES (value1, value2, value3,...)
* The second form specifies both the column names and the values to be inserted:

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

## SQL INSERT INTO Example

We have the following "Persons" table:

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

Now we want to insert a new row in the "Persons" table. We use the following SQL statement:

INSERT INTO Persons VALUES (4,'Nilsen', 'Johan', 'Bakken 2', 'Stavanger') The "Persons" table will now look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |
| 4 | Nilsen | Johan | Bakken 2 | Stavanger |

## Insert Data Only in Specified Columns

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

The following SQL statement will add a new row, but only add data in the "P\_Id", "LastName" and the "FirstName" columns:

INSERT INTO Persons (P\_Id, LastName, FirstName) VALUES (5, 'Tjessem', 'Jakob') The "Persons" table will now look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |
| 4 | Nilsen | Johan | Bakken 2 | Stavanger |
| 5 | Tjessem | Jakob |  |  |

## The UPDATE Statement

* The UPDATE statement is used to update records in a table.
* The UPDATE statement is used to update existing records in a table.
* SQL UPDATE Syntax

UPDATE table\_name

SET column1=value, column2=value2,... WHERE some\_column=some\_value

**Note:** Notice the WHERE clause in the UPDATE syntax. The WHERE clause specifies which record or records that should be updated. If you omit the WHERE clause, all records will be updated!

## SQL UPDATE Example

The "Persons" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |
| 4 | Nilsen | Johan | Bakken 2 | Stavanger |
| 5 | Tjessem | Jakob |  |  |

Now we want to update the person "Tjessem, Jakob" in the "Persons" table. We use the following SQL statement:

UPDATE Persons

SET Address='Nissestien 67', City='Sandnes'

WHERE LastName='Tjessem' AND FirstName='Jakob' The "Persons" table will now look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |
| 4 | Nilsen | Johan | Bakken 2 | Stavanger |
| 5 | Tjessem | Jakob | Nissestien 67 | Sandnes |

## SQL UPDATE Warning

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

UPDATE Persons

SET Address='Nissestien 67', City='Sandnes' The "Persons" table would have looked like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Nissestien 67 | Sandnes |
| 2 | Svendson | Tove | Nissestien 67 | Sandnes |
| 3 | Pettersen | Kari | Nissestien 67 | Sandnes |
| 4 | Nilsen | Johan | Nissestien 67 | Sandnes |
| 5 | Tjessem | Jakob | Nissestien 67 | Sandnes |

## The DELETE Statement

* The DELETE statement is used to delete records in a table.
* The DELETE statement is used to delete rows in a table.
* SQL DELETE Syntax

DELETE FROM table\_name WHERE some\_column=some\_value

**Note:** Notice the WHERE clause in the DELETE syntax. The WHERE clause specifies which record or records that should be deleted. If you omit the WHERE clause, all records will be deleted!

## SQL DELETE Example

The "Persons" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |
| 4 | Nilsen | Johan | Bakken 2 | Stavanger |
| 5 | Tjessem | Jakob | Nissestien 67 | Sandnes |

Now we want to delete the person "Tjessem, Jakob" in the "Persons" table. We use the following SQL statement:

DELETE FROM Persons

WHERE LastName='Tjessem' AND FirstName='Jakob' The "Persons" table will now look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |
| 4 | Nilsen | Johan | Bakken 2 | Stavanger |

## Delete All Rows

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

DELETE FROM table\_name or

DELETE \* FROM table\_name

# SQL ADVANCE

## The TOP Clause

* The TOP clause is used to specify the number of records to return.
* The TOP clause can be very useful on large tables with thousands of records. Returning a large number of records can impact on performance.

**Note:** Not all database systems support the TOP clause.

* SQL Server Syntax:

SELECT TOP number|percent column\_name(s) FROM table\_name

* SQL SELECT TOP Equivalent in MySQL and Oracle:
  + MySQL Syntax:

SELECT column\_name(s) FROM table\_name LIMIT number

Example:

SELECT \*

FROM Persons LIMIT 5

* + Oracle Syntax

SELECT column\_name(s) FROM table\_name

WHERE ROWNUM <= number

Example

SELECT \*

FROM Persons

WHERE ROWNUM <=5

## SQL TOP Example

The "Persons" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |
| 4 | Nilsen | Tom | Vingvn 23 | Stavanger |

Now we want to select only the two first records in the table above. We use the following SELECT statement:

SELECT TOP 2 \* FROM Persons

The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |

## SQL TOP PERCENT Example

The "Persons" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |
| 4 | Nilsen | Tom | Vingvn 23 | Stavanger |

Now we want to select only 50% of the records in the table above. We use the following SELECT statement:

SELECT TOP 50 PERCENT \* FROM Persons

The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |

## SQL Wildcards

* SQL wildcards can be used when searching for data in a database.
* SQL wildcards can substitute for one or more characters when searching for data in a database.
* SQL wildcards must be used with the SQL LIKE operator.
* With SQL, the following wildcards can be used:

|  |  |
| --- | --- |
| **Wildcard** | **Description** |
| % | A substitute for zero or more characters |
| \_ | A substitute for exactly one character |
| [charlist] | Any single character in charlist |
| [^charlist] or  [!charlist] | Any single character not in charlist |

## SQL Wildcard Examples

We have the following "Persons" table:

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

## Using the % Wildcard

Now we want to select the persons living in a city that starts with "sa" from the "Persons" table. We use the following SELECT statement:

SELECT \* FROM Persons

WHERE City LIKE 'sa%'

The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |

Next, we want to select the persons living in a city that contains the pattern "nes" from the "Persons" table.

We use the following SELECT statement:

SELECT \* FROM Persons WHERE City LIKE '%nes%'

The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |

## Using the \_ Wildcard

Now we want to select the persons with a first name that starts with any character, followed by "la" from the "Persons" table.

We use the following SELECT statement:

SELECT \* FROM Persons WHERE FirstName LIKE '\_la'

The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |

Next, we want to select the persons with a last name that starts with "S", followed by any character, followed by "end", followed by any character, followed by "on" from the "Persons" table.

We use the following SELECT statement:

SELECT \* FROM Persons

WHERE LastName LIKE 'S\_end\_on' The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |

## Using the [charlist] Wildcard

Now we want to select the persons with a last name that starts with "b" or "s" or "p" from the "Persons" table.

We use the following SELECT statement:

SELECT \* FROM Persons WHERE LastName LIKE '[bsp]%'

The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

Next, we want to select the persons with a last name that do not start with "b" or "s" or "p" from the "Persons" table.

We use the following SELECT statement:

SELECT \* FROM Persons WHERE LastName LIKE '[!bsp]%'

The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |

## The LIKE Operator

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

SELECT column\_name(s) FROM table\_name

WHERE column\_name LIKE pattern

## LIKE Operator Example

The "Persons" table:

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

Now we want to select the persons living in a city that starts with "s" from the table above. We use the following SELECT statement:

SELECT \* FROM Persons WHERE City LIKE 's%'

The "%" sign can be used to define wildcards (missing letters in the pattern) both before and after the pattern.

The result-set will look like this:

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

Next, we want to select the persons living in a city that ends with an "s" from the "Persons" table. We use the following SELECT statement:

SELECT \* FROM Persons WHERE City LIKE '%s'

The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |

Next, we want to select the persons living in a city that contains the pattern "tav" from the "Persons" table.

We use the following SELECT statement:

SELECT \* FROM Persons WHERE City LIKE '%tav%'

The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

It is also possible to select the persons living in a city that NOT contains the pattern "tav" from the "Persons" table, by using the NOT keyword.

We use the following SELECT statement:

SELECT \* FROM Persons WHERE City NOT LIKE '%tav%'

The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |

## The IN Operator

* The IN operator allows you to specify multiple values in a WHERE clause.
* SQL IN Syntax:

SELECT column\_name(s) FROM table\_name

WHERE column\_name IN (value1,value2,...)

## IN Operator Example

The "Persons" table:

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

Now we want to select the persons with a last name equal to "Hansen" or "Pettersen" from the table above.

We use the following SELECT statement: SELECT \* FROM Persons

WHERE LastName IN ('Hansen','Pettersen')

The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

## The BETWEEN Operator

* The BETWEEN operator is used in a WHERE clause to select a range of data between two values.
* The BETWEEN operator selects a range of data between two values. The values can be numbers,

text, or dates.

* SQL BETWEEN Syntax:

SELECT column\_name(s) FROM table\_name WHERE column\_name

BETWEEN value1 AND value2

## BETWEEN Operator Example

The "Persons" table:

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

Now we want to select the persons with a last name alphabetically between "Hansen" and "Pettersen" from the table above.

We use the following SELECT statement: SELECT \* FROM Persons

WHERE LastName

BETWEEN 'Hansen' AND 'Pettersen' The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |

### Note:

The BETWEEN operator is treated differently in different databases.

In some databases, persons with the LastName of "Hansen" or "Pettersen" will not be listed, because the BETWEEN operator only selects fields that are between and excluding the test values).

In other databases, persons with the LastName of "Hansen" or "Pettersen" will be listed, because the BETWEEN operator selects fields that are between and including the test values).

And in other databases, persons with the LastName of "Hansen" will be listed, but "Pettersen" will not be listed (like the example above), because the BETWEEN operator selects fields between the test values, including the first test value and excluding the last test value.

Therefore: Check how your database treats the BETWEEN operator.

## Example 2

To display the persons outside the range in the previous example, use NOT BETWEEN: SELECT \* FROM Persons

WHERE LastName

NOT BETWEEN 'Hansen' AND 'Pettersen' The result-set will look like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

## SQL Alias

* With SQL, an alias name can be given to a table or to a column.
* You can give a table or a column another name by using an alias. This can be a good thing to do if you have very long or complex table names or column names.
* An alias name could be anything, but usually it is short.
* SQL Alias Syntax for Tables:

SELECT column\_name(s) FROM table\_name

AS alias\_name

* SQL Alias Syntax for Columns:

SELECT column\_name AS alias\_name FROM table\_name

## Alias Example

Assume we have a table called "Persons" and another table called "Product\_Orders". We will give the table aliases of "p" and "po" respectively.

Now we want to list all the orders that "Ola Hansen" is responsible for. We use the following SELECT statement:

SELECT po.OrderID, p.LastName, p.FirstName FROM Persons AS p, Product\_Orders AS po

WHERE p.LastName='Hansen' AND p.FirstName='Ola' The same SELECT statement without aliases:

SELECT Product\_Orders.OrderID, Persons.LastName, Persons.FirstName FROM Persons, Product\_Orders

WHERE Persons.LastName='Hansen' AND Persons.FirstName='Ola'

**Notes:** As you'll see from the two SELECT statements above; aliases can make queries easier to both write and to read.

## SQL JOIN

* SQL joins are used to query data from two or more tables, based on a relationship between certain columns in these tables.
* The JOIN keyword is used in an SQL statement to query data from two or more tables, based on a

relationship between certain columns in these tables.

