

Introduction to SQL

SQL is a standard language for accessing and manipulating databases.

What is SQL?

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

What can SQL do?

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

SQL syntax

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) of data.
An example of a table called "People":

P_ID	Last Name	First name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	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, Name, 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 selects all records in the table "People":

```
SELECT * FROM People
```

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

Keep in mind that...

- SQL is not case sensitive
-

Comma after the SQL statements?

Some systems databases require a comma 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 assume use MS Access and SQL Server 2000 and we do not have to put a comma after each SQL statement, but some database programs forces you to use it.

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	

SQL DDL and DML

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

DDL part to create or drop tables and databases. It also allows you to define keys, specify links between tables, and impose constraints between tables. The most important DDL statements in are:

- CREATE DATABASE - creates a new database
- ALTER DATABASE - change a database
- CREATE TABLE - creates a new table
- ALTER TABLE - modifies a table
- DROP TABLE - remove a table
- CREATE INDEX - creates an index (search key)
- DROP INDEX - deletes an index

Query and update commands form the DML part of SQL:

- SELECT - extracts data in a database
- UPDATE - updates data in a database
- DELETE - deletes data from a database
- INSERT INTO - inserts data into a database for new

Data Definition Language (DDL)

SQL CREATE DATABASE Statement

- SQL CREATE DATABASE Syntax

CREATE DATABASE *dbname*;

- SQL CREATE DATABASE Example

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

CREATE DATABASE my_db;

SQL DROP DATABASE Statement

The DROP DATABASE statement is used to delete a database.

DROP DATABASE *database_name*

The SQL CREATE TABLE Statement

- SQL CREATE TABLE Syntax

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

CREATE TABLE *table_name*

(
column_name1 data_type(size),
column_name2 data_type(size),
column_name3 data_type(size),
....
);

- SQL CREATE TABLE Example

We want to create a table called "Employees" that contains five attributes: P_ID, Name, Address, and City.

We use the following CREATE TABLE statement:

CREATE TABLE Employees

(
P_Id int,
LastName varchar(255),

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

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

PersonID	LastName	FirstName	Address	City

The SQL 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 DROP TABLE Statement

The DROP TABLE statement is used to delete a table.

```
DROP TABLE table_name
```

SQL 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 "Employees" 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 "Employees" table.

We use the following SQL statement:

```
ALTER TABLE Employees  
ADD DateOfBirth date
```

The "Employees" table will now look 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 "Employees" table.

We use the following SQL statement:

```
ALTER TABLE Employees  
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 "Employees" table.

We use the following SQL statement:

```
ALTER TABLE Employees  
DROP COLUMN DateOfBirth
```

The "Employees" 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

SQL CREATE INDEX Statement

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

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

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.

- **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)
```

- **CREATE INDEX Example**

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

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

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

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

The DROP INDEX Statement

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

DROP INDEX Syntax for MS SQL Server:

```
DROP INDEX table_name.index_name
```

SQL constraints

Constraints can be specified when creating a table (with the CREATE TABLE statement) or after the creation (with the ALTER TABLE statement).

We will focus on the following constraints:

- NOT NULL
- PRIMARY KEY
- FOREIGN KEY
- CHECK
- DEFAULT

- **SQL NOT NULL constraint**

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

- **Primary key Contrainte**

PRIMARY KEY constraint uniquely identifies each record in a table. Primary keys must contain unique values, not zero. Each table must have one and only one primary key.

```
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 the appointment of a primary key constraint and definition of a PRIMARY KEY constraint on multiple columns, use the following SQL syntax:

```
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)
)
```

- ***PRIMARY KEY constraint with ALTER TABLE SQL***

To create a primary key constraint when the table is already created, use the following SQL statement:

```
ALTER TABLE Persons
ADD PRIMARY KEY (P_Id)
```

To name the constraint:

```
ALTER TABLE Persons
ADD CONSTRAINT pk_PersonID PRIMARY KEY (P_Id,LastName)
```

- ***To remove a primary key constraint***

To delete a PRIMARY KEY constraint, use the following SQL statement:

```
ALTER TABLE Persons
DROP CONSTRAINT pk_PersonID
```

- ***FOREIGN KEY constraint***

A foreign key in one table refers to a primary key in another table.

Example: The attribute "P_ID" table "Orders" the tip "P_ID" attribute of the "Persons" table.

The "P_ID" attribute of the "Persons" table is the primary key. The "P_ID" attribute of the "Orders" table is a foreign key.

FOREIGN KEY constraint prevents invalid data is inserted into the foreign key column. The value of a foreign key must be present among the values it points to another table.

- ***Foreign key constraint with the CREATE TABLE***

The following SQL statement creates a foreign key to "P_ID" when the table "Orders" is created:

```
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:

```
CREATE TABLE Orders
(
  O_id int NOT NULL,
  OrderNo int NOT NULL,
  P_ID int,
  PRIMARY KEY (o_id)
  Fk_PerOrders CONSTRAINT FOREIGN KEY (P_ID) REFERENCES Persons (P_ID)
)
```

- ***Foreign key constraint with ALTER TABLE***

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

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

To name the constraint:

```
ALTER TABLE Orders
Fk_PerOrders ADD CONSTRAINT FOREIGN KEY (P_ID) REFERENCES Persons
(P_ID)
```

- ***To remove a foreign key constraint***

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

- **CHECK constraint**

CHECK constraint is used to limit the value range of an attribute.

- ***CHECK constraint with CREATE TABLE***

```
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:

```
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')
)
```

- ***CHECK constraint with ALTER TABLE***

```
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:

```
ALTER TABLE Persons
ADD CONSTRAINT chk_Person CHECK (P_Id>0 AND City='Sandnes')
```

- ***To delete a CHECK constraint***

To delete a CHECK constraint, use the following SQL statement:

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

- ***DEFAULT constraint***

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

- ***DEFAULT constraint with CREATE TABLE***

The following SQL statement creates a DEFAULT constraint on the attribute "City" when the table "People" is created:

```
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, using functions like GETDATE ():

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

- ***DEFAULT constraint with ALTER TABLE***

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

```
ALTER TABLE Persons
ALTER COLUMN City SET DEFAULT 'SANDNES'
```

- ***To delete a DEFAULT constraint***

To delete a default constraint, use the following SQL statement:

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

The Data Manipulation Language (DML)

Query and update commands form the DML part of SQL:

- SELECT - extracts data in a database
- UPDATE - updates data in a database
- DELETE - deletes data from a database
- INSERT INTO - inserts data into a database for new

The 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 syntax**

```
SELECT column_name(s)
FROM table_name
and
SELECT * FROM table_name
```

- **SQL example**

We want to select the contents of the columns "Last_Name" and "First_Name" in the "Persons" table.

We use the following SELECT statement:

```
SELECT Name, Surname FROM Persons
```

results:

Last_Name	First_Name
Hansen	Ola
Svendsen	Tove
Pettersen	Kari

- **Example of SELECT ***

We want to select all columns from the "Persons" table.

We use the following SELECT statement:

```
SELECT * FROM Persons
```

Tip: The asterisk (*) is a quick way to select all columns!

R

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

- **The SELECT DISTINCT**

In a table, some columns may contain duplicate values. This is not a problem; however, sometimes you want to list the different values.

The DISTINCT keyword can be used to return distinct values.

SQL syntax

```
SELECT DISTINCT column_name (s)
FROM table_name
```

- **Example SELECT DISTINCT**

We want to select only the distinct values in the column "City" from the "Persons" table.

We use the following SELECT statement:

```
SELECT DISTINCT City FROM Persons
```

result:

City
Sandnes
Stavanger

- **The WHERE clause**

The WHERE clause is used to extract only those records that meet the specified criteria.

SQL syntax

```
SELECT column_name (s)
```

```
FROM table_name
```

```
WHERE column_name operator value
```

- **Example of Where**

We want to select the persons living in the city "Sandnes" from the table above.

We use the following query:

```
SELECT * FROM Persons
```

```
WHERE City = 'Sandnes'
```

result:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	Tove	Borgvn 23	Sandnes

Quotes around text fields

SQL uses single quotes around text values (most database systems will also accept the quotes).

The numerical values should not be quoted.

For text values:

This query is correct:

```
SELECT * FROM Persons WHERE Name = 'Tove'
```

this is incorrect:

```
SELECT * FROM Persons WHERE FirstName = Tove
```

For numeric values:

This query is correct:

```
SELECT * FROM Persons WHERE Year = 1965
```

this is incorrect:

```
SELECT * FROM Persons WHERE Year = '1965'
```

Licensed operators in the WHERE clause

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

Operator	Description
=	equality
<>	different
>	Greater than
<	Smaller than
> =	Greater than or equal
<=	Or less
BETWEEN	Between an inclusive range
LIKE	Search for 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 can be written as !=

Operators AND & OR

The AND operator displays a record if the first and the second condition is true.

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

Example with the AND operator

We do not want to select that people with the first name "Tove" and the name "Svendson"

We use the following SELECT statement:

```
SELECT * FROM Persons
```

```
WHERE Name = 'Tove' AND name = 'Svendson'
```

result:

P_ID	Last_Name	First_Name	Address	City
2	Svendson	Tove	Borgvn 23	Sandnes

Example with OR

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

We use the following SELECT statement:

```
SELECT * FROM Persons
```

```
WHERE Name = 'Tove' OR Name = 'Ola'
```

result:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes

2	Svendsen	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 the person whose name is equal to "Svendsen" and the first name equal to "Tove" or "Ola":

We use the following SELECT statement:

```
SELECT * FROM Persons WHERE
```

```
Name = 'Svendson'
```

```
AND (Name = 'Tove' OR FirstName = 'Ola')
```

result:

P_ID	Last_Name	First_Name	Address	City
2	Svendsen	Tove	Borgvn 23	Sandnes

The ORDER BY keyword

The key word ORDER BY is used to sort the results by a specified column. By default, it sorts the records in ascending order.

If you want to sort the records in descending order, you can use the DESC keyword.

SQL syntax

```
SELECT column_name (s)
```

```
FROM table_name
```

```
ORDER BY column_name (s) ASC | DESC
```

ORDER BY Example

"Persons" table:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	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 people by their name.

We use the following SELECT statement:

```
SELECT * FROM Persons
```

```
ORDER BY Name
```

result:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
4	Nilsen	Tom	Vingvn 23	Stavanger
3	Pettersen	Kari	Storgt 20	Stavanger
2	Svendsen	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 order by their last name.

We use the following SELECT statement:

```
SELECT * FROM Persons
```

```
ORDER BY Name DESC
```

result:

P_ID	Last_Name	First_Name	Address	City
2	Svendsen	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger
4	Nilsen	Tom	Vingvn 23	Stavanger
1	Hansen	Ola	Timoteivn 10	Sandnes

The UPDATE Statement

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 rows should be updated. If you omit the WHERE clause, all records will be updated!

- Update Example

The table "Persons":

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

4	Nilsen	Johan	Bakken 2	Stavanger
5	Tjessem	Paul		

Now we want to update the person "Tjessem, Paul."

We use the following SQL statement:

UPDATE People

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

WHERE name = 'Tjessem' AND Name = 'Paul'

The table becomes:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger
4	Nilsen	Johan	Bakken 2	Stavanger
5	Tjessem	Paul	Nissestien 67	Sandnes

The DELETE statement

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

- Syntax SQL

DELETE FROM table_name

WHERE some_column=some_value

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

- SQL Delete Example

The table "Persons":

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger
4	Nilsen	Johan	Bakken 2	Stavanger
5	Tjessem	Paul	Nissestien 67	Sandnes

Now we want to remove the person "Tjessem, Paul" the table.

We use the following SQL statement:

DELETE FROM Persons
WHERE name = 'Tjessem' AND Name = 'Paul'
The table becomes:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger
4	Nilsen	Johan	Bakken 2	Stavanger

- **Delete all lines**

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

DELETE FROM table_name

or

DELETE * FROM table_name

The INSERT INTO statement

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

- **SQL syntax**

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

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

INSERT INTO table_name

VALUES (value1, value2, value3, ...)

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

INSERT INTO table_name (column1, column2, column3, ...)

VALUES (value1, value2, value3, ...)

- **SQL Example**

We want to insert a new row in the "Persons" table.

We use the following SQL statement:

INSERT INTO People

VALUES (4, 'Nilsen', 'Johan', 'Bakken 2', 'Stavanger')

the person table is now:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger
4	Nilsen	Johan	Bakken 2	Stavanger

Insert data only in the specified columns

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

The following SQL statement will add a new line, but only in the "P_ID", "Name" and "Name":

```
INSERT INTO Persons (P_ID, Last Name, First Name)
VALUES (5, 'Tjessem', 'Paul')
```

The table is now:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger
4	Nilsen	Johan	Bakken 2	Stavanger
5	Tjessem	Paul		

The TOP clause

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

The TOP clause can be very useful on large tables with thousands of records.

SQL syntax

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