* Tables in a database are often related to each other with keys.
* A primary key is a column (or a combination of columns) with a unique value for each row. Each primary key value must be unique within the table. The purpose is to bind data together, across tables, without repeating all of the data in every table.

Look at the "Persons" table:

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

Note that the "P\_Id" column is the primary key in the "Persons" table. This means that **no** two rows can have the same P\_Id. The P\_Id distinguishes two persons even if they have the same name.

Next, we have the "Orders" table:

|  |  |  |
| --- | --- | --- |
| **O\_Id** | **OrderNo** | **P\_Id** |
| 1 | 77895 | 3 |
| 2 | 44678 | 3 |
| 3 | 22456 | 1 |
| 4 | 24562 | 1 |
| 5 | 34764 | 15 |

Note that the "O\_Id" column is the primary key in the "Orders" table and that the "P\_Id" column refers to the persons in the "Persons" table without using their names.

Notice that the relationship between the two tables above is the "P\_Id" column.

## Different SQL JOINs

Before we continue with examples, we will list the types of JOIN you can use, and the differences between them.

* **JOIN**: Return rows when there is at least one match in both tables
* **LEFT JOIN**: Return all rows from the left table, even if there are no matches in the right table
* **RIGHT JOIN**: Return all rows from the right table, even if there are no matches in the left table
* **FULL JOIN**: Return rows when there is a match in one of the tables

## SQL INNER JOIN Keyword

* The INNER JOIN keyword return rows when there is at least one match in both tables.
* SQL INNER JOIN Syntax:

SELECT column\_name(s) FROM table\_name1 INNER JOIN table\_name2

ON table\_name1.column\_name=table\_name2.column\_name

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

## SQL INNER JOIN Example

The "Persons" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

The "Orders" table:

|  |  |  |
| --- | --- | --- |
| **O\_Id** | **OrderNo** | **P\_Id** |
| 1 | 77895 | 3 |
| 2 | 44678 | 3 |
| 3 | 22456 | 1 |
| 4 | 24562 | 1 |
| 5 | 34764 | 15 |

Now we want to list all the persons with any orders. We use the following SELECT statement:

SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo FROM Persons INNER JOIN Orders ON Persons.P\_Id=Orders.P\_Id ORDER BY Persons.LastName

The result-set will look like this:

|  |  |  |
| --- | --- | --- |
| **LastName** | **FirstName** | **OrderNo** |
| Hansen | Ola | 22456 |
| Hansen | Ola | 24562 |
| Pettersen | Kari | 77895 |
| Pettersen | Kari | 44678 |

The INNER JOIN keyword return rows when there is at least one match in both tables. If there are rows in "Persons" that do not have matches in "Orders", those rows will NOT be listed.

## SQL LEFT JOIN Keyword

* The LEFT JOIN keyword returns all rows from the left table (table\_name1), even if there are no matches in the right table (table\_name2).
* SQL LEFT JOIN Syntax:

SELECT column\_name(s)

FROM table\_name1 LEFT JOIN table\_name2

ON table\_name1.column\_name=table\_name2.column\_name

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

## SQL LEFT JOIN Example

The "Persons" table:

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

The "Orders" table:

|  |  |  |
| --- | --- | --- |
| **O\_Id** | **OrderNo** | **P\_Id** |
| 1 | 77895 | 3 |
| 2 | 44678 | 3 |
| 3 | 22456 | 1 |
| 4 | 24562 | 1 |
| 5 | 34764 | 15 |

Now we want to list all the persons and their orders - if any, from the tables above.

We use the following SELECT statement:

SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo FROM Persons LEFT JOIN Orders

ON Persons.P\_Id=Orders.P\_Id ORDER BY Persons.LastName

The result-set will look like this:

|  |  |  |
| --- | --- | --- |
| **LastName** | **FirstName** | **OrderNo** |
| Hansen | Ola | 22456 |
| Hansen | Ola | 24562 |
| Pettersen | Kari | 77895 |
| Pettersen | Kari | 44678 |
| Svendson | Tove |  |

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

## SQL RIGHT JOIN Keyword

* The RIGHT JOIN keyword Return all rows from the right table (table\_name2), even if there are no matches in the left table (table\_name1).
* SQL RIGHT JOIN Syntax:

SELECT column\_name(s) FROM table\_name1 RIGHT JOIN table\_name2

ON table\_name1.column\_name=table\_name2.column\_name

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

## SQL RIGHT JOIN Example

The "Persons" table:

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

The "Orders" table:

|  |  |  |
| --- | --- | --- |
| **O\_Id** | **OrderNo** | **P\_Id** |
| 1 | 77895 | 3 |
| 2 | 44678 | 3 |
| 3 | 22456 | 1 |
| 4 | 24562 | 1 |
| 5 | 34764 | 15 |

Now we want to list all the orders with containing persons - if any, from the tables above. We use the following SELECT statement:

SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo FROM Persons

RIGHT JOIN Orders

ON Persons.P\_Id=Orders.P\_Id ORDER BY Persons.LastName

The result-set will look like this:

|  |  |  |
| --- | --- | --- |
| **LastName** | **FirstName** | **OrderNo** |
| Hansen | Ola | 22456 |
| Hansen | Ola | 24562 |
| Pettersen | Kari | 77895 |
| Pettersen | Kari | 44678 |
|  |  | 34764 |

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

## SQL FULL JOIN Keyword

* The FULL JOIN keyword return rows when there is a match in one of the tables.
* SQL FULL JOIN Syntax:

## SQL FULL JOIN Example

The "Persons" table:

SELECT column\_name(s) FROM table\_name1 FULL JOIN table\_name2

ON table\_name1.column\_name=table\_name2.column\_name

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

The "Orders" table:

|  |  |  |
| --- | --- | --- |
| **O\_Id** | **OrderNo** | **P\_Id** |

|  |  |  |
| --- | --- | --- |
|  |  |  |
| 1 | 77895 | 3 |
| 2 | 44678 | 3 |
| 3 | 22456 | 1 |
| 4 | 24562 | 1 |
| 5 | 34764 | 15 |

Now we want to list all the persons and their orders, and all the orders with their persons. We use the following SELECT statement:

SELECT Persons.LastName, Persons.FirstName, Orders.OrderNo

FROM Persons FULL JOIN Orders

ON Persons.P\_Id=Orders.P\_Id ORDER BY Persons.LastName

The result-set will look like this:

|  |  |  |
| --- | --- | --- |
| **LastName** | **FirstName** | **OrderNo** |
| Hansen | Ola | 22456 |
| Hansen | Ola | 24562 |
| Pettersen | Kari | 77895 |
| Pettersen | Kari | 44678 |
| Svendson | Tove |  |
|  |  | 34764 |

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

## The SQL UNION Operator

* The UNION operator is used to combine the result-set of two or more SELECT statements.
* Notice that each SELECT statement within the UNION must have the same number of columns. The columns must also have similar data types. Also, the columns in each SELECT statement must be in the same order.
* SQL UNION Syntax:

SELECT column\_name(s) FROM table\_name1 UNION

SELECT column\_name(s) FROM table\_name2

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

* SQL UNION ALL Syntax:

SELECT column\_name(s) FROM table\_name1 UNION ALL

SELECT column\_name(s) FROM table\_name2

* **PS:** The column names in the result-set of a UNION are always equal to the column names in the first SELECT statement in the UNION.

## SQL UNION Example

Look at the following tables:

### "Employees\_Norway":

|  |  |
| --- | --- |
| **E\_ID** | **E\_Name** |
| 01 | Hansen, Ola |
| 02 | Svendson, Tove |
| 03 | Svendson, Stephen |
| 04 | Pettersen, Kari |

**"Employees\_USA"**:

|  |  |
| --- | --- |
| **E\_ID** | **E\_Name** |
| 01 | Turner, Sally |

|  |  |
| --- | --- |
| 02 | Kent, Clark |
| 03 | Svendson, Stephen |
| 04 | Scott, Stephen |

Now we want to list **all the different** employees in Norway and USA. We use the following SELECT statement:

SELECT E\_Name FROM Employees\_Norway UNION

SELECT E\_Name FROM Employees\_USA The result-set will look like this:

|  |
| --- |
| **E\_Name** |
| Hansen, Ola |
| Svendson, Tove |
| Svendson, Stephen |
| Pettersen, Kari |
| Turner, Sally |
| Kent, Clark |
| Scott, Stephen |

**Note:** This command cannot be used to list all employees in Norway and USA. In the example above we have two employees with equal names, and only one of them will be listed. The UNION command selects only distinct values.

## SQL UNION ALL Example

Now we want to list **all** employees in Norway and USA: SELECT E\_Name FROM Employees\_Norway

UNION ALL

SELECT E\_Name FROM Employees\_USA

**Result**

|  |
| --- |
| **E\_Name** |
| Hansen, Ola |
| Svendson, Tove |
| Svendson, Stephen |
| Pettersen, Kari |
| Turner, Sally |
| Kent, Clark |
| Svendson, Stephen |
| Scott, Stephen |

## The SQL SELECT INTO Statement

* The SQL SELECT INTO statement can be used to create backup copies of tables.
* The SELECT INTO statement selects data from one table and inserts it into a different table.
* The SELECT INTO statement is most often used to create backup copies of tables.
* SQL SELECT INTO Syntax

We can select all columns into the new table: SELECT \*

INTO new\_table\_name [IN externaldatabase] FROM old\_tablename

Or we can select only the columns we want into the new table: SELECT column\_name(s)

INTO new\_table\_name [IN externaldatabase] FROM old\_tablename

## SQL SELECT INTO Example

**Make a Backup Copy** - Now we want to make an exact copy of the data in our "Persons" table. We use the following SQL statement:

SELECT \*

INTO Persons\_Backup FROM Persons

We can also use the IN clause to copy the table into another database:

SELECT \*

INTO Persons\_Backup IN 'Backup.mdb' FROM Persons

We can also copy only a few fields into the new table:

SELECT LastName,FirstName INTO Persons\_Backup

FROM Persons

## SQL SELECT INTO - With a WHERE Clause

We can also add a WHERE clause.

The following SQL statement creates a "Persons\_Backup" table with only the persons who lives in the city "Sandnes":

SELECT LastName,Firstname INTO Persons\_Backup FROM Persons

WHERE City='Sandnes'

## SQL SELECT INTO - Joined Tables

Selecting data from more than one table is also possible.

The following example creates a "Persons\_Order\_Backup" table contains data from the two tables "Persons" and "Orders":

SELECT Persons.LastName,Orders.OrderNo INTO Persons\_Order\_Backup

FROM Persons INNER JOIN Orders

ON Persons.P\_Id=Orders.P\_Id

## The CREATE DATABASE Statement

* The CREATE DATABASE statement is used to create a database.
* SQL CREATE DATABASE Syntax:

CREATE DATABASE database\_name

## CREATE DATABASE Example

Now we want to create a database called "my\_db".

We use the following CREATE DATABASE statement:

CREATE DATABASE my\_db

Database tables can be added with the CREATE TABLE statement. The CREATE TABLE Statement

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

SQL CREATE TABLE Syntax:

CREATE TABLE table\_name (

column\_name1 data\_type, column\_name2 data\_type, column\_name3 data\_type,

....

)

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.

## CREATE TABLE Example

Now we want to create a table called "Persons" that contains five columns: P\_Id, LastName, FirstName, Address, and City.

We use the following CREATE TABLE statement:

CREATE TABLE Persons (

P\_Id int,

LastName varchar(255), FirstName varchar(255), Address varchar(255), City varchar(255)

)

The P\_Id column is of type int and will hold a number. The LastName, FirstName, Address, and City columns are of type varchar with a maximum length of 255 characters.

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
|  |  |  |  |  |

The empty table can be filled with data with the INSERT INTO statement.

## SQL Constraints

* Constraints are used to limit the type of data that can go into a table.
* Constraints can be specified when a table is created (with the CREATE TABLE statement) or after the table is created (with the ALTER TABLE statement).

We will focus on the following constraints:

* NOT NULL
* UNIQUE
* PRIMARY KEY
* FOREIGN KEY
* CHECK
* DEFAULT

The next chapters will describe each constraint in details.

## SQL NOT NULL Constraint

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

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

CREATE TABLE Persons (

P\_Id int NOT NULL,

LastName varchar(255) NOT NULL, FirstName varchar(255),

Address varchar(255), City varchar(255)

)

## SQL UNIQUE Constraint

* The UNIQUE constraint uniquely identifies each record in a database table.
* The UNIQUE and PRIMARY KEY constraints both provide a guarantee for uniqueness for a column or set of columns.
* A PRIMARY KEY constraint automatically has a UNIQUE constraint defined on it.
* Note that you can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

## SQL UNIQUE Constraint on CREATE TABLE

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

### MySQL:

CREATE TABLE Persons (

P\_Id int NOT NULL,

LastName varchar(255) NOT NULL, FirstName varchar(255),

Address varchar(255), City varchar(255), UNIQUE (P\_Id)

)

### SQL Server / Oracle / MS Access:

CREATE TABLE Persons (

P\_Id int NOT NULL UNIQUE,

LastName varchar(255) NOT NULL, FirstName varchar(255),

Address varchar(255), City varchar(255)

)

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

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

CREATE TABLE Persons (

P\_Id int NOT NULL,

LastName varchar(255) NOT NULL, FirstName varchar(255),

Address varchar(255), City varchar(255),

CONSTRAINT uc\_PersonID UNIQUE (P\_Id,LastName)

)

## SQL UNIQUE Constraint on ALTER TABLE

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

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

ALTER TABLE Persons ADD UNIQUE (P\_Id)

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

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

ALTER TABLE Persons

ADD CONSTRAINT uc\_PersonID UNIQUE (P\_Id,LastName)

## To DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL:

### MySQL:

ALTER TABLE Persons DROP INDEX uc\_PersonID

### SQL Server / Oracle / MS Access:

ALTER TABLE Persons

DROP CONSTRAINT uc\_PersonID

## SQL PRIMARY KEY Constraint

* The PRIMARY KEY constraint uniquely identifies each record in a database table.
* Primary keys must contain unique values.
* A primary key column cannot contain NULL values.
* Each table should have a primary key, and each table can have only ONE primary key.

## SQL PRIMARY KEY Constraint on CREATE TABLE

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

### MySQL:

CREATE TABLE Persons (

P\_Id int NOT NULL,

LastName varchar(255) NOT NULL, FirstName varchar(255),

Address varchar(255), City varchar(255), PRIMARY KEY (P\_Id)

)

### SQL Server / Oracle / MS Access:

CREATE TABLE Persons (

P\_Id int NOT NULL PRIMARY KEY,

LastName varchar(255) NOT NULL, FirstName varchar(255),

Address varchar(255), City varchar(255)

)

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 (

P\_Id int NOT NULL,

LastName varchar(255) NOT NULL, FirstName varchar(255),

Address varchar(255), City varchar(255),

CONSTRAINT pk\_PersonID PRIMARY KEY (P\_Id,LastName)

)

## SQL PRIMARY KEY Constraint on ALTER TABLE

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

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

ALTER TABLE Persons ADD PRIMARY KEY (P\_Id)

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

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

ALTER TABLE Persons

ADD CONSTRAINT pk\_PersonID PRIMARY KEY (P\_Id,LastName)

**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).

## To 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\_PersonID

## SQL FOREIGN KEY Constraint

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

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

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

The "Orders" table:

|  |  |  |
| --- | --- | --- |
| **O\_Id** | **OrderNo** | **P\_Id** |
| 1 | 77895 | 3 |
| 2 | 44678 | 3 |
| 3 | 22456 | 2 |
| 4 | 24562 | 1 |

Note that the "P\_Id" column in the "Orders" table points to the "P\_Id" column in the "Persons" table. The "P\_Id" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.

The "P\_Id" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.

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

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

## SQL FOREIGN KEY Constraint on CREATE TABLE

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

### MySQL:

CREATE TABLE Orders (

O\_Id int NOT NULL, OrderNo int NOT NULL, P\_Id int,

PRIMARY KEY (O\_Id),

FOREIGN KEY (P\_Id) REFERENCES Persons(P\_Id)

)

### SQL Server / Oracle / MS Access:

CREATE TABLE Orders (

O\_Id int NOT NULL PRIMARY KEY,

OrderNo int NOT NULL,

P\_Id int FOREIGN KEY REFERENCES Persons(P\_Id)

)

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

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

CREATE TABLE Orders (

O\_Id int NOT NULL, OrderNo int NOT NULL, P\_Id int,

PRIMARY KEY (O\_Id),

CONSTRAINT fk\_PerOrders FOREIGN KEY (P\_Id) REFERENCES Persons(P\_Id)

)

## SQL FOREIGN KEY Constraint on ALTER TABLE

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

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

ALTER TABLE Orders ADD FOREIGN KEY (P\_Id)

REFERENCES Persons(P\_Id)

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

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

ALTER TABLE Orders

ADD CONSTRAINT fk\_PerOrders

FOREIGN KEY (P\_Id)

REFERENCES Persons(P\_Id)

## To DROP a FOREIGN KEY Constraint

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

### MySQL:

ALTER TABLE Orders

DROP FOREIGN KEY fk\_PerOrders SQL Server / Oracle / MS Access:

ALTER TABLE Orders

DROP CONSTRAINT fk\_PerOrders

## SQL CHECK Constraint

* The CHECK constraint is used to limit the value range that can be placed in a column.
* If you define a CHECK constraint on a single column it allows only certain values for this column.
* If you define a CHECK constraint on a table it can limit the values in certain columns based on

values in other columns in the row.

## SQL CHECK Constraint on CREATE TABLE

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

### My SQL:

CREATE TABLE Persons (

P\_Id int NOT NULL,

LastName varchar(255) NOT NULL, FirstName varchar(255),

Address varchar(255), City varchar(255), CHECK (P\_Id>0)

)

### SQL Server / Oracle / MS Access:

CREATE TABLE Persons (

P\_Id int NOT NULL CHECK (P\_Id>0),

LastName varchar(255) NOT NULL, FirstName varchar(255),

Address varchar(255), City varchar(255)

)

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 (

P\_Id int NOT NULL,

LastName varchar(255) NOT NULL, FirstName varchar(255),

Address varchar(255), City varchar(255),

CONSTRAINT chk\_Person CHECK (P\_Id>0 AND City='Sandnes')

)

## SQL CHECK Constraint on ALTER TABLE

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

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

ALTER TABLE Persons ADD CHECK (P\_Id>0)

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

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

ALTER TABLE Persons

ADD CONSTRAINT chk\_Person CHECK (P\_Id>0 AND City='Sandnes')

## To DROP a CHECK Constraint

To drop a CHECK constraint, use the following SQL:

### SQL Server / Oracle / MS Access:

ALTER TABLE Persons

DROP CONSTRAINT chk\_Person

## SQL DEFAULT Constraint

* The DEFAULT constraint is used to insert a default value into a column.
* The default value will be added to all new records, if no other value is specified.

## SQL DEFAULT Constraint on CREATE TABLE

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

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

CREATE TABLE Persons (

P\_Id int NOT NULL,

LastName varchar(255) NOT NULL, FirstName varchar(255),

Address varchar(255),

City varchar(255) DEFAULT 'Sandnes'

)

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

CREATE TABLE Orders (

O\_Id int NOT NULL, OrderNo int NOT NULL, P\_Id int,

OrderDate date DEFAULT GETDATE()

)

## SQL DEFAULT Constraint on ALTER TABLE

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

### MySQL:

ALTER

ALTER City SET DEFAULT 'SANDNES'

TABLE

Persons

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

ALTER TABLE Persons

ALTER COLUMN City SET DEFAULT 'SANDNES'

To DROP a DEFAULT Constraint

To drop a DEFAULT constraint, use the following SQL:

### MySQL:

ALTER TABLE Persons ALTER City DROP DEFAULT

### SQL Server / Oracle / MS Access:

ALTER TABLE Persons

ALTER COLUMN City DROP DEFAULT

## Indexes

* The CREATE INDEX statement is used to create indexes in tables.
* Indexes allow the database application to find data fast; without reading the whole table.
* An index can be created in a table to find data more quickly and efficiently.
* The users cannot see the indexes, they are just used to speed up searches/queries.

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

SQL CREATE INDEX Syntax

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

CREATE INDEX index\_name ON table\_name (column\_name)

SQL CREATE UNIQUE INDEX Syntax

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

CREATE UNIQUE INDEX index\_name ON table\_name (column\_name)

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

## CREATE INDEX Example

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

CREATE INDEX PIndex

ON Persons (LastName)

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

CREATE INDEX PIndex

ON Persons (LastName, FirstName)

## The DROP INDEX Statement

* Indexes, tables, and databases can easily be deleted/removed with the DROP statement.
* The DROP INDEX statement is used to delete an index in a table.