```
FROM table_name
```

- **SQL Syntax**

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

```
FROM table_name;
```

- **Example SQL TOP**

The table "Person":

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger
4	Nilsen	Tom	Vingvn 23	Stavanger

Now we want to select the first two records in the table above.

We use the following SELECT statement:

```
SELECT TOP 2 * FROM Persons
```

result:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	Tove	Borgvn 23	Sandnes

- TOP PERCENTAGE

Now we want to select 50% of the records in the table above.

We use the following SELECT statement:

SELECT TOP 50 PERCENT * FROM Persons

Result:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	Tove	Borgvn 23	Sandnes

The LIKE operator

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

- SQL syntax

SELECT column_name (s)

FROM table_name

WHERE column_name LIKE pattern

- SQL Example

We do not want to select only those living in a city that starts with "s" Person table.

We use the following SELECT statement:

SELECT * FROM Persons WHERE City LIKE '% s'

A "%" can be used to define wildcards (missing letters in the model) before and after the pattern.

Result:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	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".

We use the following SELECT statement:

SELECT * FROM Persons WHERE City LIKE '% s'

result:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	Tove	Borgvn 23	Sandnes

Next we want to select the persons living in a city that contains the string "TAV".

We use the following SELECT statement:

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

result:

P_ID	Last_Name	First_Name	Address	City
3	Pettersen	Kari	Storgt 20	Stavanger

It is also possible to select the persons living in a city that does not contain the string "TAV", using the NOT keyword.

We use the following SELECT statement:

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

result:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	Tove	Borgvn 23	Sandnes

SQL Wildcards

The SQL wildcard can replace one or more characters when searching for data in a database.

The SQL wildcard must be used with the SQL LIKE operator.

With SQL, the following wildcards can be used:

Wildcard	Description
%	To match 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

Example:

We have the following records in the table "Person":

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes

2	Svendsen	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 "her."

We use the following SELECT statement:

SELECT * FROM Persons WHERE City LIKE 'his%'

result:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	Tove	Borgvn 23	Sandnes

Next we want to select the persons living in a city that contains the pattern 'nos'.

We use the following SELECT statement:

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

result:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	Tove	Borgvn 23	Sandnes

Using the wildcard _

Now we want to select those with a first name that begins with any character, followed by ".".

We use the following SELECT statement:

SELECT * FROM Persons WHERE Name LIKE '_la'

result:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes

Next we want to select people with a family name starting with "S" followed by any character, followed by "end" followed by any character, followed by "in".

We use the following SELECT statement:

SELECT * FROM Persons WHERE Name LIKE 'S_end_en'

result:

P_ID	Last_Name	First_Name	Address	City
2	Svendsen	Tove	Borgvn 23	Sandnes

Using [charlist] Wildcard

Now we want to select people with a name that starts with "b" or "s" or "p".

We use the following SELECT statement:

```
SELECT * FROM Persons
WHERE Last_Name LIKE '[bsp]%'
```

result:

P_ID	Last_Name	First_Name	Address	City
2	Svendsen	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Next we want to select people with a family name that does not begin with "b" or "s" or "P".

We use the following SELECT statement:

```
SELECT * FROM Persons
WHERE Name LIKE '[! bsp]%'
```

result:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes

The IN operator

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

- **SQL syntax**

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

- **SQL Example**

We want to select the people with a name equal to "Hansen" or "Pettersen" from the table above.

We use the following SELECT statement:

```
SELECT * FROM Persons
WHERE name IN ('Hansen', 'Pettersen')
```

result:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

The BETWEEN operator

The BETWEEN operator selects a range of data between two values. The values can be numbers, text, or dates.

- **SQL syntax**

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

- SQL Example 1

We want to select the people with a name between "Hansen" and "Pettersen" (in alphabetical order) of the "People" table.

We use the following SELECT statement:

```
SELECT * FROM Persons WHERE Last_Name BETWEEN 'Hansen' AND 'Pettersen'
```

Result with SQL Server 2008:

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Note: The BETWEEN operator is treated differently in different databases! In some databases, persons with the name "Hansen" or "Pettersen" will not be listed as the operator only selects fields that are between and excluding the test values.

In other databases, persons with the name "Hansen" or "Pettersen" will be listed as the operator enters selects fields that are between and including the test values, as in SQL server.

And other bases, people with the name "Hansen" will be listed, but "Pettersen" will not be listed because the operator enters selects fields between the test values, including the value of test first and excluding the value of the last test.

Therefore: Check how your database treats the BETWEEN operator.

- SQL Example 2

To show people outside the range in the previous example, use NOT BETWEEN:

```
SELECT * FROM Persons WHERE Name NOT BETWEEN 'Hansen' AND 'Pettersen'
```

result:

P_ID	Last_Name	First_Name	Address	City
2	Svendsen	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

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 table names very long or complex or column names.

An alias name can be anything, but usually it is short.

- **SQL syntax**

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

- **SQL Alias Syntax for Columns**

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

- **SQL Example**

Suppose we have a table called "Persons" and another table called "Product_Orders." We will give the table alias "p" and "PO", respectively.

Now we want all levels including "Ola Hansen" is responsible for the list.

We use the following SELECT statement:

```
SELECT po.OrderID, p.Nom, p.Prénom FROM Persons AS p, Product_Orders AS po
WHERE p.Nom = 'Hansen' AND p.Prénom = 'Ola'
```

The same SELECT statement without an alias:

```
SELECT Product_Orders.OrderID, Personnes.Nom, Personnes.Prénom
FROM People, Product_Orders WHERE Personnes.Nom = 'Hansen' AND
Personnes.Prénom = 'Ola'
```

As you can see from the two SELECT statements above; Aliases are used to make queries easier to write and read.

Joint

The join is used to query data from two or more tables based on a relationship between certain columns in these tables.

The tables in a database are often related to one another.

A primary key is a column (or set of columns) with a single value for each line. Each primary key value must be unique in the table. The purpose is to link together the data, through the tables, without repeating the entire data in each table.

Look at the table "persons":

P_ID	Last_Name	First_Name	Address	City
1	Hansen	Ola	Timoteivn 10	Sandnes
2	Svendsen	Tove	Borgvn 23	Sandnes
3	Pettersen	Kari	Storgt 20	Stavanger

Note that the "P_ID" column is the primary key of the "Persons" table. This means that two lines cannot have the same P_ID. The P_ID are two people even if they have the same name. Then 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 attribute "O_ID" is the primary key in the table "orders" and "P_ID" refers to people in the "Persons" table without using their names. Note that the relationship between the two tables above is the attribute "P_ID."

The UNION operator

The UNION operator is used to combine the results of two or more SELECT statements. Note that each SELECT statement within the Union must have the same number of columns. The columns must also have compatible data types.

- **SQL syntax**
SELECT column_name (s) FROM tbl_name1
UNION
SELECT column_name (s) FROM tbl_name2

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

```
SELECT column_name (s) FROM tbl_name1
UNION ALL
SELECT column_name (s) FROM tbl_name2
```

PS: The column names in the result are always equal to the column names in the first SELECT of the UNION.

- **SQL Example**
Look at the following tables:
"Employees_Norway"

E_ID	E_Name
01	Hansen, Ola
02	Svendsen, Tove
03	Svendsen, Stephen
04	Pettersen, Kari

"Employees_USA" :

E_ID	E_Name
01	Turner, Sally
02	Kent, Clark
03	Svensen, Stephen
04	Scott, Stephen

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

```
SELECT FROM E_Name Employees_Norway
UNION
SELECT FROM E_Name Employees_USA
```

The result set is as follows:

E_Name
Hansen, Ola
Svensen, Tove
Svensen, 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.

- **Example UNION ALL**

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
Svensen, Tove
Svensen, Stephen
Pettersen, Kari

Turner, Sally
Kent, Clark
Svendsen, Stephen
Scott, Stephen

SELECT INTO

The SELECT INTO statement selects data from a table and insert it into another table. The SELECT INTO statement is most often used to create backup copies of tables.

- SQL syntax

We can select all columns:

```
SELECT * INTO new_table_name [IN externaldatabase]
FROM old_tablename
```

Or we can select only the columns we want in the new table:

```
SELECT column_name (s)
INTO new_table_name [IN externaldatabase]
FROM old_tablename
```

- SQL Example

Make a backup copy - Now we want to make a copy of data in the table "People".

We use the following SQL statement:

```
SELECT * INTO Personnes_Backup FROM Persons
```

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

```
SELECT LastName, FirstName INTO Personnes_Backup FROM Persons
```

SELECT INTO - With a WHERE clause

We can also add a WHERE clause.

The following SQL statement creates a "table Personnes_Backup" with only the people living in the city "Sandnes"