DROP INDEX Syntax for MS Access:

DROP INDEX index\_name ON table\_name

DROP INDEX Syntax for MS SQL Server:

DROP INDEX table\_name.index\_name

DROP INDEX Syntax for DB2/Oracle:

DROP INDEX index\_name

DROP INDEX Syntax for MySQL:

ALTER TABLE table\_name DROP INDEX index\_name

## The DROP TABLE Statement

The DROP TABLE statement is used to delete a table. DROP TABLE table\_name

## The DROP DATABASE Statement

The DROP DATABASE statement is used to delete a database. DROP DATABASE database\_name

## The TRUNCATE TABLE Statement

What if we only want to delete the data inside the table, and not the table itself? Then, use the TRUNCATE TABLE statement:

TRUNCATE TABLE table\_name

## The ALTER TABLE Statement

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

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

ALTER TABLE table\_name ADD column\_name datatype

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

ALTER TABLE table\_name DROP COLUMN column\_name

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

ALTER TABLE table\_name

ALTER COLUMN column\_name datatype

## SQL ALTER TABLE Example

Look at the "Persons" table:

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

Now we want to add a column named "DateOfBirth" in the "Persons" table. We use the following SQL statement:

ALTER TABLE Persons ADD DateOfBirth date

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

The "Persons" table will now like this:

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

## Change Data Type Example

Now we want to change the data type of the column named "DateOfBirth" in the "Persons" table. We use the following SQL statement:

ALTER TABLE Persons

ALTER COLUMN DateOfBirth year

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

## DROP COLUMN Example

Next, we want to delete the column named "DateOfBirth" in the "Persons" table. We use the following SQL statement:

ALTER TABLE Persons DROP COLUMN DateOfBirth

The "Persons" table will now like this:

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

## AUTO INCREMENT a Field

* Auto-increment allows a unique number to be generated when a new record is inserted into a table.
* Very often we would like the value of the primary key field to be created automatically every

time a new record is inserted.

We would like to create an auto-increment field in a table. Syntax for MySQL

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

CREATE TABLE Persons (

P\_Id int NOT NULL AUTO\_INCREMENT,

LastName varchar(255) NOT NULL, FirstName varchar(255),

Address varchar(255), City varchar(255), PRIMARY KEY (P\_Id)

)

MySQL uses the AUTO\_INCREMENT keyword to perform an auto-increment feature.

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

To let the AUTO\_INCREMENT sequence start with another value, use the following SQL statement: ALTER TABLE Persons AUTO\_INCREMENT=100

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

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

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

Syntax for SQL Server

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

CREATE TABLE Persons (

P\_Id int PRIMARY KEY IDENTITY,

LastName varchar(255) NOT NULL, FirstName varchar(255),

Address varchar(255), City varchar(255)

)

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

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

To specify that the "P\_Id" column should start at value 10 and increment by 5, change the identity to IDENTITY(10,5).

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

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

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

Syntax for Access

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

CREATE TABLE Persons (

P\_Id PRIMARY KEY AUTOINCREMENT,

LastName varchar(255) NOT NULL, FirstName varchar(255),

Address varchar(255), City varchar(255)

)

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.

To specify that the "P\_Id" column should start at value 10 and increment by 5, change the autoincrement to AUTOINCREMENT(10,5).

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

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

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

Syntax for Oracle

In Oracle the code is a little bit more tricky.

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

Use the following CREATE SEQUENCE syntax:

CREATE SEQUENCE seq\_person MINVALUE 1

START WITH 1

INCREMENT BY 1

CACHE 10

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

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

INSERT INTO Persons (P\_Id,FirstName,LastName) VALUES (seq\_person.nextval,'Lars','Monsen')

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

## SQL CREATE VIEW Statement

* In SQL, a view is a virtual table based on the result-set of an SQL statement.
* A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.
* You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.
* SQL CREATE VIEW Syntax:

CREATE VIEW view\_name AS SELECT column\_name(s)

FROM table\_name WHERE condition

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

## SQL CREATE VIEW Examples

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

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

CREATE VIEW [Current Product List] AS SELECT ProductID,ProductName

FROM Products

WHERE Discontinued=No

We can query the view above as follows:

SELECT \* FROM [Current Product List]

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

CREATE VIEW [Products Above Average Price] AS SELECT ProductName,UnitPrice

FROM Products

WHERE UnitPrice>(SELECT AVG(UnitPrice) FROM Products) We can query the view above as follows:

SELECT \* FROM [Products Above Average Price]

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

CREATE VIEW [Category Sales For 1997] AS

SELECT DISTINCT CategoryName,Sum(ProductSales) AS CategorySales FROM [Product Sales for 1997]

GROUP BY CategoryName

We can query the view above as follows:

SELECT \* FROM [Category Sales For 1997]

We can also add a condition to the query. Now we want to see the total sale only for the category "Beverages":

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

## SQL Updating a View

You can update a view by using the following syntax:

SQL CREATE OR REPLACE VIEW Syntax:

CREATE OR REPLACE VIEW view\_name AS SELECT column\_name(s)

FROM table\_name WHERE condition

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

CREATE VIEW [Current Product List] AS SELECT ProductID,ProductName,Category FROM Products

WHERE Discontinued=No

## SQL Dropping a View

You can delete a view with the DROP VIEW command. SQL DROP VIEW Syntax:

DROP VIEW view\_name

## 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 complicated.

Before talking about the complications of querying for dates, we will look at the most important built-in functions for working with dates.

## MySQL Date Functions

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

|  |  |
| --- | --- |
| **Function** | **Description** |
| NOW() | Returns the current date and time |
| CURDATE() | Returns the current date |
| CURTIME() | Returns the current time |
| DATE() | Extracts the date part of a date or date/time expression |
| EXTRACT() | Returns a single part of a date/time |
| DATE\_ADD() | Adds a specified time interval to a date |
| DATE\_SUB() | Subtracts a specified time interval from a date |
| DATEDIFF() | Returns the number of days between two dates |
| DATE\_FORMAT() | Displays date/time data in different formats |

## SQL Server Date Functions

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

|  |  |
| --- | --- |
| **Function** | **Description** |
| GETDATE() | Returns the current date and time |
| DATEPART() | Returns a single part of a date/time |
| DATEADD() | Adds or subtracts a specified time interval from a date |
| DATEDIFF() | Returns the time between two dates |
| CONVERT() | Displays date/time data in different formats |

## SQL Date Data Types

**MySQL** comes with the following data types for storing a date or a date/time value in the database:

* DATE - format YYYY-MM-DD
* DATETIME - format: YYYY-MM-DD HH:MM:SS
* TIMESTAMP - format: YYYY-MM-DD HH:MM:SS
* YEAR - format YYYY or YY

**SQL Server** comes with the following data types for storing a date or a date/time value in the database:

* DATE - format YYYY-MM-DD
* DATETIME - format: YYYY-MM-DD HH:MM:SS
* SMALLDATETIME - format: YYYY-MM-DD HH:MM:SS
* TIMESTAMP - format: a unique number

**Note:** The date types are chosen for a column when you create a new table in your database! For an overview of all data types available.

## SQL Working with Dates

You can compare two dates easily if there is no time component involved! Assume we have the following "Orders" table:

|  |  |  |
| --- | --- | --- |
| **OrderId** | **ProductName** | **OrderDate** |
| 1 | Geitost | 2008-11-11 |
| 2 | Camembert Pierrot | 2008-11-09 |
| 3 | Mozzarella di Giovanni | 2008-11-11 |
| 4 | Mascarpone Fabioli | 2008-10-29 |

Now we want to select the records with an OrderDate of "2008-11-11" from the table above. We use the following SELECT statement:

SELECT \* FROM Orders WHERE OrderDate='2008-11-11'

The result-set will look like this:

|  |  |  |
| --- | --- | --- |
| **OrderId** | **ProductName** | **OrderDate** |
| 1 | Geitost | 2008-11-11 |
| 3 | Mozzarella di Giovanni | 2008-11-11 |

Now, assume that the "Orders" table looks like this (notice the time component in the "OrderDate" column):

|  |  |  |
| --- | --- | --- |
| **OrderId** | **ProductName** | **OrderDate** |
| 1 | Geitost | 2008-11-11 13:23:44 |
| 2 | Camembert Pierrot | 2008-11-09 15:45:21 |
| 3 | Mozzarella di Giovanni | 2008-11-11 11:12:01 |
| 4 | Mascarpone Fabioli | 2008-10-29 14:56:59 |

If we use the same SELECT statement as above:

SELECT \* FROM Orders WHERE OrderDate='2008-11-11'

we will get no result! This is because the query is looking only for dates with no time portion.

**Tip:** If you want to keep your queries simple and easy to maintain, do not allow time components in your dates!

Definition and Usage

**NOW()** returns the current date and time.

### Syntax

NOW()

Example

The following SELECT statement:

SELECT NOW(),CURDATE(),CURTIME()

will result in something like this:

|  |  |  |
| --- | --- | --- |
| **NOW()** | **CURDATE()** | **CURTIME()** |
| 2008-11-11 12:45:34 | 2008-11-11 | 12:45:34 |

Example

The following SQL creates an "Orders" table with a datetime column (OrderDate):

CREATE TABLE Orders (

OrderId int NOT NULL,

ProductName varchar(50) NOT NULL,

OrderDate datetime NOT NULL DEFAULT NOW(), PRIMARY KEY (OrderId)

)

Notice that the OrderDate column specifies NOW() as the default value. As a result, when you insert a row into the table, the current date and time are automatically inserted into the column.

Now we want to insert a record into the "Orders" table:

INSERT INTO Orders (ProductName) VALUES ('Jarlsberg Cheese') The "Orders" table will now look something like this:

|  |  |  |
| --- | --- | --- |
| **OrderId** | **ProductName** | **OrderDate** |
| 1 | Jarlsberg Cheese | 2008-11-11 13:23:44.657 |

Definition and Usage

**CURDATE()** returns the current date. Syntax

CURDATE()

Example

The following SELECT statement:

SELECT NOW(),CURDATE(),CURTIME()

will result in something like this:

|  |  |  |
| --- | --- | --- |
| **NOW()** | **CURDATE()** | **CURTIME()** |
| 2008-11-11 12:45:34 | 2008-11-11 | 12:45:34 |

Example

The following SQL creates an "Orders" table with a datetime column (OrderDate):

CREATE TABLE Orders (

OrderId int NOT NULL,

ProductName varchar(50) NOT NULL,

OrderDate datetime NOT NULL DEFAULT CURDATE(), PRIMARY KEY (OrderId)

)

Notice that the OrderDate column specifies CURDATE() as the default value. As a result, when you insert a row into the table, the current date are automatically inserted into the column.

Now we want to insert a record into the "Orders" table:

INSERT INTO Orders (ProductName) VALUES ('Jarlsberg Cheese') The "Orders" table will now look something like this:

|  |  |  |
| --- | --- | --- |
| **OrderId** | **ProductName** | **OrderDate** |
| 1 | Jarlsberg Cheese | 2008-11-11 |

Definition and Usage

**CURTIME()** returns the current time.

### Syntax

CURTIME()

Example

The following SELECT statement:

SELECT NOW(),CURDATE(),CURTIME()

will result in something like this:

|  |  |  |
| --- | --- | --- |
| **NOW()** | **CURDATE()** | **CURTIME()** |
| 2008-11-11 12:45:34 | 2008-11-11 | 12:45:34 |

Definition and Usage

The DATE() function extracts the date part of a date or date/time expression.

### Syntax

DATE(date)

Where date is a valid date expression.

### Example

Assume we have the following "Orders" table:

|  |  |  |
| --- | --- | --- |
| **OrderId** | **ProductName** | **OrderDate** |
| 1 | Jarlsberg Cheese | 2008-11-11 13:23:44.657 |

The following SELECT statement:

SELECT ProductName, DATE(OrderDate) AS OrderDate FROM Orders

WHERE OrderId=1 will result in this:

|  |  |
| --- | --- |
| **ProductName** | **OrderDate** |
| Jarlsberg Cheese | 2008-11-11 |

Definition and Usage

The **EXTRACT()** function is used to return a single part of a date/time, such as year, month, day, hour, minute, etc.

### Syntax

EXTRACT(unit FROM date)

Where date is a valid date expression and unit can be one of the following:

|  |
| --- |
| **Unit Value** |
| MICROSECOND |
| SECOND |
| MINUTE |
| HOUR |
| DAY |
| WEEK |
| MONTH |
| QUARTER |
| YEAR |
| SECOND\_MICROSECOND |
| MINUTE\_MICROSECOND |
| MINUTE\_SECOND |
| HOUR\_MICROSECOND |
| HOUR\_SECOND |

|  |
| --- |
| HOUR\_MINUTE |
| DAY\_MICROSECOND |
| DAY\_SECOND |
| DAY\_MINUTE |
| DAY\_HOUR |
| YEAR\_MONTH |

Example

Assume we have the following "Orders" table:

|  |  |  |
| --- | --- | --- |
| **OrderId** | **ProductName** | **OrderDate** |
| 1 | Jarlsberg Cheese | 2008-11-11 13:23:44.657 |

The following SELECT statement:

SELECT EXTRACT(YEAR FROM OrderDate) AS OrderYear, EXTRACT(MONTH FROM OrderDate) AS OrderMonth, EXTRACT(DAY FROM OrderDate) AS OrderDay,

FROM Orders WHERE OrderId=1

will result in this:

|  |  |  |
| --- | --- | --- |
| **OrderYear** | **OrderMonth** | **OrderDay** |
| 2008 | 11 | 11 |

Definition and Usage

The **DATE\_ADD()** function adds a specified time interval to a date.

### Syntax

DATE\_ADD(date,INTERVAL expr type)

Where date is a valid date expression and expr is the number of interval you want to add. type can be one of the following:

|  |
| --- |
| **Type Value** |
| MICROSECOND |
| SECOND |
| MINUTE |
| HOUR |
| DAY |
| WEEK |
| MONTH |
| QUARTER |
| YEAR |
| SECOND\_MICROSECOND |
| MINUTE\_MICROSECOND |
| MINUTE\_SECOND |
| HOUR\_MICROSECOND |
| HOUR\_SECOND |
| HOUR\_MINUTE |
| DAY\_MICROSECOND |
| DAY\_SECOND |
| DAY\_MINUTE |
| DAY\_HOUR |
| YEAR\_MONTH |

### Example

Assume we have the following "Orders" table:

|  |  |  |
| --- | --- | --- |
| **OrderId** | **ProductName** | **OrderDate** |
| 1 | Jarlsberg Cheese | 2008-11-11 13:23:44.657 |

Now we want to add 45 days to the "OrderDate", to find the payment date. We use the following SELECT statement:

SELECT OrderId,DATE\_ADD(OrderDate,INTERVAL 45 DAY) AS OrderPayDate FROM Orders

Result:

|  |  |
| --- | --- |
| **OrderId** | **OrderPayDate** |
| 1 | 2008-12-26 13:23:44.657 |

Definition and Usage

The **DATE\_SUB()** function subtracts a specified time interval from a date.

### Syntax

DATE\_SUB(date,INTERVAL expr type)

Where date is a valid date expression and expr is the number of interval you want to subtract. type can be one of the following:

|  |
| --- |
| **Type Value** |
| MICROSECOND |
| SECOND |
| MINUTE |
| HOUR |
| DAY |
| WEEK |
| MONTH |

|  |
| --- |
| QUARTER |
| YEAR |
| SECOND\_MICROSECOND |
| MINUTE\_MICROSECOND |
| MINUTE\_SECOND |
| HOUR\_MICROSECOND |
| HOUR\_SECOND |
| HOUR\_MINUTE |
| DAY\_MICROSECOND |
| DAY\_SECOND |
| DAY\_MINUTE |
| DAY\_HOUR |
| YEAR\_MONTH |

### Example

Assume we have the following "Orders" table:

|  |  |  |
| --- | --- | --- |
| **OrderId** | **ProductName** | **OrderDate** |
| 1 | Jarlsberg Cheese | 2008-11-11 13:23:44.657 |

Now we want to subtract 5 days from the "OrderDate" date. We use the following SELECT statement:

SELECT OrderId,DATE\_SUB(OrderDate,INTERVAL 5 DAY) AS SubtractDate FROM Orders

Result:

|  |  |
| --- | --- |
| **OrderId** | **SubtractDate** |
| 1 | 2008-11-06 13:23:44.657 |

Definition and Usage

The **DATEDIFF()** function returns the time between two dates.

### Syntax

DATEDIFF(date1,date2)

Where date1 and date2 are valid date or date/time expressions.

**Note:** Only the date parts of the values are used in the calculation.

### Example

The following SELECT statement:

SELECT DATEDIFF('2008-11-30','2008-11-29') AS DiffDate

will result in this:

1

**DiffDate**

### Example

The following SELECT statement:

SELECT DATEDIFF('2008-11-29','2008-11-30') AS DiffDate

will result in this:

-1

**DiffDate**

Definition and Usage

The **DATE\_FORMAT()** function is used to display date/time data in different formats.

### Syntax

DATE\_FORMAT(date,format)

Where date is a valid date and format specifies the output format for the date/time. The formats that can be used are:

|  |  |
| --- | --- |
| **Format** | **Description** |
| %a | Abbreviated weekday name |
| %b | Abbreviated month name |
| %c | Month, numeric |
| %D | Day of month with English suffix |
| %d | Day of month, numeric (00-31) |
| %e | Day of month, numeric (0-31) |
| %f | Microseconds |
| %H | Hour (00-23) |
| %h | Hour (01-12) |
| %I | Hour (01-12) |
| %i | Minutes, numeric (00-59) |
| %j | Day of year (001-366) |
| %k | Hour (0-23) |
| %l | Hour (1-12) |
| %M | Month name |
| %m | Month, numeric (00-12) |
| %p | AM or PM |
| %r | Time, 12-hour (hh:mm:ss AM or PM) |

|  |  |
| --- | --- |
|  |  |
| %S | Seconds (00-59) |
| %s | Seconds (00-59) |
| %T | Time, 24-hour (hh:mm:ss) |
| %U | Week (00-53) where Sunday is the first day of week |
| %u | Week (00-53) where Monday is the first day of week |
| %V | Week (01-53) where Sunday is the first day of week, used with %X |
| %v | Week (01-53) where Monday is the first day of week, used with %x |
| %W | Weekday name |
| %w | Day of the week (0=Sunday, 6=Saturday) |
| %X | Year of the week where Sunday is the first day of week, four digits, used with %V |
| %x | Year of the week where Monday is the first day of week, four digits, used with %v |
| %Y | Year, four digits |
| %y | Year, two digits |

### Example

The following script uses the DATE\_FORMAT() function to display different formats. We will use the NOW() function to get the current date/time:

DATE\_FORMAT(NOW(),'%b %d %Y %h:%i %p') DATE\_FORMAT(NOW(),'%m-%d-%Y') DATE\_FORMAT(NOW(),'%d %b %y') DATE\_FORMAT(NOW(),'%d %b %Y %T:%f')

The result would look something like this: Nov 04 2008 11:45 PM

11-04-2008

04 Nov 08

04 Nov 2008 11:45:34:243

Definition and Usage

The **GETDATE()** function returns the current date and time from the SQL Server.

### Syntax

GETDATE()

### Example

The following SELECT statement: SELECT GETDATE() AS CurrentDateTime will result in something like this:

2008-11-11 12:45:34.243

**CurrentDateTime**

**Note:** The time part above goes all the way to milliseconds.

### Example

The following SQL creates an "Orders" table with a datetime column (OrderDate):

CREATE TABLE Orders (

OrderId int NOT NULL PRIMARY KEY,

ProductName varchar(50) NOT NULL,

OrderDate datetime NOT NULL DEFAULT GETDATE()

)

Notice that the OrderDate column specifies GETDATE() as the default value. As a result, when you insert a row into the table, the current date and time are automatically inserted into the column.

Now we want to insert a record into the "Orders" table:

INSERT INTO Orders (ProductName) VALUES ('Jarlsberg Cheese') The "Orders" table will now look something like this:

|  |  |  |
| --- | --- | --- |
| **OrderId** | **ProductName** | **OrderDate** |
| 1 | Jarlsberg Cheese | 2008-11-11 13:23:44.657 |

Definition and Usage

The **DATEPART()** function is used to return a single part of a date/time, such as year, month, day, hour, minute, etc.

### Syntax

DATEPART(datepart,date)

Where date is a valid date expression and datepart can be one of the following:

|  |  |
| --- | --- |
| **datepart** | **Abbreviation** |
| year | yy, yyyy |
| quarter | qq, q |
| month | mm, m |
| dayofyear | dy, y |
| day | dd, d |
| week | wk, ww |
| weekday | dw, w |
| hour | hh |
| minute | mi, n |
| second | ss, s |
| millisecond | ms |
| microsecond | mcs |
| nanosecond | ns |

### Example

Assume we have the following "Orders" table:

|  |  |  |
| --- | --- | --- |
| **OrderId** | **ProductName** | **OrderDate** |
| 1 | Jarlsberg Cheese | 2008-11-11 13:23:44.657 |

The following SELECT statement:

SELECT DATEPART(yyyy,OrderDate) AS OrderYear, DATEPART(mm,OrderDate) AS OrderMonth, DATEPART(dd,OrderDate) AS OrderDay,

FROM Orders WHERE OrderId=1

will result in this:

|  |  |  |
| --- | --- | --- |
| **OrderYear** | **OrderMonth** | **OrderDay** |
| 2008 | 11 | 11 |

Definition and Usage

The **DATEADD()** function is adds or subtracts a specified time interval from a date.