```
SELECT LastName, Firstname INTO Personnes_Backup FROM Persons WHERE City =
'Sandnes'
```

SELECT INTO - With joint

Selecting data from more than one table is also possible.

The following example creates a "Personnes_Order_Backup table" that contains the data of the two tables "Persons" and "Orders":

```
SELECT Personnes.LastName, Orders.OrderNo
INTO Personnes_Order_Backup
```

FROM People, Orders
Where Persons.P_Id = Orders.P_Id

Auto Increment a field

Very often we want the value of the primary key field is automatically incremented each time a new record is inserted.

- **Syntax for SQL Server**

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

SQL Views

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

- **SQL syntax**

```
CREATE VIEW AS view_name  
SELECT column_name (s)  
FROM table_name  
WHERE condition
```

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

```
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]
```

Deleting a View

You can delete a view with the DROP VIEW command.

```
DROP VIEW view_name
```

SQL Functions

SQL Dates

The most difficult part when working with dates is to be sure that the date format you are trying to insert, matches the format of the date column in the database.

SQL Server date functions

The most important 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 data date / time in different formats

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

- DATE - format AAAA-MM-JJ
- DATETIME - format: AAAA-MM-JJ HH: MM: SS
- SMALLDATETIME - format: AAAA-MM-JJ HH: MM: SS
- TIMESTAMP - format: a unique number

Working with dates

Suppose we have the following "orders" table:

OrderId	ProductID	OrderDate
1	4	2008-11-11
2	3	2008-11-09
3	6	2008-11-11
4	3	2008-10-29

We want to select the recordings 2008-11-11.

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

DATEPART

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)

datepart can be one of:

datepart	abbreviation
year	yy, yyyy
quarter	qq, q
month	mm, m
dayofyear	dy, y
days	dd, d
week	wk, ww
weekday	ps, w
hours	hh
minutes	mi, n
second	ss, s
millisecond	ms
microsecond	mcs
nanosecond	ns

SQL Example

Suppose we have the following "orders" table:

OrderId	ProductID	OrderDate
1	45	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
```

Gives the following result:

OrderYear	OrderMonth	OrderDay
2008	11	11

DateAdd

Syntax

DATEADD(datepart,number,date)

Example

Suppose we have the following "orders" table:

OrderId	ProductID	OrderDate
1	33	2008-11-11 13:23:44.657

And we want to add 45 days to the "OrderDate", to find the date of payment

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

Result :

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

DATEDIFF

Syntax

DATEDIFF(datepart,startdate,enddate)

Example

We want to get the number of days between two dates.

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

result:

DiffDate
61

NULL values in SQL

NULL values represent missing data.

For example, to see the records from the table People whose addresses are not provided:

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

Or records whose addresses are provided:

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

ISNULL function

Watch the "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 the "UnitsOnOrder" column is optional and can contain null values we have the following **SELECT statement**:

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

the result is null because UnitsOnOrder worth is zero.

So use the ISNULL () function to address this problem. ISNULL () returns a zero if the value is NULL:

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

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
------	-----------	------------	----------

Now we want to find the average value of the "OrderPrice" fields.

```
SELECT AVG(OrderPrice) AS OrderAverage FROM Orders
```

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)
```

SQL COUNT() Function

The COUNT() function returns the number of rows that matches a specified criteria (NULL values will not be counted).

```
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
```

SQL MAX() Function

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

SQL MAX() Syntax

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

SQL MIN() Function

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

SQL MIN() Syntax

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

SQL SUM() Function

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

SQL SUM() Syntax

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

SQL 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
```

Example:

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

SQL HAVING Clause

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

SQL HAVING Syntax

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
HAVING aggregate_function(column_name) operator value
```

Example:

```
SELECT Customer,SUM(OrderPrice) FROM Orders
GROUP BY Customer
HAVING SUM(OrderPrice)<2000
```

SQL UPPER () Function

The UPPER () function converts the value of a field to uppercase.

Syntax for SQL Server

```
SELECT UPPER(column_name) FROM table_name
```

SQL LOWER() Function

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

Syntax for SQL Server

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

SQL 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
```

Example:

```
SELECT LEN(Address) as LengthOfAddress FROM Persons
```

SQL 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
```

column_name: Required. The field to round.

decimals: Required. Specifies the number of decimals to be returned.

Example:

```
SELECT ProductName, ROUND(UnitPrice,0) as UnitPrice FROM Products
```