### Syntax

DATEADD(datepart,number,date)

Where date is a valid date expression and number is the number of interval you want to add. The number can either be positive, for dates in the future, or negative, for dates in the past.

datepart can be one of the following:

|  |  |
| --- | --- |
| **datepart** | **Abbreviation** |
| year | yy, yyyy |
| quarter | qq, q |
| month | mm, m |
| dayofyear | dy, y |
| day | dd, d |
| week | wk, ww |

|  |  |
| --- | --- |
| weekday | dw, w |
| hour | hh |
| minute | mi, n |
| second | ss, s |
| millisecond | ms |
| microsecond | mcs |
| nanosecond | ns |

### Example

Assume we have the following "Orders" table:

|  |  |  |
| --- | --- | --- |
| **OrderId** | **ProductName** | **OrderDate** |
| 1 | Jarlsberg Cheese | 2008-11-11 13:23:44.657 |

Now we want to add 45 days to the "OrderDate", to find the payment date. We use the following SELECT statement:

SELECT OrderId,DATEADD(day,45,OrderDate) AS OrderPayDate FROM Orders

Result:

|  |  |
| --- | --- |
| **OrderId** | **OrderPayDate** |
| 1 | 2008-12-26 13:23:44.657 |

Definition and Usage

The **DATEDIFF()** function returns the time between two dates.

### Syntax

DATEDIFF(datepart,startdate,enddate)

Where startdate and enddate are valid date expressions and datepart can be one of the following:

|  |  |
| --- | --- |
| **datepart** | **Abbreviation** |
| year | yy, yyyy |
| quarter | qq, q |
| month | mm, m |
| dayofyear | dy, y |
| day | dd, d |
| week | wk, ww |
| weekday | dw, w |
| hour | hh |
| minute | mi, n |
| second | ss, s |
| millisecond | ms |
| microsecond | mcs |
| nanosecond | ns |

### Example

Now we want to get the number of days between two dates. We use the following SELECT statement:

SELECT DATEDIFF(day,'2008-06-05','2008-08-05') AS DiffDate

Result:

61

**DiffDate**

### Example

Now we want to get the number of days between two dates (notice that the second date is "earlier" than the first date, and will result in a negative number).

We use the following SELECT statement:

SELECT DATEDIFF(day,'2008-08-05','2008-06-05') AS DiffDate

Result:

-61

**DiffDate**

Definition and Usage

The **CONVERT()** function is a general function for converting data into a new data type. The CONVERT() function can be used to display date/time data in different formats.

### Syntax

CONVERT(data\_type(length),data\_to\_be\_converted,style)

Where data\_type(length) specifies the target data type (with an optional length), data\_to\_be\_converted contains the value to be converted, and style specifies the output format for the date/time.

The styles that can be used are:

|  |  |
| --- | --- |
| **Style ID** | **Style Format** |
| 100 or 0 | mon dd yyyy hh:miAM (or PM) |
| 101 | mm/dd/yy |
| 102 | yy.mm.dd |
| 103 | dd/mm/yy |
| 104 | dd.mm.yy |
| 105 | dd-mm-yy |

|  |  |
| --- | --- |
| 106 | dd mon yy |
| 107 | Mon dd, yy |
| 108 | hh:mm:ss |
| 109 or 9 | mon dd yyyy hh:mi:ss:mmmAM (or PM) |
| 110 | mm-dd-yy |
| 111 | yy/mm/dd |
| 112 | yymmdd |
| 113 or 13 | dd mon yyyy hh:mm:ss:mmm(24h) |
| 114 | hh:mi:ss:mmm(24h) |
| 120 or 20 | yyyy-mm-dd hh:mi:ss(24h) |
| 121 or 21 | yyyy-mm-dd hh:mi:ss.mmm(24h) |
| 126 | yyyy-mm-ddThh:mm:ss.mmm(no spaces) |
| 130 | dd mon yyyy hh:mi:ss:mmmAM |
| 131 | dd/mm/yy hh:mi:ss:mmmAM |

### Example

The following script uses the CONVERT() function to display different formats. We will use the GETDATE() function to get the current date/time:

CONVERT(VARCHAR(19),GETDATE()) CONVERT(VARCHAR(10),GETDATE(),110) CONVERT(VARCHAR(11),GETDATE(),106) CONVERT(VARCHAR(24),GETDATE(),113)

The result would look something like this: Nov 04 2008 11:45 PM

11-04-2008

04 Nov 08

04 Nov 2008 11:45:34:243

NULL values represent missing unknown data. By default, a table column can hold NULL values.

## SQL NULL Values

* If a column in a table is optional, we can insert a new record or update an existing record without adding a value to this column. This means that the field will be saved with a NULL value.
* NULL values are treated differently from other values.
* NULL is used as a placeholder for unknown or inapplicable values.
* **Note:** It is not possible to compare NULL and 0; they are not equivalent.

## SQL Working with NULL Values

Look at the following "Persons" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola |  | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari |  | Stavanger |

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

How can we test for NULL values?

It is not possible to test for NULL values with comparison operators, such as =, <, or <>. We will have to use the IS NULL and IS NOT NULL operators instead.

## SQL IS NULL

How do we select only the records with NULL values in the "Address" column? We will have to use the IS NULL operator:

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

The result-set will look like this:

|  |  |  |
| --- | --- | --- |
| **LastName** | **FirstName** | **Address** |
| Hansen | Ola |  |
| Pettersen | Kari |  |

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

## SQL IS NOT NULL

How do we select only the records with no NULL values in the "Address" column? We will have to use the IS NOT NULL operator:

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

The result-set will look like this:

|  |  |  |
| --- | --- | --- |
| **LastName** | **FirstName** | **Address** |
| Svendson | Tove | Borgvn 23 |

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

Look at the following "Products" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **ProductName** | **UnitPrice** | **UnitsInStock** | **UnitsOnOrder** |
| 1 | Jarlsberg | 10.45 | 16 | 15 |
| 2 | Mascarpone | 32.56 | 23 |  |
| 3 | Gorgonzola | 15.67 | 9 | 20 |

Suppose that the "UnitsOnOrder" column is optional, and may contain NULL values. We have the following SELECT statement:

SELECT ProductName,UnitPrice\*(UnitsInStock+UnitsOnOrder) FROM Products

In the example above, if any of the "UnitsOnOrder" values are NULL, the result is NULL. Microsoft's ISNULL() function is used to specify how we want to treat NULL values.

The NVL(), IFNULL(), and COALESCE() functions can also be used to achieve the same result. In this case we want NULL values to be zero.

Below, if "UnitsOnOrder" is NULL it will not harm the calculation, because ISNULL() returns a zero if the value is NULL:

### SQL Server / MS Access

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

### Oracle

Oracle does not have an ISNULL() function. However, we can use the NVL() function to achieve the same result:

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

### MySQL

MySQL does have an ISNULL() function. However, it works a little bit different from Microsoft's ISNULL() function.

In MySQL we can use the IFNULL() function, like this:

SELECT ProductName,UnitPrice\*(UnitsInStock+IFNULL(UnitsOnOrder,0))

FROM Products

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

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

## Microsoft Access Data Types

Data types and ranges for Microsoft Access, MySQL and SQL Server.

|  |  |  |
| --- | --- | --- |
| **Data type** | **Description** | **Storage** |
| Text | Use for text or combinations of text and numbers. 255 characters maximum |  |
| Memo | Memo is used for larger amounts of text. Stores up to 65,536 characters. **Note:** You cannot sort a memo field. However, they are searchable |  |
| Byte | Allows whole numbers from 0 to 255 | 1 byte |
| Integer | Allows whole numbers between -32,768 and 32,767 | 2 bytes |
| Long | Allows whole numbers between -2,147,483,648 and 2,147,483,647 | 4 bytes |
| Single | Single precision floating-point. Will handle most decimals | 4 bytes |
| Double | Double precision floating-point. Will handle most decimals | 8 bytes |
| Currency | Use for currency. Holds up to 15 digits of whole dollars, plus 4 decimal places. **Tip:** You can choose which country's currency to use | 8 bytes |
| AutoNumber | AutoNumber fields automatically give each record its own number, usually starting at 1 | 4 bytes |

|  |  |  |
| --- | --- | --- |
| Date/Time | Use for dates and times | 8 bytes |
| Yes/No | A logical field can be displayed as Yes/No, True/False, or On/Off. In code, use the constants True and False (equivalent to -1 and 0).**Note:** Null values are not allowed in Yes/No fields | 1 bit |
| Ole Object | Can store pictures, audio, video, or other BLOBs (Binary Large OBjects) | up to 1GB |
| Hyperlink | Contain links to other files, including web pages |  |
| Lookup Wizard | Let you type a list of options, which can then be chosen from a drop-down list | 4 bytes |

## MySQL Data Types

In MySQL there are three main types : text, number, and Date/Time types.

### Text types:

|  |  |
| --- | --- |
| **Data type** | **Description** |
| CHAR(size) | Holds a fixed length string (can contain letters, numbers, and special characters). The fixed size is specified in parenthesis. Can store up to 255 characters |
| VARCHAR(size) | Holds a variable length string (can contain letters, numbers, and special characters). The maximum size is specified in parenthesis. Can store up to 255 characters. **Note:** If you put a greater value than 255 it will be converted to a TEXT type |
| TINYTEXT | Holds a string with a maximum length of 255 characters |
| TEXT | Holds a string with a maximum length of 65,535 characters |
| BLOB | For BLOBs (Binary Large OBjects). Holds up to 65,535 bytes of data |
| MEDIUMTEXT | Holds a string with a maximum length of 16,777,215 characters |
| MEDIUMBLOB | For BLOBs (Binary Large OBjects). Holds up to 16,777,215 bytes of data |
| LONGTEXT | Holds a string with a maximum length of 4,294,967,295 characters |
| LONGBLOB | For BLOBs (Binary Large OBjects). Holds up to 4,294,967,295 bytes of data |
| ENUM(x,y,z,etc.) | Let you enter a list of possible values. You can list up to 65535 values in an ENUM list. If a value is inserted that is not in the list, a blank value will be inserted.  **Note:** The values are sorted in the order you enter them.  You enter the possible values in this format: ENUM('X','Y','Z') |
| SET | Similar to ENUM except that SET may contain up to 64 list items and can store more than one choice |

**Number types:**

|  |  |
| --- | --- |
| **Data type** | **Description** |
| TINYINT(size) | -128 to 127 normal. 0 to 255 UNSIGNED\*. The maximum number of digits may be |

|  |  |
| --- | --- |
|  | specified in parenthesis |
| SMALLINT(size) | -32768 to 32767 normal. 0 to 65535 UNSIGNED\*. The maximum number of digits may be specified in parenthesis |
| MEDIUMINT(size  ) | -8388608 to 8388607 normal. 0 to 16777215 UNSIGNED\*. The maximum number of digits may be specified in parenthesis |
| INT(size) | -2147483648 to 2147483647 normal. 0 to 4294967295 UNSIGNED\*. The maximum number of digits may be specified in parenthesis |
| BIGINT(size) | -9223372036854775808 to 9223372036854775807 normal. 0 to  18446744073709551615 UNSIGNED\*. The maximum number of digits may be specified in parenthesis |
| FLOAT(size,d) | A small number with a floating decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter |
| DOUBLE(size,d) | A large number with a floating decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter |
| DECIMAL(size,d) | A DOUBLE stored as a string , allowing for a fixed decimal point. The maximum number of digits may be specified in the size parameter. The maximum number of digits to the right of the decimal point is specified in the d parameter |

\*The integer types have an extra option called UNSIGNED. Normally, the integer goes from an negative to positive value. Adding the UNSIGNED attribute will move that range up so it starts at zero instead of a negative number.

### Date types:

|  |  |
| --- | --- |
| **Data type** | **Description** |
| DATE() | A date. Format: YYYY-MM-DD  **Note:** The supported range is from '1000-01-01' to '9999-12-31' |
| DATETIME() | \*A date and time combination. Format: YYYY-MM-DD HH:MM:SS  **Note:** The supported range is from '1000-01-01 00:00:00' to '9999-12-31 23:59:59' |
| TIMESTAMP() | \*A timestamp. TIMESTAMP values are stored as the number of seconds since the Unix epoch ('1970-01-01 00:00:00' UTC). Format: YYYY-MM-DD HH:MM:SS  **Note:** The supported range is from '1970-01-01 00:00:01' UTC to '2038-01-09 03:14:07' UTC |
| TIME() | A time. Format: HH:MM:SS  **Note:** The supported range is from '-838:59:59' to '838:59:59' |
| YEAR() | A year in two-digit or four-digit format.  **Note:** Values allowed in four-digit format: 1901 to 2155. Values allowed in two-digit format: 70 to 69, representing years from 1970 to 2069 |

\*Even if DATETIME and TIMESTAMP return the same format, they work very differently. In an INSERT or UPDATE query, the TIMESTAMP automatically set itself to the current date and time.

TIMESTAMP also accepts various formats, like YYYYMMDDHHMMSS, YYMMDDHHMMSS, YYYYMMDD, or YYMMDD.

## SQL Server Data Types

### Character strings:

|  |  |  |
| --- | --- | --- |
| **Data type** | **Description** | **Storage** |
| char(n) | Fixed-length character string. Maximum 8,000 characters | n |
| varchar(n) | Variable-length character string. Maximum 8,000 characters |  |
| varchar(max) | Variable-length character string. Maximum 1,073,741,824 characters |  |
| text | Variable-length character string. Maximum 2GB of text data |  |

**Unicode strings:**

|  |  |  |
| --- | --- | --- |
| **Data type** | **Description** | **Storage** |
| nchar(n) | Fixed-length Unicode data. Maximum 4,000 characters |  |
| nvarchar(n) | Variable-length Unicode data. Maximum 4,000 characters |  |
| nvarchar(max) | Variable-length Unicode data. Maximum 536,870,912 characters |  |
| ntext | Variable-length Unicode data. Maximum 2GB of text data |  |

### Binary types:

|  |  |  |
| --- | --- | --- |
| **Data type** | **Description** | **Storage** |
| bit | Allows 0, 1, or NULL |  |
| binary(n) | Fixed-length binary data. Maximum 8,000 bytes |  |
| varbinary(n) | Variable-length binary data. Maximum 8,000 bytes |  |
| varbinary(max) | Variable-length binary data. Maximum 2GB |  |
| image | Variable-length binary data. Maximum 2GB |  |

**Number types:**

|  |  |  |
| --- | --- | --- |
| **Data type** | **Description** | **Storage** |
| tinyint | Allows whole numbers from 0 to 255 | 1 byte |
| smallint | Allows whole numbers between -32,768 and 32,767 | 2 bytes |

|  |  |  |
| --- | --- | --- |
| int | Allows whole numbers between -2,147,483,648 and 2,147,483,647 | 4 bytes |
| bigint | Allows whole numbers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807 | 8 bytes |
| decimal(p,s) | Fixed precision and scale numbers.  Allows numbers from -10^38 +1 to 10^38 –1.  The p parameter indicates the maximum total number of digits that can be stored (both to the left and to the right of the decimal point). p must be a value from 1 to  38. Default is 18.  The s parameter indicates the maximum number of digits stored to the right of the decimal point. s must be a value from 0 to p. Default value is 0 | 5-17 bytes |
| numeric(p,s  ) | Fixed precision and scale numbers.  Allows numbers from -10^38 +1 to 10^38 –1.  The p parameter indicates the maximum total number of digits that can be stored (both to the left and to the right of the decimal point). p must be a value from 1 to  38. Default is 18.  The s parameter indicates the maximum number of digits stored to the right of the decimal point. s must be a value from 0 to p. Default value is 0 | 5-17 bytes |
| smallmoney | Monetary data from -214,748.3648 to 214,748.3647 | 4 bytes |
| money | Monetary data from -922,337,203,685,477.5808 to 922,337,203,685,477.5807 | 8 bytes |
| float(n) | Floating precision number data from -1.79E + 308 to 1.79E + 308.  The n parameter indicates whether the field should hold 4 or 8 bytes. float(24) holds a 4-byte field and float(53) holds an 8-byte field. Default value of n is 53. | 4 or 8 bytes |
| real | Floating precision number data from -3.40E + 38 to 3.40E + 38 | 4 bytes |

### Date types:

|  |  |  |
| --- | --- | --- |
| **Data type** | **Description** | **Storage** |
| datetime | From January 1, 1753 to December 31, 9999 with an accuracy of 3.33 milliseconds | 8 bytes |
| datetime2 | From January 1, 0001 to December 31, 9999 with an accuracy of 100 nanoseconds | 6-8 bytes |
| smalldatetime | From January 1, 1900 to June 6, 2079 with an accuracy of 1 minute | 4 bytes |
| date | Store a date only. From January 1, 0001 to December 31, 9999 | 3 bytes |
| time | Store a time only to an accuracy of 100 nanoseconds | 3-5 bytes |
| datetimeoffset | The same as datetime2 with the addition of a time zone offset | 8-10  bytes |
| timestamp | Stores a unique number that gets updated every time a row gets created or |  |

|  |  |  |
| --- | --- | --- |
|  | modified. The timestamp value is based upon an internal clock and does not correspond to real time. Each table may have only one timestamp variable |  |

**Other data types:**

|  |  |
| --- | --- |
| **Data type** | **Description** |
| sql\_variant | Stores up to 8,000 bytes of data of various data types, except text, ntext, and timestamp |
| uniqueidentifier | Stores a globally unique identifier (GUID) |
| xml | Stores XML formatted data. Maximum 2GB |
| cursor | Stores a reference to a cursor used for database operations |
| table | Stores a result-set for later processing |

# SQL FUNCTIONS

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

## SQL Aggregate Functions

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

* AVG() - Returns the average value
* COUNT() - Returns the number of rows
* FIRST() - Returns the first value
* LAST() - Returns the last value
* MAX() - Returns the largest value
* MIN() - Returns the smallest value
* SUM() - Returns the sum

## SQL Scalar functions

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

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

## The AVG() Function

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

SELECT AVG(column\_name) FROM table\_name

## SQL AVG() Example

We have the following "Orders" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **O\_Id** | **OrderDate** | **OrderPrice** | **Customer** |
| 1 | 2008/11/12 | 1000 | Hansen |
| 2 | 2008/10/23 | 1600 | Nilsen |
| 3 | 2008/09/02 | 700 | Hansen |
| 4 | 2008/09/03 | 300 | Hansen |
| 5 | 2008/08/30 | 2000 | Jensen |
| 6 | 2008/10/04 | 100 | Nilsen |

Now we want to find the average value of the "OrderPrice" fields. We use the following SQL statement:

SELECT AVG(OrderPrice) AS OrderAverage FROM Orders The result-set will look like this:

950

**OrderAverage**

Now we want to find the customers that have an OrderPrice value higher than the average OrderPrice value.

We use the following SQL statement:

SELECT Customer FROM Orders

WHERE OrderPrice>(SELECT AVG(OrderPrice) FROM Orders) The result-set will look like this:

|  |
| --- |
| **Customer** |
| Hansen |
| Nilsen |
| Jensen |

## SQL COUNT

The COUNT() function returns the number of rows that matches a specified criteria. SQL COUNT(column\_name) Syntax

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

SELECT COUNT(column\_name) FROM table\_name SQL COUNT(\*) Syntax

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

SELECT COUNT(\*) FROM table\_name

SQL COUNT(DISTINCT column\_name) Syntax

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

SELECT COUNT(DISTINCT column\_name) FROM table\_name

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

## SQL COUNT(column\_name) Example

We have the following "Orders" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **O\_Id** | **OrderDate** | **OrderPrice** | **Customer** |
| 1 | 2008/11/12 | 1000 | Hansen |
| 2 | 2008/10/23 | 1600 | Nilsen |
| 3 | 2008/09/02 | 700 | Hansen |
| 4 | 2008/09/03 | 300 | Hansen |
| 5 | 2008/08/30 | 2000 | Jensen |
| 6 | 2008/10/04 | 100 | Nilsen |

Now we want to count the number of orders from "Customer Nilsen". We use the following SQL statement:

SELECT COUNT(Customer) AS CustomerNilsen FROM Orders WHERE Customer='Nilsen'

The result of the SQL statement above will be 2, because the customer Nilsen has made 2 orders in total:

2

**CustomerNilsen**

## SQL COUNT(\*) Example

If we omit the WHERE clause, like this:

SELECT COUNT(\*) AS NumberOfOrders FROM Orders The result-set will look like this:

**NumberOfOrders**

6

which is the total number of rows in the table.

## SQL COUNT(DISTINCT column\_name) Example

Now we want to count the number of unique customers in the "Orders" table. We use the following SQL statement:

SELECT COUNT(DISTINCT Customer) AS NumberOfCustomers FROM Orders The result-set will look like this:

3

**NumberOfCustomers**

which is the number of unique customers (Hansen, Nilsen, and Jensen) in the "Orders" table.

## The FIRST() Function

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

SELECT FIRST(column\_name) FROM table\_name

## SQL FIRST() Example

We have the following "Orders" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **O\_Id** | **OrderDate** | **OrderPrice** | **Customer** |
| 1 | 2008/11/12 | 1000 | Hansen |
| 2 | 2008/10/23 | 1600 | Nilsen |
| 3 | 2008/09/02 | 700 | Hansen |
| 4 | 2008/09/03 | 300 | Hansen |
| 5 | 2008/08/30 | 2000 | Jensen |
| 6 | 2008/10/04 | 100 | Nilsen |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

Now we want to find the first value of the "OrderPrice" column. We use the following SQL statement:

SELECT FIRST(OrderPrice) AS FirstOrderPrice FROM Orders **Tip:** Workaround if FIRST() function is not supported: SELECT OrderPrice FROM Orders ORDER BY O\_Id LIMIT 1 The result-set will look like this:

1000

**FirstOrderPrice**

## The LAST() Function

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

SELECT LAST(column\_name) FROM table\_name

## SQL LAST() Example

We have the following "Orders" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **O\_Id** | **OrderDate** | **OrderPrice** | **Customer** |
| 1 | 2008/11/12 | 1000 | Hansen |
| 2 | 2008/10/23 | 1600 | Nilsen |
| 3 | 2008/09/02 | 700 | Hansen |
| 4 | 2008/09/03 | 300 | Hansen |
| 5 | 2008/08/30 | 2000 | Jensen |
| 6 | 2008/10/04 | 100 | Nilsen |

Now we want to find the last value of the "OrderPrice" column. We use the following SQL statement:

SELECT LAST(OrderPrice) AS LastOrderPrice FROM Orders

**Tip:** Workaround if LAST() function is not supported:

SELECT OrderPrice FROM Orders ORDER BY O\_Id DESC LIMIT 1 The result-set will look like this:

100

**LastOrderPrice**

## The MAX() Function

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

SELECT MAX(column\_name) FROM table\_name

## SQL MAX() Example

We have the following "Orders" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **O\_Id** | **OrderDate** | **OrderPrice** | **Customer** |
| 1 | 2008/11/12 | 1000 | Hansen |
| 2 | 2008/10/23 | 1600 | Nilsen |
| 3 | 2008/09/02 | 700 | Hansen |
| 4 | 2008/09/03 | 300 | Hansen |
| 5 | 2008/08/30 | 2000 | Jensen |
| 6 | 2008/10/04 | 100 | Nilsen |

Now we want to find the largest value of the "OrderPrice" column. We use the following SQL statement:

SELECT MAX(OrderPrice) AS LargestOrderPrice FROM Orders The result-set will look like this:

2000

**LargestOrderPrice**

## The MIN() Function

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

SELECT MIN(column\_name) FROM table\_name

## SQL MIN() Example

We have the following "Orders" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **O\_Id** | **OrderDate** | **OrderPrice** | **Customer** |
| 1 | 2008/11/12 | 1000 | Hansen |
| 2 | 2008/10/23 | 1600 | Nilsen |
| 3 | 2008/09/02 | 700 | Hansen |
| 4 | 2008/09/03 | 300 | Hansen |
| 5 | 2008/08/30 | 2000 | Jensen |
| 6 | 2008/10/04 | 100 | Nilsen |

Now we want to find the smallest value of the "OrderPrice" column. We use the following SQL statement:

SELECT MIN(OrderPrice) AS SmallestOrderPrice FROM Orders The result-set will look like this:

100

**SmallestOrderPrice**

## The SUM() Function

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

SELECT SUM(column\_name) FROM table\_name

## SQL SUM() Example

We have the following "Orders" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **O\_Id** | **OrderDate** | **OrderPrice** | **Customer** |
| 1 | 2008/11/12 | 1000 | Hansen |
| 2 | 2008/10/23 | 1600 | Nilsen |
| 3 | 2008/09/02 | 700 | Hansen |
| 4 | 2008/09/03 | 300 | Hansen |
| 5 | 2008/08/30 | 2000 | Jensen |
| 6 | 2008/10/04 | 100 | Nilsen |

Now we want to find the sum of all "OrderPrice" fields". We use the following SQL statement:

SELECT SUM(OrderPrice) AS OrderTotal FROM Orders The result-set will look like this:

5700

**OrderTotal**

Aggregate functions often need an added GROUP BY statement.

## The GROUP BY Statement

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

SQL GROUP BY Syntax

SELECT column\_name, aggregate\_function(column\_name) FROM table\_name

WHERE column\_name operator value GROUP BY column\_name

## SQL GROUP BY Example

We have the following "Orders" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **O\_Id** | **OrderDate** | **OrderPrice** | **Customer** |
| 1 | 2008/11/12 | 1000 | Hansen |
| 2 | 2008/10/23 | 1600 | Nilsen |
| 3 | 2008/09/02 | 700 | Hansen |
| 4 | 2008/09/03 | 300 | Hansen |
| 5 | 2008/08/30 | 2000 | Jensen |
| 6 | 2008/10/04 | 100 | Nilsen |

Now we want to find the total sum (total order) of each customer.

We will have to use the GROUP BY statement to group the customers. We use the following SQL statement:

SELECT Customer,SUM(OrderPrice) FROM Orders GROUP BY Customer

The result-set will look like this:

|  |  |
| --- | --- |
| **Customer** | **SUM(OrderPrice)** |
| Hansen | 2000 |
| Nilsen | 1700 |
| Jensen | 2000 |

Nice! Isn't it? :)

Let's see what happens if we omit the GROUP BY statement:

SELECT Customer,SUM(OrderPrice) FROM Orders The result-set will look like this:

|  |  |
| --- | --- |
| **Customer** | **SUM(OrderPrice)** |
| Hansen | 5700 |
| Nilsen | 5700 |
| Hansen | 5700 |
| Hansen | 5700 |
| Jensen | 5700 |
| Nilsen | 5700 |

The result-set above is not what we wanted.

**Explanation of why the above SELECT statement cannot be used:** The SELECT statement above has two columns specified (Customer and SUM(OrderPrice). The "SUM(OrderPrice)" returns a single value (that is the total sum of the "OrderPrice" column), while "Customer" returns 6 values (one value for each row in the "Orders" table). This will therefore not give us the correct result. However, you have seen that the GROUP BY statement solves this problem.

## GROUP BY More Than One Column

We can also use the GROUP BY statement on more than one column, like this: SELECT Customer,OrderDate,SUM(OrderPrice) FROM Orders

GROUP BY Customer,OrderDate

## The UCASE() Function

The UCASE() function converts the value of a field to uppercase. SQL UCASE() Syntax

SELECT UCASE(column\_name) FROM table\_name

Syntax for SQL Server

SELECT UPPER(column\_name) FROM table\_name

## SQL UCASE() Example

We have the following "Persons" table:

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

Now we want to select the content of the "LastName" and "FirstName" columns above, and convert the "LastName" column to uppercase.

We use the following SELECT statement:

SELECT UCASE(LastName) as LastName,FirstName FROM Persons The result-set will look like this:

|  |  |
| --- | --- |
| **LastName** | **FirstName** |
| HANSEN | Ola |
| SVENDSON | Tove |
| PETTERSEN | Kari |

## The LCASE() Function

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

SELECT LCASE(column\_name) FROM table\_name

Syntax for SQL Server

SELECT LOWER(column\_name) FROM table\_name

## SQL LCASE() Example

We have the following "Persons" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

Now we want to select the content of the "LastName" and "FirstName" columns above, and convert the "LastName" column to lowercase.

We use the following SELECT statement:

SELECT LCASE(LastName) as LastName,FirstName FROM Persons The result-set will look like this:

|  |  |
| --- | --- |
| **LastName** | **FirstName** |
| hansen | Ola |
| svendson | Tove |
| pettersen | Kari |

## The MID() Function

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

SELECT MID(column\_name,start[,length]) FROM table\_name

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

## SQL MID() Example

We have the following "Persons" table:

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

Now we want to extract the first four characters of the "City" column above. We use the following SELECT statement:

SELECT MID(City,1,4) as SmallCity FROM Persons The result-set will look like this:

|  |
| --- |
| **SmallCity** |
| Sand |
| Sand |
| Stav |

## The LEN() Function

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

SELECT LEN(column\_name) FROM table\_name

## SQL LEN() Example

We have the following "Persons" table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **P\_Id** | **LastName** | **FirstName** | **Address** | **City** |
| 1 | Hansen | Ola | Timoteivn 10 | Sandnes |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| 2 | Svendson | Tove | Borgvn 23 | Sandnes |
| 3 | Pettersen | Kari | Storgt 20 | Stavanger |

Now we want to select the length of the values in the "Address" column above. We use the following SELECT statement:

SELECT LEN(Address) as LengthOfAddress FROM Persons The result-set will look like this:

|  |
| --- |
| **LengthOfAddress** |
| 12 |
| 9 |
| 9 |

## The ROUND() Function

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

SELECT ROUND(column\_name,decimals) FROM table\_name

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| column\_name | Required. The field to round. |
| decimals | Required. Specifies the number of decimals to be returned. |

## SQL ROUND() Example

We have the following "Products" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Prod\_Id** | **ProductName** | **Unit** | **UnitPrice** |
| 1 | Jarlsberg | 1000 g | 10.45 |
| 2 | Mascarpone | 1000 g | 32.56 |
| 3 | Gorgonzola | 1000 g | 15.67 |

Now we want to display the product name and the price rounded to the nearest integer. We use the following SELECT statement:

SELECT ProductName, ROUND(UnitPrice,0) as UnitPrice FROM Products The result-set will look like this:

|  |  |
| --- | --- |
| **ProductName** | **UnitPrice** |
| Jarlsberg | 10 |
| Mascarpone | 33 |
| Gorgonzola | 16 |

## The NOW() Function

The NOW() function returns the current system date and time. SQL NOW() Syntax

SELECT NOW() FROM table\_name

## SQL NOW() Example

We have the following "Products" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Prod\_Id** | **ProductName** | **Unit** | **UnitPrice** |
| 1 | Jarlsberg | 1000 g | 10.45 |
| 2 | Mascarpone | 1000 g | 32.56 |
| 3 | Gorgonzola | 1000 g | 15.67 |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

Now we want to display the products and prices per today's date. We use the following SELECT statement:

SELECT ProductName, UnitPrice, Now() as PerDate FROM Products The result-set will look like this:

|  |  |  |
| --- | --- | --- |
| **ProductName** | **UnitPrice** | **PerDate** |
| Jarlsberg | 10.45 | 10/7/2008 11:25:02 AM |
| Mascarpone | 32.56 | 10/7/2008 11:25:02 AM |
| Gorgonzola | 15.67 | 10/7/2008 11:25:02 AM |

## The FORMAT() Function

The FORMAT() function is used to format how a field is to be displayed. SQL FORMAT() Syntax

SELECT FORMAT(column\_name,format) FROM table\_name

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| column\_name | Required. The field to be formatted. |
| format | Required. Specifies the format. |

## SQL FORMAT() Example

We have the following "Products" table:

|  |  |  |  |
| --- | --- | --- | --- |
| **Prod\_Id** | **ProductName** | **Unit** | **UnitPrice** |
| 1 | Jarlsberg | 1000 g | 10.45 |
| 2 | Mascarpone | 1000 g | 32.56 |
| 3 | Gorgonzola | 1000 g | 15.67 |

Now we want to display the products and prices per today's date (with today's date displayed in the following format "YYYY-MM-DD").

We use the following SELECT statement:

SELECT ProductName, UnitPrice, FORMAT(Now(),'YYYY-MM-DD') as PerDate FROM Products

The result-set will look like this:

|  |  |  |
| --- | --- | --- |
| **ProductName** | **UnitPrice** | **PerDate** |
| Jarlsberg | 10.45 | 2008-10-07 |
| Mascarpone | 32.56 | 2008-10-07 |
| Gorgonzola | 15.67 | 2008-10-07 |